Applied Artificial Intelligence

An International Journal

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/uaai20

Distributed Learning of CNNs on Heterogeneous CPU/GPU Architectures

Jose Marques, Gabriel Falcao & Luís A. Alexandre

To cite this article: Jose Marques, Gabriel Falcao & Luís A. Alexandre (2018) Distributed Learning of CNNs on Heterogeneous CPU/GPU Architectures, Applied Artificial Intelligence, 32:9-10, 822-844, DOI: 10.1080/08839514.2018.1508814

To link to this article: https://doi.org/10.1080/08839514.2018.1508814

Published online: 10 Sep 2018.

Submit your article to this journal

Article views: 500

View related articles

View Crossmark data

Citing articles: 2 View citing articles

Full Terms & Conditions of access and use can be found at
https://www.tandfonline.com/action/journalInformation?journalCode=uaai20

APPLIED ARTIFICIAL INTELLIGENCE
2018, VOL. 32, NOS. 9–10, 822–844
https://doi.org/10.1080/08839514.2018.1508814

Distributed Learning of CNNs on Heterogeneous CPU/GPU Architectures

Jose Marques[a], Gabriel Falcao[a,b], and Luís A. Alexandre[c]

[a]Instituto de Telecomunicações, Department of Electrical and Computer Engineering, University of Coimbra, Coimbra, Portugal; [b]Visiting Professor at École Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland; [c]Instituto de Telecomunicações, Departamento de Informática, Universidade da Beira Interior, Covilhã, Portugal

ABSTRACT

The convolutional neural networks (CNNs) have proven to be powerful classification tools in tasks that range from check reading to medical diagnosis, reaching close to human perception, and in some cases surpassing it. However, the problems to solve are becoming larger and more complex, which translates to larger CNNs, leading to longer training times that not even the adoption of Graphics Processing Units (GPUs) could keep up to. This problem is partially solved by using more processing units and distributed training methods that are offered by several frameworks dedicated to neural network training, such as Caffe, Torch, or TensorFlow. However, these techniques do not take full advantage of the possible parallelization offered by CNNs and the cooperative use of heterogeneous devices with different processing capabilities, clock speeds, memory size, among others. This paper presents a new method for the parallel training of CNNs where only the convolutional layer is distributed. The paper analyzes the influence of network size, bandwidth, batch size, number of devices, including their processing capabilities, and other parameters. Results show that this technique is capable of diminishing the training time without affecting the classification performance for both CPUs and GPUs. For the CIFAR-10 dataset, using a CNN with two convolutional layers, and 500 and 1500 kernels, respectively, best speedups achieve 3.28 × using four CPUs and 2.45 × with three GPUs. Larger datasets will certainly require more than 60-90% of processing time calculating convolutions, and speedups will tend to increase accordingly.

ARTICLE HISTORY
Received 15 January 2018
Accepted 23 July 2018

Introduction

Deep learning has been the engine behind tasks that are considered common nowadays. One of the most used models within deep learning is the CNN. The technological development allowed the access to computational resources capable of training increasingly larger neural networks, and also

CONTACT Luís A. Alexandre lfbaa@di.ubi.pt Instituto de Telecomunicações, Departamento de Informática, Universidade da Beira Interior, Covilhã, Portugal

This article has been republished with minor changes. These changes do not impact the academic content of the article. Color versions of one or more of the figures in the article can be found online at www.tandfonline.com/uaai.

© 2018 Taylor & Francis

larger datasets. More specifically, it was due to the development of faster CPUs and 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 (Dean et al. 2012), capable of training networks with as much as 1.7 billion parameters, currently among the largest of their type. However, it should be noted that this framework uses thousands of CPU cores distributed along hundreds of machines and the training takes days to complete.

Another important technological development for the evolution of deep learning was the adoption of GPU architectures, more specifically the use of GPGPU for scientific and generic computation.

A distinctive aspect of CNN lies on the so-called convolutional layers, which makes it the ideal choice for image and speech recognition, since both tasks rely heavily on the correlation of neighboring data. The major problem with it is that the processing of convolutions is computationally intensive, requiring from 60% to 90% of the total training time only using about 5% of the parameters of the whole network (Krizhevsky 2014; Ward et al. 2011). Although for such scenario, Amdahl's Law constrains speedups to the range 2.5~10, this work shows that it is possible to work near those limits. Also, while tools such as Caffe or TensorFlow usually explore the compute power of a single GPU or a small group of homogeneous GPUs on the same node/server, this work proposes an alternative for filling in the gaps of the above mentioned frameworks, namely by using truly heterogeneous CPU and GPU parallel computing architectures, isolated or grouped in distinct nodes/clusters, eventually in different physical locations, for the compute-intensive training of deep learning with balanced workloads.

Thus, the main contribution of this paper consists in an open-source distribution technique that makes use of the potential parallelization that convolutional layers have to offer, feeding platforms of conventional heterogeneous CPU and GPU devices the same feature maps, but providing them with different kernels and balanced workloads, gaining speed up during the computation of convolutions that compensates communication times for orchestrating the different nodes. This contribution is likely to produce significant impact, since under the current context of training CNNs, computation times can easily vary from days to weeks (Keuper and Preundt 2016).

Distributed Convolutional Neural Networks

Over the last years, Deep Neural Networks (DNNs) were able to achieve performances, better than all humans or the best ones at it, in games like chess (Lai 2015) and Go (Silver et al. 2016).

Despite having shown to be a powerful machine learning technique, when applied to large inputs, like images, most DNN like deep belief nets or stacked

autoencoders, became rather complex and sizable, capable of reaching millions of weights for simple inputs that consist of RGB images of size 32×32.

Another problem is that these networks neglect correlation between neighboring data, like translations and distortions, despite there are local correlations in pattern recognition problems. Ideally, local features would be extracted and analyzed in order to be able to detect certain beings or objects. CNN, however, are able to overcome those issues by making use of 3 key factors: local receptive fields, weight sharing, and spatial pooling.

Convolutional Neural Networks

The first proposal of a model similar to a CNN can be attributed to Fukushima (1980) with the neocognitron. It served as inspiration to the modern concept of CNN, which was introduced by LeCun and Bengio (1995) and also 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, such as 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 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 achieving better classification performances when compared to other approaches.

The main differences between CNN and other DNN 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 CNN is LeNet-5 (LeCun et al. 1989). It contains two convolutional layers and two 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.

Distributed Training Techniques

Training the largest CNN is becoming a real challenge even using GPU, either because datasets are growing fast in size and these parallel machines are limited in memory, or simply because the training times still remain quite long. Performing the distributed training of CNN fosters accelerating this type of complex processing. This section aims to provide some insight regarding the most recent techniques of distributed training.

Distributed training can refer to distributing the training of the network across several GPU or CPUS in the same or in different computers. There are mainly two types of techniques for performing the distribution: data parallelism and model parallelism.

Data Parallelism

In data parallelism, the batch of data is split across the several nodes of the cluster, such as CPU, GPU, 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 parameter 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 GPU 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.

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 (Dean et al. 2012) 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 possible implementation (Yadan et al. 2013) 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.

A different 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, with a tile representing a different image (Ward et al. 2011). 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 device will need to have every kernel.

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

Related Work

There are a few works that address the speedup achievable with distribution techniques, mostly data parallel ones. Among the few frameworks that allow the distributed training of convolutional networks, TensorFlow (Abadi, Agarwal, and Barham et al. 2015) offers the possibility of training a model in a parallel, distributed fashion, providing the code to do so with the CIFAR-10 dataset (Krizhevsky 2009), using several GPUs on the same machine. Each GPU is envisioned to have similar speed and have enough

memory for the entire network, so a model replica is placed in each GPU, with the model parameters being updated synchronously, having to wait for all GPUs to complete the processing of the corresponding data.

The results provided by TensorFlow in file *cifar10_multi_gpu_train.py* of its source code show that the introduction of a second GPU is able to reduce the step time to less than half of the case with a single device; however, for the remaining cases, with three and four GPUs, the step time is barely reduced, showing that it does not seem to scale.

There are also studies that try to distribute the training of a CNN across different machines, such as (Vishnu, Siegel, and Daily 2016), that use several CPUs connected using InfiniBand. They also used TensorFlow, taking advantage of data parallelism. All the samples are divided equally across devices, as each device is considered to have the same exact computational capabilities. The updates are performed synchronously using MPI, which is heavily optimized, allowing for a minimal time being spent in communications. The results relative to a CNN trained with CIFAR-10 show a speedup of $3.01\times$ when scaling from 4 to 64 cores.

The facts that distinct devices may receive the same amount of data and update the parameters synchronously are limitations to the use of real-life machines that have different capabilities. This is the main motivation in developing a significantly distinct approach capable of performing CNN training in truly heterogeneous devices. Another problem is the fact that these studies seem to be limited to distribute the training across a maximum of four GPUs placed in a single machine.

Thus, our proposal is to develop a parallelization scheme that is able to train a CNN using the resources of different machines available on a network, with distinct computational resources. To avoid limiting the training time to the slowest machine used, a quick test is performed on all machines, so as to grasp the computational capabilities of each device. Since the distribution of the workload is performed during runtime, it allows the use of a wide variety of devices, each one receiving a proportional share of the workload.

Distributed Convolutional Learning

Hybrid CPU–CPU and GPU–GPU Computing

One of the major problems that arises with the usage of computers having different CPU and GPU is that distinct devices have different computational resources and thus are able to complete the same workload in different times. This can become a problem, especially when one or several devices are relatively slow compared to others.

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 its convolutions at approximately the same time.

To do so, a pre-processing procedure is performed, where every device runs an N-dimensional convolution with both the size of the images and the size of the kernels provided by the master device, trying to simulate part of the convolutional layer. The convolution is run using random values, since only the time spent performing calculations is relevant. After the respective simulations complete, the computation time is reported to the master node in order to find the performance ratio between devices, either CPU or GPU. The slave nodes only need to know the IP address of the master node, while the master node needs to know the number of slave nodes and their respective IP addresses.

However, considering that during the experiment three to four 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 two devices. In general, for n devices, each with a time to complete the task given by t_i, $i = 1, \ldots, n$, we define the workload for each device as follows:

$$w_i = \frac{\frac{\max(t)}{t_i}}{\sum_{j=1}^{n} \frac{\max(t)}{t_j}}. \quad (1)$$

Workload Distribution

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 calculates convolutions for the convolutional layer. This exploits the fact that convolutional layers use less than 10% of the parameters (Krizhevsky 2014), indicating that communication overheads will not become a relevant problem when compared to the computation time saved.

For this approach, one of the nodes orchestrates and is designated as master node, while the remaining ones are slave nodes. Considering only the convolutional layer is the subject of distributed training, the master node is in charge of training the remaining network.

For the convolution distribution, 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. 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, each slave sends the feature maps, after which 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.

Experiments

Hardware Platforms Setup

As can be seen in Table 1, the computers are all composed by a set of distinct devices, so the need to perform hybrid CPU–CPU and GPU–GPU processing has emerged. The code for this experiment was written in Matlab. Thus, the compatible framework for performing parallel computing is CUDA, meaning that the GPU cluster uses three computers (PC2, PC3, and PC4), since only NVIDIA GPUs are supported. The CPU cluster runs with all computers available.

Network Architecture and Dataset

For this experiment, the dataset used was CIFAR-10 (Krizhevsky 2009). It consists of a labeled subset of the 80 million tiny images dataset in Torralba, Fergus, and Freeman (2008). The dataset contains 60000 32×32 color images grouped into 10 classes, with each class having 6000 images. Of the total 60000 images, 50000 are intended to be used for training and the remaining 10000 for testing. The classes present in this dataset are as follows: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. This dataset was chosen particularly for consisting of color 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 CNN architectures in shorter periods of time compared to other datasets like Imagenet (Russakovsky et al. 2015) and is therefore able to serve as a proof of concept. A third reason for choosing CIFAR-10 is related with comparison purposes, since it is a popular dataset used many times in the literature.

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

Table 1. CPUs and GPUs used in the experiments.

PC	CPU	RAM	GPU	RAM
PC1	i5-3210M @ 2.5GHz	6GB	Radeon HD 7500M	N/A
PC2	i7-4700HQ @ 2.4GHz	8GB	GeForce 840M	2GB
PC3	i7-5500U @ 2.4GHz	8GB	GeForce 940M	2GB
PC4	i7-6700HQ @ 2.6GHz	16GB	GeForce 950M	4GB

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) quantify the influence that the number of kernels in each convolutional layer has on speedups; and (3) evaluate how the batch size impacts the speedups.

To achieve that, the number of kernels on each convolutional layer was varied, testing four different network architectures. The smallest tested CNN has 50 kernels in the first convolutional layer and 500 on the second one. The next architectures 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, respectively.

Experimental Results

Speedups Using CPU-cluster

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 two, three, and four devices, respectively.

Figure 1 shows the results by maintaining the same network architecture and varying the batch size; thus, it is possible to understand the influence of batch sizes by analyzing each subfigure individually, and to study how the number of kernels affects the attained speedup by comparing the result of each batch size across different subfigures.

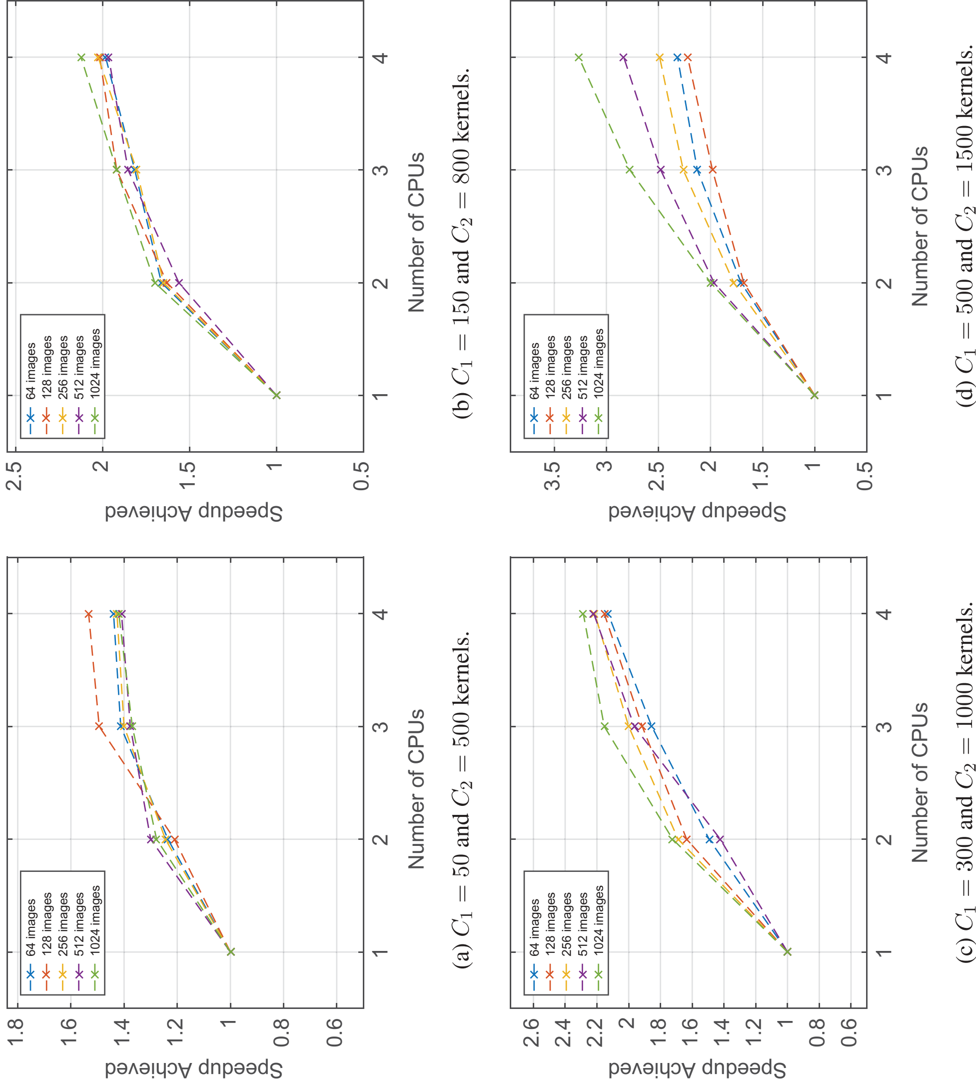
Figure 1 shows that a speedup always exists, even when considering the smallest network and batch size. Figure 3 shows that the introduction of more CPU contributes to an improvement on processing time, achieving speedups of $1.3\times$ for 2 CPUs, $1.5\times$ for 3 and slightly above $1.5\times$ for 4 CPUs.

A batch size influence analysis on the distribution technique performance starts by comparing each subfigure individually. For the smallest considered network, the difference in batch size does not introduce significant changes, since the speedups attained for four CPUs are between $1.4\times$ and $1.55\times$.

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 continue to be almost nonexistent.

However, for the largest network tested, there is a more prominent difference when training it with different batch sizes, with the speedups for 4 CPUs ranging from $2.21\times$ to $3.28\times$. Nonetheless, the difference of speedups between different batch sizes across the other trained networks is very small.

To understand the effects that the number of kernels has on the attained speedups, it is necessary to analyze the results pertaining a batch size across the different network architectures.



(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 1. Attained speedup for all batch sizes, using a CPU cluster ranging from 1 to 4 machines.

Considering the case with a batch of 64 images, it is visible that the speedup increases from 1.45 × using the smallest network to almost 2.25 × , with the largest one.

By further analyzing the effects of the network architecture, a quick study involving the remaining batch sizes show that the increase in convolutional layers always leads to an improvement in speedup, with the worst case being with 128 images, where the 4 CPU case improves from 1.52x to 2.2 × when considering the largest network. The best case scenario comes when using a batch of 1024 images, considering the speedups climb from 1.4 × to 3.25 × when considering 4 CPUs training the largest network.

For the largest considered architecture, the speedups are always above 1.65 × with 2 CPUs and are capable of achieving speedups greater than 2 × for 3 or more CPUs. To understand how communication times differ, it is necessary to analyze how the training time is distributed. Figure 2 shows the elapsed time relative to only one batch of 1024 images, since the time for the training of an entire epoch is mostly linear. The full training time is divided into three parts: *Comm. time* refers to the communication time between master node and 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 other than convolutions.

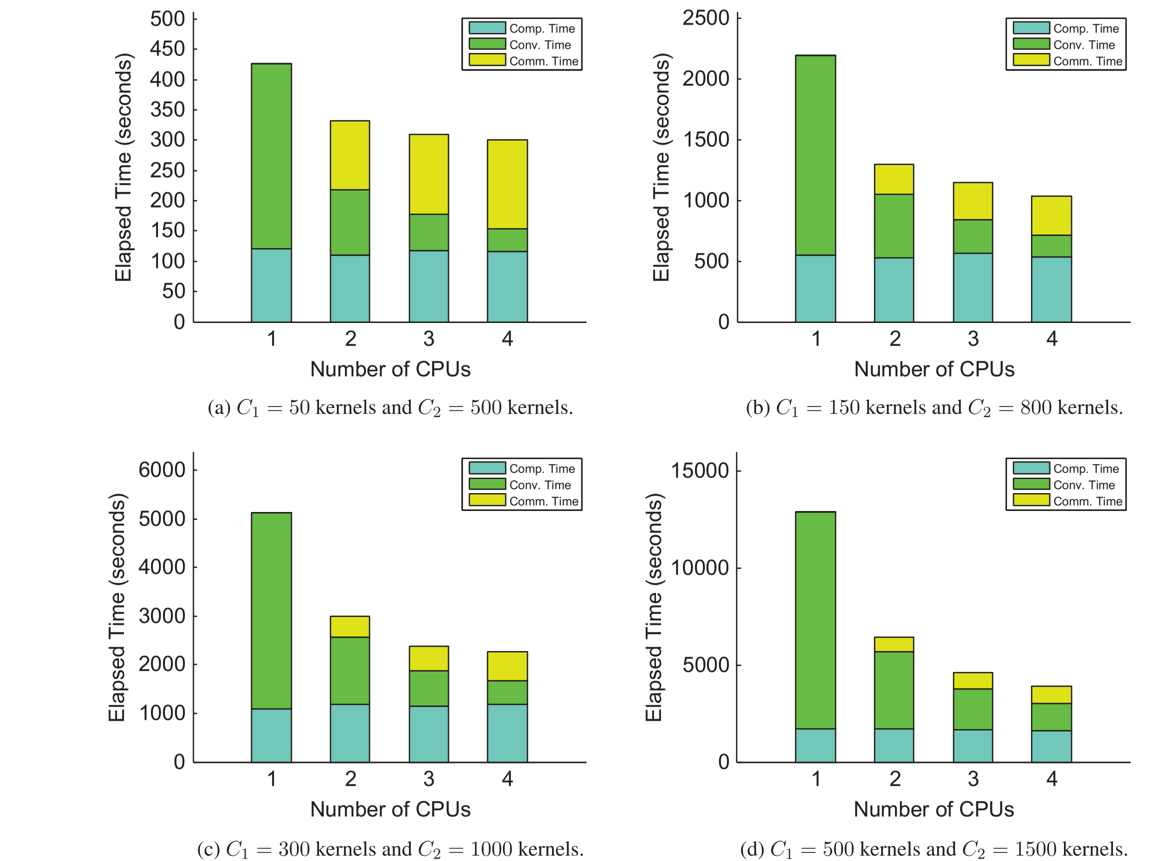
The training time using one CPU presents a decrease in the percentage of time dedicated to the computation of different layers, going from 25% with the smallest network to 13% when training the largest one. A more thorough analysis of the largest network using a batch with 1024 images shows that, as it can be seen in Figure 2d, the use of 2 CPUs achieves a speedup of 1.98 x, while for 3 and 4 CPUs the attained speedup is 2.73 x and 3.28 x, 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 7.76 x. Therefore, networks that rely more heavily on convolutions will be able to achieve better speedups.

Thus, it is possible to see that speedups are more dependent on the number of kernels than batch size, for the CPU case.

Vishnu *et al.* also parallelize a CNN training with CIFAR-10, using several CPUs connected with InfiniBand and are able to achieve a speedup of 3.01 x using 64 cores, relative to the training time using 4 cores (Vishnu, Siegel, and Daily 2016).

Speedups Using GPU-cluster

Only 3 of the 4 computers could run matlab and CUDA; hence, the maximum size of the GPU cluster is 3 machines, which allows the comparison between CPU and GPU up to a certain point.



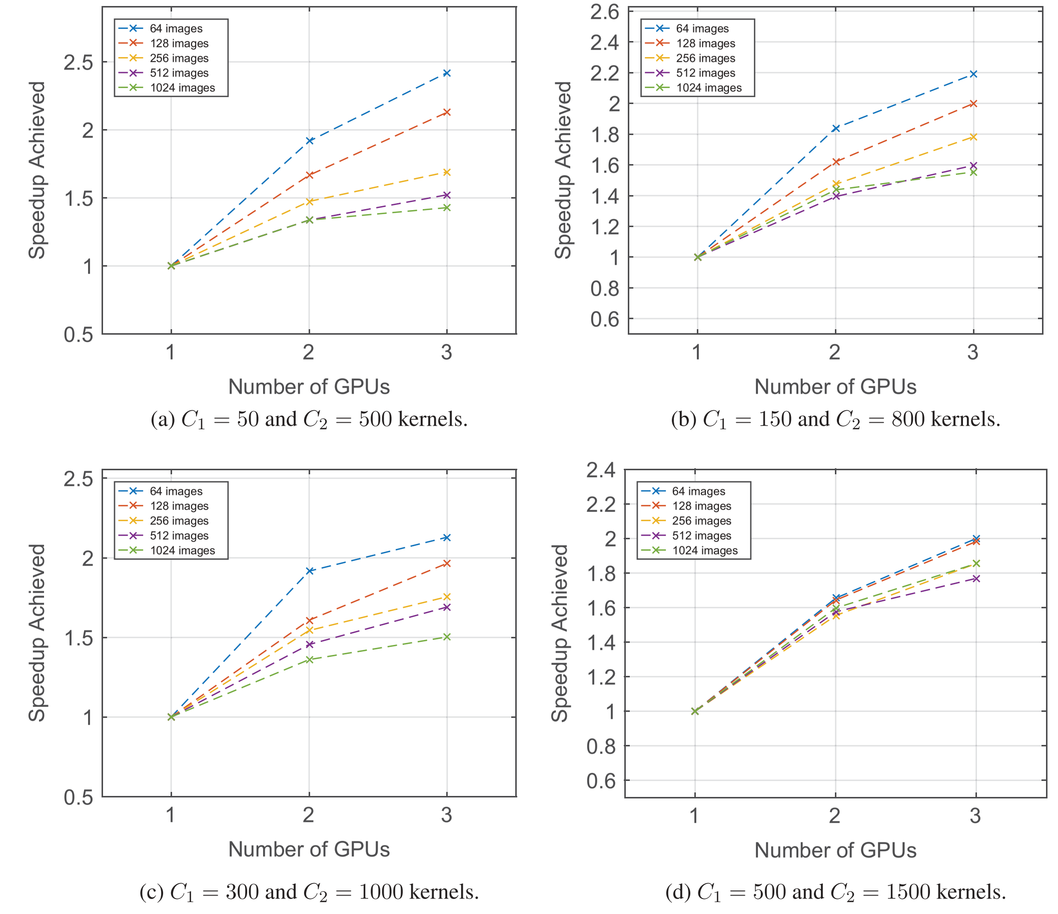
(a) $C_1 = 50$ kernels and $C_2 = 500$ kernels.

(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 2. Elapsed time for a batch size with 1024 images, using a CPU cluster ranging from 1 to 4 machines.



(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 3. Attained speedup for all batch sizes, using a GPU cluster ranging from 1 to 3 machines.

Another aspect to consider is the computational capabilities of the GPU. As stated previously, as long as divergent operations are negligible or controlled 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 multiplication, very quickly, due to the large number of parallel cores available. However, for smaller amounts of data globally the GPU handles these tasks less efficiently 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.

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 addition of more nodes for the cases with 2 and 3 devices, respectively. This notation respects the one used in Section 5.1.

The analysis of the influence of batch sizes on the distribution technique performance is performed by comparing each subfigure individually from Figure 3, like in the CPU case. Considering the smallest trained network, there is a considerable difference between the distinct batch sizes, with speedups for 3 GPUs ranging from $1.45\times$ to $2.45\times$.

This is a trend that continues with the next two architectures, where speedups range from $1.5\times$ to $2.2\times$ on both cases, using 3 GPUs for the training.

However, for the largest trained network, the range of speedups is much smaller, when analyzing the different batch sizes, fluctuating between $1.75\times$ and $2\times$.

To understand the effects that the number of kernels has on the attained speedups, it is necessary to analyze the results pertaining a batch size across the different network architectures, as in the CPU case.

Considering the case with a batch of 64 images, it is visible that the speedup decreases from $2.45\times$ using the smallest network to $2\times$, with the largest one.

This trend continues with a batch size of 128, with speedups decreasing from $2.15\times$ to $1.95\times$, but change with the remaining batches, with speedups increasing with larger networks. To understand the differences between the CPU and GPU cases, it is necessary to analyze how the training time is distributed. Figure 4 shows the elapsed time relative to only one batch of 1024 images, since the time for the training of an entire epoch is mostly linear. The full training period is divided into three parts: *Comm. time* refers to the communication time between 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 nodes. Finally, *Comp. time* is the time spent on computation of layers other than performing convolutions.

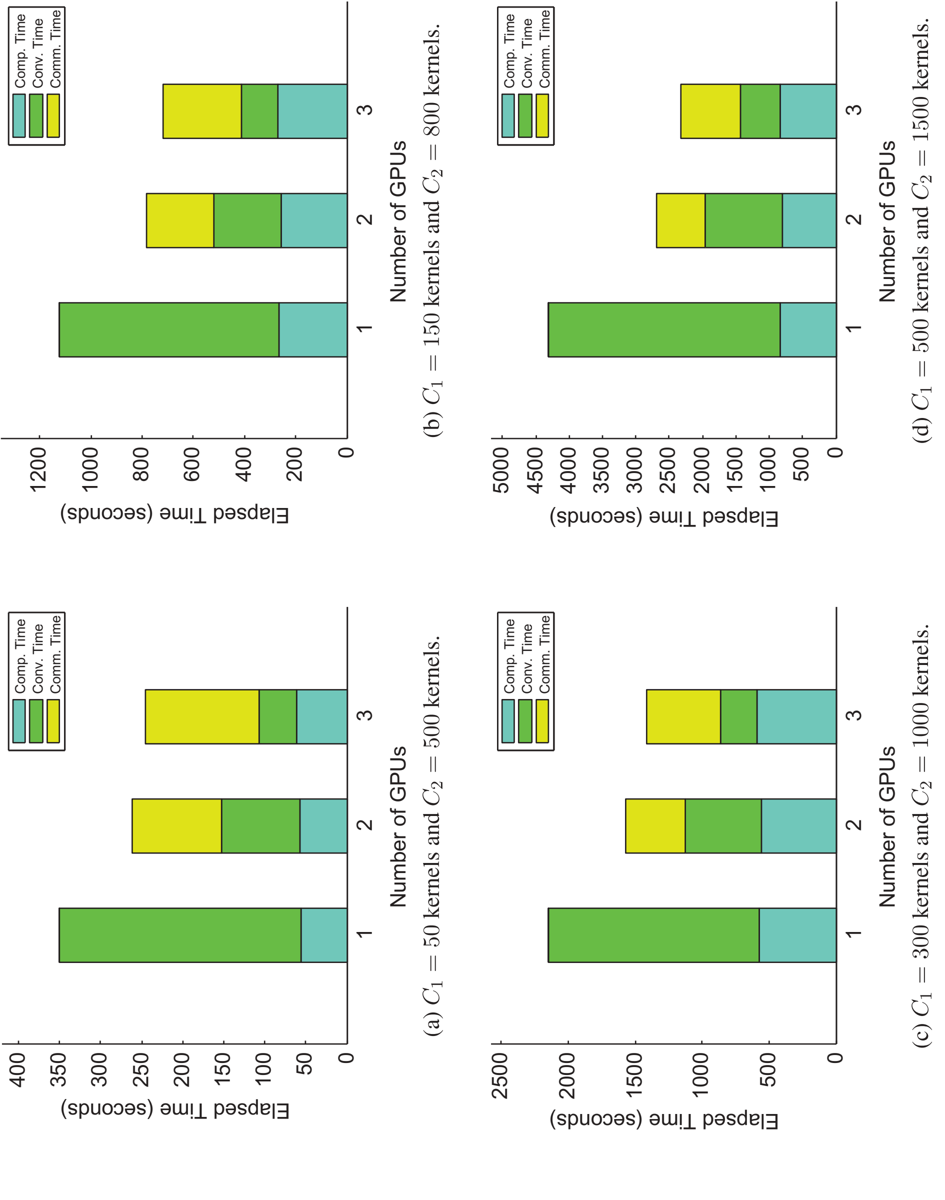


Figure 4. Elapsed time for a batch size with 1024 images, using a GPU cluster ranging from 1 to 3 machines.

As shown in Figure 4, an increase of kernels in the GPU case makes almost no difference concerning communication time and speedup, and this is also visible in the rest of the tested architectures, trained using batches of 1024 images. All the architectures tested with this batch size show an attained speedup between $1.45\times$ and $1.80\times$ for 2 GPU and between $1.45\times$ and $2\times$ using 3 GPU, with the ratio between communication, convolution and computation time being virtually the same on the 3 considered experiments, with communication time rising from 19% with 2 GPU to 30% when using all 3 GPU.

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

Using the code provided by TensorFlow to train a CNN with CIFAR-10 with multiple GPUs on the same machine, it is possible to reduce the step time from $0.35-0.60$ seconds per batch with 1 GPU to $0.13-0.20$ seconds with 2 GPUs. However, the addition of more GPUs does not correlate to better speedups, since 3 GPUs are able to reduce the step time to only $0.13-0.18$ seconds and 4 GPUs still take 0.10 seconds per batch.

Comparison between CPU and GPU

Table 2 shows the best-attained speedups for each network architecture and a given number of devices, for both CPU and GPU. It should be noted that, for each case, speedup is obtained by comparing execution time against a single device of the same type:

As shown in Table 2, the difference between speedups using multiple CPU, for a given architecture, increases with the growing convolutional layers. The speedup improvement using 2 CPU is particularly small, although it reaches $1.98\times$ on the largest tested network. However, this tendency fades with the increase in CPU. By training the network with 3 CPU, the speedup is $1.93\times$ for the second smallest network, reaching $2.74\times$ for the largest architecture. Using 4 CPU gives a considerable gain in speedup, particularly for the network with 300 kernels on the first

Table 2. Best speedups achieved by network architecture and number of CPU and GPU used.

	CPU			GPU	
Network	2	3	4	2	3
50:500	1.40x	1.51x	1.56x	1.96x	2.45x
150:800	1.68x	1.93x	2.10x	1.89x	2.23x
300:1000	1.69x	2.15x	2.33x	1.78x	2.09x
500:1500	1.98x	2.74x	3.28x	1.66x	2.00x

convolutional layer and 1000 kernels on the second one, and the largest trained network. This is explained with the increase in communication time due to sending dozens more kernels to other nodes that are only a couple of KBs, being counterbalanced by convolutions' parallelization.

However, for the GPU implementation case, in Table 2, the speedups diminish with the enlargement of the convolutional layers. This happens because although the GPU is being used more efficiently with larger networks, the addition of more devices incurs in larger communication times, due to the need of sending a substantially higher number of kernels to the other devices. Thus, for larger networks, the attained speedup is significantly less than for smaller ones.

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 analyze 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 *upload* that are necessary to exchange between master and slave nodes can be described as follows:

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

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. All values transmitted are of 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 achieved shows that the bandwidth is approximately constant, averaging at 5 Mbps. Another aspect to consider is the number of kernels that should be passed to each worker, which is explored more in detail in Section 4.1.

By understanding these details, it is possible to accurately predict new communication times when more nodes are added, as well as convolution times and therefore the total processing time. Three different cases where considered. The first two pertain to the CPU case (depicted in Figure 5), where the processing time for the smallest and largest

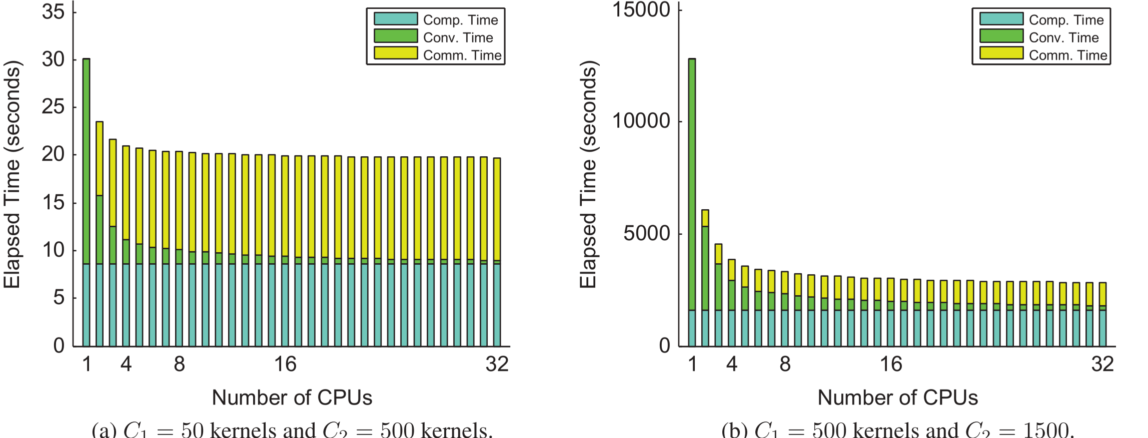


Figure 5. Elapsed time for the smallest network, using a batch with 64 images, and the largest network, with a batch size of 1024 images, using a CPU cluster ranging from 1 to 32 machines.

networks were simulated adding 32 CPU nodes. For these simulations, the CPU were considered to have computational capabilities similar to the devices used in the experiment (shown in Table 1), being assigned random performance values with Gaussian distribution, varying between worst and best case scenario for each CPU used. These results are shown in Figure 5.

As results show, the method is scalable without incurring in performance loss, despite becoming irrelevant the introduction of more nodes after a certain value. Both the case of the smallest network and the largest one benefit little from the addition of more nodes from 4 CPU, and there is a stabilization in speedup after 8 nodes. This occurs because the inclusion of more CPU 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. But when using several CPU, 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 with parallelization.

The final case refers to the GPU case, where only the largest network was simulated up to 32 nodes (Figure 6). This is justified with the fact that the most efficient use of the GPU occurs with the largest network,

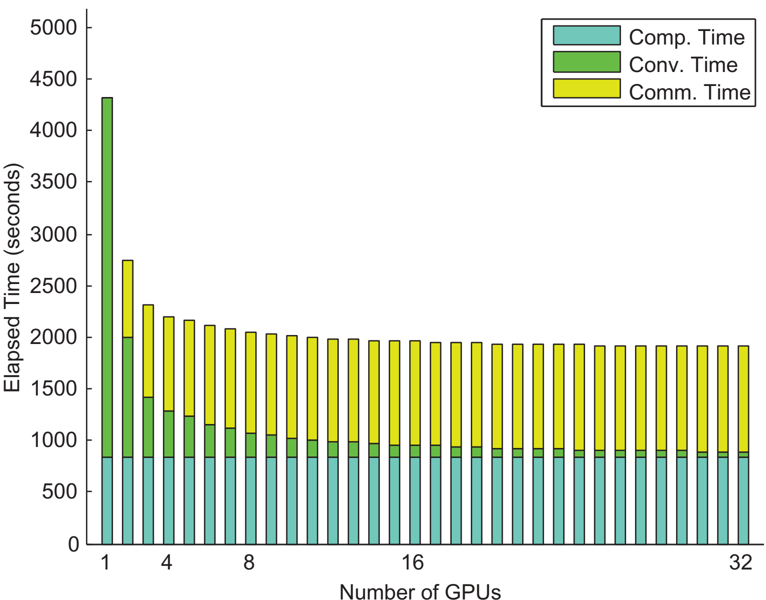


Figure 6. Elapsed time for the largest network, with a batch size of 1024 images, using a GPU cluster ranging from 1 to 32 machines.

trained with the highest batch size of 1024 images. As in the CPU case, the added nodes were considered to have computational capabilities similar to the devices used (shown in Table 1), being assigned performance values between worst and best GPU case scenario used in the experiment. The results are detailed in Figure 6.

As in the cases considered for simulation using CPU, the solution using GPU is also scalable, with the speedup virtually stagnating for 8 or more nodes, as visible in Figures 7 and 8. Since the convolution is done more quickly on a GPU than on a CPU, communication and computation times assume greater impact as bottlenecks. Figures 7, 8 and 9 indicate speedups achieved for different data rates assumed in communications among distinct computing nodes. As stated previously, the communication time can be diminished with a faster data transmission, while the computation time can be improved with parallelization.

The results provided by TensorFlow source code serve as a mean of comparison to the scalability of this method. Said results show that the average attained speedups range from 2.6 × up to 3.5 × , respectively, for 2 up to 4 GPUs. Although these results are substantially better, there are two aspects that should be addressed. First, the GPUs used for the training with TensorFlow were all part of the same machine that requires specific hardware configuration and presents limitations (same hardware, same GPUs, max number of GPUs limited by motherboard), thus the data transmission rate is considerably higher than using devices in different machines, physically separated by the network, which incurs in significantly lower communication times.

Secondly, TensorFlow parallelizes the entire network, whereas our distribution technique focuses on the convolutional layers, thus turning the training of the remaining layers a bottleneck. Considering the case where communication times are virtually nonexistent (which is achievable with faster communications), the speedup obtainable with our method is about 4.3 × , which is slightly better.

Conclusions

The developed solution proves to be a useful tool for the distributed training of CNN. Although good performances were achieved, there is one other aspect that could, and should, be further explored, and that is implementation using OpenCL, as opposed to CUDA. Not only would that mean that other GPU could be used, such as AMDs, but more importantly, it would allow for the distribution of the training to be done using mobile GPU, as well as FPGA and other low-power devices. Despite not having the same computational resources as desktop CPU and GPU, they can be far more energy efficient, and would allow to achieve smaller energy consumption levels without compromising the desired

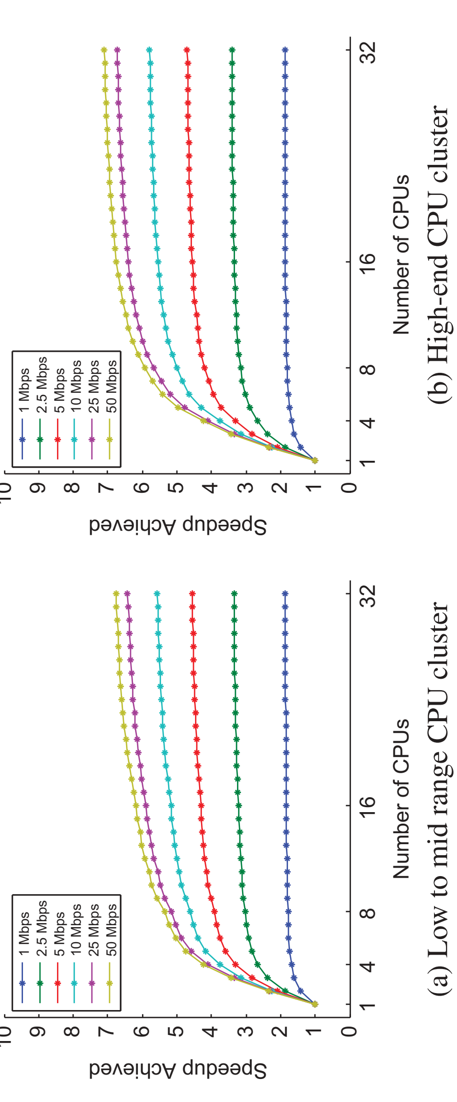


Figure 7. Speedups achieved on the largest network, trained with 1024 images for a cluster of 32 nodes using (a) low-to-mid range and (b) high-end CPU.

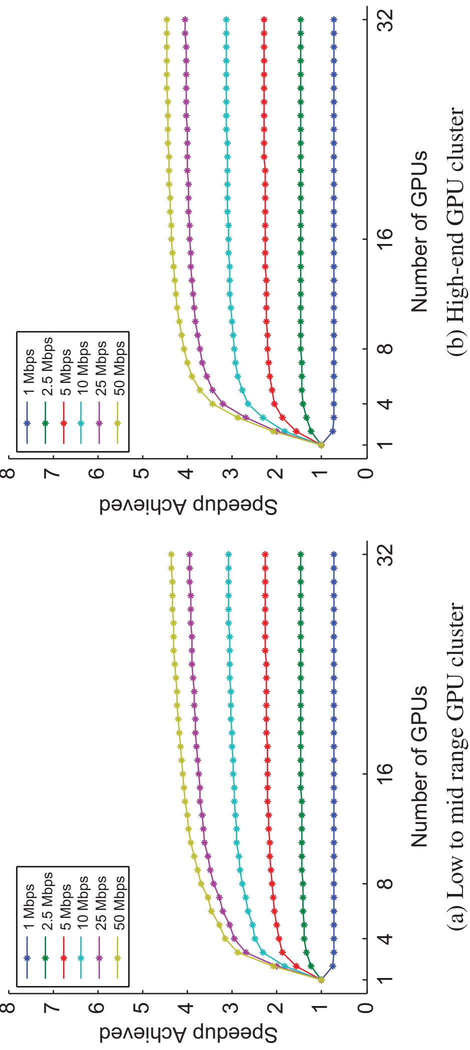


Figure 8. Speedups achieved on the largest network, trained with 1024 images for a cluster of 32 nodes using (a) low-to-mid range and (b) high-end GPU.

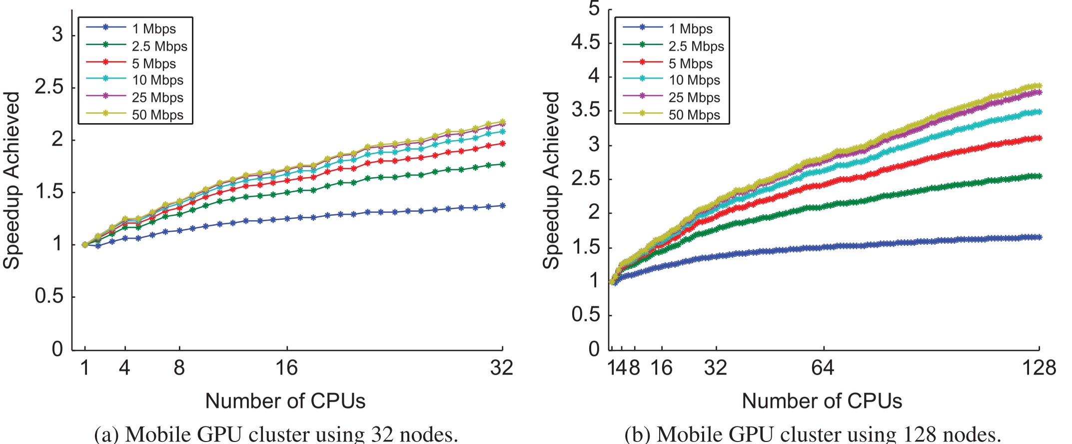


Figure 9. Speedup achieved on the largest network, trained with 1024 images for a mobile GPU cluster of (a) 32 and (b) 128 nodes.

throughput and classification performance, as the extrapolated curves in Figure 9 seem to indicate for the mobile low-power GPU case.

Acknowledgement

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

References

Abadi, M., et al. 2015. TensorFlow: Largescale machine learning on heterogeneous systems. Software available from tensorflow.org.

Dean, J., G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, et al. 2012. *Large scale distributed deep networks*. NIPS.

Fukushima, K. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* 36 (4):193–202. doi:10.1007/BF00344251.

Keuper, J., and F.-J. Preundt (2016). Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *2nd Workshop on Machine Learning in HPC Environments*.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto.

Krizhevsky, A. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR*. abs/1404.5997. https://arxiv.org/abs/1404.5997

Lai, M. 2015. Giraffe: Using deep reinforcement learning to play chess. *CoRR*. abs/1509.01549. https://arxiv.org/abs/1509.01549

LeCun, Y., and Y. Bengio. 1995. Convolutional networks for images, speech, and time-series. In *The handbook of brain theory and neural networks*, ed. M. A. Arbib. Cambridge, Massachusetts: MIT Press.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural Computation* 1 (4):541–51. doi:10.1162/neco.1989.1.4.541.

Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. 2015. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)* 115 (3):211–52. doi:10.1007/s11263-015-0816-y.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (7587):484–89. doi:10.1038/nature16961.

Torralba, A., R. Fergus, and W. T. Freeman. 2008. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30 (11):1958–70. doi:10.1109/TPAMI.2007.70837.

Vishnu, A., C. Siegel, and J. Daily. 2016. Distributed tensorflow with MPI. *CoRR*. abs/1603.02339. https://arxiv.org/abs/1603.02339

Ward, J., S. Andreev, F. Heredia, B. Lazar, and Z. Manevska. 2011. *Efficient mapping of the training of convolutional neural networks to a cuda-based cluster*. http://parse.ele.tue.nl/education/cluster2

Yadan, O., K. Adams, Y. Taigman, and M. Ranzato. 2013. Multi-gpu training of convnets. *CoRR*. https://arxiv.org/abs/1312.5853