José Fernando Duarte Marques

Distributed Learning of Convolutional Neural Networks on Heterogeneous Processing Units

Dissertação submetida para a satisfação parcial dos requisitos do grau de Mestre em Engenharia Electrotécnica e de Computadores

Setembro de 2016

· U C ·

Universidade de Coimbra

Background of cover image source: *Distributed Learning of Convolutional Neural Networks across different machines*.

· U C ·

UNIVERSIDADE DE COIMBRA

FACULDADE DE CIÊNCIAS E TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Distributed Learning of Convolutional Neural Networks on Heterogeneous Processing Units

José Fernando Duarte Marques

Dissertação para a obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Júri

Presidente:	Doutor Fernando Santos Perdigão
Vogal:	Doutor Jorge Nuno de Almeida e Sousa Almada Lobo

Orientadores

Orientador:	Doutor Gabriel Falcão Paiva Fernandes
Co-orientador:	Doutor Luís A. Alexandre

Coimbra

Setembro 2016

José Fernando Duarte Marques: *Distributed Learning of Convolutional Neural Networks on Heterogeneous Processing Units*, Tese de Mestrado em Engenharia Electrotécnica e de Computadores, ramo de especialização em Computadores, Faculdade de Ciências e Tecnologia, Universidade de Coimbra © Setembro 2016

Acknowledgments

Gostaria de começar por agradecer aos meus orientadores, Doutor Gabriel Falcão e Doutor Luís Alexandre por toda a ajuda, pelo acompanhamento dado ao meu trabalho, bem como as ideias propostas sem as quais esta tese não teria sido feita. Quero ainda agradecer especialmente toda a confiança que depositaram em mim e principalmente a motivação dada, mesmo quando as coisas correram menos bem.

Em seguida o meu agradecimento vai para os meus companheiros de laboratório, pela boa disposição e toda a ajuda oferecida, em especial para a Andreia e o César, pela companhia durante as noitadas no laboratório, com algum trabalho q.b. à mistura.

Agradeço também a todos os meus amigos que me ajudaram, mesmo que indirectamente com palavras de incentivo e principalmente com distrações, lembrando-me que há mais vida para além da tese. Como diz o provérbio: "All work and no play makes Jack a dull boy". Em especial agradeço ao Ricardo, a Bruna, o Girão e o Pedro, não só por termos sofrido juntos, mas porque isso nunca foi impedimento para a ajuda e diversão.

E porque não podia nem ia deixar passar sem vos agradecer a vocês: Joca, Lúcia, Filipa, Inês e Joana. Porque se apenas tivesse uma linha para os agradecimentos, vocês iam lá estar. Se fui eu que escrevi esta tese, então vocês foram os que me deram força para a escrever. Porque quando achava que já não era capaz de escrever mais, vocês me deram motivação. Porque quando estava mais em baixo, vocês me animavam. Porque estão sempre presentes nos bons e nos maus momentos. Porque isto não é o agradecimento que vocês merecem por tudo o que fizeram por mim, mas é o máximo que as palavras me permitem. Porque são de sempre, e vão ser para sempre.

Quero ainda agradecer todo o apoio que a minha família me deu, por ter acreditado sempre em mim. Aos meus pais, porque sem eles nada disto tinha sido possível. Mais que 5 anos de curso, esta tese é o culminar de 23 anos de apoio incondicional dos meus pais. Porque me incentivaram sempre a aprender mais. Porque me motivaram a ser uma pessoa melhor. Por tudo aquilo que fizeram por mim, e que não sou capaz sequer de exprimir em palavras. Mais que tudo, o que desejo agora é pagar tudo o que vocês fizeram por mim em dobro. Sei que nunca vou chegar sequer perto de o fazer, mas isso não me vai impedir de tentar. Porque não seria nada do que sou sem vocês.

E sei que os agradecimentos já vão alongados, mas se há alguém que não poderia deixar de agradecer é à Cláudia. Agradecer toda a paciência que teve comigo, mesmo quando a tese me começava a consumir. Agradecer todo o apoio que me deu, mesmo quando estava mais distante. Agradecer toda a preocupação que teve comigo. Agradecer todo o carinho que me deu. Sei que isto não chega para compensar tudo o que passou comigo, mas vou esforçar-me para a recompensar. Porque eu sei que não há palavras que cheguem para expressar tudo o que ela fez por mim, nem o amor que me manteve são ao longo destes anos. Porque sem ti, não seria nada.

A todos sem excepção, muito obrigado.

*"No matter what kind of wisdom dictates you the option you should pick,
no one will be able to tell if it's right or wrong until you arrive to some
sort of outcome, resulting from your choice. The only thing that we are allowed to do
is to believe that we won't regret the choice we made."*

- Hajime Isayama, Shingeki no Kyojin

El Psy Congroo

Abstract

The field of deep learning has been the focus of plenty of research and development over the last years. *Deep Neural Networks* (DNNs), and more specifically *Convolutional Neural Networks* (CNNs) have shown to be powerful tools in tasks that range from ordinary, like check reading, to the most essential, being used in medical diagnosis. This evolution in the field has lead to the development of frameworks, such as Torch and Theano, that simplified the training process of a CNN, where the user only needs to create the network architecture, select the ideal hyper-parameters and provide the inputs and desired outputs. However, the easy access to these frameworks lead to an increase in both network size as well as dataset size, since the networks had to become bigger and more complex to be able to achieve more significant results. This lead to larger training times, that not even the improvement of *Graphics Processing Unit*s (GPUs) and more specifically the use of *General-Purpose GPU* (GPGPU) could keep up to.

To solve that problem, several distributed training methods were developed, dividing the workload through several GPUs on the same machine or through *Central Processing Unit*s (CPUs) or GPUs on different machines. This distribution techniques can be divided into 2 groups: data parallelism and model parallelism. The first method consists on using replicas of the same network on each device and train them using different data. Model parallelism divides the workload of the entire network through the different devices used. However, none of these techniques used by the different frameworks takes advantage of the parallelization offered by the CNNs, and trying to use a different method with those frameworks ends up being a task too complex or even impossible.

In the present thesis, a new distributed training technique is developed, that makes use of the parallelization that CNNs have to offer. The method is a variation of the model parallelism, but only the convolutional layer is distributed. Every machine receives the same inputs but a different set of kernels, and the result of the convolutions is then sent to a main machine, known as master node.

This method was subjected to a series of test, varying the number of machines involved as well as the network architecture, with the results being presented in this document. The results show that this technique is capable of diminishing the training times considerably without classification performance loss, for both the CPUs as well as the GPUs. A detailed analysis regarding the influence of the network size and batch size was also included in this document. Finally, a simulation was executed, that shows results using a higher number of machines as well as a possible use of mobile GPUs, whose energy efficiency applied to deep learning was also explored in this work, supported by the contents of Appendix A.

Keywords

Convolutional Neural Networks, Parallel Computing, Distributed Computing, Deep Learning, GPGPU, CUDA

Resumo

A área de deep learning tem sido o foco de muita pesquisa e desenvolvimento ao longo dos últimos anos. As DNNs, e mais concretamente as CNNs provaram ser ferramentas poderosas em tarefas que vão desde as mais comuns, como leitura de cheques, às mais essenciais, sendo usadas em diagnóstico médico. Esta evolução na área levou ao desenvolvimento de frameworks, como o Torch e o Theano, que simplificaram o processo de treino de uma CNN, sendo necessário apenas estruturar a rede, escolhendo os parâmetros ideais e fornecer os inputs e outputs desejados. No entanto, o fácil acesso a essas frameworks levou a um aumento no tamanho tanto das redes como dos conjuntos de dados usados, uma vez que as redes tiveram que se tornar maiores e mais complexas para obter resultados mais significativos. Isto levou a tempos de treinos maiores, que nem a melhoria de GPUs e mais especificamente o uso de GPGPU conseguiu acompanhar.

Para dar resposta a isso, foram desenvolvidos métodos de treino distribuído, dividindo o trabalho quer por várias GPUs na mesma máquina, quer por CPUs e GPUs em máquinas distintas. As diferentes técnicas de distribuição podem ser dividas em 2 grupos: paralelismo de dados e paralelismo de modelo. O primeiro método consiste em usar réplicas de uma rede e treinar fornecendo dados diferentes. O paralelismo de modelo passa por dividir o trabalho de toda a rede pelos diferentes dispositivos usados. No entanto, nenhuma destas técnicas usadas pela diferentes frameworks existentes tira partido da paralelização oferecida pelas CNNs, e tentar usar um outro método com essas frameworks revela-se um trabalho demasiado complexo e muitas vezes impossível.

Nesta tese, é apresentada uma nova técnica de treino distribuído, que faz uso da paralelização que as CNNs oferecem. O método é uma variação do paralelismo de modelo, onde apenas a camada de convolução é distribuída. Todas as máquinas recebem as mesmas entradas mas um conjunto diferente de filtros, sendo que no final das convoluções os resultados são enviados para uma máquina central, designada como nó mestre.

Este método foi alvo de uma série de testes, variando o número de máquinas envolvidas e a arquitectura da rede, cujos resultados se encontram neste documento. Os resultados mostram que esta técnica é capaz de diminuir os tempos de treino consideravelmente sem perda de desempenho de classificação, tanto para CPU como para GPU. Também foi feita uma análise detalhada sobre a influência do tamanho da rede e do batchsize no speedup conseguido. Por fim, foram também simulados resultados para um número superior de máquinas usadas, bem como o possível uso de GPUs de dispositivos móveis, cuja eficiência energética aplicada ao deep learning foi também explorada neste trabalhado, suportado pelo conteúdo do Appendix A.

Palavras Chave

Redes Neuronais de Convolução, Computação Paralela, Computação Distribuída, Deep Learning, GPGPU, CUDA

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ALU	Arithmetic and Logic Unit
CNN	Convolutional Neural Network
COTS HPC	Commodity Off-The-Shelf High Performance Computing
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
FLOPs	Floating-Point Operations per second
FPGA	Field-Programmable Gate Array
GPGPU	General-Purpose GPU
GPU	Graphics Processing Unit
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IP	Internet Protocol
NN	Neural Network
OpenCL	Open Computing Language
RAM	Random-Access Memory
ReLU	Rectified Linear Unit
RGB	Red Green Blue

1 Introduction

Contents

Machine learning has been the engine behind tasks that are considered normal nowadays. From using search engines to depositing checks at the ATM[1], including filtering content on social media, to tasks that range from medical diagnosis[2] to game playing[3]. It is ever more present all around, particularly on smart appliances (like smart homes[4,5] and smartphones).

Currently, one of the major problems regarding machine learning in several real world applications is the large amount of variables on which the data depends. For instance, the image of a green shirt might look darker when there is low illumination, or the format of a car might depend on the image's angle. Simply using representation learning, which is when a machine takes raw data to discover representations needed for classification, is not very helpful, since it is necessary to obtain abstract features that seem normal to humans but are difficult to express using logic or mathematical reasoning.

Deep learning is able to complement representation learning and solve this problem, since it allows the creation of complex concepts out of simpler ones. This is achieved by using multiple levels of representation, each more abstract than the previous layers. Figure 1.1 shows this concept clearly. The layers from this type of model can be classified according to three types: visible, hidden and output layer. The visible layer is named so because it represents the variables that can be seen (like the pixel values of an image). On the other hand, the values from the hidden layer are not explicit in the input data. Following the given example, the first hidden layer is able to detect the presence or absence of edges in certain orientations. Having the edge representations, the second hidden layer is able to detect corners and contours, that are nothing more than arrangements of edges. With the corners and contours representations, the third hidden layer is capable of identifying parts of certain objects. Finally, the output layer is able to say what is the object present in the image. It should be noted that the most important aspect of these types of models is that such layers are all learned from the data without any human intervention.

1.1 Motivation

Despite being around since the 1970's and 80's[7], even with commercial applications[8], it wasn't until recently that deep learning really boomed. This can be attributed to mainly two factors: the size of the datasets and the increase in model's size.

Most datasets prior to the 1990's, such as the Iris dataset[9,10] or the TIMIT dataset[11] had less than a couple thousand instances available to train the networks. While it may have been sufficient for classifying between three types of Iris, it became insufficient as machine learning evolved and more complex tasks were required. The creation of new datasets was almost nonexistent prior to 2000, but that changed as society evolved and

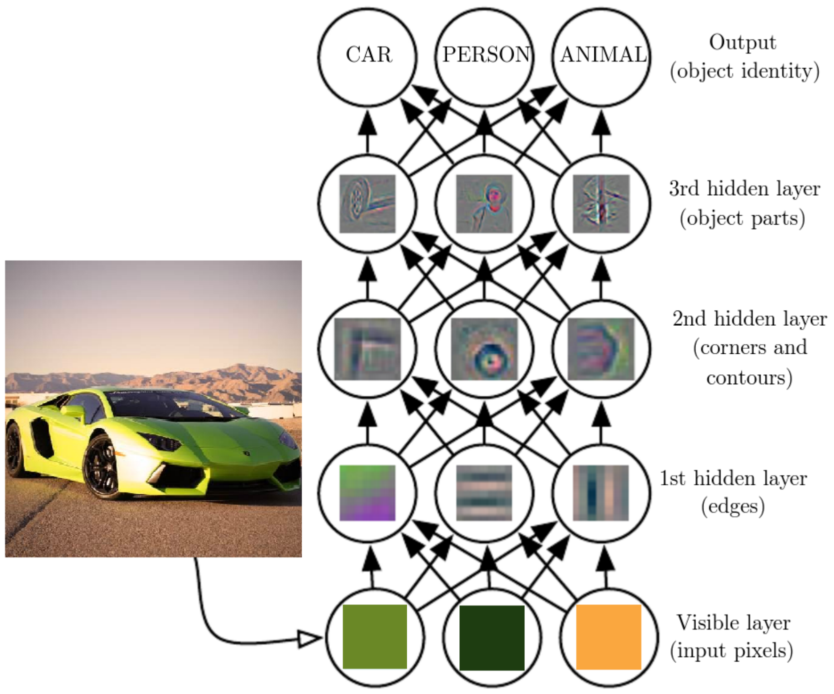


Figure 1.1: Example of a deep learning model. Each layer represents a higher level of abstraction than the previous layer[6].

became more dependent on new technologies. Since computers are evermore connected and considering that information is kept, the number of datasets spiked, as well as the number of instances they contain. This progression is visible in Figure 1.2.

On the other hand, the increase in model's size relates to the number of neurons required to obtain significant results, as neural network with few neurons cannot solve the most recent machine learning problems. Therefore, it was necessary to create larger models and networks, capable of achieving better results, even for the most complex tasks. Thus, deep learning and particularly *Deep Neural Networks* (DNNs) emerged over the last years.

Obviously this boom was only possible thanks to the technological development that occurred since the 1980's, that allowed the access to computational resources capable of training increasingly larger neural networks, and also larger datasets. More specifically, it was due to the development of faster *Central Processing Unit*s (CPUs) and *Random-Access Memories* (RAMs), the increase in available memory/storage, and also due to the improvement of distributed training infrastructures. It was only then that it was possible to create frameworks like DistBelief[12], capable of training networks with as much as 1.7 billion parameters, currently one of the largest networks. However, it should be noted that this framework uses thousands of CPU cores distributed along hundreds of

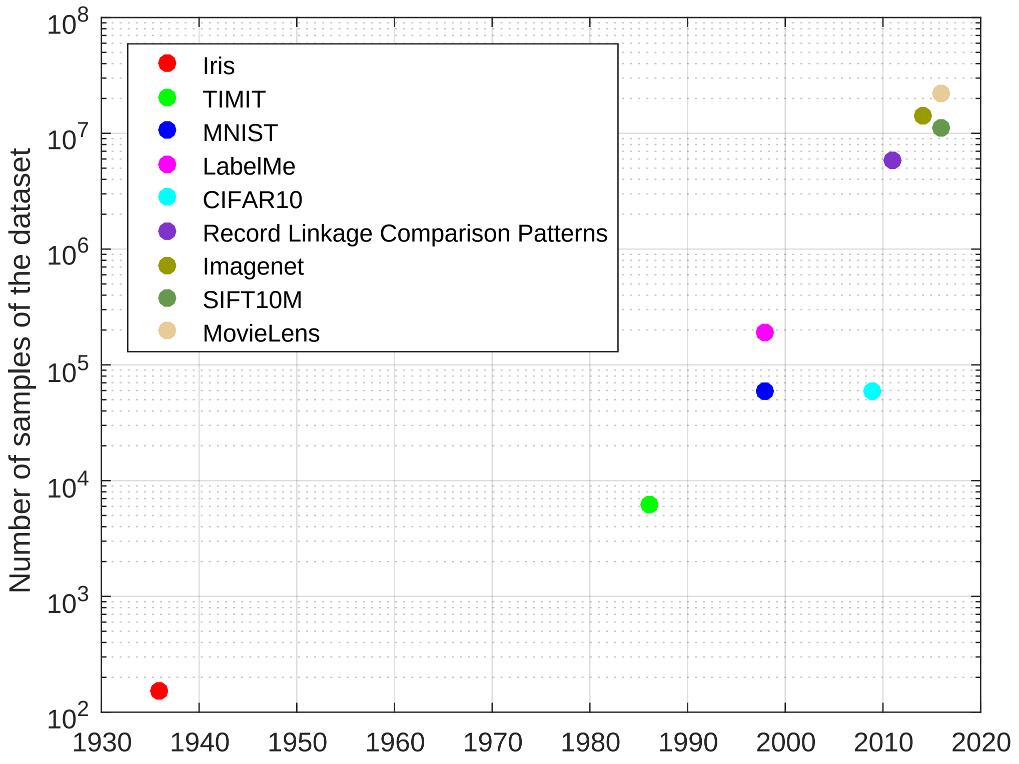


Figure 1.2: Evolution of machine learning datasets through time.

machines and the training takes a few days. It produces very good results, achieving a cross-validated classification accuracy of 15% on ImageNet dataset[13].

However, one of the most important technological developments that helped the evolution of deep learning was the improvement of *Graphics Processing Unit* (GPU), more specifically the use of *General-Purpose GPU* (GPGPU). These newly available resources allowed the speeding up of network training time, through the development of parallel computing frameworks like *Open Computing Language* (OpenCL) and *Compute Unified Device Architecture* (CUDA). This is only possible because GPUs, despite working at smaller frequencies than CPUs, have a higher number of cores and are more efficient at receiving a large batch of data and repeating the same operation very quickly, something that happens during neural network training.

A demonstration of the GPU's efficiency can be seen by analyzing the *Commodity Off-The-Shelf High Performance Computing* (COTS HPC) system, that is able to train a network with the same size as the one trained by DistBelief in only a couple of days using only 3 machines, each one with 4 GPUs, which represents a resource usage decrease by two orders of magnitude[14]. Moreover, it is capable of training a network with over 11 billion parameters using 16 machines in only 3 days.

However, deep learning evolution evolved into a new kind of neural network: the *Convolutional Neural Network* (CNN). The first major contribuition from a CNN to the growth of deep learning appeared when this kind of network was used to win the *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC), the largest contest in object recognition, by diminishing the top-5 error rate from 26.1% to 15.3%[15]. This means that the CNN creates a list of possible categories for each image, from 1000 possible categories, and the correct one always appears amongst the first 5 except 15.3% of the times.

The major evolution from DNNs towards CNNs lies in the convolution that is performed during the training of this last type of network, which makes it the ideal choice for the image and speech recognition, since both of these tasks rely heavily on the correlation of neighboring data. The major problem with it is that the computation of the convolution is computationally intensive, with 60% to as much as 90% of the total training time being spent doing convolutions, that only use about 5% of the parameters of the whole network[16,17].

Although proprietary and closed methods for the training of CNNs exist, they either distribute the whole network across different machines, making use of unnecessary communications in intermediate layers, or creating replicas of the same network across those devices, not being able to create large networks and losing some of the information when the parameters of all the networks are averaged. Thus, the main motivation for this thesis consists of developing an open source distribution technique that makes use of the potential parallelization that convolutional layers have to offer, feeding different devices the same feature maps but providing them with different kernels, gaining speed up during the convolution computation that can even compensate the communication time between the different nodes.

1.2 Objectives

Considering the above, the proposed goal of this thesis is to provide a new method to distribute the training of a CNN, making use of the parallelization of the convolutional layer. Therefore, the main focus is to be able to make the convolutions needed for the network training across different machines, using different devices, like CPUs, GPUs and hybrid combinations of both and be able to achieve significant speedups. The final goal consists of analyzing the impact of the network hyperparameters, such as number of kernels, latency time and specially the impact that the number of used devices has on the global network training time.

1.3 Main Contributions

In this thesis, a distribution of kernels among different machines is proposed, to optimize the convolutional phase of the training of a CNN, using several machines. Below are highlighted the main contributions of this work:

- Distributed learning on clusters of CPUs;
- Distributed learning on clusters of GPUs;
- Distributed learning using hybrid CPU-CPU and GPU-GPU computing;
- Tradeoff analysis based on CNN size and number of devices available.

1.4 Dissertation Outline

This thesis is organized in 6 chapters. Following the introduction, chapter 2 focuses on CNN, and in alternative ways of performing the training of these *Neural Networks* (NNs) distributed over a group of computing machines, in order to achieve real speedups in terms of learning/training period/time, providing some insight on the most recent techniques in that area. Chapter 3 discusses the major principles of GPU architectures, which is then complemented with an explanation of the CUDA framework. Chapter 4 details the algorithm changes implemented on the training of a CNN as well as the dataset and hardware adopted in the experimental results section. The experimental results are presented in chapter 5 and the conclusions and future work are detailed in chapter 6.

2

Convolutional Neural Network and Distribution Techniques

Contents

Over the last years, *Deep Neural Networks* (DNNs) have shown great promise in several practical applications, achieving state-of-the-art performance on a variety of different tasks, including object recognition[18,19] and speech recognition[20,21], among others. They were even able to achieve strong super-human performances, performing better than all humans or the best ones at it, in games like chess[22] and Go[23].

However, despite having shown to be a powerful machine learning technique, DNNs are usually applied to cases where the input dimension is large, like images or sound bits. Because of this they often encompass complex and sizable networks, where a considerably high number of connections and weights coexist. Additionally, their training also requires the use of large datasets, which in addition to the aforementioned structure, makes them very demanding in terms of computational resources. For example, given Figure 2.1 from the ImageNet dataset[13], which contains over 14 million images separated over 21 thousand categories, considering the image size is 500×333 and is *Red Green Blue* (RGB), the input layer would have 499500 inputs. For a fully connected hidden layer having only 100 neurons, the number of weights would be 49950000.

Figure 2.1: Example from the ImageNet dataset[13] depicting a domestic dog.

Even considering the CIFAR-10 dataset[24], which consists of much smaller RGB images of 32×32 size, the input layer would have 3072 inputs, with 307200 weights using a fully connected hidden layer with 100 neurons, without even considering more hidden layers or the output layer. Given that this dataset only has 60000 images, the network would be too complex and overfitting would tend to happen.

Another problem is that these networks neglect correlation between neighbouring data, like translations and distortions, despite there being a relatively high amount of local correlation in pattern recognition problems. Ideally, local features would be extracted and analyzed to be able to detect certain beings or objects. *Convolutional Neural Network* (CNN) however are able to overcome those issues by making use of 3 key factors: local receptive fields, weight sharing and spatial pooling.

2.1 Convolutional Neural Networks

The concept of CNN was first introduced in 1995 by Yann LeCun and Yoshua Bengio[25] and was greatly inspired by the discovery of locally sensitive and orientation-selective neurons in the visual cortex of a cat. By using local receptive fields it is possible to exploit local visual features, like edges, corners and end-points (in images). This is advantageous because adjacent pixels tend to be strongly correlated while pixels that are farther apart are usually uncorrelated, or have a weak correlation. Having the ability to share weights across locally connected neurons allows reducing the amount of parameters to train, decreasing the amount of data needed, making the training faster and easier, achieving better classification performances when compared to DNNs.

The main differences between CNNs and DNNs are the use of convolutions and pooling (or subsampling) operations, instead of simple matrix multiplication in at least one layer. One of the most popular CNNs is LeNet-5[8], illustrated in Figure 2.2. It contains 2 convolutional layers and 2 subsampling layers interleaved, ending with fully connected layers. The network can be tuned by changing the number of layers, the number and size of filters from the convolutional layer or the stride of the subsampling layer.

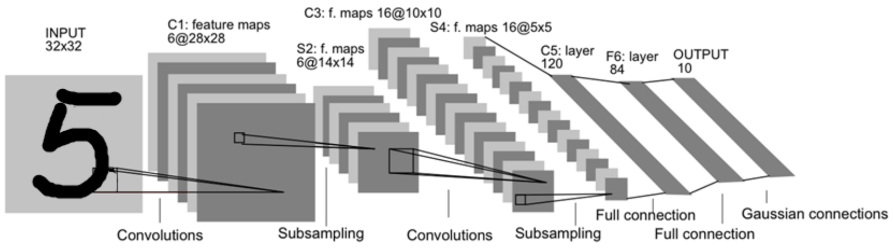


Figure 2.2: The architecture of LeNet-5[8], a CNN used for digits recognition.

2.1.1 Convolutional Layer

The input of a convolutional layer is usually a multidimensional array of data, while the kernel is a multidimensional array of parameters that readjusts through the network training. A convolution operation then applies those kernels to the inputs, as to detect the most appropriate features. Thus, the outputs of the convolution are called feature maps. Mathematically, for a 2D image and a 2D kernel:

$$F(i,j) = (K * I)(i,j) = \sum_h \sum_w I(i-h, j-w)K(h,w) \tag{2.1}$$

where F is the output, or feature map, I is the image and K is the kernel. Despite the equation requiring to flip the kernel, this is not a necessary operation of a DNN implementation, as most neural network frameworks implement the cross-correlation operation, as depicted in Equation 2.2, with the same objective. Strictly speaking, the two operations are different, as one requires the kernel to be flipped while the other does not. However, most of the times, the difference is irrelevant, since a network will learn the best values for the kernel using both operations. It will just learn how to detect different features.

$$F(i,j) = (K * I)(i,j) = \sum_h \sum_w I(i+h, j+w)K(h,w) \tag{2.2}$$

The reason for the popularity of convolutional layers is due to their ability to work with variable sized inputs, to which sparse connectivity and parameter sharing provided important contributions.

Sparse connectivity (or sparse weights) happens when the outputs only have a certain amount of connections. Considering the case of an image, that may have thousands to millions of pixels, a kernel is used to detect small features, thus storing few parameters and limiting the number of outputs. Figure 2.3 is able to demonstrate how sparse connectivity works on CNNs.

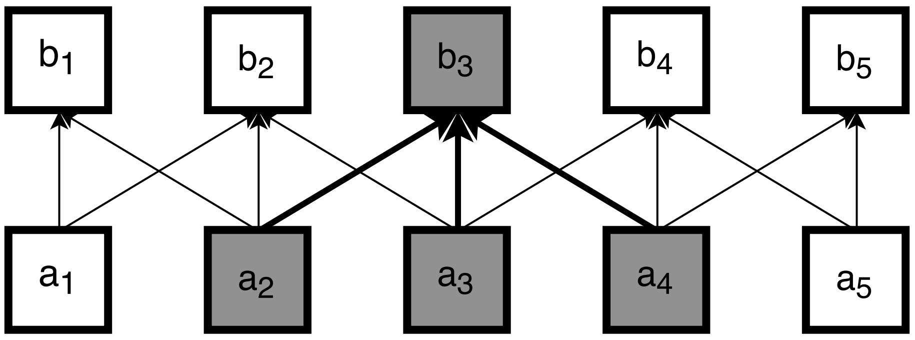


Figure 2.3: Example of sparse connectivity with two layers. Considering the case where layer a is formed by a convolution with a width 3 kernel, the output b_3 is only affected by three inputs.

As depicted in Figure 2.3, the use of convolutional layers limits the direct interaction between inputs and outputs, meaning that a change in one neuron only directly influences the value of 3 neurons, when the kernel width is 3.

However, in the case of CNNs with several layers, the value of a neuron from a deep layer can be indirectly influenced by the value of neurons from shallow layers. It is then possible to create bigger and more convoluted filters, built from far simpler ones, as can be seen in Figure 2.4.

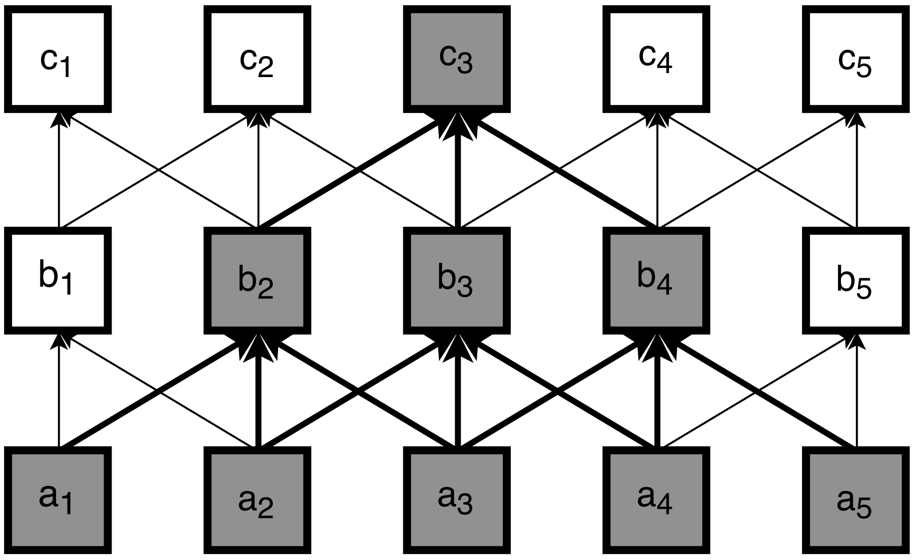


Figure 2.4: Example of sparse connectivity with three layers. Even though the neuron c_3 only depends on three neurons on layer b, those neurons depend each on three neurons from layer a, meaning that neurons from deeper layers are indirectly influenced by neurons from the first layers.

Parameter sharing is used to reduce the number of parameters, which can be achieved using the same filter across the entire input. This means that instead of learning a separate set of parameters for every possible location, it is only necessary to learn one set, allowing to detect features regardless of their position in the input. To learn more features, more filters must be used, so that they can be trained to detect different features.

After detecting a certain feature, only the approximate position in comparison to other features is relevant. Considering the CIFAR10 dataset, by detecting antlers and a snout relatively close to each other, it is easy to assert that the image contains a deer. It is imperative to use only the relative positions, since the precise position of features is likely to change in different images, and it may ruin the training of the network. A simple way to relativize the positions of certain features is to reduce the spatial resolution of the feature map, which can be done using pooling, or subsampling, that are the final key factor in a CNN. The goals at this point are to reduce the size of convolutional responses and to add invariance to small transformations.

2.1.2 Pooling Layer

The pooling layer is a form of non-linear down-sampling, that partitions its input into several non-overlapping blocks and evaluates a pooling function over each block. The most used pooling function is the max pooling, as seen in Figure 2.5, that outputs the maximum value on each block, but there are several others like the average pooling and the L^2-norm pooling, that compute the average and the L^2-norm of each block, respectively.

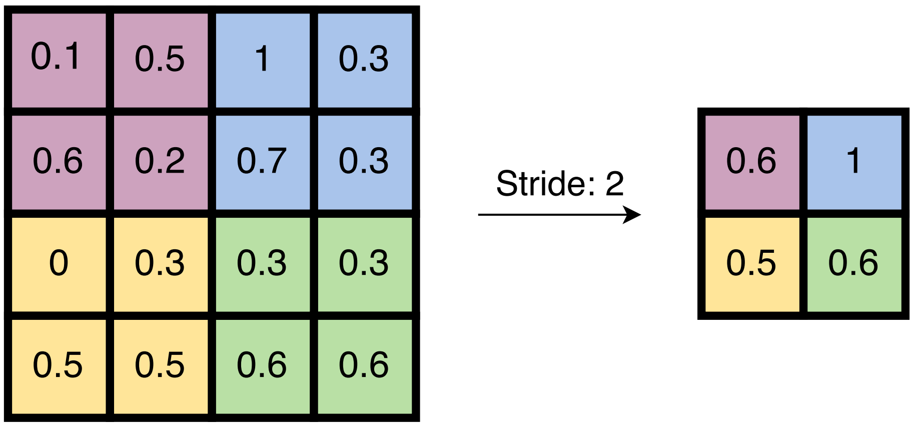


Figure 2.5: Example of max pooling with a stride of 2. The inputs are separated into 2x2 blocks and the maximum of each block is computed.

In every pooling function, the goal is to make the network invariant to small transformations, meaning that if the input was translated by a small amount, the values of most pooled outputs would remain the same, which is particularly important if the presence of a certain feature is more relevant than its position. Figure 2.6 shows exactly this point. By taking the inputs of Figure 2.5 and shifting them to the right by 1 pixel, almost all the values of the inputs were changed, but the output after the pooling function only changed one value.

The pooling layer can also be used to perform dimensionality reduction in the feature map, trimming the amount of parameters and computation required to train the network, thus controlling the overfitting.

However, despite convolutional and pooling layers being the major differences between DNNs and CNNs, this type of neural networks makes use of several other layers, particularly *Rectified Linear Unit* (ReLU), fully connected and loss layers.

2.1.3 ReLU Layer

The ReLU layer applies an activation function, named rectifier, defined as $f(x) = \max(0, x)$, where x is the input of the neuron. Although before the creation of this layer,

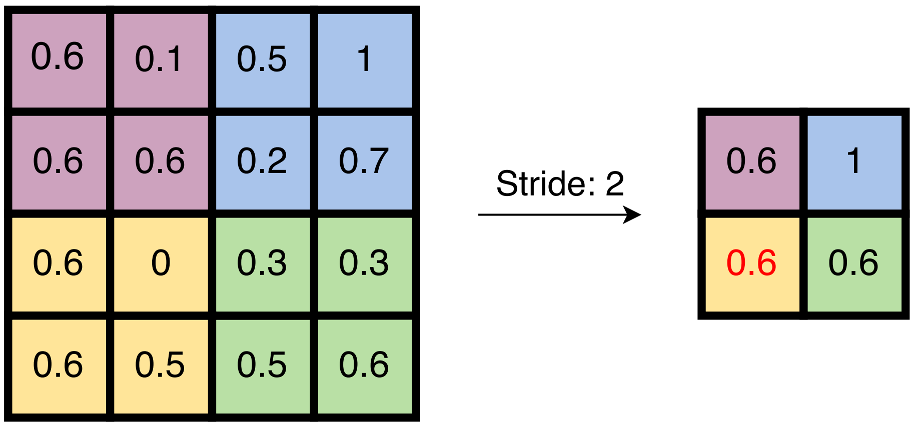


Figure 2.6: Example of max pooling with a stride of 2. The inputs are separated into 2x2 blocks and the maximum of each block is computed. The inputs from Figure 2.5 where shifted to the right 1 pixel, but only one value from the output was affected.

other types of activation functions were used, such as the logistic sigmoid or the hyperbolic tangent, it is considered that the rectifier is more biologically plausible[26]. Not only that, but it is significantly less computationally expensive then other activation functions and it is also able to train networks several times faster than equivalent ones using hyperbolic tangent[15]. There are cases where networks with such activation function can have neurons that become inactive, since the weights can be updated in such a way that the input of the neuron will always be negative, meaning the output of the neuron will always be 0. However, selecting a proper value for the learning rate will diminish the probability of it happening.

2.1.4 Fully Connected Layer

The fully connected layer has neurons with full connections to all activations, like in regular DNNs. Their activations are calculated with a simple matrix multiplication, with a bias offset. Usually they are the last layers used in CNNs, since they are not spatially located and are one-dimensional, meaning that there cannot be convolutional layers after.

2.1.5 Loss Layer

The final layer of CNNs is the loss layer, that is a fully connected layer where a loss function is applied, that allows to calculate the difference between the predicted and true labels. Different tasks use the appropriate functions. The most used loss function for classifying images is softmax, that predicts a single class of N mutually exclusive classes, by normalizing the vector, which implies that every class has values from 0 to 1, and the sum is always 1.

However training the largest CNNs is becoming a real challenge even using *Graphics Processing Unit*s (GPUs), either because these parallel machines are limited in memory or simply because the training times are just too long. Performing the distributed training of CNNs is an appropriate strategy that fosters accelerating this type of complex processing. The next section aims to provide some insight regarding the most recent techniques of distributed training.

2.2 Distribution Techniques

Distributed training can refer to distributing the training of the network across several GPUs in the same computer or in different computers, with the latter also being applied to *Central Processing Unit*s (CPUs). There are mainly two types of techniques for the distribution: data parallelism and model parallelism.

2.2.1 Data Parallelism

In data parallelism the batch of data is split across the several nodes from the cluster, may they be CPUs, GPUs or a combination of both. Each node is then responsible for computing the gradients with respect to all the parameters, but does so using part of the batch. However, since every node is running a replica, it is necessary to communicate the gradients and parameters values on every update step. Another problem with this approach is that since every node calculates different gradients, they need to be averaged, and that causes the loss of information and may hinder the training process.

Another condition for the use of this type of parallelism, especially when using GPUs is that the batch size must be large enough to be distributed and still be able to exploit the highly parallel capabilities of the GPU. An example of data parallelism is present in Figure 2.7.

2.2.2 Model Parallelism

Model parallelism consists of dividing the network's computation across the several nodes, that may differ considering the type of network used. In the DistBelief[12] case, the DNN is partitioned across several nodes and only the nodes with edges that cross partition boundaries need to have their state transmitted between nodes. Another implementation[27] separates the first convolutional layer across several nodes, dividing the number of kernels, with each node calculating a part of the network, having only cross connections at one intermediate layer and at the very top fully connected layers. This implementation is visible in Figure 2.8.

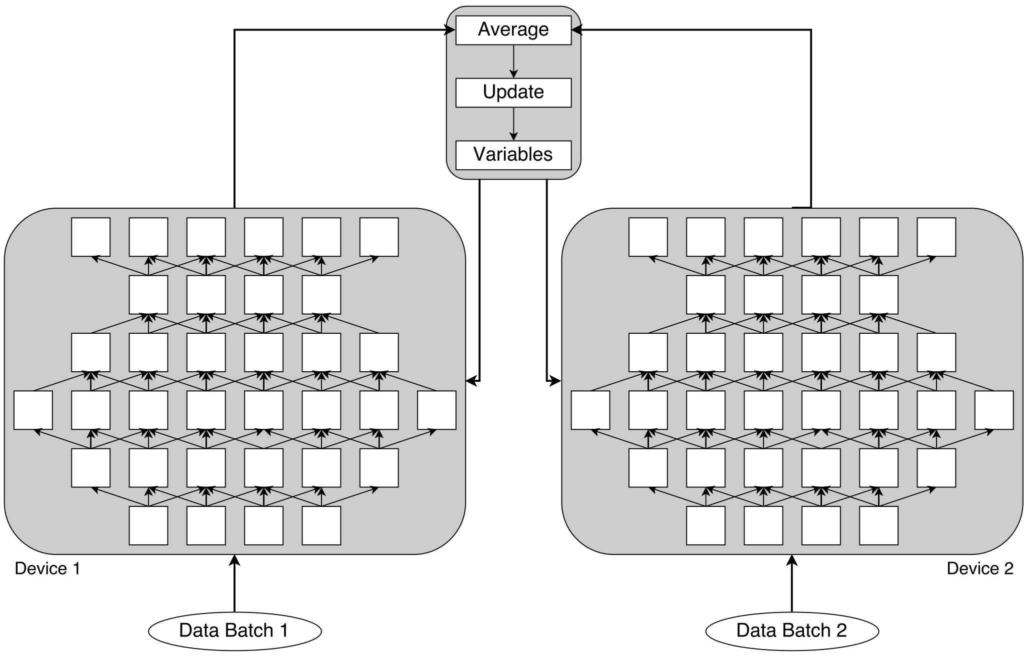


Figure 2.7: Data parallelism.

Another type of model parallelism can also be considered by splitting the image in tiles that are represented by thread blocks per output feature map. Each tile is analogous to a thread block and each pixel is represented by a thread, where a tile represents a different image[16]. However, this type of distribution is only efficient in cases where the image and batch size is large enough, and when there are not many kernels to be convoluted, since every node will need to have every kernel necessary.

The distribution technique devised in this thesis can be thought as another type of model parallelism, since the workload of the network is distributed across several machines. This will be further detailed in Section 4.

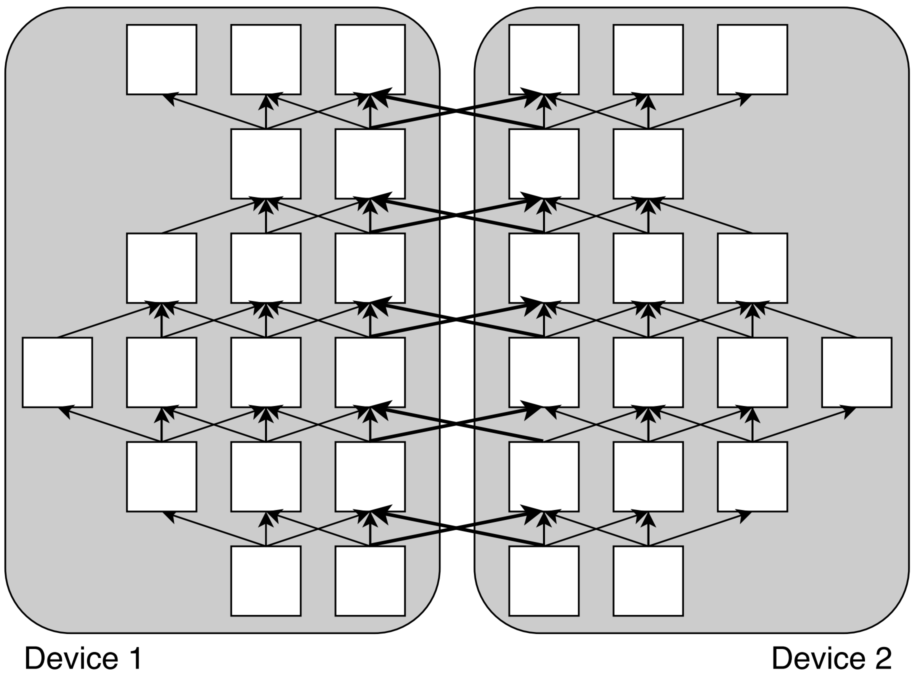


Figure 2.8: Model parallelism.

3 Parallel Computing

Contents

Moore's Law stated, in 1965, that the number of transistors in integrated circuits had doubled every two years, and would continue at that rate or even increase over the years[28]. It turned out to be true, as chips nowadays can contain billions of transistors[29,30]. With that increase in computational power, the single-core was quickly replaced with multicore, whose computational resources were exploited with the development of parallel computing. With it, computer architectures evolved even further and are currently monopolized by the massive use of two distinct platforms: the *Central Processing Unit* (CPU) and the *Graphics Processing Unit* (GPU).

This chapter presents an overview of CPU and GPU architectures, as well as a focus on GPU programming, specifically using the *Compute Unified Device Architecture* (CUDA) framework.

3.1 Central Processing Unit (CPU) Overview

The CPU was created to execute general-purpose tasks, able to perform basic arithmetic, as well as logical operations, but also input and output tasks, that are specified by instructions. Although the CPU is in constant evolution, as stated above, the conceptual hardware model stayed virtually the same. This architecture comprehends an *Arithmetic and Logic Unit* (ALU), that performs arithmetic and logic operations, registers, that supply operands to the ALU and store results of ALU operations, and a control unit, which is responsible for managing the flow of instructions.

A high-level view of the CPU is depicted in Figure 3.1.

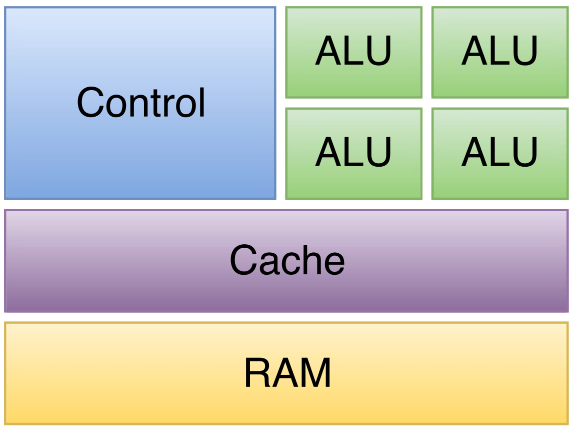


Figure 3.1: High-level view of a CPU architecture.

Since the CPUs were developed to complete general-purpose tasks, one of the most important aspects is memory access time, so the CPU is able to perform calculations and make decisions as quickly as possible. In order to minimize that time, a pyramidal model for the memory was developed, with the top corresponding to the fastest cache, but also the smallest, with the bottom containing the largest memories with the longer memory access times.

As technological resources continue to evolve, CPUs increase their processing power, adding more cores and enabling thread execution. This, allied with the enclosure of several cores on certain levels of cache systems, provides a shared memory for threads to access each other's data with fast on-chip memories.

3.2 Graphics Processing Unit (GPU) Overview

Initially, the GPU was developed exclusively as a graphics processor, managing graphics related functions, like image rendering. This allowed the GPU to evolve as a highly specialized graphical oriented parallel machine, where most of the available chip area is used by computational units. Unlike the CPU, that minimizes memory access time using a hierarchic model of memory with small but fast caches, the GPU relegates that pyramidal system to accommodate more ALUs, forgoing latency in favor of better throughput performance, so as to complete the tasks as fast as possible. This allows the GPUs to handle the many calculations required to manipulate and render the graphics, which requires the execution of the same calculation over a large batch of data. Since that processing must occur in real time, it needs to be completed as fast as possible.

This way, the GPU benefits from kernels with high arithmetic intensity, which is the ratio between arithmetic and memory instructions, since it will overshadow the memory latency. This is also possible due to their architecture, or more specifically due to the number of cores that GPUs normally contain, which are all able to execute threads. As each thread performs a task on a data sample, a large batch of information can be worked on by all the threads simultaneously.

The memory system present in GPUs is *Dynamic Random Access Memory* (DRAM), which uses capacitors to store the bit value. However, despite allowing the incorporation of more memory on the chip area, it increases the latency. A high-level view of the GPU is depicted in Figure 3.2 and is designed to serve as a comparison to the architecture of a CPU, shown in Figure 3.1.

The DRAM is divided into cells, that store the actual data and have a rectangular, grid like pattern. Since consecutive cells in the same row have successive addresses, the latency of sequential accesses to the same row is low. Also, as a way to reduce overall memory access time, the controller schedules accesses the same row before scheduling accesses to different rows.

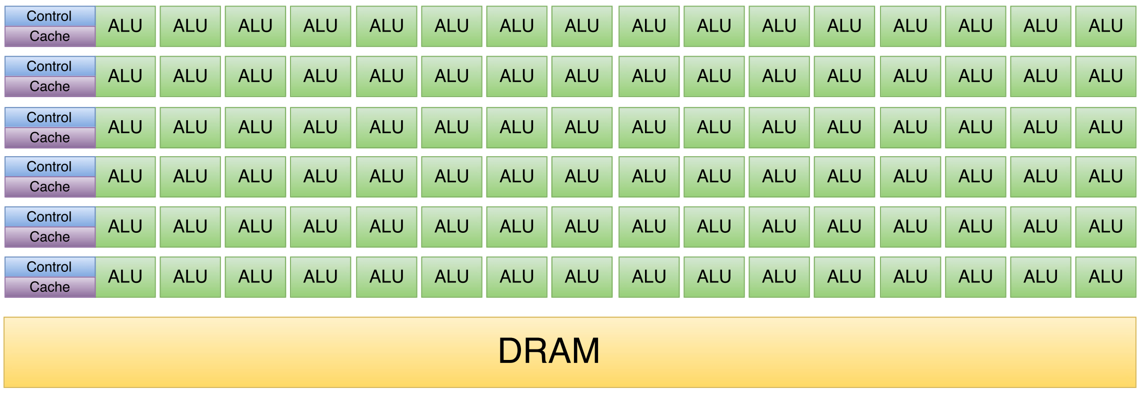


Figure 3.2: High-level view of a GPU architecture.

Another key feature of the GPUs is the incorporation of large buses, that are able to fetch large quantities of data to the compute units, thanks to the bandwidths they provide.

All this confers the GPU an ability for high levels of parallelization, capable of outperforming the CPU on arithmetic intensive algorithms.

3.3 GPU programming

As stated above, the GPU's main advantage is its ability to execute a program in parallel. When programming on a GPU, one of the most important things to do is locate the potential parallelization in the algorithm. This is the best way to take advantage of the GPU's full computational potential. However, a CPU is always required when using the GPU, since the CPU is required to manage the execution of the program. Specifically, the CPU is necessary to determine which portions of the program are performed by the GPU and the parameters needed. The CPU is also responsible for the memory management, transfering memory from and to the GPU. However, it is a time costly operation and should be limited as much as possible.

Other drawbacks from the GPU include the clock speed, as it is slower than the CPU's, less memory and cache. This limits the ability of the GPU to run in serial execution. For a better use of the GPU's computational resources, it is necessary to separate the portions of the program that are serial from the ones that are parallel, and run the serial ones with the CPU and the other with the GPU. If an operation has to be executed enough times and can be done in parallel, then the GPU can more than compensate for its slower execution.

3.3.1 CUDA

In order to turn GPUs into fully programmable devices, programming languages had to be created. With the objective of having a dedicated language, NVIDIA created the CUDA framework, that exclusively works with the company's devices, as is the case of almost all the computers used in this thesis. It provides a data parallel programming language with C++ support, handling the GPU specificities through parallel data structure and keywords, for parallel programing.

There are three key aspects in CUDA: the hierarchy of thread groups, the shared memory model and barrier synchronization. Bundled together the three offer the programmer the design tools required to express algorithms for parallel execution.

The parallel part of a program is known as a kernel, an extension to the function feature of the C programming language, that is executed across the GPU hardware, where the execution grid is composed of threads and blocks of threads. Blocks can be one to three dimensional and are defined by the programmer, limited by the amount of memory resources that each thread consumes. This strategy is limited in communication and synchronization allowed for threads in the execution grid. Threads can communicate via shared memory and synchronize with the other threads in the same block, where each block runs independently from one another.

The heterogeneous programming model behind CUDA assumes that the CPU acts as a controller to the co-processor, i.e. the GPU, where the actual execution of a kernel will take place. This model comes at the expense of having, at least, two different memory regions, one controlled by the CPU, that acts as the host, and the other by the GPU, that acts as the device. However, most recent architectures come with *Direct Memory Access* (DMA), where the programmer no longer needs to maintain the memory spaces coherence manually.

The CPU, or host, allocates the space in the device's global memory and sends the data segment to the device. This allows the GPU to already have all the necessary data available to perform calculations when the kernel is called. Once the kernel finishes its execution, the host copies the data to its memory, followed by deallocating the memory space in the device.

The GPU is responsible for running the parallel kernels, where the kernel specifies the amount of work each thread performs. It is built in a way that allows each thread to be responsible for an index of the matrix. If memory accesses have the data pre-aligned on the device's memory, all indexes are calculated and stored at the same time.

However, one of the problems of developing parallel CUDA code programs lies with using the resources in the most efficient way. In order to do so, there are some techniques that can significantly improve the program's performance. Beside the shared memory already discussed above, one of the things to avoid is thread divergence. Divergent conditions such as if-else statements influence the processing time, because all the paths have to be verified by all threads, adding more cycles. So, these branching statements should be avoided while parallelizing. Another technique is to use loop unrolling, so as to reduce the number of times the condition to perform the jump is met. This can be done by the programmer or by the CUDA compiler. The final technique is asynchronous memory transfers, where the memory transfers are performed while executing the kernel, reducing the impact of memory transactions, increasing the throughput of the program.

4 New methodology for distributed training of CNNs

Contents

The goal of this work is to provide a different method of distributed training that takes advantage of the structure of a *Convolutional Neural Network* (CNN). As said in a previous chapter, the convolutional phase makes up for nearly 60% of the training time[16] and may even contain up to 90% of computation time, using about 5% of the network's parameters[17].

4.1 Problem analysis

The first step towards implementation lies in the analysis and definition of the task at hand. Chapter 2 provides the theory behind CNNs and the different methods of distributed training. It is important to note that the this method shares similarities with the model parallelism, as part of the network is shared among several nodes and the next section explains the differences and how they benefit the training of the network.

4.1.1 Distributed approach

The similarities between this distributed approach and model parallelism lie in sharing part of the network. However, where the nodes using model parallelism always compute the same part of the network and communications are kept to a minimum, this approach only does the convolutions for the convolutional layer. This works because the convolutional layers have high representation using less than 10% of the parameters[17], meaning that the communication overhead will not be a relevant problem when compared to the computation time saved.

The master node sends the size and number of inputs, that can be images or feature maps from previous layers. It also sends the size and number of kernels needed for the convolution, with different nodes receiving a different number of kernels, as further explained in Section 4.1.2. All this information regarding input, kernel size and number is necessary so that the slave knows how much data to read from the socket and how it should reshape it, since data read from sockets comes in vector form. After every node concludes their part of the convolutions, every slave sends their feature maps, where the master node reshapes and rearranges them.

The process is repeated until the training of the network is over, with the master node sending a shutdown flag to every slave. This training distribution technique can be better evaluated by analyzing the Algorithms 4.1 and 4.2, referring to the master node and the slave nodes, respectively.

Algorithm 4.1 Master node

```
for $slave = 1$ to $numSlaves$ do
  $connectSocket(slave)$
end for

while training do
  for $layer = 1$ to $numLayers$ do
    if convolutional layer then
      for $slave = 1$ to $numSlaves$ do
        /*All slaves receive the same inputs*/
        $inputWidth, inputHeight, numInputs \Rightarrow writeSocket(slave)$
        $input \Rightarrow writeSocket(slave)$

        /*Different slaves receive different kernels*/
        $numOutputMaps(slave) \Rightarrow writeSocket(slave)$
        $kernelWidth, kernelHeight, numKernels(slave) \Rightarrow writeSocket(slave)$
        $kernels(slave) \Rightarrow writeSocket(slave)$
      end for

      for $outputMaps = 1$ to $numOutputMaps$ do
        $output \Leftarrow convn(input, kernels)$        /* The outputs are the feature maps */
      end for

      /*Master node receives and combines all the feature maps*/
      for $slave = 1$ to $numSlaves$ do
        $outputWidth, outputHeight, numOutputs \Leftarrow readSocket(slave)$
        $output \Leftarrow readSocket(slave)$
        $output \Leftarrow reshape(output, [outputWidth, outputHeight, numOutputs])$
        $allOK \Rightarrow writeSocket(slave)$
      end for
    end if
  end for
end while

for $slave = 1$ to $numSlaves$ do
  $trainOver \Rightarrow writeSocket(slave)$
end for
```

4.1.2 Hybrid CPU-CPU and GPU-GPU computing

One of the major problems that arises with the usage of computers having different *Central Processing Unit*s (CPUs) and *Graphics Processing Unit*s (GPUs) is that different devices are able to complete the same workload in different times. This can become a problem, especially when one or more of the devices are relatively slower. For example, considering two devices: Device 1, that can complete an arbitrary workload in only 10 seconds, and Device 2, that completes the same work in 20 seconds. If the workload were

Algorithm 4.2 Slave nodes

```
$connectSocket(server)$

/*When the training is over, the master sends a bit that tells the slaves to shutdown*/
while $trainOver = 0$ do
    while $bytesReceived = 0$ do
        pause(1)
    end while

    $inputWidth, inputHeight, numInputs \Leftarrow readSocket(server)$
    $input \Leftarrow readSocket(server)$
    $input \Leftarrow reshape(input, [inputWidth, inputHeight, numInputs])$

    $numOutputMaps \Leftarrow readSocket(server)$
    $kernelWidth, kernelHeight, numKernels \Leftarrow readSocket(server)$
    $kernels \Leftarrow readSocket(server)$
    $kernels \Leftarrow reshape(kernels, [kernelWidth, kernelHeight, numKernels])$

    for $outputMaps = 1$ to $numOutputMaps$ do
        $output \Leftarrow convn(input, kernels)$        /* The outputs are the feature maps */
    end for

    $outputWidth, outputHeight, numOutputs \Rightarrow writeSocket(server)$
    $output \Rightarrow writeSocket(server)$

    /*After every batch, the server sends a bit that acknowledges that it received the
    feature maps*/
    $allOK \Leftarrow readSocket(server)$
end while
```

to be distributed equally, Device 1 would complete the task in 5 seconds while Device 2 would take 10 seconds. If Device 1 were to be used as the comparison basis, the speed up would be below 1x, as computation time would remain the same, but communication times would be introduced.

In order to mitigate this problem, it is necessary to find beforehand the suitable workload for each device, which in this case is the number of kernels, so that each device can finish all it's convolutions at approximately the same time.

To do so, every device runs a N-dimensional convolution with both the size of the images as well as the size of the kernels provided by the master device, that tries to simulate part of the convolutional layer. The convolution is run using random values, because only the time spent on it is relevant. After the respective simulations, the computation time is sent to the master node in order to find the performance ratio between devices, either CPUs or GPUs. The slave nodes only need to know the *Internet Protocol* (IP) address of the master node, while the master node only needs to know the number of slave nodes, making it 100% usable across devices that have MATLAB.

Considering the same example as before, the performance values would be [2, 1], for Device 1 and Device 2, respectively. Device 1 would then have twice as much kernels as Device 2, for the same arbitrary workload, with Device 1 in charge of two thirds of it, while Device 2 convolving using one third of the kernels. This means that both devices would finish their convolutions in about 6.67 seconds, which taking into consideration the previous computing time of 10 seconds represents a speed up of 1.5x. This difference in performance between devices comes from the differences regarding data transfers and computing capabilities.

However, considering that during the experiment 3 to 4 computers will be used, it is necessary to further clarify how the attribution of performance values and subsequent distribution of work is done, in cases with more than 2 devices. For example, considering an arbitrary workload, the 4 devices complete the task in 10, 15, 20 and 30 seconds respectively. To calculate how the workload should be distributed to each device, one must divide all the processing times by the slowest computer's time and invert that result. This way, the performance values of the devices for this particular example are [3, 2, 1.5, 1], where the first device is in charge of $\frac{3}{3+2+1.5+1}$, or $\frac{3}{7.5}$ of the workload, while the remaining devices handle $\frac{2}{7.5}$, $\frac{1.5}{7.5}$ and $\frac{1}{7.5}$ respectively.

4.2 Hardware platforms setup

This section discloses the computer platforms, and their specifications, used to execute the experiment.

	CPU		GPU		OS
	Name	RAM	Name	RAM	
PC1	Intel Core i5-3210M CPU @ 2.50 GHz	6GB	Radeon HD 7500M	N/A	Arch Linux
PC2	Intel Core i7-4700HQ CPU @ 2.40 GHz	8GB	NVIDIA GeForce 840M	2GB	Windows 8.1
PC3	Intel Core i7-5500U CPU @ 2.40 GHz	8GB	NVIDIA GeForce 940M	2GB	Ubuntu 14.04
PC4	Intel Core i7-6700HQ CPU @ 2.60 GHz	16GB	NVIDIA GeForce GTX 950M	4GB	Ubuntu 14.04

Table 4.1: Platforms used during the experiment.

As it can be noticed from Table 4.1, the computers are all composed by different devices, so both hybrid CPU-CPU and GPU-GPU computing was necessary. As the code from this experiment was all written in Matlab, the only framework for parallel computing compatible is *Compute Unified Device Architecture* (CUDA), meaning that the GPU cluster has only 3 computers (PC2, PC3 and PC4), since only NVIDIA GPUs are supported, while the CPU cluster can run with all computers available.

4.3 Network architecture and Dataset

For this experiment, the dataset used was the CIFAR10[24]. It is a labeled subset of the 80 million tiny images dataset[31] and was collected by Alex Krizhevsky, Vinod Nair and Geoffrey Hinton.

The dataset consists of 60000 32×32 colour images separated into 10 classes, with each class having 6000 images. Of the 60000 images, 50000 are intended to be used for training and the remaining 10000 for testing. The classes present in this dataset are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck.

This dataset was chosen particularly for consisting of colour images, which is the norm for most recent image datasets, but also for having a considerably small dataset with small images, which allows to test several CNNs architectures in a shorter period of time than other datasets like Imagenet[13] and is able to serve as a proof of concept.

The chosen architecture for the network is as follows:

- Convolutional layer (henceforth known as C_1), with kernels with 5×5 pixels size;
- Normalization layer;
- Pooling layer, with stride 2;
- Convolutional layer (henceforth known as C_2), with kernels with 5×5 pixels size;
- Normalization layer;
- Pooling layer, with stride 2;
- Fully connected layer;
- Loss layer, with softmax loss;

The goal of the experiment is threefold: 1) analyze the speedup achieved using a varying number of devices; 2) study the influence that the number of kernels in each convolutional layer have on the speedups; 3) evaluate how the batch size impacts the speedups.

To achieve that, the number of kernels on each convolutional layer was varied, testing 4 different network architectures. The smallest tested CNN has 50 kernels in the first convolutional layer and 500 on the second one. The remaining layers use 150 and 300 kernels for the first layer and 800 and 1000 kernels for the second one, while the largest tested network has 500 and 1500 kernels on each layer.

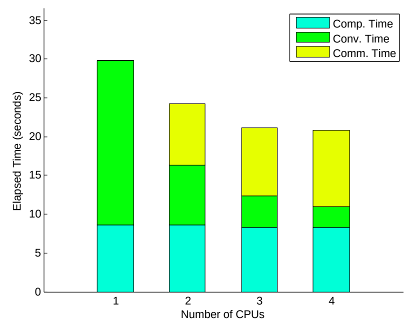
5 Experimental Results

Contents

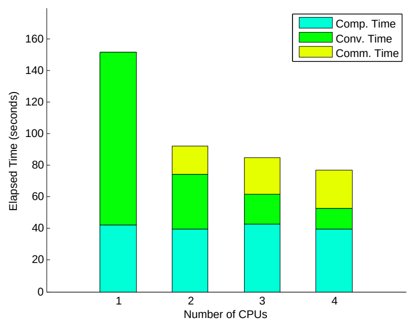
This chapter presents the results obtained by applying the distribution devised in the previous chapter and assess its performance for the two case studies considered: *Central Processing Unit* (CPU) cluster and *Graphics Processing Unit* (GPU) cluster. This section begins with an analysis of the speedup achieved using a variation in number of devices. At a second stage, the effects of the batch size and number of kernels per convolutional layer are also considered for this type of distribution method. Finally a comparison of the overall performance of the CPU and GPU (and combinations of both) considering the same experimental parameters is performed.

The elapsed time is relative to only one batch of data, since the time for the training of an entire epoch is linear. The full training time is divided into 3 parts: *Comm. time* refers to the communication time between the master node and the slaves. *Conv. time* is the time spent in convolutions by each node, or by the slowest node, as opposed to being the cumulative time spent in convolutions by all the nodes. Finally, *Comp. time* is the time spent on computation of layers other than the convolutional ones.

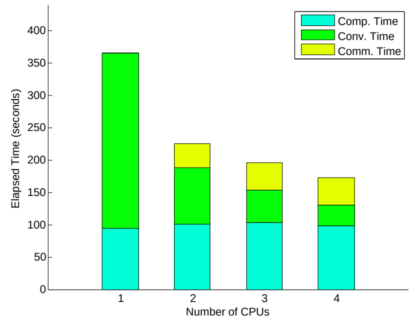
5.1 Results using CPU-cluster



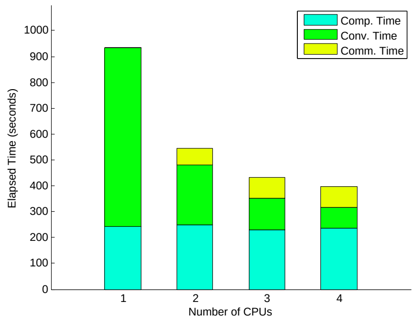
(a) $C_1 = 50$ and $C_2 = 500$ kernels.



(b) $C_1 = 150$ and $C_2 = 800$ kernels.

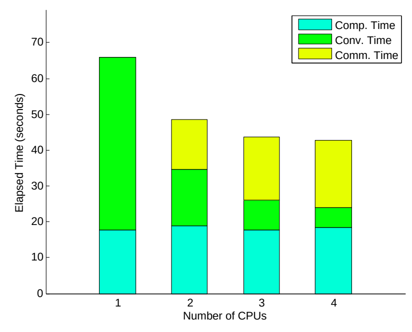


(c) $C_1 = 300$ and $C_2 = 1000$ kernels.

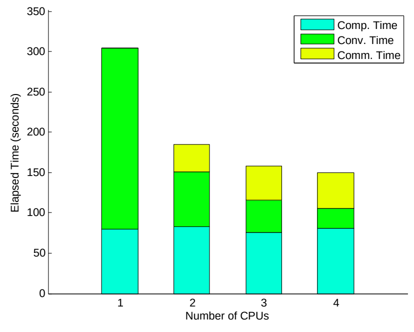


(d) $C_1 = 500$ and $C_2 = 1500$ kernels.

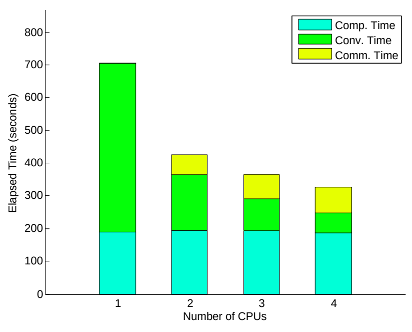
Figure 5.1: Elapsed time for a batch size with 64 images, with different network architectures, using a CPU cluster raging from 1 to 4 PCs.



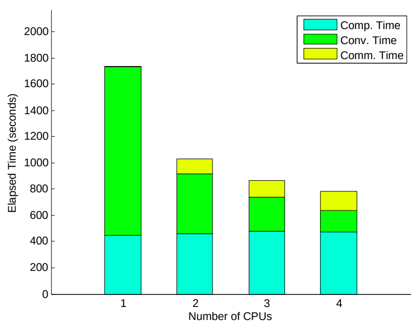
(a) $C_1 = 50$ and $C_2 = 500$ kernels.



(b) $C_1 = 150$ and $C_2 = 800$ kernels.

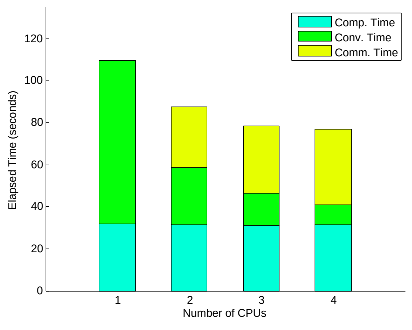


(c) $C_1 = 300$ and $C_2 = 1000$ kernels.

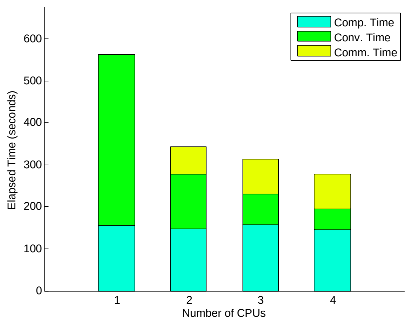


(d) $C_1 = 500$ and $C_2 = 1500$ kernels.

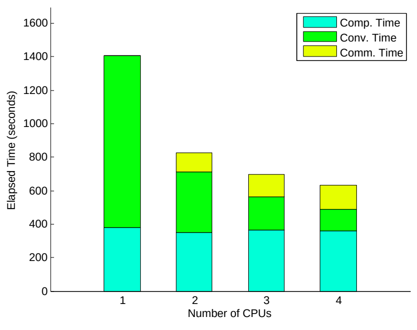
Figure 5.2: Elapsed time for a batch size with 128 images, with different network architectures, using a CPU cluster raging from 1 to 4 PCs.



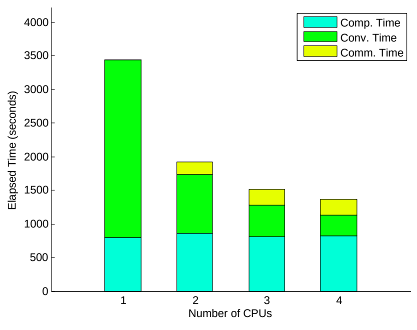
(a) $C_1 = 50$ kernels and $C_2 = 500$ kernels.



(b) $C_1 = 150$ kernels and $C_2 = 800$ kernels.

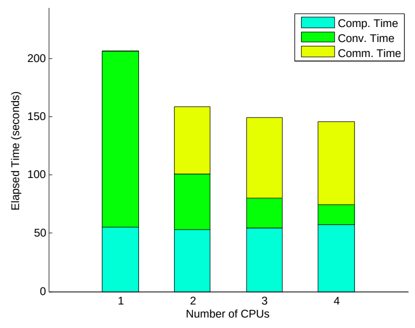


(c) $C_1 = 300$ kernels and $C_2 = 1000$ kernels.

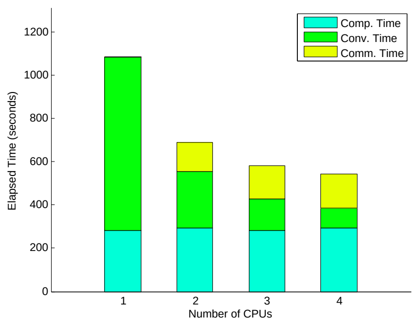


(d) $C_1 = 500$ kernels and $C_2 = 1500$ kernels.

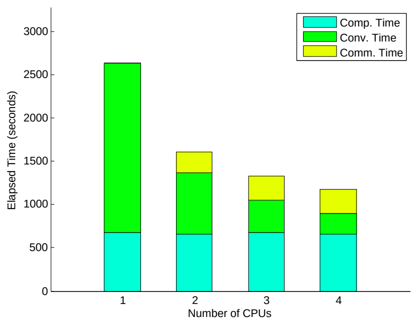
Figure 5.3: Elapsed time for a batch size with 256 images, with different architectures, using a CPU cluster raging from 1 to 4 PCs.



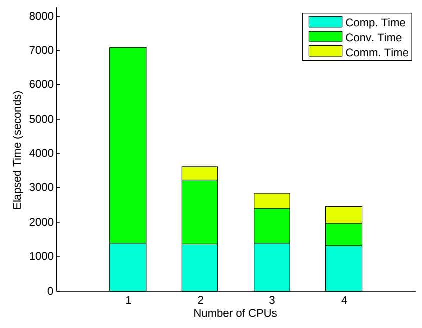
(a) $C_1 = 50$ kernels and $C_2 = 500$ kernels.



(b) $C_1 = 150$ kernels and $C_2 = 800$ kernels.

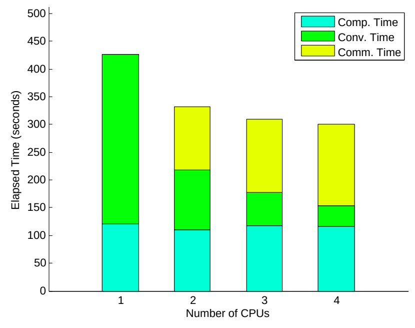


(c) $C_1 = 300$ kernels and $C_2 = 1000$ kernels.

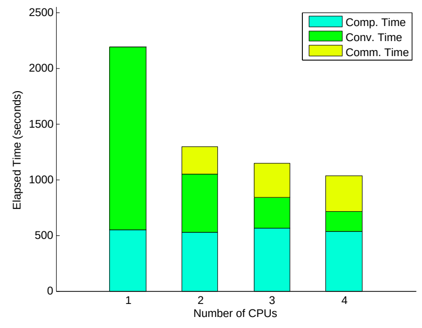


(d) $C_1 = 500$ kernels and $C_2 = 1500$ kernels.

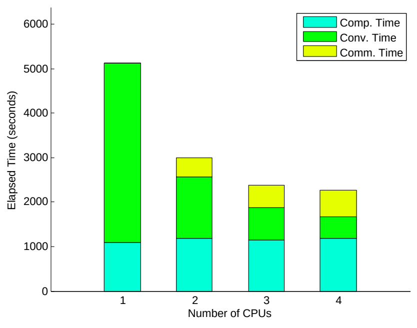
Figure 5.4: Elapsed time for a batch size with 512 images, with different architectures, using a CPU cluster raging from 1 to 4 PCs.



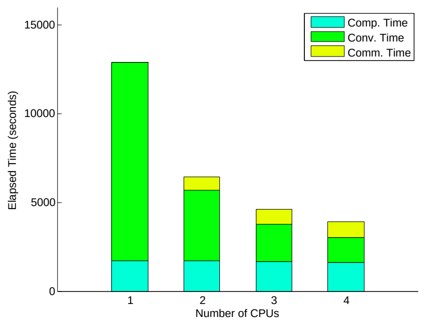
(a) $C_1 = 50$ kernels and $C_2 = 500$ kernels.



(b) $C_1 = 150$ kernels and $C_2 = 800$ kernels.



(c) $C_1 = 300$ kernels and $C_2 = 1000$ kernels.



(d) $C_1 = 500$ kernels and $C_2 = 1500$ kernels.

Figure 5.5: Elapsed time for a batch size with 1024 images, with different architectures, using a CPU cluster raging from 1 to 4 PCs.

5.1.1 Introduction

In order to better understand the analysis of the results, there is a key aspect in the methodology followed that should be taken into consideration. Since only the convolutional phase was parallelized, the computation of the remaining layers is always made by the master node, this explains why the *comp. time* always remains the same when considering the same architecture trained with the same batch size, apart from minor fluctuations that are bound to happen in every practical experiment.

Following the notation used in Chapter 4.2, PC1 serves as the master node for the CPU implementation, being the reference of comparison when using a single CPU. The rest of the devices considered, PC2, PC3, and PC4 are introduced in this order to test the introduction of more nodes for the cases with 2, 3, and 4 devices, respectively.

All subfigures present in Figures 5.1 to 5.5 document results by keeping the same batch size and varying the network architecture, more specifically altering the number of kernels in both convolutional layers. Thus it is possible to make a detailed analysis of the influence of the convolutional layer on the performance of this distribution technique.

By analyzing Figure 5.1, it is visible that a speedup always exists, even when considering the smallest network, which contain 50 kernels on the first convolutional layer and 500 kernels on the second one. Looking at the results of that network, visible in subfigure 5.1a), it is noticeable that the introduction of more CPUs contributes to an improvement on processing time, achieving speedups of 1.20x with 2 CPUs and nearing 1.35 on both 3 and 4 CPUs. However, the communication time clearly overshadows the rest of the processing time, representing almost 4.51x the convolution time in the case of 4 CPUs, which results in much less significant speedups. The time relative to the computation of other layers always stays constant with any number of devices used, since that was not a target of any kind of parallelization.

5.1.2 Number of Kernels

To understand the effects of the number of kernels, it is necessary to analyze the remaining subfigures. As subfigure 5.1b) shows, an increase of kernels on both convolutional layers leads to a significant decrease in the proportion of time spent in communication. Considering the specific case of 4 CPUs, where in the architecture with 50 kernels on the first layer and 500 on the second the communication time corresponded to 47% of total training time, as presented in subfigure 5.1a), in this case the communication time totals 30% of global training time. The obtained speedup is 1.67x, 1.80x and 1.95x to 2, 3 and 4 CPUs respectively, showing an improvement over the last architecture.

By further analyzing the effect of the network architecture, a quick study of subfigure 5.1c) shows that a new increase in number of kernels on both layers leads to enhancing performances, with speedups between 1.61x and 2.18x, according to the number of devices used. For the largest network tested, with 500 kernels on the first convolutional layer and 1500 on the second one, the speedup reaches 2.27x for 4 CPUs and communication time dwindles to 19%, showing that this type of distribution is not communication bounded. As communication time depends solely on the amount of data sent and bandwidth, even considering a data transmission 5 times slower, the attained speedup would still be 1.27x. However, it should be noted that the computation time of the remaining layers has become the bottleneck, occupying from 25% to 56% of total training time. Without any kind of parallelization of the remaining layers, the increase in number of CPUs used has a theoretical maximum speedup of 4 times, for this particular architecture, using CPUs.

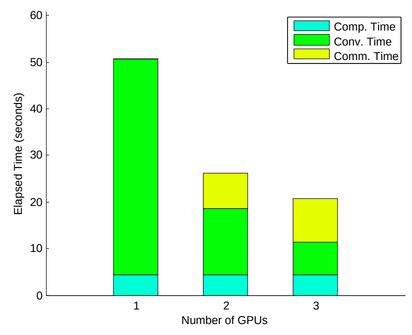
5.1.3 Batch Size

A batch size influence analysis on the distribution technique performance starts by comparing subfigures that represent the same network architecture but use different batch sizes by comparing subfigures 5.1a) to 5.5a), subfigures 5.1b) to 5.5b), all the way to the largest networks tested. For the smallest network considered, the difference in batch sizes does not introduce significant changes. The speedup for 2 CPUs is between 1.27x and 1.33x, for 3 CPUs is between 1.33x and 1.47x and it reaches a 1.50x speedup with 4 CPUs. Using any batch size, the communication time continues to overshadow the remaining training time, showing that convolution distribution for smaller networks is not really worth it.

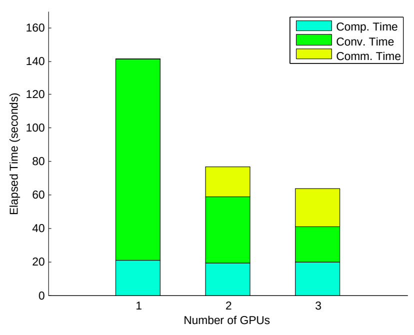
For the two next architectures, with 150 kernels on the first convolutional layer and 800 kernels on the second one, and 300 kernels on the first layer and 1000 on the second, the differences are almost non existent: there is a performance gain accompanying the increase in convolutional layers sizes that is considerably constant with the increment of batch size. Furthermore, an analysis of equal sized networks trained with different batch sizes shows that the ratio between communication, convolution and computation time also stays constant.

However, for the largest network tested, there is a more prominent difference when training it with different batch sizes. By analyzing the training time using 1 CPU on the different subfigures, it is visible a decrease in percentage of time dedicated to the computation of different layers, going from 25% with a batch size of 64 images to 13% when using 1024 images to train. Taking it into consideration, a more thorough analysis of the largest network using a batch with 1024 images shows that, as it can be seen in subfigure 5.5d), the use of 2 CPUs achieves a speedup of 1.98x, while for 3 and 4 CPUs the attained speedup is 2.73x and 3.28x, respectively. Considering that the computation of the remaining layers only occupies 13% of the total training time using one CPU, the theoretical maximum speedup achievable for this particular case would be about 7.76x.

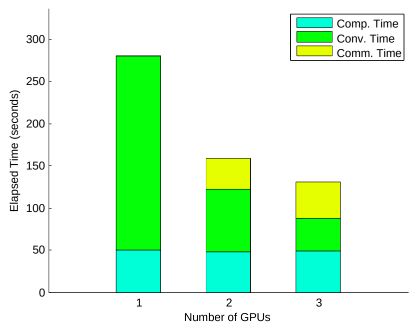
5.2 Results using GPU-cluster



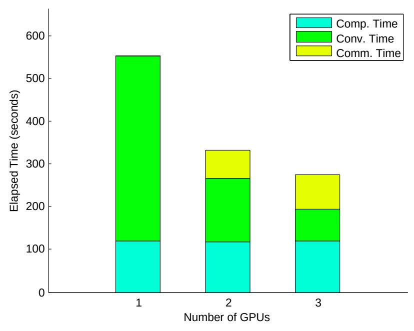
(a) $C_1 = 50$ and $C_2 = 500$ kernels.



(b) $C_1 = 150$ and $C_2 = 800$ kernels.



(c) $C_1 = 300$ and $C_2 = 1000$ kernels.



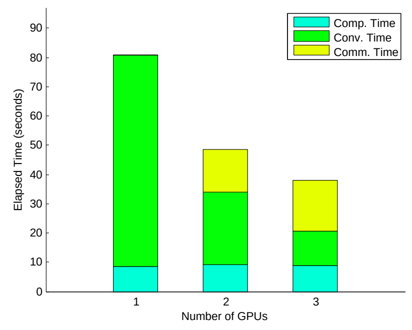
(d) $C_1 = 500$ and $C_2 = 1500$ kernels.

Figure 5.6: Elapsed time for a batch size with 64 images, with different network architectures, using a GPU cluster raging from 1 to 3 PCs.

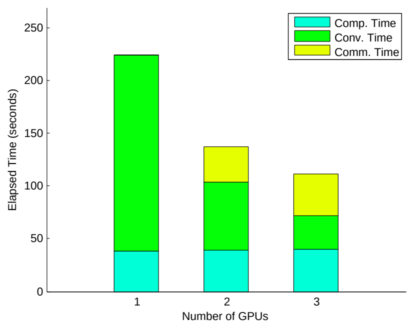
5.2.1 Introduction

In order to analyze the results obtained with the GPU clusters, it is necessary to clarify some constraints, namely regarding the number of GPUs used in the experiments. Considering that the code was developed using MATLAB, GPU integration is only possible using CUDA, that only allows its execution on NVIDIA GPUs. Taking into account that only 3 of the 4 computers complete that requirement, the maximum size of the GPU cluster is 3 machines, which only allows the comparison between CPU and GPU up to a certain point.

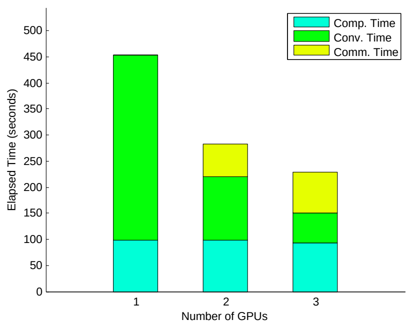
Another aspect to consider is the computational capabilities of the GPUs. As stated previously in Section 3.2, the GPU is much more effective than the CPU when it comes to receiving large quantities of data and repeat the same operation, mostly sum and mul-



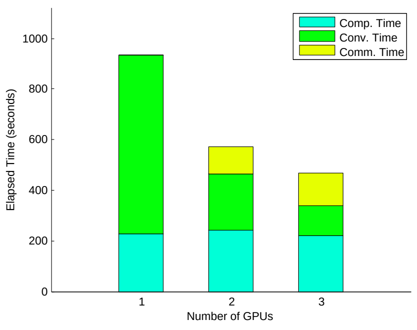
(a) $C_1 = 50$ and $C_2 = 500$ kernels.



(b) $C_1 = 150$ and $C_2 = 800$ kernels.

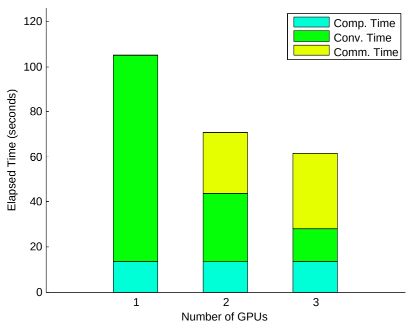


(c) $C_1 = 300$ and $C_2 = 1000$ kernels.

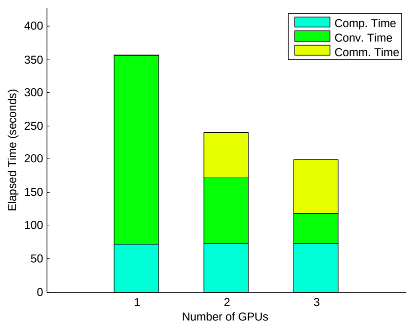


(d) $C_1 = 500$ and $C_2 = 1500$ kernels.

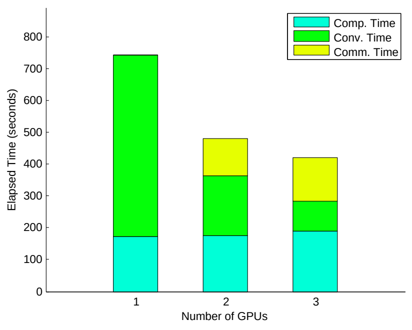
Figure 5.7: Elapsed time for a batch size with 128 images, with different network architectures, using a GPU cluster raging from 1 to 3 PCs.



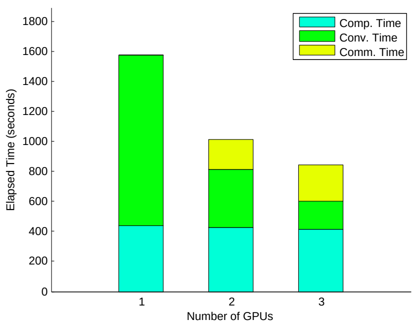
(a) $C_1 = 50$ kernels and $C_2 = 500$ kernels.



(b) $C_1 = 150$ kernels and $C_2 = 800$ kernels.

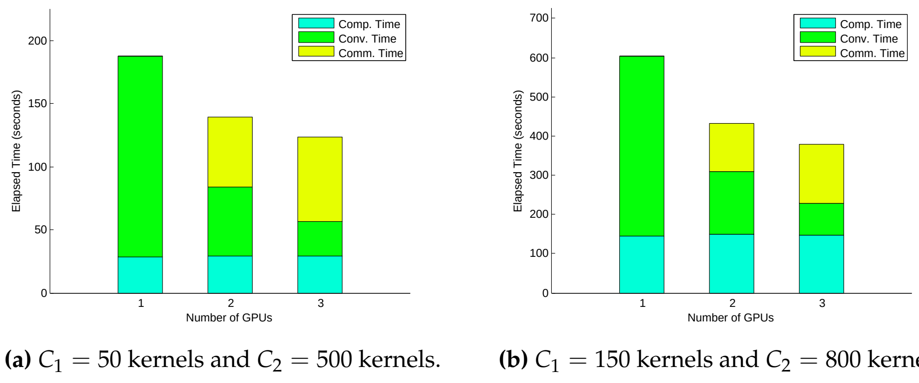


(c) $C_1 = 300$ kernels and $C_2 = 1000$ kernels.

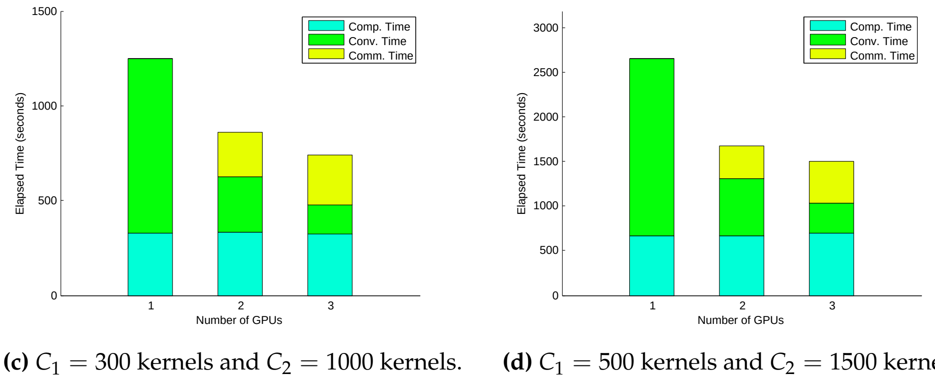


(d) $C_1 = 500$ kernels and $C_2 = 1500$ kernels.

Figure 5.8: Elapsed time for a batch size with 256 images, with different architectures, using a GPU cluster raging from 1 to 3 PCs.

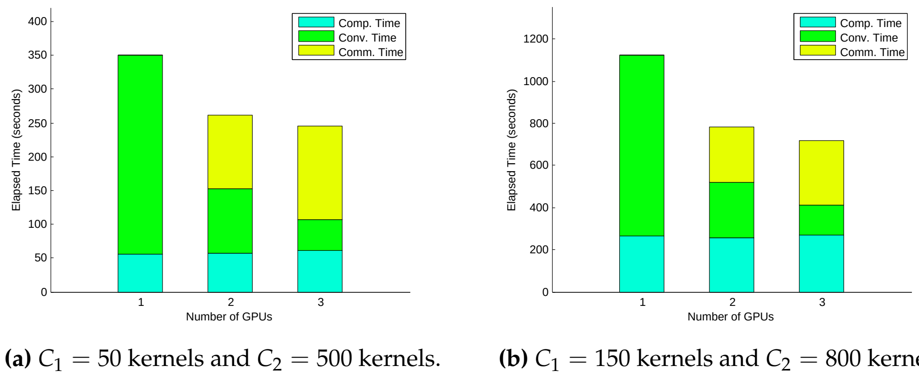


(a) $C_1 = 50$ kernels and $C_2 = 500$ kernels. **(b)** $C_1 = 150$ kernels and $C_2 = 800$ kernels.

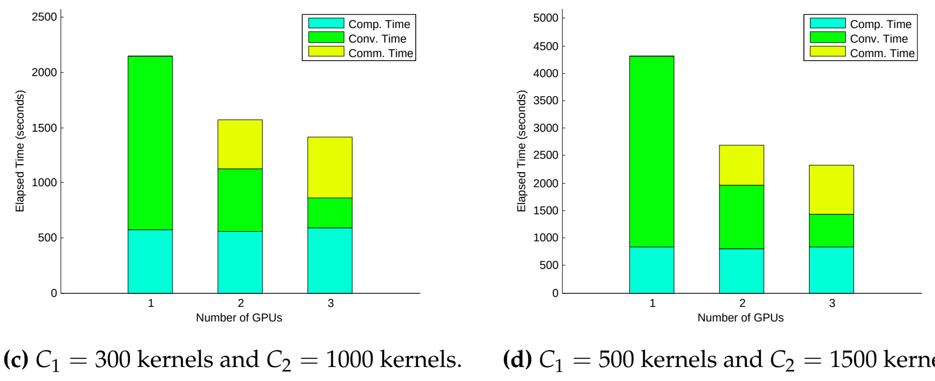


(c) $C_1 = 300$ kernels and $C_2 = 1000$ kernels. **(d)** $C_1 = 500$ kernels and $C_2 = 1500$ kernels.

Figure 5.9: Elapsed time for a batch size with 512 images, with different architectures, using a GPU cluster raging from 1 to 3 PCs.



(a) $C_1 = 50$ kernels and $C_2 = 500$ kernels. **(b)** $C_1 = 150$ kernels and $C_2 = 800$ kernels.



(c) $C_1 = 300$ kernels and $C_2 = 1000$ kernels. **(d)** $C_1 = 500$ kernels and $C_2 = 1500$ kernels.

Figure 5.10: Elapsed time for a batch size with 1024 images, with different architectures, using a GPU cluster raging from 1 to 3 PCs.

tiplication, very quickly, due to its large number of parallel cores. However, for smaller amounts of data globally the GPU handles these tasks more slowly than the CPU would.

Finally, as in the CPU case, only the convolutional layers were parallelized using GPU computing, which implies that the computation of the remaining layers is performed on the CPU. Since the node used as master for the CPU cluster is not available for GPU parallelization, the values of the computation time will be different between CPU and GPU implementations.

For this particular case, since PC1 is not a NVIDIA GPU, the PC2 serves as the master node, also being the reference for the case of a single GPU. PC3 and PC4 are introduced in this order to test the introduction of more nodes for the cases with 2 and 3 devices, respectively. This notation respects the one used in Chapter 4.2.

As before, all subfigures present in Figures 5.6 to 5.10 document results by keeping the same batch size and varying the network architecture, thus allowing a detailed analysis of the influence of the convolutional layer on the performance of this distribution technique.

By analyzing Figure 5.6, it is possible to observe that speedup is always achievable for any type of network architecture considered. However, the speedup presented in some specific cases is not completely linear. By observing the results from the smallest network, with 50 kernels on the first convolutional layer and 500 and the second one, visible in subfigure 5.6a), the introduction of GPUs brings considerable improvements on processing time, reaching speedups of 1.92x with 2 GPUs and nearing 2.50x with 3 GPUs. However, it is necessary to perform a deeper analysis, comparing this result with the one obtained using the CPU cluster because, by comparing these results with the ones in subfigure 5.1a), it is possible to see that in the case of only 1 device, the CPU runs faster than using the GPU. This happens because the amount of data to parallelize is too small to make use of the vast GPU parallel processing resources. Taking into consideration the result obtained using only 1 CPU, the speedups achieved would only be 1.15x and 1.50x for 2 and 3 GPUs respectively, which is close to the ones obtained using CPU. It is also possible to notice that the communication time rises considerably with the increase in number of GPUs, but that effect is minimized with the ability to do faster convolutions using the different machines available.

5.2.2 Number of kernels

To understand the effects that the number of kernels have on the speedups, it is necessary to analyze the remaining subfigures. As subfigure 5.6b) shows, an increase of kernels in the GPU case makes almost no difference concerning communication time and speedup, and that is also visible in the rest of the tested architectures, trained using batches of 64 images. All the architectures tested with this batch size show an attained speedup between 1.63x and 1.80x for 2 GPUs and between 2.02x and 2.20x using 3 GPUs, with the ratio between communication, convolution and computation time being virtually the same on the 3 considered experiments, with the communication time rising from 19% with 2 GPUs to 30% when using all 3 GPUs.

The major difference between the CPU and GPU results is that while using the CPUs, the computation time was the major bottleneck on that experiment. However, in the GPU, the communication and computation time share about the same percentage of full training time, when using 3 GPUs, which is explained by the fact that the GPU is able to accelerate the convolutional phase.

Another comparison between the different devices can be made using subfigures 5.1b) and 5.6b), because using only 1 of the devices gives about the same processing time. Analyzing both cases, it is possible to see that where the CPUs only achieve speedups of 1.62x and 1.75x with 2 and 3 devices, the GPUs are capable of achieving 1.85x and 2.22x on those same conditions.

5.2.3 Batch Size

The analysis of the influence of batch sizes on the distribution technique performance starts by comparing subfigures that represent the same network architecture but use different batch sizes, like in the CPU case. Considering the smallest network tested, the difference in batch size brings a significant change: the increase, in ratio, of the communication case, with the percentage going from 25% using 2 GPUs and 40% for 3 GPUs, with a batch of 64 images, to about 33% and 52% with 2 and 3 GPUs, when training the network with 1024 images.

However, the remaining network architectures show little to no difference with the increase in batch size, with the obtained speedups fluctuating between 1.55x for 2 GPUs and 1.90x when using 3 GPUs, with the communication, convolution and computation time percentage staying approximately the same.

5.3 Comparison between CPU and GPU

The following two tables show the best attained speedups between each network architecture for a given number of devices, for both CPU and GPU:

As table 5.1 shows, the difference between speedups using multiple CPUs, for a given architecture, increases with the growing convolutional layers. The speedup improvement using 2 CPUs is particularly small, although it reaches almost 2.00x on the largest tested network. However, this tendency fades with the increase in CPUs. By training the network with 3 CPUs, the speedup is almost 2.00x for the second smallest network, reaching

Network \ Num. of CPU	2	3	4
50:500	1.40x	1.51x	1.56x
150:800	1.68x	1.93x	2.10x
300:1000	1.69x	2.15x	2.33x
500:1500	1.98x	2.74x	3.28x

Table 5.1: Best speedups achieved by network architecture and number of CPUs used.

Network \ Num. of GPU	2	3
50:500	1.96x	2.45x
150:800	1.89x	2.23x
300:1000	1.78x	2.09x
500:1500	1.66x	2.00x

Table 5.2: Best speedups achieved by network architecture and number of GPUs used.

2.75x for the largest architecture. Using 4 CPUs gives a considerable gain in speedup, particularly between the network with 300 kernels on the first convolutional layer and 1000 kernels on the second one, and the largest trained network. This is explained because the increase in communication time due to sending dozens more kernels to other nodes, that are only a couple of kBs, is counterbalanced by the convolutions parallelization.

However, for the GPU implementation case, the values of the speedups diminish with the enlargement of the convolutional layers. This happens because for smaller layers the computation complexity is low and the GPU is not using its computational abilities to their fullest, even being slower than the CPU in some cases. Therefore, the convolution time is going to be higher than it was supposed to and any parallelization will benefit from a high speedup. In cases where the GPU is being used more efficiently, the speedup is smaller, since the convolutions are made very quickly and the communication time between nodes ends up being a larger portion of the total processing time, becoming a bottleneck. Even the transmission of a couple of kernels ends up not compensating since the convolutions are made much faster than the increase in communication time.

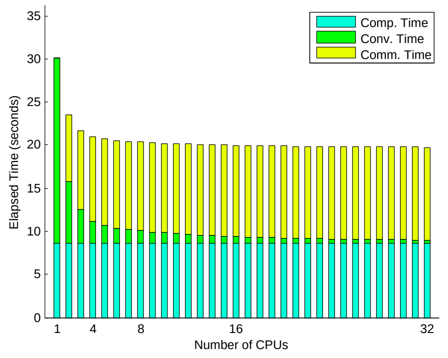
5.4 Scalability

One of the most important details when developing a distribution technique is scalability. As with other methods for distributed learning, speedups may only exist when using up to a certain number of nodes. For a more detailed study on scalability, it is necessary to know some details regarding the experiments conducted. First, the amount of data transmitted between master and slave nodes on the convolutional layers. This depends only on the number of convolutional layers and the size of their inputs, including width, height and number of input channels, size and number of kernels and the batch size. Taking this information into consideration, the number of elements that are necessary to exchange between master and slave nodes can be described as:

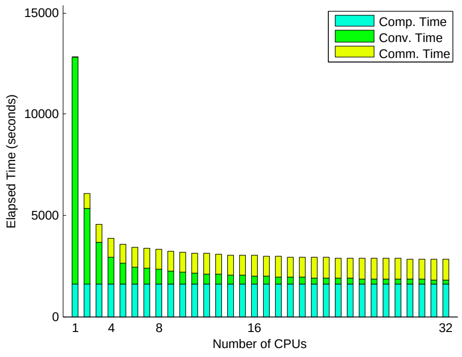
$$upload = \sum_{i=1}^{layers} in_i^2 \times inCh_i \times batch + k_i^2 \times numK_i \times inCh_i + out_i^2 \times numK_i \times batch \quad (5.1)$$

where *layers* refers to the number of convolutional layers that need to be distributed, *in* is the convolutional layer's input width or height, considering a square image, like this particular case, *inCh* represents the input channels, *k* is the kernel size, *numK* represents the number of kernels for each convolutional layer, *out* refers to the output's size and *batch* is the batch size. The size in MB of the values passed between nodes is given by $uploadMB = upload \times 8/1024/1024$, since all values transmitted are of the type *double*. The next detail to consider is the velocity at which the data is transmitted across nodes. A quick study of the several results show that the bandwidth is constant, averaging at 5 MBps. Another thing to take into account is the number of kernels that should be passed to each worker, which is explored more in detail in Section 4.1.2.

Knowing these details, it is possible to accurately predict the communication time when more nodes are added, the convolution time and therefore the total processing time. Three different cases where considered. The first two pertain to the CPU case, where the processing time for the smallest and largest networks were simulated for the addition of more nodes, up to 32 CPUs. For the simulations, the remaining CPUs where considered to have a processing power between the worst and best CPU. These results are shown in Figure 5.11.



(a) $C_1 = 50$ kernels and $C_2 = 500$ kernels.



(b) $C_1 = 500$ kernels and $C_2 = 1500$ kernels.

Figure 5.11: Elapsed time for the smallest network, using a batch with 64 images, and the largest network, with a batch size of 1024 images, using a CPU cluster raging from 1 to 32 PCs.

As results show, the method is scalable without incurring in performance losses, despite becoming irrelevant the introduction of more nodes up to a certain value. Both the case of the smallest network as the largest one benefit little from the addition of more nodes from 4 CPUs, and there is a stabilization in speedup from 8 nodes. This occurs because the inclusion of more CPUs leads to a slight increase in information to be sent by the master node, that is counterbalanced by the decrease in time obtained by the parallelization. It is also possible to notice that while using 1 CPU, the convolution time is the bottleneck using several CPUs, this situation is reversed and the communication and computation times become the bottlenecks. The former can be solved with faster data transmission, but the latter can only be fixed by parallelizing.

The final case refers to the GPU case, where only the largest network was simulated up to 32 nodes. This has to do with the fact that the GPU needs to be used efficiently, and that happens more in the largest network, trained with the largest batch of 1024 images. As in the CPU case, the added nodes where considered to have processing power between the worst and best GPU. The results are detailed in Figure 5.12.

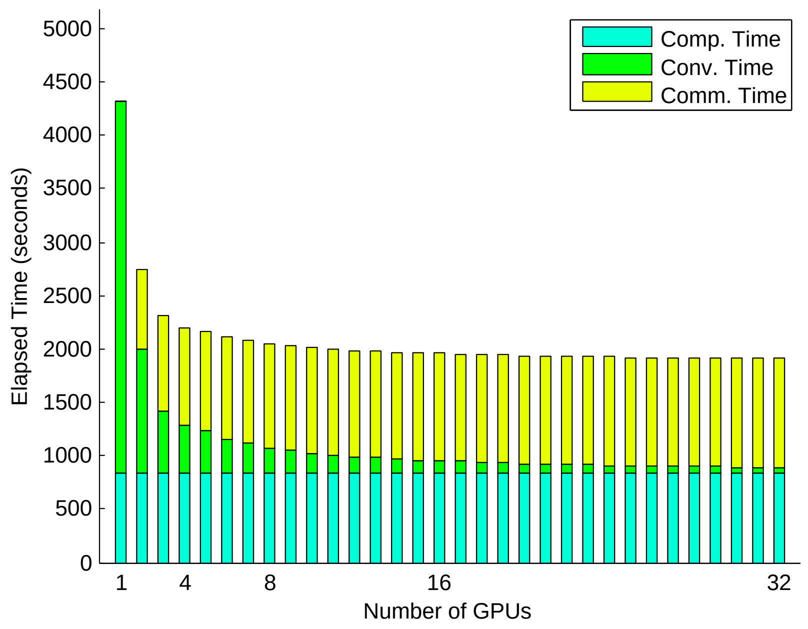


Figure 5.12: Elapsed time for the largest network, with a batch size of 1024 images, using a GPU cluster raging from 1 to 32 PCs.

As in the cases considered for simulations using CPUs, the solution using GPUs is also scalable, with the speedup virtually stagnating for 8 or more nodes. Since the convolution is done more quickly on a GPU then on a CPU, the maximum theoretical speedup in this case is much smaller (6.70x for the CPU vs. 2.15x for the GPU) and the communication and computation times assume greater impact as bottlenecks. As stated previously, the communication time can be diminished with a faster data transmission, while the computation time can be dealt with parallelization.

5.5 Discussion

Despite achieving considerable speedups for the tested networks, some essential aspects of the experiment are to be considered.

The first aspect to highlight is the difference between CPU and GPU speedups. This is explained by the fact that speedups are calculated with respect to the use of a single device. Since the convolutional layer is computed significantly faster with a GPU, the computation phase will occupy a larger percentage in total processing time, thus decreasing the maximum speedup achievable. So, to better analyze the real speedup of the GPU, it is necessary to directly compare it with the results using the same number of CPUs. The speedups of the GPU over the CPU were only calculated for the largest and computationally more intensive tested network, using a batch of 1024 images, because it represents the case where the GPU is more efficiently used, and the results are presented in Table 5.3.

Network \ Num. of Devices	1	2	3
50:500	1.22x	1.28x	1.28x
150:800	1.94x	1.67x	1.62x
300:1000	2.35x	1.92x	1.67x
500:1500	2.96x	2.41x	2.01x

Table 5.3: Speedups achieved by the GPU over CPU, using the same number of devices, by network architecture and number of devices used.

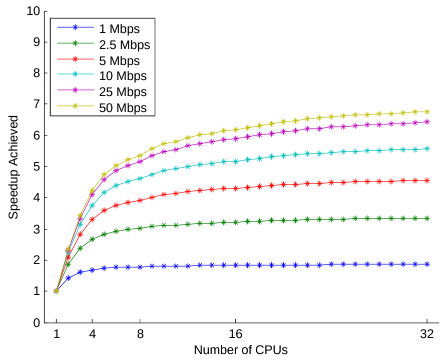
The most visible effect is the gain in speedup that accompanies the expansion of the convolutional layers. This happens because with the increase in kernel numbers, the GPU is able to exploit more efficiently its computational resources, in comparison to a CPU. The general decrease in speedup with an increasing number of devices is also related to the GPU's computational abilities. Considering that the maximum theoretical speedup for the GPU is smaller, the decrease in processing time with each added GPU is considerably less compared to the decrease adding more CPUs.

Another aspect is internet speed, where data transmission rate is bound to vary. This impacts communication times, that influences final processing time, leading to a better or worst speedup.

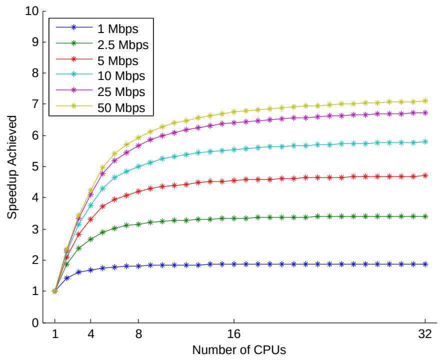
The final aspect pertains the used devices. Some of the laptops used are over 2 years old, which makes both the CPU and GPU considered low to mid range devices by today's standards.

So, as a way to generalize even more this distribution technique, the values of the data transmission speed are varied and two cases are considered. The first is to use a cluster, both CPU and GPU for low to mid range devices, while the second case if to use high end devices on the cluster.

The results for the CPU and GPU are present in Figures 5.13 and 5.14.

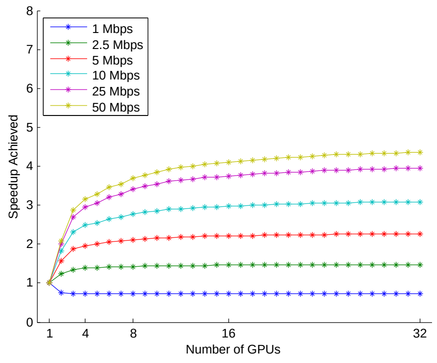


(a) Low to mid range CPU cluster

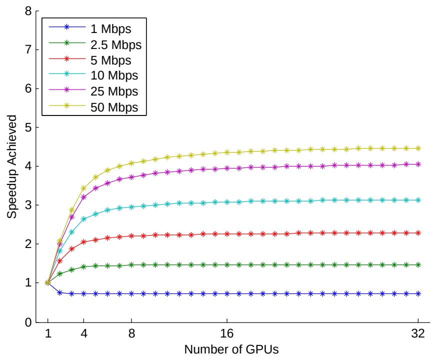


(b) High end CPU cluster

Figure 5.13: Speedup achieved on the largest network, trained with 1024 images for a cluster of 32 nodes using low mid range and high end CPUs.



(a) Low mid range GPU cluster



(b) High end GPU cluster

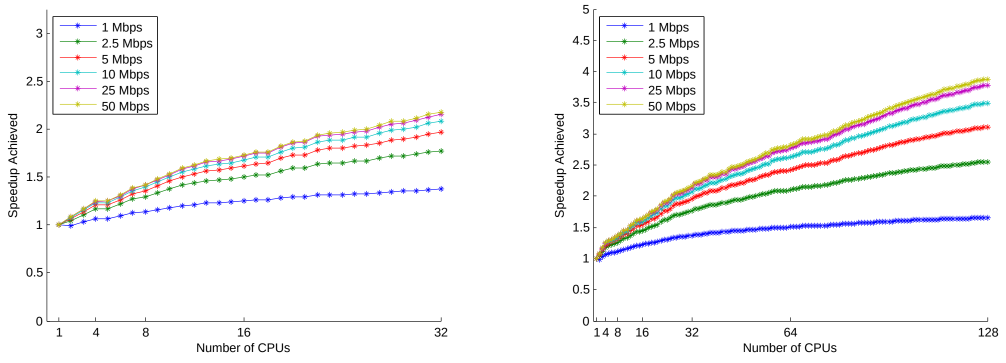
Figure 5.14: Speedup achieved on the largest network, trained with 1024 images for a cluster of 32 nodes using low mid range and high end GPUs.

Interestingly, results show that, the difference between using low end or high end devices is almost non-existent. This happens because the bottlenecks of this distribution technique end up being the communication and computation time. This means that the only difference between using the two types of devices has to do with how many nodes are needed for the speedup to start stabilizing around a maximum, with fewer nodes required for the high end devices.

However, the Internet speed is extremely important, and this has to do with the fact that with faster data transmissions, the communication time stops acting as the main cause for a bottleneck and the network has the ability to achieve higher speedups. The contrary also stands true, with a slower data transmission diminishing the speedup, with the GPU case showing that the training may even be slower than using only 1 GPU.

5.5.1 Mobile GPUs

Another aspect to consider is devices that are considerably slower than the ones used. Specifically speaking, about 10 times slower than the GPUs used, as are mobile GPUs, which is more detailed in the experimental results from Appendix A. For this simulation, the same variables as before are varied: internet speed and number of nodes. One particular difference is that the master node was still a desktop GPU. The results for this simulation are shown in Figure 5.15.



(a) Mobile GPU cluster using 32 nodes. **(b)** Mobile GPU cluster using 128 nodes.

Figure 5.15: Speedup achieved on the largest network, trained with 1024 images for a mobile GPU cluster of 32 and 128 nodes.

An initial simulation had only considered 32 nodes. As subfigure 5.15a) shows, 31 mobile GPUs are not sufficient to achieve the same values of speedups as desktop GPUs, so a new simulation had to be ran, with a maximum of 128 nodes, with the results documented in subfigure 5.15b).

Although mobile GPUs have only a tenth of the processing power as their desktop counterpart and achieve considerably worse throughput performance, they should still be considered as a viable alternative, specially because of their power consumption. As Appendix A explains in detail, various different mobile GPUs had their computational capabilities tested and compared against one of the best GPU in the market, Nvidia Titan GTX, achieving the same results but taking ten times as long. However, their average power was nearly three orders of magnitude lower than the reference GPU, which resulted in energy consumption around two orders of magnitude lower.

This means that almost 100 mobile phones could be used as a replacement for one single GPU and still be able to run a similar workload and approximately consume the same amount of energy.

5.5.2 Roofline Model

The final aspect that should be addressed regarding the GPU implementation is check the convolution performance regarding the limitations of the used hardware, so as to ponder ways to optimize the algorithm. This study can be accomplished using the Roofline model[32] which combines bandwidth, floating point performance and arithmetic intensity to evaluate the attained performance regarding the used GPUs.

The most basic Roofline model is depicted in Figure 5.16, and plots the floating point performance against the arithmetic intensity. The resulting graphic illustrates the two platform specific performance ceilings: a ceiling derived from the memory bandwidth and one derived from the processor's peak performance.

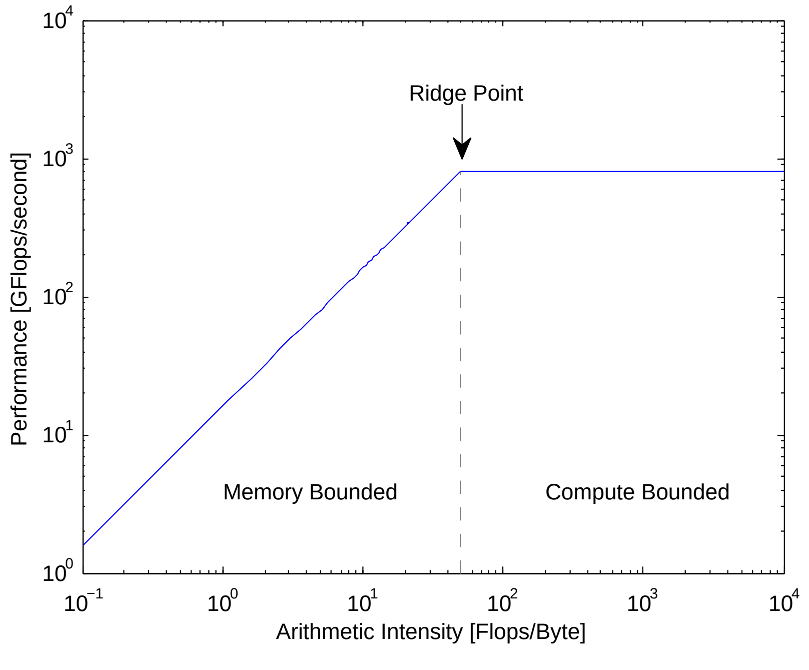


Figure 5.16: Roofline Model example.

Thus, the upper bound of a kernel's performance depends on two things. If the arithmetic intensity of the kernel reaches the flat part of the roof, then the performance is compute bounded. If, instead, it hits the diagonal part of the roof, then the performance is memory bounded. Ideally, the arithmetic intensity of a kernel would lie in the ridge point, where the horizontal and diagonal roofs meet, because the kernel would exhaust the memory bandwidth and *Floating-Point Operations per second* (FLOPs) capacity at the same time, thus requiring the minimum arithmetic intensity to achieve maximum performance.

Figure 5.17 shows the Roofline model for each covered architecture used in this experiment. The values of memory bandwidth and FLOPs were taken from the vendor's specification manual.

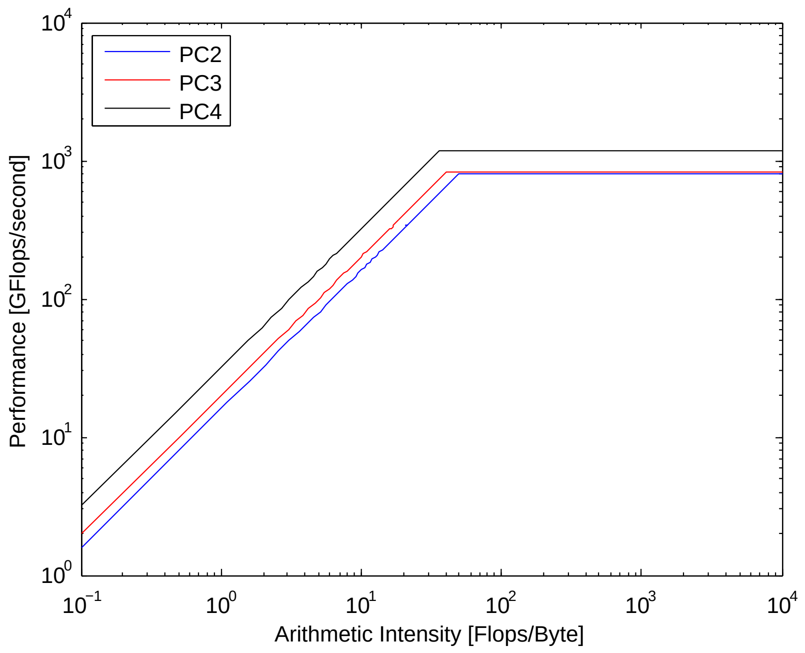


Figure 5.17: Roofline Model for the used GPUs.

The roofline model shows that both the PC2 and PC3 have similar peak performances with the PC3 having better memory bandwidth than PC2. PC4 is the best GPU from the ones used in this experiment, having a larger bandwidth and more processing power. Knowing the roofline models for the hardware used, the next step is measure the FLOPs and arithmetic intensity to know the kernel's performance upper bound. To do so, the *nvprof* profiling tool from NVIDIA was used, which retrieves information regarding the amount of transferred bytes and floating-point operations. These results are presented in Figure 5.18.

As the results show, this kernel is memory bounded, which means that the accesses to memory overshadowed the floating points operations, particularly the sums and multiplications. As such, the kernel was not able to efficiently exploit the GPUs available computational resources, which results in a speedup lower than it could potentially be for the GPU implementation.

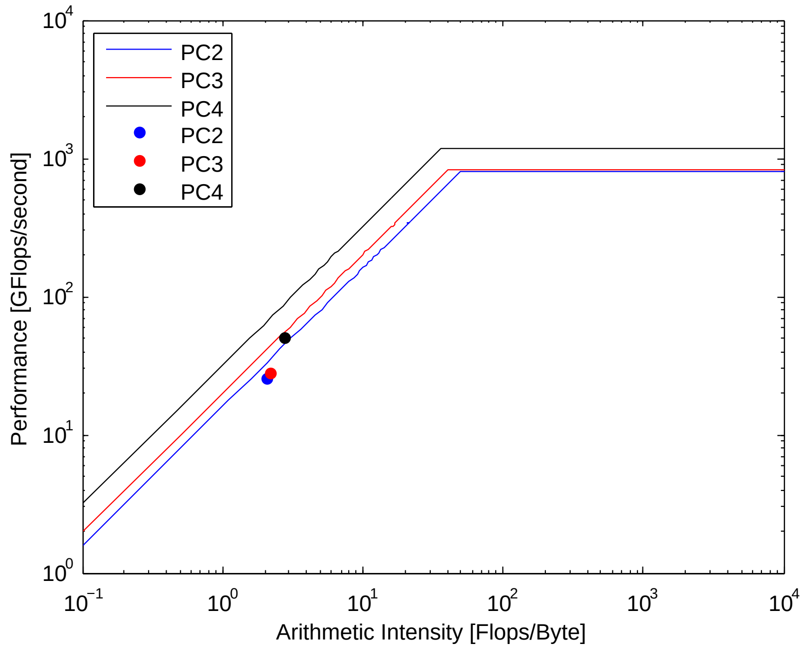


Figure 5.18: Roofline Model for the used GPUs.

To improve the algorithms performance and move the point of operation closer to the ridge point, the memory accesses should be kept to a minimum, in favor of floating point operations. To do so, the best course of action is to move a larger portion of the image to the registers[33]. Considering the kernels used are 5x5, the region of the image prefetched should be at least 6x6. Then, storing the convolution filter in the constant memory, the output pixels are computed and stored. Thus, with a 5x5 kernel and a 6x6 region, each thread produces 4 output pixels, reducing the memory accesses and maintaining the number of floating point operations.

6 Conclusions

Contents

The emergence of *Convolutional Neural Networks* (CNNs) brought an improvement in classification performance, when compared to *Deep Neural Networks* (DNNs), but it also hauled a higher computational necessity, that not even the use of *Graphics Processing Unit*s (GPUs) could fully support, due to the size of the networks involved, as well as the datasets. So it was necessary to develop distributed training techniques, that could use all processing units available in a machine, or across differently located machines. The existing techniques of distributed training are based on data parallelism and model parallelism. In the first one each device receives part of the data batch and does its own training individually, only to report the final parameters to a node that serves as master, that averages them and sends the new parameters to all the nodes used. The model parallelism allows the training of only a single replica of the network, with each node processing part of each layer, but requiring a constant communication across every layer. However, none of these methods makes use of the CNN architecture, since about 60%[16] to 90%[17] of training time is spent convolving, depending on the number and size of the kernels, as well as amount of convolutional layers. So the main goal of this thesis was the development of a method for distributed training that made use of the parallelization that the convolutional layer allows.

The proposed technique is of easy implementation, because unlike the methods of model parallelism, it only occurs in the convolutional layer and the only data that needs to be transmitted are the inputs as well as a number of kernels that is calculated during runtime, so that each node can do its part of the workload, which eliminates the convolution phase as a bottleneck, for an implementation using both *Central Processing Unit*s (CPUs) as well as GPUs.

That could be achieved with all the tested networks, with attained speedups for every architecture trained with all the considered batch sizes, when adding more devices. The best reached speedup using CPUs is 3.28x for 4 nodes, when training the largest network, with 1024 images in the batch, and is 2.45x for 3 GPUs. However, these speedups could be largely surpassed using a faster data transmission, as the results from the simulations show. Even considering the case where only 2 devices are used (both CPU and GPU), the speedup is always existent, being very close to 2x, on both tested cases.

Furthermore, this is the best technique to use with CPUs and/or GPUs with different processing resources, thanks to the hybrid CPU-CPU and GPU-GPU computing. Using data parallelism, the training time is always dependent on the slowest device, or in the case of asynchronism, the slowest device might train with old parameters. The alternative would be to split the data batch unevenly, but that would cause loss of information during the averaging of the parameters. With model parallelism, since it is necessary to define which neurons must communication between the nodes, it would be required to know all the processing information of each device *a priori*.

The simulations further show that the attained speedups depend very little of the processing capabilities of each individual device, for laptops ranging from low and mid range to high end, for a number larger than 4, since the lower computational resources are largely compensated by the parallelization.

Finally, the decision to develop this method in Matlab was made not only because of its intuitive nature regarding matrices, and operations involving them, but mainly due to the possibility of working using devices with different operating systems, without the need to develop cross-platform software, which can become a highly complex task.

6.1 Future Work

The developed solution proves to be a useful tool for the distributed training of CNNs. Although good performances were achieved, there is one other aspect that could, and should, be further explored, and that is implementation using *Open Computing Language* (OpenCL), as opposed to *Compute Unified Device Architecture* (CUDA). Not only would that mean that other GPUs could be used, like AMD's, but more importantly, it would allow for the distribution of the training to be done using mobile GPUs, as well as *Field-Programmable Gate Array*s (FPGAs) and other lower power processors. Despite not having the same computational resources as desktop CPUs and GPUs, they are far more energy efficient, and it would allow to study the impact on the energy consumption performance, trying to achieve a smaller energy consumption without compromising the throughput and classification performance.

A

Appendix A

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

G. Falcao, *Senior Member, IEEE*, L. A. Alexandre, J. Marques, X. Frazao, J. Maria

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

***Index Terms*—Embedded processors, Parallel processing, Low-power, Energy saving, Hardware Limits, Deep Learning, Stacked Autoencoders**

I. Introduction

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

While the current trend in machine learning is using convolutional neural networks (CNNs), such current state-of-the-art implementations tend to consume high levels of energy in order to produce the expected results, which directly impacts the processing costs of big data and creates constraints in their utilization in low-power-driven autonomous vehicles/robots.

In this paper we propose a scalable parallel solution for stacked autoencoder (SAE) architectures in mobile GPUs. The paper builds upon [2] as a first step towards the implementation of more complex approaches to deep learning, such as CNNs, so as to understand the possible gains in terms of energy savings that arrive from switching from desktop GPUs to low powered devices, as well as comprehend the limitations at hardware and software levels, namely the maximum allowed size of the neural network of said transition.

Mobile platforms concede having small autonomous robots/vehicles, such as drones, with deep learning capabilities. Such platforms consume at least one order of magnitude less energy while providing similar throughput and classification error, when compared to desktop GPUs or CPUs. Thus the envisioned application scenarios consist of deep learning mobile apps and autonomous robots that learn online being retrained without the need for accessing a GPU cluster, adapting on-the-fly to environmental conditions. Such an approach offers higher flexibility since there is no need of a permanent high-quality wireless connection or access to a GPU cluster.

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

Also, the hardware and code development of unique OpenCL-based parallel kernels, which address this computational challenge under a set of different low-power embedded devices, is addressed. The source code is provided to the scientific community that wishes to replicate these experiments.

Currently, several frameworks allow the training and evaluation of deep learning models, such as Theano [3] and Torch7 [4]. However, these do not allow to change all aspects of the algorithm necessary for the proposed experiment. Which required the development of code that allows higher degrees of control over all aspects of execution and model parallelization.

II. Deep Learning and Stacked Autoencoders

The use of more than two hidden layers in neural network supervised learning was seen as unnecessary until around 2006. After that, it has become a major trend in machine learning and deep learning is currently the state-of-the-art approach in multiple domains.

The potential advantages that come with using deep learning are the possibility of having increasingly more abstract levels of representation, reusing the intermediate level representations across different tasks and also obtaining a more compact and efficient representation for certain types of problems [5].

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

To obtain a deep architecture using AEs they are stacked on top of each other such that the output of an AE is the input for the next one. This stacking can produce a deep network: the SAE. The SAE is obtained as follows: first pre-train several AEs such that the first learns to approximate the inputs from the dataset, the second learns to approximate the

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

G. Falcao, J. Marques and J. Maria are with Instituto de Telecomunicações and Department of Electrical and Computer Engineering, University of Coimbra, 3030-290 Coimbra, Portugal. E-mail:{gff, jmarques, jmaria}@co.it.pt

L. A. Alexandre and X. Frazao are with Instituto de Telecomunicações and Department of Informatics, University of Beira Interior, 6201-001 Covilhã, Portugal. E-mail:{luis.alexandre, xavierfrazao}@co.it.pt

Algorithm 1: Training Phase

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

hidden representations of the first and so on. A final layer of neurons is placed on top of the AE that is the output layer and will have as many neurons as there are classes in the problem (e.g. a softmax layer). The training is then performed for all layers in a supervised manner (called fine-tuning).

III. Hardware parallelism for Neural Networks

A. *Mapping parallel OpenCL kernels on the device*

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

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

Algorithm 2: Testing Phase

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

1) Feed-forward: When the samples from the dataset and weights for that layer are loaded to the device's global memory, the initial phase is started by sending data through the network. The kernel is launched on the device across two dimensions, the first being equal to the output nodes of the current layer and the second relative to the amount of samples from the dataset. This means that one particular work-item is responsible for one output node when all the input nodes from one sample go through it. Inside the kernel, a weighted sum is computed in a loop, over all the layer input nodes and respective weights for that particular output node, computing the overall sum of that product. An activation function (the sigmoid function), is then applied to that sum plus the bias of that output node. This kernel is valid for both the encoder and decoder phase of the AE, the only difference being the input varying between the original image for the encoder layer and the encoder output for the decoder layer.

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

3) Back-propagation (hidden layer): The kernel used for the back propagation in the hidden layer is close to that of the output layer. We do not have a reconstruction error for this layer but we are dependent on the gradient calculated in the output layer. The kernel is then launched with one dimension, the size of the hidden layer output nodes.

The product of the weights of this layer and the output gradient is summed across the input nodes, with the resulting sum replacing the error in the previous algorithm, finally obtaining the gradient for this layer. The kernel then proceeds to compute the partial derivatives as described in the output layer kernel.

When the back propagation for this hidden layer comes to an end, the partial derivatives are then copied to the host where a simple loop updates the weights and bias, this being a fast and low computationally demanding operation.

In order to implement the aforementioned parallel kernels, we developed parallel kernels for mobile GPUs that are exploited under the OpenCL framework context. In the next subsections we identify the main optimizations performed for this novel approach.

B. Mobile GPU specific high-level memory optimizations

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

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

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

2) Floating-Point Processing: Since the dynamic range of the weights will vary even if the initial random generation is limited to a small interval, we can expect a superior behavior from the SAE by mapping weights and inputs/outputs as floating-point values. In fact, although fixed-point computations are more efficient in both throughput performance and resource usage, the most recent mobile phones are being deployed with DSP blocks capable of raising floating-point computing resources and minimizing the impact in throughput performance.

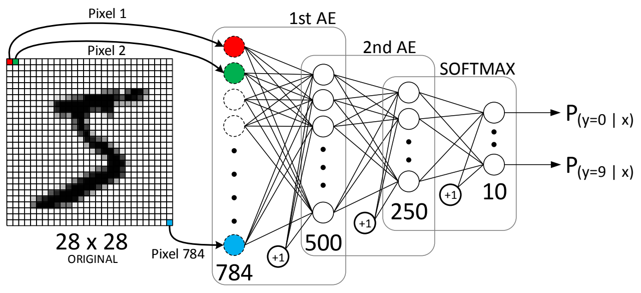


Fig. 1. Topology of the Stacked Autoencoder for the MNIST dataset.

IV. Experimental Results on Low-power Architectures

The goal of the experiments was three-fold: 1) validate the implementations in all devices; 2) allow comparing energy consumption between the tested platforms; and 3) find their hardware limitations namely due to memory and processing capabilities. For this we have chosen a well known dataset, the MNIST [6]. It 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 was divided into a training set with 50000 images, and validation and test sets with 10000 images each. The five computing platforms used in these experiments are listed in Table I.

TABLE I
Computing Platforms. Mobile devices have shared RAM.

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

The desktop GPU is used only for reference, since our focus is on low-power devices. The training hyper-parameters defined for the SAE consist of a training batch of 64 images and an initial learning rate of 0.45 on a network of size $784 - 500 - 250 - 10$, as depicted in Fig 1. For this particular SAE we achieved a classification error of 1.47% training during 1500 epochs.

For the energy consumption analysis in Table II we kept the same SAE architecture using 1 epoch. These measurments scale linearly with the number of epochs. Power consumption was calculated measuring the idle requirements of the entire system (host and device) and then launching the application, measuring the power difference (load - idle) over the SAE execution time, using a power meter for the desktop refGPU and the PowerTutor [7] application for the remaining devices.

By analyzing Table II, it is possible to verify that regarding energy consumption, mobile devices are clearly better than the reference desktop refGPU, achieving the same results while consuming only from 0.69% to 1.61% of the energy, which

TABLE II
EXECUTION TIME OF 1 EPOCH AND ENERGY CONSUMPTION ON A NETWORK OF SIZE 784-500-250-10 (*LOWER IS BETTER)

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

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

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

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

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

As a term of comparison, the desktop GPU ran the SAE for each mobile devices' largest architecture in 5m43s, 9m13s, 7m40s and 11m40s, respectively.

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

To further grasp hardware limitations results, there are several factors that need to be considered: first, mobile GPUs do not have dedicated memory, so the memory that is available is small and managed by the SoC, varying between devices (as can be seen with mGPU2 and mGPU3 that have the same amount of RAM but support different maximum sizes of the deep neural networks); also, even using the same SoC, results can vary by simply using different OS versions that can implement different resource management policies; and finally, we have to consider that mobile devices only recently started supporting OpenCL, so these implementations have still margin to progress. With the expected advances of hardware and new OpenCL implementations, OpenCL capabilities in mobile devices will likely improve considerably in the near future.

V. CONCLUSIONS

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

Despite being one order of magnitude slower than on a desktop GPU, the training on mobile GPUs uses less than 2% energy than it would on the desktop counterpart, opening the doors to start performing demanding algorithms directly on autonomous vehicles, robots and other low-power applications, and even cases where online training is not possible.

Moreover, this study paves the way for a future technology progression, as mobile GPUs with more hardware resources are developed. This may use state-of-the-art networks, such as CNNs, running exclusively on low-power devices achieving top results in energy savings as well as classification accuracy. Additionaly, the use of approaches such as BinaryConnect [8] and Deep compression [9] to improve speed and reduce storage needs can further contribute to this goal.

Regarding the hardware, higher integration between RAM access of CPUs and GPUs can introduce a substantial reduction in processing times, as suggested by the Heterogeneous System Architecture(HSA) proposed by AMD and ARM.

Finally, we have made the OpenCL source code available[1] to the community that wishes to replicate these experiments.

REFERENCES

[1] A. Coates, B. Huval *et al.*, "Deep learning with COTS HPC systems," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 2013, pp. 1337–1345.

[2] J. Maria, J. Amaro *et al.*, "Stacked autoencoders using low-power accelerated architectures for object recognition in autonomous systems," *Neural Processing Letters*, vol. 43, no. 2, pp. 445–458, 2015.

[3] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[4] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[5] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, Jan. 2009.

[6] Y. LeCun. (2014) MNIST Dataset. http://yann.lecun.com/exdb/mnist/. Accessed: 2015-04-15.

[7] "PowerTutor: A Power Monitor for Android-Based Mobile Platforms," http://ziyang.eecs.umich.edu/projects/powertutor, Accessed: 2015-04-15.

[8] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016.

[9] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[1]https://goo.gl/KzeDWx

Bibliography

[1] Y. LeCun, L. Bottou, and Y. Bengio, "Reading checks with graph transformer networks," in *International Conference on Acoustics, Speech, and Signal Processing*, vol. 1. Munich: IEEE, 1997, pp. 151–154.

[2] I. Kononenko, "Machine learning for medical diagnosis: History, state of the art and perspective," *Artif. Intell. Med.*, vol. 23, no. 1, pp. 89–109.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[4] M. H. Kabir, M. R. Hoque, H. Seo, and S.-H. Yang, "Machine learning based adaptive context-aware system for smart home environment," *International Journal of Smart Home*, vol. 9, pp. 55–62, 2015.

[5] A. Dixit and A. Naik, "Use of prediction algorithms in smart homes," *International Journal of Machine Learning and Computing*, vol. 4, 2014.

[6] I. G. Y. Bengio and A. Courville, "Deep learning," 2016, book in preparation for MIT Press.

[7] A. G. Ivakhnenko, "Polynomial theory of complex systems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-1, no. 4, pp. 364–378, Oct 1971.

[8] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, Dec. 1989.

[9] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, no. 7, pp. 179–188, 1936.

[10] E. Anderson, "The species problems in iris," *Annals of the Missouri Botanical Garden*, vol. 23, pp. 179–188, 1936.

[11] W. M. Fisher, G. R. Doddington, and K. M. Goudie-Marshall, "The darpa speech recognition research database: Specifications and status," in *Proceedings of DARPA Workshop on Speech Recognition*, 1986, pp. 93–99.

[12] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[14] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, "Deep learning with COTS HPC systems," in *Proceedings of the 30th International Conference on Machine Learning, Cycle 3*, vol. 28, 2013, pp. 1337–1345.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[16] J. Ward, S. Andreev, F. Heredia, B. Lazar, and Z. Manevska, "Efficient mapping of the training of convolutional neural networks to a cuda-based cluster," 2011.

[17] A. Krizhevsky, “One weird trick for parallelizing convolutional neural networks,” *CoRR*, vol. abs/1404.5997, 2014.

[18] D. Erhan, C. Szegedy, A. Toshev, and D. Anguelov, “Scalable object detection using deep neural networks,” *CoRR*, vol. abs/1312.2249, 2013.

[19] C. Szegedy, A. Toshev, and D. Erhan, “Deep neural networks for object detection,” in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 2553–2561.

[20] L. Tóth, “Combining time- and frequency-domain convolution in convolutional neural network-based phone recognition,” in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2014, pp. 190–194.

[21] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, “Convolutional neural networks for speech recognition,” *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, vol. 22, no. 10, pp. 1533–1545, 2014.

[22] M. Lai, “Giraffe: Using deep reinforcement learning to play chess,” *CoRR*, vol. abs/1509.01549, 2015.

[23] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[24] A. Krizhevsky, “Learning multiple layers of features from tiny images,” University of Toronto, Tech. Rep., 2009.

[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] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, G. J. Gordon and D. B. Dunson, Eds., vol. 15, 2011, pp. 315–323.

[27] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato, “Multi-gpu training of convnets,” *CoRR*, 2013.

[28] G. E. Moore, “Cramming More Components onto Integrated Circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[29] NVIDIA, “Inside pascal: Nvidia’s newest computing platform,” https://devblogs.nvidia.com/parallelforall/inside-pascal/, accessed: 2016-08-30.

[30] NVIDIA, “Kepler GK110 whitepaper,” http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, accessed: 2016-08-30.

[31] A. Torralba, R. Fergus, and W. T. Freeman, “80 million tiny images: A large data set for nonparametric object and scene recognition,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970.

[32] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[33] F. N. Iandola, D. Sheffield, M. J. Anderson, P. M. Phothilimthana, and K. Keutzer, “Communication-minimizing 2d convolution in GPU registers,” in *IEEE International Conference on Image Processing, ICIP 2013, Melbourne, Australia, September 15-18, 2013*, 2013.