DOI: 10.14529/jsfi210104

Computational Resource Consumption in Convolutional Neural Network Training – A Focus on Memory

Luis A. Torres[1,2,3], *Carlos J. Barrios*[1,2,3], *Yves Denneulin*[4,5,6,7,8]

Deep neural networks (DNNs) have grown in popularity in recent years thanks to the increase in computing power and the size and relevance of data sets. This has made it possible to build more complex models and include more areas of research and application. At the same time, the amount of data generated during the training process of these models puts great pressure on the capacity and bandwidth of the memory subsystem and, as a direct consequence, has become one of the biggest bottlenecks for the scalability of neural networks. Therefore, the optimizing of the workloads produced by DNNs in the memory subsystem requires a detailed understanding of access to the memory and the interactions between the processor, accelerator devices, and the system memory hierarchy. However, contrary to what would be expected, most DNN profilers work at a high level, so they only perform an analysis of the model and individual layers of the network leaving aside the complex interactions between all the hardware components involved in the training. This article shows the characterization performed using a convolutional neural network implemented in the two most popular frameworks: TensorFlow and Pytorch. Likewise, the behavior of the component interactions is discussed by varying the batch size for two sets of synthetic data and showing the results obtained by the profiler created for the study. Moreover, the results obtained when evaluating the AlexNet version on TensorFlow and its similarity in behavior when using a basic CNN are included.

Keywords: High-Performance Computing, deep learning, profiling, performance characterization, memory consumption.

Introduction

In recent years, Deep Learning (DL) algorithms have become popular due to their ability to extract features from input data. They are based on artificial neural networks and contain a lot of hidden layers, which is why they are known as Deep Neural Networks [18]. This type of network can contain millions of internal parameters resulting from multiple nonlinear and iterative transformations that occur in matrix or tensory form.

Deep neural networks have been divided into several groups including Convolutional Neural Networks, Recurrent Neural Networks (RNNs), and Long Short Term Memory (LSTM) [25]. These types of structures are used to create specific models according to a goal and require a great computing capacity and significant time (depending on the complexity of the model and computational resources) in their training to obtain a suitable model. It should be noted that the vast majority of this training time is used in Matrix-by-Vector Multiplication (MVM) tasks produced by convolutional or classification layers (dense layers). As mentioned earlier, such structures contain a large number of parameters and that, together with MVM operations performed on hidden layers, lead to a high transfer rate between memory, processor, and accelerator devices, requiring higher processing and memory capabilities. In this way, memory has

[1]Universidad Industrial de Santander, Bucaramanga, Colombia
[2]Supercomputación y Cálculo Científico UIS (SC3UIS), Bucaramanga, Colombia
[3]Cómputo Avanzado y a Gran Escala (CAGE), Bucaramanga, Colombia
[4]Université Grenoble Alpes, Grenoble, France
[5]French National Centre for Scientific Research (CNRS), Grenoble, France
[6]Inria, Grenoble, France
[7]Grenoble INP Institute of Engineering Univ. Grenoble Alpes, Grenoble, France
[8]Laboratoire d'Informatique de Grenoble (LIG), Grenoble, France

become a limit to the model that can be studied and the amount of data used for your training. Finally, it should be noted that MVM operations are highly parallelisable and therefore so are the Deep Learning algorithms.

Most studies using memory profilers are based on high-level understanding of the individual DNN layers or analytical models such as the Roofline [30] (roofline analysis helps to visualize the limits imposed by the hardware, as well as to determine the main limiting factor - memory bandwidth or computational capacity - thus leading to an ideal roadmap of possible optimization steps [29]). These approaches do not capture the complex interaction between the CPU, memory, and accelerator devices. As such, it only provides high-level memory performance sources in a static CPU / Memory / Device configuration.

The purpose of this work is to present the analysis of the characterization of the resources consumed in the training of a convolutional neural network implemented in both Tensorflow and Pytorch in order to determine the behavior of the workloads and identify possible causes of the bottlenecks in the training phase. The characterization has been performed using two synthetic datasets of sixty thousand and six hundred thousand 32x32x3 tensioners and using values of 32, 64, 128, 256 and 512 for the batch size.

In Section 1, we present a summary of related works that address solutions for memory problems in neural network training and where we start from as motivation for the realization of this work. Section 2 describes the methodology used to carry out the study, detailing the resources implemented in it. In Section 3, the reason for the memory problem presented during training is described. In Section 4 the results are shown and an evaluation of them is made according to the behavior obtained by the profiler. The last section concludes this paper.

1. Related Works

The requirements of computing capabilities for performing deep neural network training have increased significantly in recent years. Similarly, each new model that emerges for a specific field requires greater precision; much more complex and sophisticated models, with a larger number of layers and neurons, increase the number of trainable parameters of the network [9, 12]. These models dynamically generate tens or hundreds of MegaBytes of intermediate data (activations or feature maps for convolutional layers) for each layer of the network, data that often exceeds the capacity of the first levels of the memory hierarchy and puts enormous pressure on the bandwidth of the main memory [5].

All of the above have forced accelerator designers for deep neural networks to use high-cost memory solutions such as HBM (High Bandwidth Memory) used in Google TPUs [11]. Other solutions have been proposed to overcome these limitations such as the development of new techniques to improve training by working directly on the neural network graph [3, 15] or working with sparse matrices [20]. Designing specialized dense nodes for the effective use of accelerators has also been done. These nodes include: NVIDIA Tesla A100, Google TPU, or Intel GAUDI. On such nodes, training efficiency depends on model parallelization (HoroVod [24] and KARMA [28]) and effective communications between accelerators performed by a specialized network such as NVIDIA NVLink [16].

Now, despite these innovative designs, the use of increasingly deep and dense network topologies has made the resources available for training still a problem, particularly memory capacity. In light of this, techniques such as using throttle memory as an application-level cache relative to host memory have emerged during the training process [21]. This technique is very sensitive

to communication bandwidth and can incur latency due to communication and data synchronization. The use of disaggregated memory has also been considered [17], but it presents the same bottleneck as the previous one: the PCIe bus.

Other solutions include ASICs (Application-specific integrated circuit) which can be up to three times faster than general-purpose devices [8] and specialized accelerators such as DaDianNao [4] where a nearby data processing approach is adopted and has a neural functional unit that performs arithmetic operations and receives the values of the network weights from adjacent eDRAM (Embedded DRAM) memory. Finally, other types of architectures have been proposed which are memory-centric and where memory modules are added within the accelerators. This type of architecture discusses memory modules decoupled from the PCIe and parked locally within the interconnection of devices, using NVlink for example. This maximizes the communication bandwidth between them while expanding the total memory capacity of the system [14].

It should be noted that all these improvements in the hardware and the training methods of deep neural networks aim both at increasing the performance of the model and reducing the training times. Most of the research on the hardware used for training has been focused on scaling the computational capabilities and their performances [7, 26, 32]. However, few have addressed studies on the interactions of the hardware components involved in the process and especially in the memory subsystem. The latter has become the most important bottleneck for the scalability and performance of the models.

Finally, studies such as Chishti's [5] from Intel Laboratories showed in a simulation environment called DLSim the problem previously raised along with memory barriers. None of the profilers found, do a study of the problem from the point of view of communication and interactions of the components involved in model training. This is the motivation for the study presented in this paper.

2. Methodology

2.1. Model Selection and Dataset

There is now a large number of models and datasets covering fields such as image classification, object detection, speech recognition, generative adversarial nets, and deep reinforcement learning. These range from models with few hidden layers such as AlexNet [12] as well as other highly complex ones that can require huge computational capabilities for their training (Inception-v3 [27] as such as Resnet [9]). It is also important to note that many of the current datasets are large because of the increase in data available to create them. As a result, it is decided to first work with two synthetic datasets to control the total size of the dataset as well as its dimensions. On the other hand, it has been decided to create a small convolutional network model to analyze the interactions made during training with the main computational resources. In general, convolutional neural networks repeat the same process layer by layer to abstract more detailed features and finally end up with a fully connected layer to determine the class to which the image entered at the beginning of the network belongs. Therefore, the use of a denser model was ruled out, estimating that the process in each convolutional layer would have similar behavior and that memory use would increase as a function of the number of filters used among other hyperparameters and, would not present variability in the interaction of the hardware components involved. The model used for testing has been implemented in both Tensorflow and

Pytorch, its architecture is shown in Tab. 1. Finally, the AlexNet implementation in Tensorflow is shown in order to observe the variation presented in the interaction of the components when varying the model.

The two synthetic datasets were generated by simulating images of three channels each with dimensions of 32x32. The sizes selected for the datasets were 60,000 (DT1) and 600,000 (DT2), each representing an in-memory occupancy of 703 Mbytes and 6,867 Mbytes respectively.

Table 1. Convolutional Neural Network Model Architecture

	Layer (type)	**Parameters**
1	Convolutional	filters = 32, kernel_size = 3x3
2	Max Pooling	pool_size = 2
3	Convolutional	filters = 64, kernel_size = 3x3
4	Max Pooling	pool_size = 2
5	Convolutional	filters = 64, kernel_size = 3x3
6	Flatten	null
7	Dense	units = 32
8	Dense	units = 10

2.2. Frameworks Selection

Currently, there has been a constant development of both hardware and software tools that are available for the implementation of the different machine learning algorithms. In terms of software, the open-source frameworks available are Tensorflow [2], PyTorch [19], Keras [1], Torch [6], CNTK [31], Caffe [10], among others.

Among the frameworks mentioned above, none has emerged as dominant in the field, and therefore the choice of the TensorFlow and PyTorch frameworks has been made for their popularity [13], their ability to work on accelerators such as GPUs, the simplicity of implementation, and their programming similarity. Another essential reason for this choice is that the monitor created to capture the interactions between training and computational resources is designed to capture Python processes and threads in the operating system.

2.3. Tool for Characterizing the Resources Consumed

This section describes the tool developed for analysis. It is designed to capture the exchanges of the different computational resources that interact during the training process of a convolutional neural network. Profilers exist to analyze the behavior of the models, such as Tensorboard for TensorFlow and PyTorch Profiler for PyTorch, but they do not measure the actual interactions between the different components involved during the training process.

The tool relies on two libraries, psutil and pynvml. The first one gives the information of the running processes as well as the resources consumed by them in the system (CPU, Memory, Disk, Network, Sensors). The second one is a wrapper for the NVML library that monitors and manages several of the states of NVIDIA GPU devices: GPU Usage, Device Memory, PCIe Bus Interaction, among others. The monitor[9] captures and records the main computational

[9]https://github.com/alejandrotorresn/PhD/blob/master/monitor.py

components that interact during model execution and training. It is designed to capture the percentage of CPU usage, the amount of memory consumed, the percentage of GPU usage, the total GPU memory, and the traffic on the PCIe bus generated by the GPU. The latter returns two data: the first measures the data transferred and the second the data received by the device through this channel. These values are recorded for the entire duration of the process execution with an interval of approximately 0.5 seconds between samples. This interval is the maximum reduction allowed by the library used for the monitor development.

2.4. Experimental Environment

To conduct these experiments we use two servers. One with the Centos 7.9 operating system and the other with Debian 10. The first compute node is equipped with 4 Intel(R) Xeon(R) CPU X7560 at 2.27 GHz processors with 64 cores in total, 125 GB of RAM, and 2 x Nvidia GeForce GTX TITAN X cards each with 12 GB of memory. This is one of the nodes of the FELIX-denomid GUANE cluster. On the other hand, to verify the results obtained, tests are carried out on a node of the chifflot cluster in Lille belonging to Grid5000[10]. This node is equipped with two Intel(R) Xeon(R) Gold 6126 at 2.60 GHz processors with 48 cores in total, 192 GB of RAM, and one 2 x Nvidia Tesla P100 cards each with 16 GB of memory. For both architectures, we use only one card per server. The versions of the frameworks used are Tensorflow 2.2.0 and PyTorch 1.4.0. Essentially, it aims at verifying and compare resource consumption and its interaction by using two types of GPUs with different compute capabilities and to show the orchestration of resources of both frameworks in different architectures. The specifications of the GPUs used in the test can be seen in Tab. 2.

Table 2. NVIDIA GPUs specifications

	GTX TITAN X	**TESLA P100**
Engine Specs		
Architecture	Maxwell	Pascal
CUDA Cores	3072	3584
Base Clock (MHz)	1000	1189
Boost Clock (MHz)	1075	1328
Memory Specs		
Memory clock (MHz)	1752.5	715
Memory clock - effective (MHz)	7010	1430
Memory size (GB)	12	16
Memory type	GDDR5	HBM2
Memory Interface Width	384-bit	4096-bit
Memory Bandwidth (GB/sec)	336.5	732

These experimental environments will be used to measure the memory requirements necessary for the training of CNNs, its source will be detailed in the next section.

[10]https://www.grid5000.fr/w/Grid5000:Home

3. Memory Requirements and Use

As mentioned earlier, a variety of deep neural network types are specialized for a set of specific fields. The study uses the model defined in Tab. 1, a model of convolutional neural networks, which are designed for tasks such as computer vision, image recognition, or object detection. The basic design of any neural network consists of an input layer, an output layer, and multiple hidden layers. With regards to convolutional neural networks, each convolutional hidden layer is responsible for extracting certain characteristics from the input data and sending them to the next one. The first convolutional layers extract more general features while the latter gets more refined features. Finally, the convolution block output data is usually sent to fully connected layers to perform the classification task if that is the purpose of the model.

Convolutional neural networks are essentially composed of three types of layers that are responsible for extracting characteristics, reducing the output data sizes of each layer, and performing classification tasks as in the case of the simulated model in this work. The first is known as a convolutional layer and applies a group of filters to the inputs generating a feature map that is obtained from the convolution between the inputs and filters and the crossing of these values by an activation function. Normally in convolutional layers, the Rectified Linear Unit activation function (ReLU[11]) is used. This function has advantages over the preceding activation functions such as its ease of calculation and greatly accelerates the convergence of stochastic gradient descent compared to the sigmoid and Tanh functions due to its linear, non-saturating form [23].

The following layer is known as Pooling and is responsible for reducing feature map sizes using one of two widespread techniques: avg-pooling and max-pooling [22]. The process of extracting features by convolutional layers and reducing data in pooling layers is performed sequentially to the last layers that are known as Fully Connect layers. This type of neural network contains three different types of data: inputs, weights, and feature maps. It is important to note that the size of the feature map data is based on the batch parameter and the number of filters used in the layer.

Once the network models are designed, they must go through the training process to obtain the desired accuracy. This process is divided into a series of iterations where each involves a path of the model both forward and backward, known as forward and backward propagation. In forward propagation, the input data traverses the network from the first to the last layer in subgroups of the size specified by the batch parameter. At the end of this step, the outputs obtained are compared with the expected outputs (Supervised Learning), and with the back-propagation algorithm gradient maps are generated that will allow updating the weights of the network.

Within this order of ideas, memory and bandwidth needs arise due to the three types of data mentioned above: inputs, weights, and feature maps. Also, another important source is the gradient maps that are generated to update the weight. It is important to note that the feature maps obtained from forward propagation should be kept in memory until the gradient maps update the network values. This large amount of data generated by the network, along with the size of the data set, is responsible for the memory limits on some hardware architectures.

The next section will show the results obtained from the monitor. This measures the consumption of computational resources involved, as well as the times of the workloads. In partic-

[11] https://cs231n.github.io/neural-networks-1

ular, the limits reached in the memory generated by the data set, the data types generated by the model, and their training will be observed.

4. Evaluation and Results

4.1. Training Times

In this first part, the training times that both frameworks took in both architectures are shown using both datasets and various batch size. In Tab. 3 it can be observed that increasing the batch size significantly reduces training times but does not affect the behavior of resource use by frameworks, as will be seen in the next section. It is important to note that by keeping the same data set and increasing the batch size, the time differences between TensorFlow and PyTorch were significantly reduced.

Table 3. Training times (sec)

	FELIX				**Chifflot**			
	TensorFlow		**PyTorch**		**TensorFlow**		**PyTorch**	
Batch size	**DT1**	**DT2**	**DT1**	**DT2**	**DT1**	**DT2**	**DT1**	**DT2**
32	165	1591	313	3048	32	338	79	637
64	100	1026	165	1682	24	212	42	394
128	74	658	87	796	20	173	32	295
256	62	501	59	535	19	156	27	243
512	52	423	52	466	18	147	24	219

On the other hand, a significant difference can be observed in the training times of both frameworks in which small batches are used. The main reason for this, as shown in Fig. 1 and Fig. 3, is the degree of CPU and GPU usage by each of the frameworks. TensorFlow has maximized device memory usage and GPU usage, while PyTorch has prioritized host memory usage with less GPU usage, behavior that has been observed in both architectures and will be discussed in the next section.

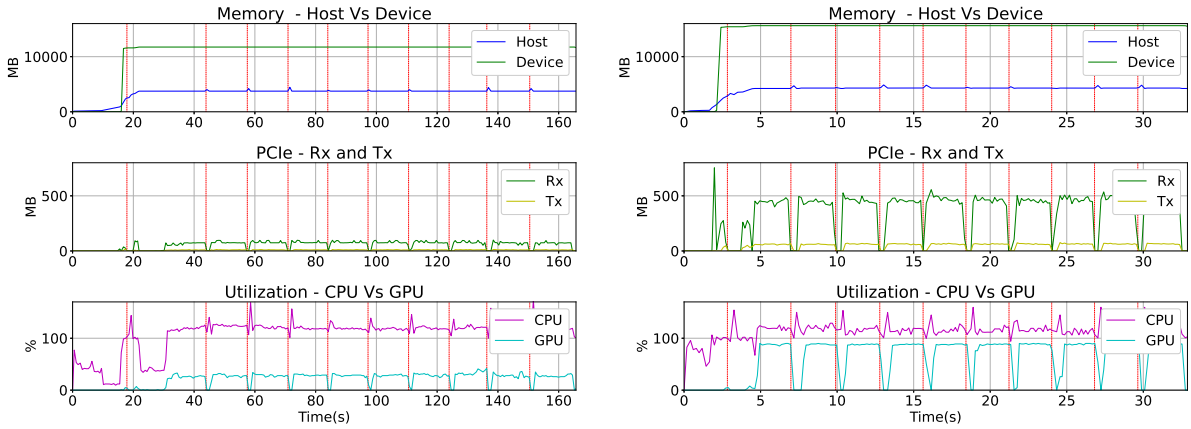
4.2. Interrelationship of Hardware Components

The experiments have been performed for different batch sizes, but only the particular results for batch size 32 are shown, since the results with the other batch sizes show similar behavior in terms of the orchestration of hardware resources measured by the monitor and allows us show the behavior studied more clearly. As mentioned in Section 2.4, two different platforms are used to carry out the experiments in order to be able to perform a contrast regarding the use of resources and their orchestration.

Figure 1 shows the results obtained during the model training process using the TensorFlow framework and a batch size of 32 for the 60,000 images dataset. The vertical red lines represent the beginning of each of the model training epochs that were set at a value of 10 for the study. The first part shows the memory consumption of the host and the device, the second part of the figure shows the communication of the device in its input channel (Rx) and its output channel (Tx) through the PCIe. Finally, the third part of the figure shows the percentage of memory consumption in both CPU and GPU. It should be noted that this third part of the graph

shows values that may exceed values of 100% due to the current configuration of the monitor developed. This monitor sums the percentage of each of the cores involved in the network training process without discriminating the amount involved, which means that the servers involved in the experiment could reach values of 4,800% or 6,400% for Chifflot and FELIX respectively.

When the datasets used in the experiment have been described, it has been commented that their sizes have been 703 MB and 6,867 MB for DT1 and DT2 respectively. These sizes are the space occupied by them in RAM memory when loaded in Python. The memory occupied by the model (inputs, weights, feature maps and gradient maps) reaches a maximum of 4,446 MB in the Host and 11,743 MB in the Device of the FELIX server and 4,880 MB in the Host and 15,621 MB on the Device of the Chifflot serve using DT1 and present little variability during the model training time as can be seen in the first part of Fig. 1. This difference between what the dataset occupies in memory and what it occupies together with the model shows that despite the small convolutional model used, there is a great impact on the memory of both the host and the device.



(a) FELIX - GUANE

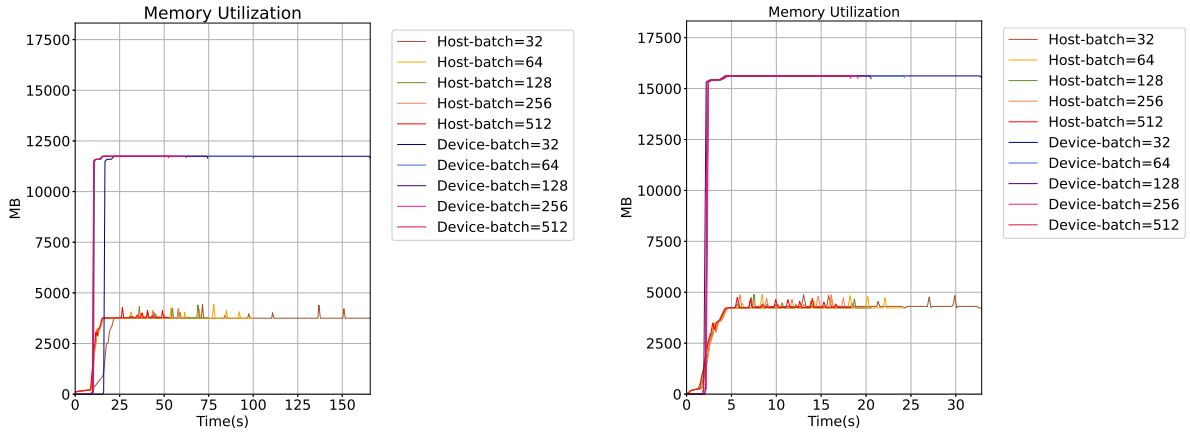
(b) Chifflot - Lille

Figure 1. Interaction of hardware components - DT1 - batch -32 - TensorFlow

The second part of Fig. 1 shows the interaction of the GPU through the PCIe bus. For both Felix and Chifflot, once the first training epoch has been initialized, a behavior with very little variability is presented with the difference that when the use of the GPU increases, the traffic in it increases in the same way. Finally, the third part of Fig. 1 shows the CPU and GPU usage during model training. The most important thing to highlight is that the change in the architecture allows the framework to make greater use of the GPU and reduce the training times of the model, but there is not a great variability in the use of resources, they are very similar during each epoch.

During the first training epoch fluctuating consumption of resources, as well as low values in the data transmitted and received by the device due to the start of the neural network by TensorFlow, are observed. Once the initial configurations are complete, the processes stabilize and resources consumption presents similar values for each epoch with very little variability during training time as mentioned before.

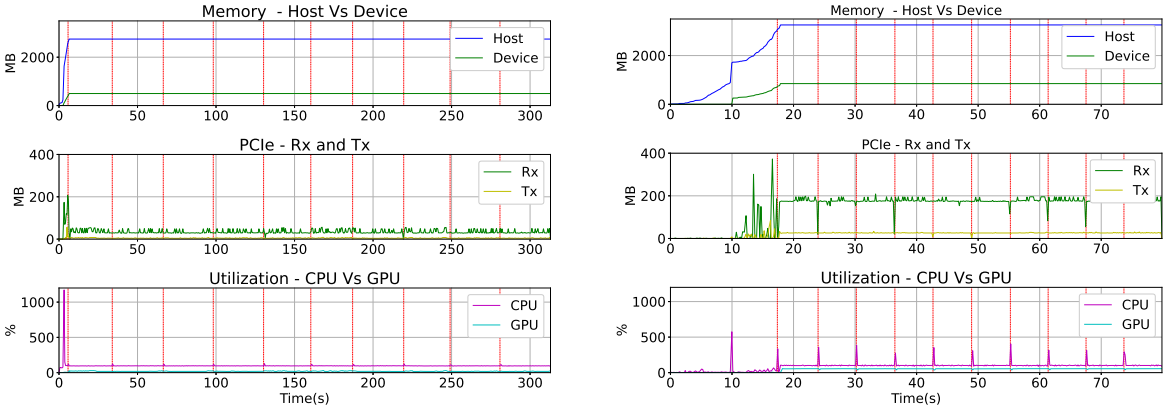
Figure 2 shows memory consumption for batches 32, 64, 128, 256 and 512 on both architectures. In addition, it is shown that for the same data set and the same neural network model, the memory consumption of the host does not present significant variations while the memory



(a) FELIX - GUANE (b) Chifflot - Lille

Figure 2. Host and device memory consumption for each of the batches set - DT1 - TensorFlow

consumption in the device presents small fluctuations that stabilize at a baseline of consumption. It should be noted that the upper part of the figure shows the memory consumption in the host while the lower part is the consumption in the device. For each batch, the duration of the training is different, and therefore, in the figure, it is observed how each corresponding line is cut.

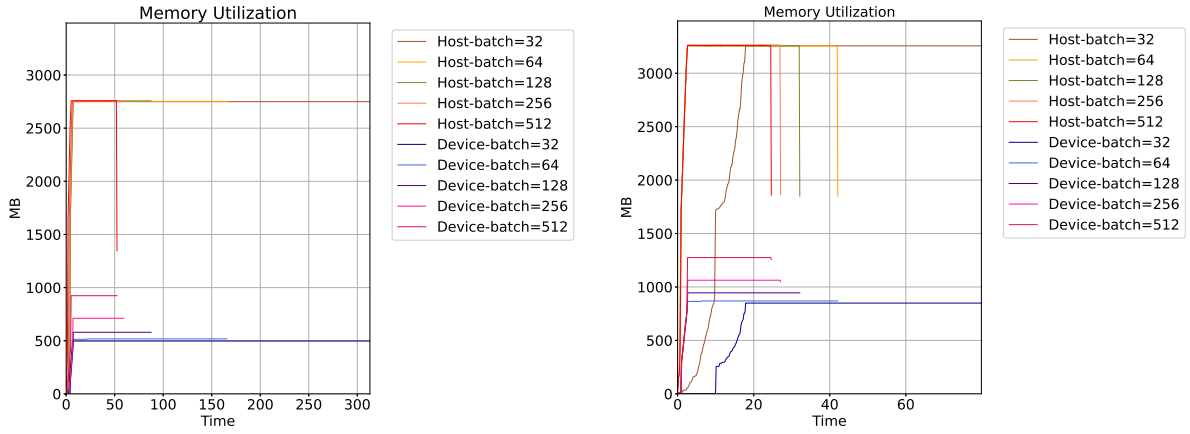


(a) FELIX - GUANE (b) Chifflot - Lille

Figure 3. Interaction of hardware components - DT1 - batch -32 - PyTorch

Figure 3 and Fig. 4 display the results for the same neural network, the same batch values as well as the same dataset using the PyTorch framework. As for PyTorch, the maximum value of memory consumption in the host has been 2,749 MB while in the device memory it has been 497 MB for FELIX and 3,256 MB in the host memory, and 849 MB in device memory in Chifflot using DT1. As with Tensorflow, there are no major variations in resource consumption after the first training period as can be seen in Fig. 1 and Fig. 3. In the same way, by increasing the use of the GPU in Chifflot, the traffic on the PCIe bus through the RX and Tx increases.

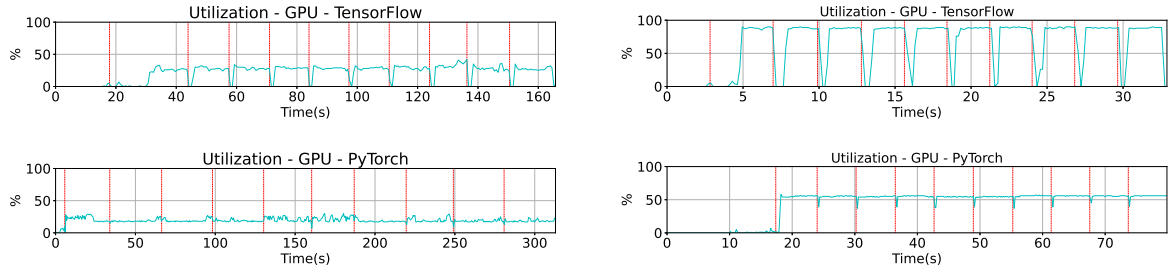
PyTorch has large peaks in CPU usage during model initialization and at the beginning of each training epoch, unlike Tensorflow, which tends to have more homogeneous behavior in the use of resources. But, the use of the device by PyTorch is less for this particular case of a



(a) FELIX - GUANE (b) Chifflot - Lille

Figure 4. Host and device memory consumption for each of the batches set - DT1 - PyTorch

dataset and a small model. This differentiation of the use of the GPU is observed in Fig. 5. The difference remains despite the change in server and therefore the architecture used in the tests.



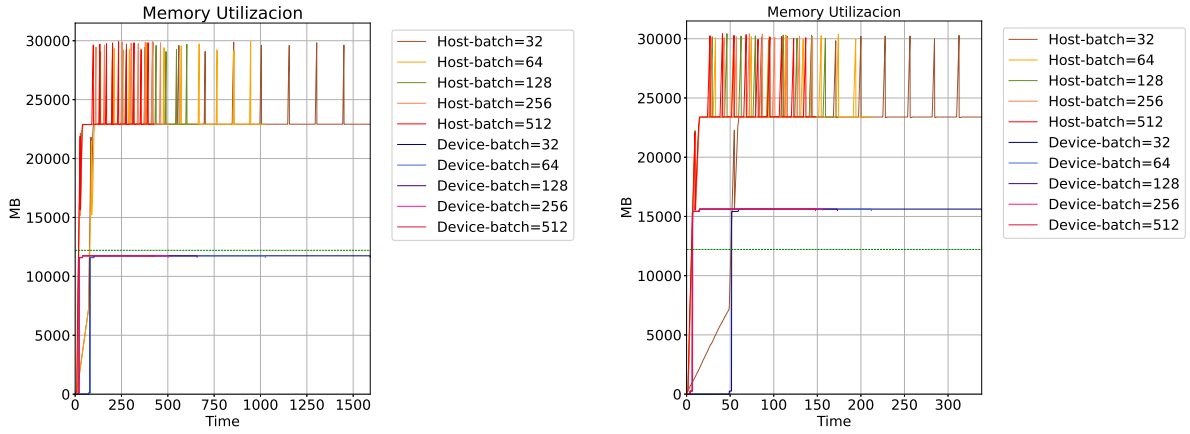
(a) FELIX - GUANE (b) Chifflot - Lille

Figure 5. GPU Utilization - DT1 - Tensorflow vs PyTorch

Figure 6 and Fig. 7 show the memory consumption during network training for the DT2 data set using the same CNN model implemented in both frameworks. The behavior in terms of memory use is similar despite the change in the dataset, increasing the memory requirement for both the host and the device as expected. Regarding the behavior of the two servers with different architectures, the training times decrease considerably with the use of more recent architecture, but the behavior of the use of the resources remains similar despite the change in the dataset and the architecture of the device for both frameworks.

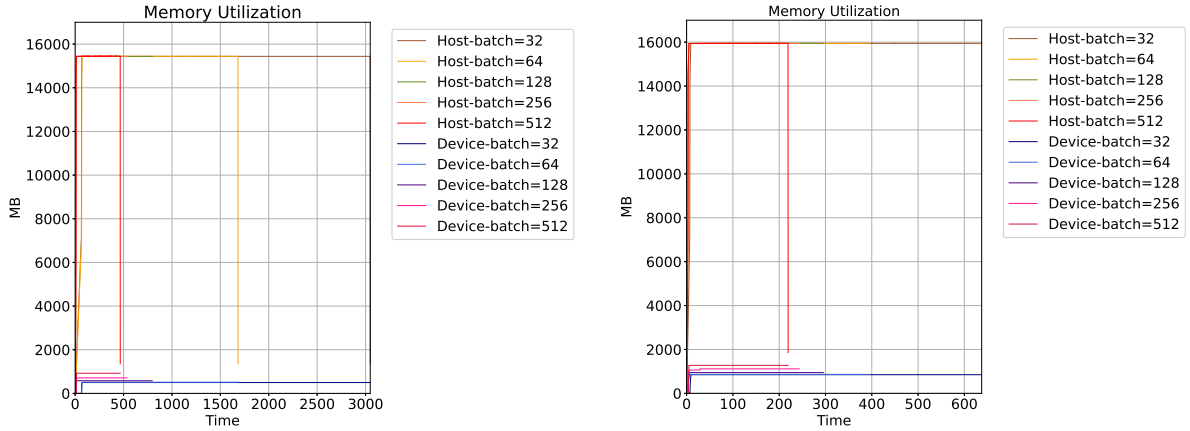
Figure 5 showed us that Tensorflow made better use of GPU for batch size 32 and that training times were improved by making an architecture change. Although the memory use behavior of both frameworks is similar, as the batch and dataset size is increased using the same model, improvements in the use of resources by PyTorch begin to show. These behaviors can be observed in Fig. 8 and Fig. 9 where the results obtained using DT2 with a batch of size 512 are shown. An increase in dataset size and batch size results in a significant increase in GPU usage by PyTorch without exceeding Tensorflow's resource handling. In the same way, the change in architecture has improved training times but hasn't shown differentiation in the orchestration of resources by the frameworks.

Finally, emphasizing memory consumption by both frameworks, it is observed that both TensorFlow and PyTorch present a maximum limit of both the host and device memory, presenting



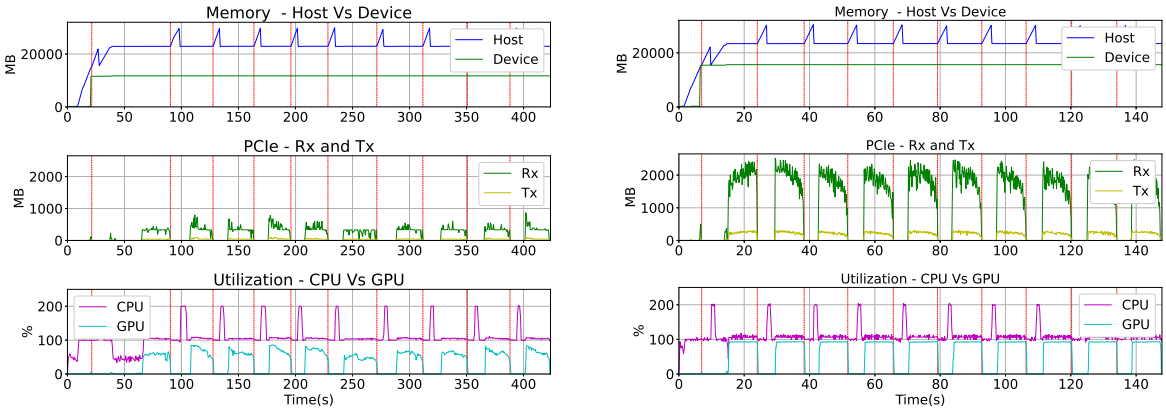
(a) FELIX - GUANE (b) Chifflot - Lille

Figure 6. Host and device memory consumption for each of the batches set - DT2 - TensorFlow



(a) FELIX - GUANE (b) Chifflot - Lille

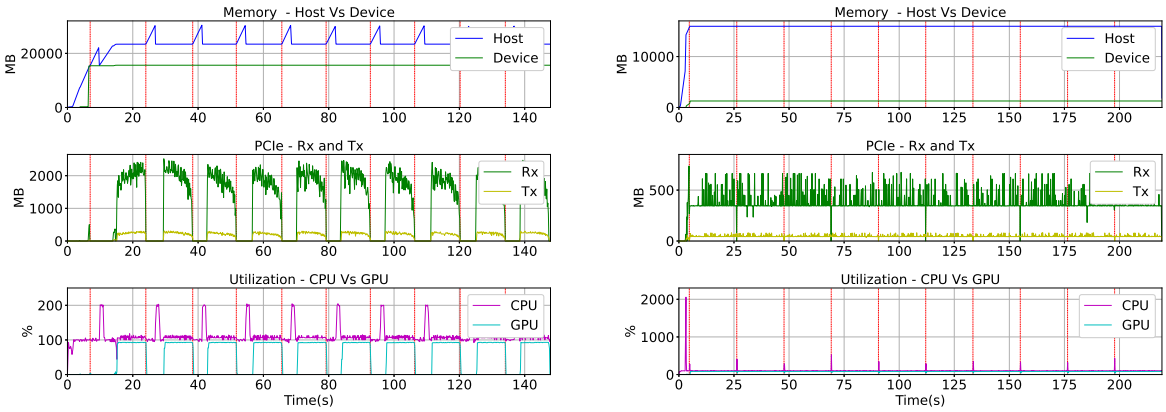
Figure 7. Host and device memory consumption for each of the batches set - DT2 - PyTorch



(a) FELIX - GUANE (b) Chifflot - Lille

Figure 8. Interaction of hardware components - DT2 - batch 512 - Tensorflow

some fluctuations at the beginning of each training epoch. PyTorch presents an interesting feature in terms of memory usage, where batch variation incurs variations of device memory usage. Table 4 shows the maximum memory consumption peaks by the host and the device for the



(a) FELIX - GUANE

(b) Chifflot - Lille

Figure 9. Interaction of hardware components - DT2 - batch 512 - PyTorch

DT1 and DT2 datasets for both frameworks and architectures with a batch size of 32. It should be noted that PyTorch maintains a better use of the memory of both the host and the device at the cost of increasing the training time of the model.

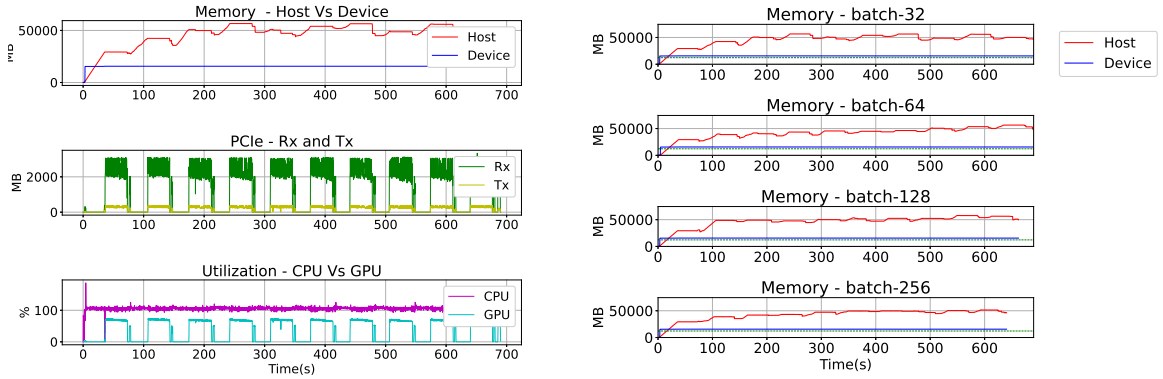
Table 4. Maximum Memory Consumption (MB) - batch-32

		FELIX		**Chifflot**	
		DT1	DT2	DT1	DT2
TensorFlow	Host	4446	29883	4880	30282
	Device	11743	11743	15621	15621
PyTorch	Host	2749	15439	3256	15945
	Device	497	497	849	849

4.3. AlexNet

This section shows the similarity of resource use and interaction by TensorFlow during AlexNet training. The training of this model is carried out exclusively on the Chifflot server using a batch size of 32 to measure the orchestration of the components and batches of size 32, 64, 128, and 256 are used for the study of both memory usage on the host as well as on the device, as shown in Fig. 10.

The results presented in this section show the behavior of memory and allowed observing the interaction of the hardware components that intervene from the beginning of the training workload. This allowes us to observe that the computational load is established during the first epoch of training and to see its little variability during the iterative process. In the following section, the results of other works will be discussed and related to those obtained during this study.



(a) Interaction of hardware components (b) Host and device memory consumption

Figure 10. Resource use by TensorFlow during AlexNet training

5. Discussion

The works presented by [5], Zhu [32] and Dai [7] show the bottleneck generated mainly by the memory subsystem when trying to train dense models with large datasets. These last two studies are mainly focused on resource consumption by model to high-level, but Chishti's work, done in a simulation environment, studies the behavior and interaction of the components involved during training. It is limited to the interaction between the processor and memory leaving aside accelerator devices such as the GPU. In order to solve or significantly reduce this problem, methods have emerged such as the optimization of the directed asynchronous graph created at the time of the execution of the model, as proposed by Le [15] and Boemer [3] or, methods of working with sparses matrices [20] to reduce memory consumption during training processes. They have also considered changing the way, memory levels are used during training as mentioned in Rhu [21] and Lim [17] works but they also present bottlenecks due to the high level of communication that occurs through the PCIe bus. In relation to the latter, new solutions are developed on specialized servers. They use a high-speed network for communication between throttled memory allowing to expand the memory capacity available in training by using multiple accelerators without using the system bus as Kwon proposes [14].

Following these ideas, most of the work done to address the problem of memory limit is using methods that reduce the memory size, avoiding the problem of the limits in communication between devices and between the same memory subsystem or the creation of new architectures specialized in the training of deep neural network models. The biggest advantage of these solutions is the implementation of sparse matrices in the training process because they can run on conventional architectures and be highly parallel.

Therefore, this work seeks to analyze the interaction between the main hardware components involved during training to look from another approach for bottlenecks that may occur in this process and then be able to address new optimization approaches. The results presented show above all the importance of finding new methods to optimize the use of the memory subsystem, host-device communications, highlighting the variability of training times when there is no effective communication with the accelerators. It is important to note that the traffic generated by the two communication lines of the device (Tx and Rx) is greater when there is greater use of the memory of the device. As a result, as the memory of the device becomes saturated, the traffic on the Rx line of the device increases, and also the traffic on the PCIe bus increases.

These latter measurements have not been studied in-depth, but they represent an interesting fact for the memory optimization used by CNN.

Conclusions

In this work, a characterization of the resources consumed and their interaction during the training of a convolutional neural network has been performed using two synthetic data sets. Batch size variability has not affected overall host or device memory consumption, and maximum rooflines have been set from the start of the training process based on dataset size and model complexity. On the other hand, training times, if they are affected by the batch size as observed in Fig. 2 and Fig. 4, the fact that they consume more host memory than the device are closely related to this variability. Finally, the monitor created to capture the resources and interaction between them has helped show the bottleneck that can become the memory subsystem. It is available for download.

Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (`https://www.grid5000.fr`) in France and, the advanced computing platforms of the High Performance and Scientific Computing Center at Universidad Industrial de Santander (SC3UIS) (`http://www.sc3.uis.edu.co`) in Colombia.

References

1. Keras: Deep Learning for humans. `https://github.com/keras-team/keras` (2015), accessed: 2020-11-10

2. Abadi, M., Barham, P., Chen, J., et al.: TensorFlow: A System for Large-Scale Machine Learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2-4 Nov. 2016, Savannah, GA, USA. pp. 265–283. USENIX Association, USA (2016), DOI: 10.5555/3026877.3026899

3. Boemer, F., Lao, Y., Cammarota, R., et al.: NGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data. In: Proceedings of the 16th ACM International Conference on Computing Frontiers, 30 April-2 May 2019, Alghero, Italy. pp. 3–13. Association for Computing Machinery, New York, NY, USA (2019), DOI: 10.1145/3310273.3323047

4. Chen, Y., Luo, T., Liu, S., et al.: DaDianNao: A Machine-Learning Supercomputer. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, 13-17 Dec. 2014, Cambridge, UK. pp. 609–622. IEEE (2014), DOI: 10.1109/MICRO.2014.58

5. Chishti, Z., Akin, B.: Memory System Characterization of Deep Learning Workloads. In: Proceedings of the International Symposium on Memory Systems, 30 Sept.-3 Oct. 2019,

Washington, District of Columbia, USA. pp. 497–505. Association for Computing Machinery, New York, NY, USA (2019), DOI: 10.1145/3357526.3357569

6. Collobert, R., Kavukcuoglu, K.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011)

7. Dai, W., Berleant, D.: Benchmarking Contemporary Deep Learning Hardware and Frameworks: A Survey of Qualitative Metrics. In: 2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI), 12-14 Dec. 2019, Los Angeles, CA, USA. IEEE (2019), DOI: 10.1109/cogmi48466.2019.00029

8. Hameed, R., Qadeer, W., Wachs, M., et al.: Understanding Sources of Inefficiency in General-Purpose Chips. SIGARCH Comput. Archit. News 38(3), 37–47 (2010), DOI: 10.1145/1816038.1815968

9. He, K., Zhang, X., Ren, S., et al.: Deep Residual Learning for Image Recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 27-30 June 2016, Las Vegas, NV, USA. pp. 770–778. IEEE (2016), DOI: 10.1109/CVPR.2016.90

10. Jia, Y., Shelhamer, E., Donahue, J., et al.: Caffe: Convolutional Architecture for Fast Feature Embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia, 3-7 Nov. 2014, Orlando, Florida, USA. pp. 675–678. Association for Computing Machinery, New York, NY, USA (2014), DOI: 10.1145/2647868.2654889

11. Jouppi, N.P., Young, C., Patil, N., et al.: In-Datacenter Performance Analysis of a Tensor Processing Unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, 24-28 June 2017, Toronto, ON, Canada. pp. 1–12. Association for Computing Machinery, New York, NY, USA (2017), DOI: 10.1145/3079856.3080246

12. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. Commun. ACM 60(6), 84–90 (2017), DOI: 10.1145/3065386

13. Kumar, V.: 5 Deep Learning Frameworks to Consider for 2020. `https://opendatascience.com/5-deep-learning-frameworks-to-consider-for-2020` (2020), accessed: 2020-11-10

14. Kwon, Y., Rhu, M.: Beyond the Memory Wall: A Case for Memory-Centric HPC System for Deep Learning. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 20-24 Oct. 2018, Fukuoka, Japan. pp. 148–161. IEEE (2018), DOI: 10.1109/MICRO.2018.00021

15. Le, T.D., Imai, H., Negishi, Y., et al.: Automatic GPU Memory Management for Large Neural Models in TensorFlow. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management, 23 June 2019, Phoenix, AZ, USA. pp. 1–13. Association for Computing Machinery (2019), DOI: 10.1145/3315573.3329984

16. Li, A., Song, S.L., Chen, J., et al.: Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. IEEE Transactions on Parallel and Distributed Systems 31(1), 94–110 (2020), DOI: 10.1109/TPDS.2019.2928289

17. Lim, K., Turner, Y., Santos, J.R., et al.: System-level implications of disaggregated memory. In: IEEE International Symposium on High-Performance Comp Architecture, 25-29 Feb. 2012, New Orleans, LA, USA. pp. 1–12. IEEE (2012), DOI: 10.1109/HPCA.2012.6168955

18. Mayer, R., Jacobsen, H.A.: Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools. ACM Comput. Surv. 53(1) (2020), DOI: 10.1145/3363554

19. Paszke, A., Gross, S., Chintala, S., Chanan, G.: PyTorch. `https://github.com/pytorch/pytorch` (2016), accessed: 2020-11-10

20. Qin, E., Samajdar, A., Kwon, H., et al.: SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), 22-26 Feb. 2020, San Diego, CA, USA. pp. 58–70. IEEE (2020), DOI: 10.1109/HPCA47549.2020.00015

21. Rhu, M., Gimelshein, N., Clemons, J., et al.: vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 15-19 Oct. 2016, Taipei, Taiwan. pp. 1–13. IEEE (2016), DOI: 10.1109/MICRO.2016.7783721

22. Saeedan, F., Weber, N., Goesele, M., Roth, S.: Detail-Preserving Pooling in Deep Networks. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 18-23 June 2018, Salt Lake City, UT, USA. pp. 9108–9116. IEEE (2018), DOI: 10.1109/CVPR.2018.00949

23. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Networks 61, 85–117 (2015), DOI: 10.1016/j.neunet.2014.09.003

24. Sergeev, l.: Horovod. `https://github.com/horovod/horovod` (2017), accessed: 2020-11-10

25. Shatnawi, A., Al-Bdour, G., Al-Qurran, R., et al.: A comparative study of open source deep learning frameworks. In: 2018 9th International Conference on Information and Communication Systems (ICICS), 3-5 April 2018, Irbid, Jordan. pp. 72–77. IEEE (2018), DOI: 10.1109/IACS.2018.8355444

26. Simmons, C., Holliday, M.A.: A Comparison of Two Popular Machine Learning Frameworks. J. Comput. Sci. Coll. 35(4), 20–25 (2019), DOI: 10.5555/3381631.3381635

27. Szegedy, C., Vanhoucke, V., Ioffe, S., et al.: Rethinking the Inception Architecture for Computer Vision. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 27-30 June 2016, Las Vegas, NV, USA. pp. 2818–2826. IEEE (2016), DOI: 10.1109/CVPR.2016.308

28. Wahib, M., Zhang, H., Nguyen, T.T., et al.: Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 9-19 Nov. 2020, Atlanta, Georgia. IEEE Press (2020), DOI: 10.5555/3433701.3433726

29. Wang, Y., Yang, C., Farrell, S., et al.: Time-Based Roofline for Deep Learning Performance Analysis. In: 2020 IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS), 11 Nov. 2020, Atlanta, GA, USA. pp. 10–19. IEEE (2020), DOI: 10.1109/DLS51937.2020.00007

30. Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM 52(4), 65–76 (2009), DOI: 10.1145/1498765.1498785

31. Yu, D., Eversole, A., Seltzer, M., et al.: An Introduction to Computational Networks and the Computational Network Toolkit (2014), https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/, accessed: 2020-11-10

32. Zhu, H., Akrout, M., Zheng, B., et al.: Benchmarking and Analyzing Deep Neural Network Training. In: 2018 IEEE International Symposium on Workload Characterization (IISWC), 30 Sept.-2 Oct. 2018, Raleigh, NC, USA. pp. 88–100. IEEE (2018), DOI: 10.1109/IISWC.2018.8573476