




Using machine learning to model the training scalability of convolutional neural networks on clusters of GPUs

Sergio Barrachina¹ · Adrián Castelló¹ · Mar Catalán¹ · Manuel F. Dolz¹  · Jose I. Mestre¹

Received: 30 April 2021 / Accepted: 4 August 2021
© The Author(s) 2021

Abstract

In this work, we build a general piece-wise model to analyze data-parallel (DP) training costs of convolutional neural networks (CNNs) on clusters of GPUs. This general model is based on *i*) multi-layer perceptrons (MLPs) in charge of modeling the NVIDIA cuDNN/cuBLAS library kernels involved in the training of some of the state-of-the-art CNNs; and *ii*) an analytical model in charge of modeling the NVIDIA NCCL Allreduce collective primitive using the Ring algorithm. The CNN training scalability study performed using this model in combination with the Roofline technique on varying batch sizes, node (floating-point) arithmetic performance, node memory bandwidth, network link bandwidth, and cluster dimension unveil some crucial bottlenecks at both GPU and cluster level. To provide evidence of this analysis, we validate the accuracy of the proposed model against a Python library for distributed deep learning training.

Keywords Deep neural networks (DNNs) · Distributed training · Multi-layer perceptron (MLP) based modeling · Analytical modeling · Clusters · GPUs

Mathematics Subject Classification 65Y20 · 68M20 · 68T07

✉ Manuel F. Dolz
dolzm@uji.es

Sergio Barrachina
barrachi@uji.es

Adrián Castelló
adcastel@uji.es

Mar Catalán
catalama@uji.es

Jose I. Mestre
jmiravet@uji.es

¹ Universitat Jaume I, Castellón, Spain

1 Introduction

The deep learning hype we are experiencing in the last years is a consequence of new mathematical and algorithmic advances, the availability of huge amounts of training data, the deployment of powerful specialized computer hardware, and the development of user-friendly yet powerful Deep Neural Networks (DNN) training frameworks [18,21]. Roughly speaking, for supervised training, a DNN is a generic algorithm that semi-automatically “learns” from a number of problem-solution pairs in order to adapt itself to produce a solution for future instances of that particular problem class. This adaptation occurs via an off-line training procedure, especially expensive for complex DNNs, which often requires the use of large-scale computing platforms [4].

The simplest scheme for extracting parallelism from a batched training process on a cluster is to replicate the model in all nodes and to distribute the computations across the “batch” dimension among the nodes. Most parallel DNN frameworks leverage this approach, referred to as data parallelism (DP), to accelerate the training process, due to its simplicity and superior scalability [24]. Alternatively, model parallelism (MP) becomes mandatory when the model is so large that it does not fit into each node memory. In [5] and [7], we independently analyzed the asymptotic performance of distributed training based on DP and MP, respectively; while in [6] we performed a full comparison between both parallel models. The strategy used to model the computational kernels in those studies was based on the Roofline model [23], while the communication costs were estimated using analytical models from the state-of-the-art [12].

In this paper, we extend our previous works with a refined piece-wise model consisting of a collection of simple yet accurate multilayer perceptrons (MLPs) in charge of capturing the behaviour of the individual computational kernels appearing in the DP training of CNNs on clusters of GPUs. Whereas, the collective communications appearing in the general model for DP training are modeled via analytical models. In particular, our work makes the following contributions:

- We validate our MLPs and analytical models using PyDTNN,¹ a Python library for distributed deep learning (DL) that covers DL training for the most common DNN models: MLPs, CNNs, and residual networks (ResNets). PyDTNN exploits DP [4], relying on specialized message-passing libraries for communication and on kernels from high-performance multi-threaded libraries for computation. For clusters of NVIDIA GPUs, PyDTNN leverages cuDNN and cuBLAS to realize the major computational kernels appearing in the DNNs layers and NCCL for the collective operations. In comparison, in [6] we only validated the major computational kernels using the Roofline-based models for distributed DP training on clusters of multi-core CPUs.
- As a result of the validation process against PyDTNN, we have refined our performance models to more accurately reproduce the actual computations and data communication exchanges appearing in the distributed DP training. Specifically, the next model aspects have been improved:

¹ The PyDTNN framework is available at <https://github.com/hpca-uji/PyDTNN/>, under a GNU General Public License v3.0.

- The Roofline-based analytical models have been extended with a collection of regression-based MLPs that produce more accurate estimations of the execution time for the NVIDIA cuDNN/cuBLAS numerical kernels, the NVIDIA CUDA (host to device) memory copies, and the optimizing functions used on the training of CNNs.
- The MLP models have been tailored to two state-of-the-art GPUs: NVIDIA Tesla A100 and V100S.
- An analytical model has been designed to mimic the behavior of the Allreduce collective communication primitive from the NVIDIA NCCL library required in DP training. This model has been validated on a small-scale cluster with nodes equipped with NVIDIA V100 GPUs, interconnected via an Infiniband EDR network.
In summary, the new MLP-based models are considerably more realistic and, as our experiments demonstrate, provide accurate estimates of the execution costs of a real distributed framework for DL on clusters of GPUs.
- As a final and major contribution of this work, we perform an extensive analysis of the DP strategy on three CNN models and two datasets, as a function of five parameters: batch size, node arithmetic performance, node memory bandwidth, network link bandwidth, and cluster size.

This paper is organized as follows. In Sect. 2, we offer a short review of distributed training for supervised DL. Next, in Sect. 3 we present and validate the MLP-based models for the computational kernels involved in a training step as well as the analytical model for the NCCL Allreduce communication primitive. In Sect. 4, we evaluate in detail the performance of three representative CNNs combined with two datasets. Finally, in Sects. 5 and 6, respectively, we discuss a number of related works and close this paper with a few remarks.

2 Distributed training of CNNs

2.1 Overview of training

A neural network (NN) can be viewed as a nonlinear function that performs a mapping between its inputs and outputs. A NN is organized into L layers, each one consisting of n_l neurons and contributing toward the output with a particular intermediate computation.

The objective of the training process is then to learn from the training data, adapting the NN model in order to minimize the deviation between the outputs computed by the NN and the expected outputs (ground truth). Provided the training data is representative, the trained model can then be used to produce accurate responses when operating with “unseen” (new) data. The adaptation of the NN is usually performed via an optimization algorithm such as the *stochastic gradient descent* (SGD) method; see, e.g., [13].

2.2 High-performance batched training

Advanced realizations of the training process perform the forward-backward passes of SGD for batches of b simultaneous inputs. This formulation converts the memory-bound kernels of the single-input case into compute-bound operations in the batched counterpart [21].

2.2.1 High performance GEMM

Fully-connected (FC) layers in (multilayer perceptrons, or MLPs, and) CNNs can be simply cast in terms of a general matrix multiplication (GEMM). This computation can be performed by invoking a multi-threaded highly tuned instance of BLAS, such as that in Intel MKL, OpenBLAS, GotoBLAS2, and BLIS for multicore processors. For GPUs from NVIDIA, it is possible to rely on the cuBLAS library to perform the corresponding GEMM.

Similarly, convolutional (CONV) layers in CNNs can also be mapped into a GEMM through a transformation of the input operand I via the IM2COL operation [9], which generates an augmented version of I (re-organizing and partially replicating its elements). In NVIDIA GPUs, the CONV layers can be efficiently processed using cuDNN, a library of primitives for DNNs that provides highly tuned implementations of the most frequent DNN kernels. This library includes support for forward and backward convolutions, including cross-correlation, using two primary methods: GEMM-based and transform-based. The former approach provides three different variants: *i*) the GEMM variant constructs the augmented IM2COL-related matrix required for the GEMM; *ii*) the IMPLICIT GEMM variant avoids explicitly forming the whole augmented matrix; and *iii*) the IMPLICIT PRECOMPUTE GEMM is similar to *i*) but precomputes some indices that facilitate the construction of the augmented matrix.

2.3 Distributed data-parallel training

During the training process, the activations are propagated between adjacent layers, from “left” to “right” of the DNN, for the FP stage. Afterwards, in the backward pass (BP), the gradients are computed and the weights are updated (WU) following the opposite direction. In both cases, there exist strict inter-layer data dependencies between consecutive layers, which impedes an inter-layer parallelization of the training process.

In the DP scheme, the GEMM kernels are parallelized by partitioning the problem data (and distributing the workload) across the batch size b among the P “processes”, which may correspond, e.g., to the number of GPU accelerators in a cluster platform. Since the number of samples that are usually employed to train a NN is quite high, provided some algorithmic parameters which could affect the convergence of the training process are conveniently adjusted, it is possible to increase the batch size b proportionally to P up to a certain dimension [25]. Also, the weights/filter tensors are replicated so that each process maintains a local copy while the input batch tensor involved in a training step is distributed among the P processes, which will thus process

only b/P samples of the batch. Therefore, in DP there is no need for inter-process communications during the FP and BP computations. In contrast, for the WU stage, an Allreduce [8] is required to aggregate the partially computed gradients related to the weights/filters in each layer across all P processes prior to updating the weights.

In clusters of NVIDIA GPUs, the Allreduce operation can be realized using any of the high-performance MPI libraries, such as MPICH, MVAPICH2, OpenMPI, etc. CUDA supports Remote Direct Memory Access, which allows buffers to be directly transferred from the GPU memory to a network adapter without staging through host memory [1]. Alternatively, the NVIDIA Collective Communication Library (NCCL) implements highly-efficient communication primitives for NVIDIA GPUs on PCIe and NVLink interconnects within a node as well as over NVIDIA Mellanox networks across nodes [17].

3 Performance modeling of CNNs

In this section, we detail the methodology employed to build the performance piecewise models based on MLP and analytical models that account for the training costs of CNNs using the DP training scheme on clusters of GPUs.

3.1 Modeling the computations using MLPs

Training CNNs involves the processing of the FP and BP for all the batches in the dataset during a given number of epochs. Given that the number of training epochs depends on the desired final accuracy, which in turn depends on the combination of the CNN model and the training dataset being used, in this work we focus on modeling the performance of a single step (FP+BP+WU) of the target CNN model.

In a previous work [6], we modeled the CNN computational training costs as the sum of the GEMM and IM2COL transformations involved in each FC and CONV layer of a CNN using the so-called Roofline model. Recall that this model receives, as inputs, the kernel arithmetic intensity in flops/byte, the peak performance, and the memory bandwidth of the processor to perform an estimation of the upper bound of flops/s that can be attained in a given processing unit. However, tuning the Roofline model to perform accurate predictions of a specific kernel is not straightforward and presents the following drawbacks: *i*) a deep knowledge of the memory hierarchy of the processor is necessary to set the right memory bandwidth; *ii*) for the case of GPUs, it is difficult to account for the different capabilities of recent accelerators (e.g., streaming multiprocessors and tensor cores); and *iii*) the model cannot fully capture the internal algorithmic implementation details.

Although the Roofline model is still a valid solution to calculate the general asymptotic costs, in this work we extend our basic approach in [6] to accurately model each of the cuDNN/cuBLAS kernels involved in the CNN training on GPUs using regression-based MLP models. Apart from that, we also use MLPs to account for the costs related

Table 1 Specification of the regression-based MLPs used for modeling each of the cuDNN/cuBLAS kernels in Table 2. The number of input neurons in layer 0 corresponds to cardinality of the tuple *Inputs* appearing in the last column of Table 2

Id.	Layer type	#Neurons
0	INPUT	#Inputs
1–10	FC RELU } $\times 5$	50
11	FC	1

to the SGD optimizing function² and the per-training-step batch (host to device) memory copy. The main advantage of this approach is that the MLP models are capable of learning the complex nonlinear behaviors inherent to the execution of these kernels on different input operand sizes and related parameters, abstracting them from the processor/accelerator parameters. With such models, the behavior of these kernels can be easily learned by each MLP from a collection of previous executions performed on the selected GPUs in order to achieve accurate execution time predictions. However, we also recognize the limitations of the MLPs as, once trained, they can only perform execution time estimations for a concrete GPU model, being the FLOPS and the memory bandwidth fixed parameters. For these reasons, the scalability study performed later in this paper stills uses the Roofline model when varying the floating-point and memory bandwidth performance.

3.1.1 Network design

We first designed a simple regression-based MLP architecture comprised of an input FC layer followed by $5 \times$ FC+RELU blocks of 50 neurons each, and a final FC layer consisting of a single neuron in charge of performing the execution time prediction; see Table 1. The inputs of the MLP models correspond to the sizes of the input operands and parameters of the modeled kernel. For instance, the MLP inputs corresponding to the `cublasSgemm` kernel responsible for realizing the FP in a FC layer l , $O^{(l)} = I^{(l)} \cdot W^{(l)}$, are b , n_{l-1} , and n_l , where b denotes the number of rows of the input/output matrices $I^{(l)}$ and $O^{(l)}$; n_{l-1} the number of columns of $I^{(l)}$ and rows of the weights matrix $W^{(l)}$; and n_l stands for the number of columns of $W^{(l)}$. Table 2 lists the cuDNN/cuBLAS kernels that have been modeled via a MLP for each type of layer and stage. The table also includes the MLPs for the SGD optimizing function as well as for the `cudaMemcpy` CUDA kernel.

3.1.2 Obtaining the datasets

To generate the necessary data to train the proposed MLP models, we developed a series of micro-benchmarks for the kernels in Table 2, using all the combinations of the operand sizes and input parameters specified in Table 3. Each micro-benchmark

² For the SGD optimizer we used the PyCUDA kernel implementation from our PyDTNN training framework, as optimizing functions are not available in cuDNN.

Table 2 Trained MLPs for each of the cuDNN, cuBLAS, SGD, Copy kernels and algorithms/modes appearing in a training step the selected CNNs. For simplicity we assume that the vertical and horizontal strides ($s_y = s_y^v = s_y^h$) and paddings ($p_l = p_l^v = p_l^h$) are equal

Layer/Kernel	Stage	Modeled kernel	Algorithms/Modes	Inputs
CONV	FP	cudaConvolutionForward	AUTO, GEMM, IMPL. GEMM, IMPL. PRECOMP. GEMM	$\{b, c_{l-1}, h_{l-1}, w_{l-1}, c_l, k_l^h, k_l^w, h_l, w_l, s_l, p_l\}$
	BP	cudaConvolutionBackwardFilter		
	BP	cudaConvolutionBackwardData	AUTO, ALGO 0, ALGO 1	
	FP	cudaAddTensor (add biases)	AUTO, ALGO 0, ALGO 1	$\{b, c_l, h_l, w_l\}$
	BP	cudaConvolutionBackwardBias	-	
	FP	cublasSgemv	-	$\{b, n_{l-1}, n_l\}$
FC	BP	cublasSgemm (weights+data gradients)	-	$\{b, n_l\}$
	FP	cudaAddTensor (add biases)	-	
	BP	cublasSgemv (biases gradient)	-	
BATCHNORM	FP	cudaBatchNormalizationForwardTraining	SPATIAL	$\{b, c_l, h_l, w_l\}$
	BP	cudaBatchNormalizationBackward		
MAXPOOL	FP	cudaPoolingForward	POOLING MAX	$\{b, c_l, h_l, w_l, k_l^h, k_l^w, s_l, p_l\}$
	BP	cudaPoolingBackward		
AVGPOOL	FP	cudaPoolingForward	POOLING AVERAGE COUNT	$\{b, c_l, h_l, w_l, k_l^h, k_l^w, s_l, p_l\}$
	BP	cudaPoolingBackward		
RELU	FP	cudaActivationForward	ACTIVATION RELU	$\{b, c_l, h_l, w_l\}$
	BP	cudaActivationBackward		
SOFTMAX	FP	cudaSoftmaxForward	MODE INSTANCE, ACCURATE	$\{b, c_l, h_l, w_l\}$
	BP	cudaSoftmaxBackward		
SGD	WU	pydtnnSgdKernel	-	$\{b, c_l, h_l, w_l\}, \{b, n_{l-1}, n_l\}$
COPY	FP	cudaMemcpy	-	$\{b, c_l, h_l, w_l\}$

Table 3 Ranges of input sizes operands selected for the generation of the training/testing dataset of the MLP models

Description	Parameter	Range of values
#Inputs, #outputs of layer l	n_{l-1}, n_l	$\{128, 256, 512, \dots, 1280 \times 10^3\}$
#Channels of layer l	c_{l-1}	$\{3, 4, 8, 16, 32, 64, 128, 256, 512\}$
Input/Output height of layer l	h_{l-1}, w_{l-1}	$\{8, 14, 16, 28, 32, 56, 112, 224\}$
Filter width/height of layer l	k_l^h, k_l^v	$\{1, 3, 5, 7, 9\}$
Vertical/Horizontal stride of layer l	s_l^v, s_l^h	$\{1, 2, 4\}$
Vertical/Horizontal padding of layer l	p_l^v, p_l^h	$\{1, 3, 5, 7, 9\}$
Batch size	b	$\{16, 32, 64, 128, 256\}$

generates the dataset for a single MLP, the samples being the tuples that contain the input operand sizes/parameters and their corresponding measured execution time. To tackle the system noise, we executed each kernel 100 times and reported the average execution time. The datasets associated with each kernel were obtained on two NVIDIA GPUs: a Tesla A100 and a Tesla V100 PCIe with cuDNN v8.0.4 and cuBLAS v11.3.0.106. The server used to execute these experiments comprises $2 \times$ Intel Xeon 6126 processors (24 cores in total at 2.60 GHz), 64 GiB of DDR4 RAM, and the two afore-mentioned NVIDIA Tesla GPUs.

3.1.3 Training the MLP models

Once the datasets are obtained, the models can be trained using the Tensorflow v2.2.0 framework. To permit the MLPs learning, we normalize the inputs and outputs using the log function to narrow the range of values of the operand sizes/parameters and the measured execution times. Next, we partition the datasets, leaving 80% for training and the remaining 20% for testing. Analogously, 80% of the training dataset is utilized only for training, while the remaining 20% is used for validation, to prevent overfitting and to guide the training process. All the MLP models are trained using the Adam optimizer on the Minimum Square Error (MSE) loss function. In this process, the initial learning rate was set to 10^{-3} and multiplied by 0.1 each time that the validation loss did not improve for 15 consecutive epochs. Additionally, the training stopping criteria terminated the training process when the validation MSE did not improve after 20 epochs.

Once the models are trained, the next step is to evaluate them using the testing dataset. For that purpose, we use the relative error (RE) as the metric to account for the differences between the predicted and measured kernel execution times. Concretely, the RE for the run time of a kernel k is defined as:

$$RE_k = \frac{T_k^{estimated} - T_k^{measured}}{T_k^{measured}}.$$

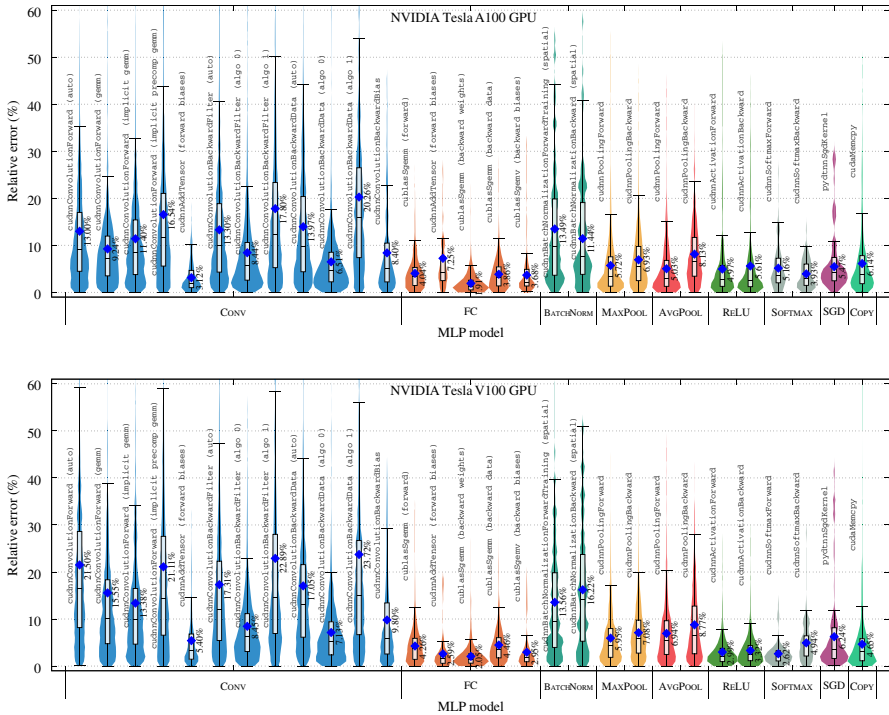


Fig. 1 Histogram and box-plots for the relative estimation errors obtained by the MLP models for the cuDNN and cuBLAS libraries as well as the SGD optimizing function and memory copies on the testing datasets and the NVIDIA Tesla A100/V100 GPUs

Figure 1 displays the histograms, boxplots, and average REs (blue diamonds) obtained for the MLP modeling the cuDNN/cuBLAS kernels on the testing dataset for both GPUs, A100 and V100. In general, we observe that the prediction error for CONV layers is around 15%, with the GEMM and IMPLICIT GEMM algorithmic variants for FP and ALGO 0 for BP, providing very accurate estimations. The RE for FC layers is in general around 5%, similar to that incurred for the MAXPOOL/AVGPPOOL, RELU, and SOFTMAX layers. The RE for BATCHNORM layers is, in general, around 14%. Finally, the RE for the SGD and the cudaMemcopy was on average, 5% and 7% for both GPU models, respectively. From these results, we can conclude that the trained MLPs provide fairly good estimations, and therefore they can be used as part of the global piece-wise model to predict the cost of a CNN training step.

3.2 Modeling the communications

To complete our piece-wise model, we need to estimate the communication exchanges appearing in CNNs trainable layers (CONV, FC, and BATCHNORM). As previously mentioned, to estimate the communication costs we leverage an analytical model to predict Allreduce exchanges performed by the NCCL implementation among the

GPUs.³ From the cluster perspective, we consider a single MPI rank mapped onto a GPU and an interconnection network among the P nodes with a star topology.⁴ In such a realization, a single link connects each node to a central switch that provides simultaneous full bandwidth between any two nodes.

NCCL supports different algorithmic schemes to perform the Allreduce collective, with a throughput that depends on the message size n , the number of GPUs, and the network topology. Nonetheless, we focus on the RING algorithm [22], which is competitive for large messages, as those which typically arise when training CNNs [17]. Following the implementation in NCCL, the Allreduce operation realized on P GPUs via the RING scheme has a cost of:

$$2(P-1)\alpha + 2\frac{P-1}{P}n\beta + \frac{P-1}{P}\frac{n}{\delta}\gamma, \quad (1)$$

where α and β respectively represent the link latency and bandwidth, γ corresponds to the peak arithmetic performance, and n stands for the message size.

3.3 Building the general model

To build our general model, we leverage the MLP models to account for the computational costs of the layers and the analytical model to estimate the communication exchange run times according to the modelled CNN architecture. For that, we developed a simulator mimicking the FP and BP stages, which is responsible to call the individual MLP/analytical models according to: *i*) the CNN structure, as a collection of layer parameters; *ii*) the architectural cluster parameters, such as the number of nodes/GPUs (processes), the GPU type, and the network configuration; and *iii*) the algorithmic parameters, i.e., the batch size and the computer floating-point format (specifically, the number of bytes per floating-point element), e.g. FP32. The complete list of parameters that can be adjusted in our general model is displayed in Table 4.

3.4 Validation of the performance model

Before applying our general performance models to analyze the scalability of the distributed DP scheme, we first assess the accuracy of MLP computation models and the Allreduce model. For this purpose, we contrast the cost estimations obtained from the individual models with the actual execution times measured for PyDTNN, a Python library for distributed CNN training on clusters of CPUs and GPUs that is competitive with current state-of-the-art frameworks, such as TensorFlow [2,3].⁵

³ We did not consider an MLP model for estimating the Allreduce communication costs as we find it time-consuming to generate a training dataset including a wide range of input real parameters for link bandwidth, latencies and number of nodes. Without this dataset, we cannot expect an MLP to perform accurate estimations.

⁴ In this work, for simplicity, we consider nodes equipped with a single GPU only.

⁵ We used PyDTNN to validate the experiments as this framework can be easily configured to leverage a data-parallel scheme using synchronous communications. Other frameworks, such as TensorFlow+Horovod,

Table 4 Parameters of the performance model

(a) CNN model	
l	#Layers id
T	Layer type
n_{l-1}, n_l	#Inputs, #outputs of layer l
h_{l-1}, h_l	Input/Output height of layer l
w_{l-1}, w_l	Input/Output width of layer l
k_l^v, k_l^h	Filter width/height of layer l
s_l^v, s_l^h	Vertical/Horizontal stride of layer l
p_l^v, p_l^h	Vertical/Horizontal padding of layer l
c_l	#Channels of layer l
b	Batch size
δ	Bytes per floating-point number
(b) Cluster	
A	Processor/accelerator type
P	#Nodes in the cluster
α	Link latency (in s)
β	Link bandwidth (in bits/s)

3.4.1 Hardware setup and calibration

The validation of the performance models was carried out on our ALTEC cluster. This platform consists of 8 nodes, each equipped with an Intel Xeon Gold 5120 processor (14 cores with a nominal frequency of 2.20 GHz), 187 GiB of DDR4 RAM, and an NVIDIA Tesla V100 PCIe with 32 GiB of HBM2. The nodes are interconnected via an Infiniband EDR network with a bandwidth of 100 Gbps. Regarding the software layer, we use Intel Python 3.7.4 to run PyDTNN on top of cuDNN v8.0.4 and cuBLAS v11.3.0.106. The communication layer is provided by NCCL v2.8.3 configured to use the Infiniband network. It is important to remark that, since we do not have access to a cluster equipped with A100 GPUs, we can only carry out the validation using the ALTEC cluster, consisting of V100 GPUs. In any case, considering that the relative estimation errors for A100 GPU are slightly smaller than those for the V100 GPU (as shown in Fig. 1), we can expect that a validation carried out on a A100 GPU would lead to similar prediction errors.

Table 5 displays the main characteristics of the cluster configuration. (Besides, it also shows the parameters of an hypothetical cluster, AMPERE, used for the performance analysis described in the next subsection.) For the validation, we adapted the models for IEEE 32-bit arithmetic by considering δ (bytes per floating-point number) and γ (theoretical peak performance, in FLOPS) to match the baseline 32-bit floating-point datatype used by PyDTNN.

also exploit data parallelism, but use auxiliary communication threads that aim to overlap computation with communications, which makes more difficult the modeling task.

Table 5 Cluster architectures employed in the validation and analysis

Parameters	ALTEC	AMPERE
#Cluster nodes	8	100
CPU model (Intel Xeon)	5120	8180M
Frequency (GHz)	2.2	2.5
#Cores	14	28
RAM memory (GBytes)	187	256
GPU model (NVIDIA Tesla)	V100	A100
HBM2 memory (GBytes)	32	40
GPU tensor core peak FP32/TF32 (TFLOPS)	133	156
GPU memory bandwidth (GBytes/s)	900	1555
Network (Infiniband)	EDR	HDR
Link bandwidth (Gbps)	200	400
Maximum link latency (μ s)	0.5	0.5

We calibrated the ALTEC cluster network parameters α and β (see Table 4) via the NCCL test benchmark, which reported a practical link bandwidth of $\beta = 12.24$ Gbps and a link latency of $\alpha = 30 \mu$ s [16].

3.4.2 Validation

The general model validation on PyDTNN was performed with one MPI rank (process) per cluster node, each mapped to a V100 GPU of the ALTEC cluster. For that, we obtained a full profile of the cuBLAS, cuDNN, memory copies, and SGD kernels realized in a training step of the VGG11 network [20]. Figure 2 shows the actual (i.e., measured) execution time per layer of VGG11 executed using a batch size of $b = 128$, along with the corresponding estimations obtained with MLP models for each of the kernels. The plots also offer the RE of the time predictions per kernel, and an alternative “weighted” relative error, WRE , that is explained below. In general, we observe that the relative errors for most kernels are moderate, remaining in the range $[-20\%, +20\%]$, with a good fraction of those being considerably lower, around $\pm 5\%$. The relative error is large for a few kernels only (related to FC and RELU layers at the end of the VGG11 in the FP, some of the MAXPOOL layers in the BP, and the FC in the WU stage). Nevertheless, the contribution of these to the total execution time is small. As a result, the effect of these deviations on the total error remain small, around $\pm 5\%$ only.

The low relative error is partially due to cancellations between underestimations and overestimations. In order to obtain a more representative metric, we have calculated the error with respect to the average execution time, as:

$$WRE_k = \frac{T_k^{estimated} - T_k^{measured}}{T_{average}}, \quad (2)$$

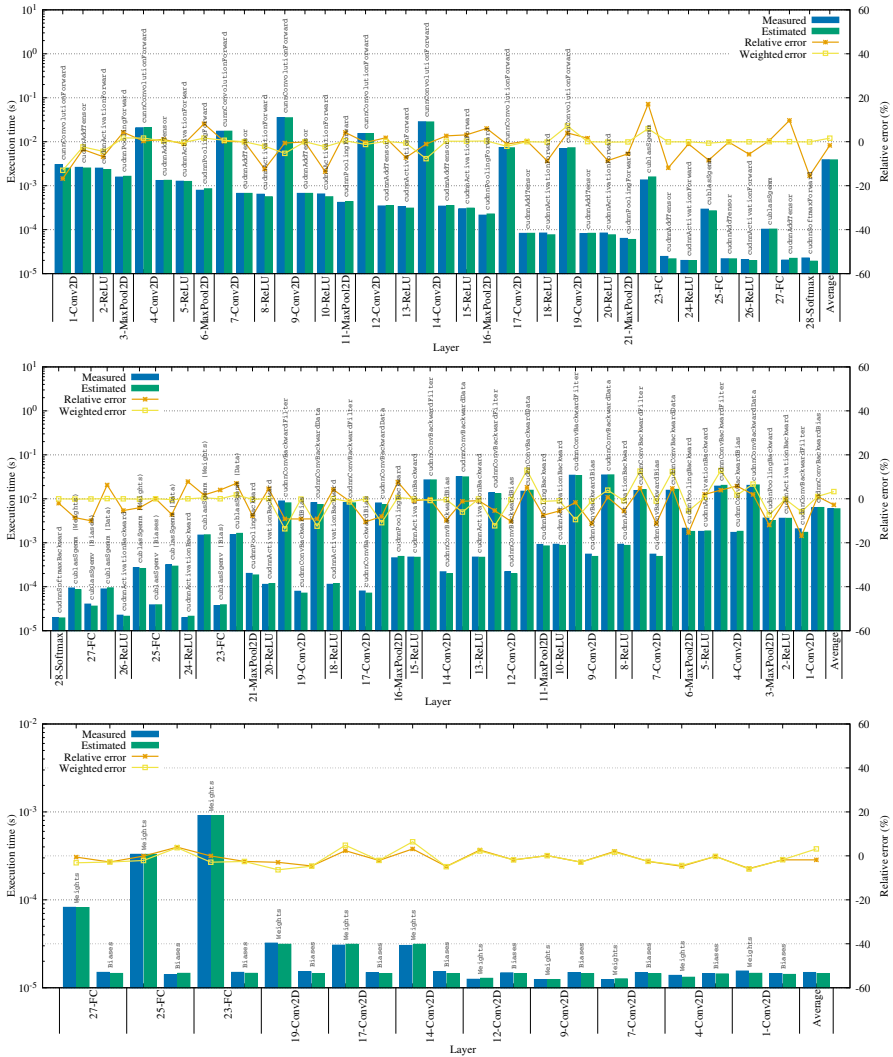


Fig. 2 Execution time (FP+BP+WU) prediction error for the VGG11 model on the V100 GPU using a batch size of 128

where the “average cost per kernel” is given as $T_{average} = \sum_{k=0}^K T_k^{measured} / K$, with K denoting the total number of kernels in a stage (FP, BP or WU). With that, the total WRE for a stage S is defined as:

$$WRE_S^T = \frac{\sum_{k=0}^K |WRE^k|}{K}. \tag{3}$$

When this metric is applied to the modeling of VGG11, the average weighted errors are $WRE_{FP}^T = 1.6\%$, $WRE_{BP}^T = 3.2\%$, and $WRE_{WU}^T = 3.1\%$. In a separate

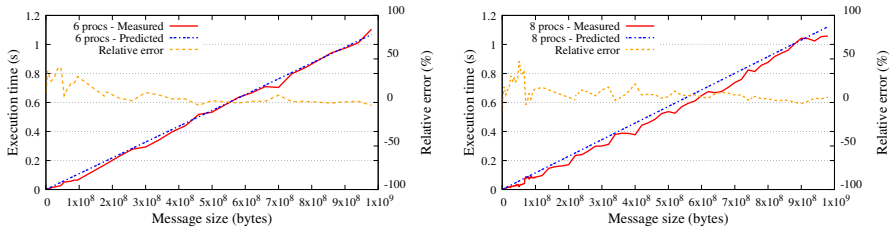


Fig. 3 Measured vs predicted execution times and relative error for the NCCL Allreduce collective communication using 6 and 8 processes, each mapped onto a V100 GPU (node) of the ALTEC cluster

experiment, we validated the general model using the same VGG11 architecture on a smaller batch size of 64. There we observed that the relative errors for most kernels also remained in the range $[-20\%, +20\%]$, with $WRE_{FP}^T = 3.2\%$ and $WRE_{BP}^T = 2.5\%$. Note that the WU validation is independent to the batch size, so it has not been repeated.

Finally, the validation of the analytical model for the NCCL Allreduce communication primitive is performed on a separate benchmark that realizes and measures the collective communication costs by exchanging a message of various sizes, from 500 KiB to 1 GiB on a different number of GPUs. Figure 3 shows the measured and predicted run times with their corresponding relative error for the NCCL Allreduce primitive using 6 (left) and 8 (right) MPI ranks, each mapped onto a single V100 GPU (node) of the ALTEC cluster. The results confirm that the analytical model delivers fairly good estimations, with a relative error that is on average below 10% in message sizes above 80 MiB.

In the light of these results, we consider that the piece-wise performance model, comprised of a collection of MLPs models and an analytical model, provides reasonable predictions of the training costs on the ALTEC cluster, so it seems reasonable to leverage it in the following section to assess the parallel scalability of the training phase on other configurations.

4 Performance analysis

In this section, we study the DP training scalability of three popular CNNs, VGG16, ResNet-50, and DenseNet-121, tailored for two datasets, CIFAR-10 and ImageNet. The inputs for the former dataset are RGB images of size 32×32 , while for ImageNet, the RGB images are of size 224×224 . Also, the classification task of a given image into one out of 10 classes for CIFAR-10 is extended to a total of 1,000 classes in ImageNet. The CNN models specifications have been gathered from the Tensorflow benchmark suite [11].

Hardware setup To analyze the performance and DP training scalability of the three CNNs on the two datasets, we employ a parameterized hypothetical cluster AMPERE; see Table 5 for more details. In general, the AMPERE cluster setup configures a state-of-the-art prototypical supercomputer. Concretely, the number of nodes selection reflects a mid-size cluster, while the processor and GPU models respectively correspond to the Intel Xeon Skylake and NVIDIA A100 leading families. Also, the chosen Infiniband

HDR network conforms with a recent interconnection technologies available from NVIDIA/Mellanox.

Configuration of experiments In all experiments, we employ our general performance models based on MLPs for the computations and the analytical model for the communication, to estimate the total execution time of a training step for the selected CNNs and datasets, on the AMPERE cluster. For each experiment we vary the following parameters: node performance or accelerator performance (floating-point operations per second, or FLOPS, and memory bandwidth); cluster configuration (number of nodes and link bandwidth); and algorithmic batch size. Recall that, given the nature of the MLPs, it is not possible to vary the FLOPS nor the memory bandwidth parameters, as these models were trained for a specific accelerator. Thus, when varying these parameters, the estimations in the following analysis rely on the Roofline models, as done in [6].

For the AMPERE cluster, we set the “baseline” values for γ , μ , α and β to the corresponding theoretical peak thresholds listed in Table 5.

Moreover, in the baseline configuration, the cluster is assumed to consist of $P=100$ nodes, and the batch size b per process (GPU) is fixed to 128. Then, for each plot, we vary a single experimental parameter while setting the remaining ones to the baseline values. The black vertical line in each plot indicates the “position” of this reference configuration according to these experimentation parameters.

4.1 Scalability analysis

Our scalability experiments in Figs. 4 and 5 respectively report the cost for the distributed DP training of the selected models on the CIFAR-10 and ImageNet datasets using the Roofline model (RFL) and the MLP models for the total computation time. The plots also show the communication costs, estimated using the analytical model (AM) for the Allreduce collective.

The first observation from those results is that, for the baseline configuration, the total execution time is mildly dictated by the node computational performance of the GPU (as a combination of peak FLOPS and memory bandwidth). This can be identified in all the figure plots by looking into the component that has a higher contribution to the total execution time in the reference point of the x -axis marked by the black vertical line.

We next analyze the effects of varying the selected parameters, grouping them into the node, cluster, and algorithmic components. Recall that, in the DP scheme, the volume of communication is proportional to the number of trainable parameters of the CNNs and the number of nodes. Also, while scaling the number of nodes, the batch size processed by each GPU is fixed to 128.

GPU performance As shown in the first and second rows of plots, increasing the GPU performance reduces the execution time for all models, but only to a certain point, e.g. $\gamma = 50$ for DenseNet-121. From there, increasing the FLOPS reaches a plateau on the total time, meaning that the costs are dominated by the memory accesses. For the VGG16 model on CIFAR-10, there appears a crossover point around $\gamma = 30$ TFLOPS: below that threshold, the communications represent a bottleneck. Conversely, varying

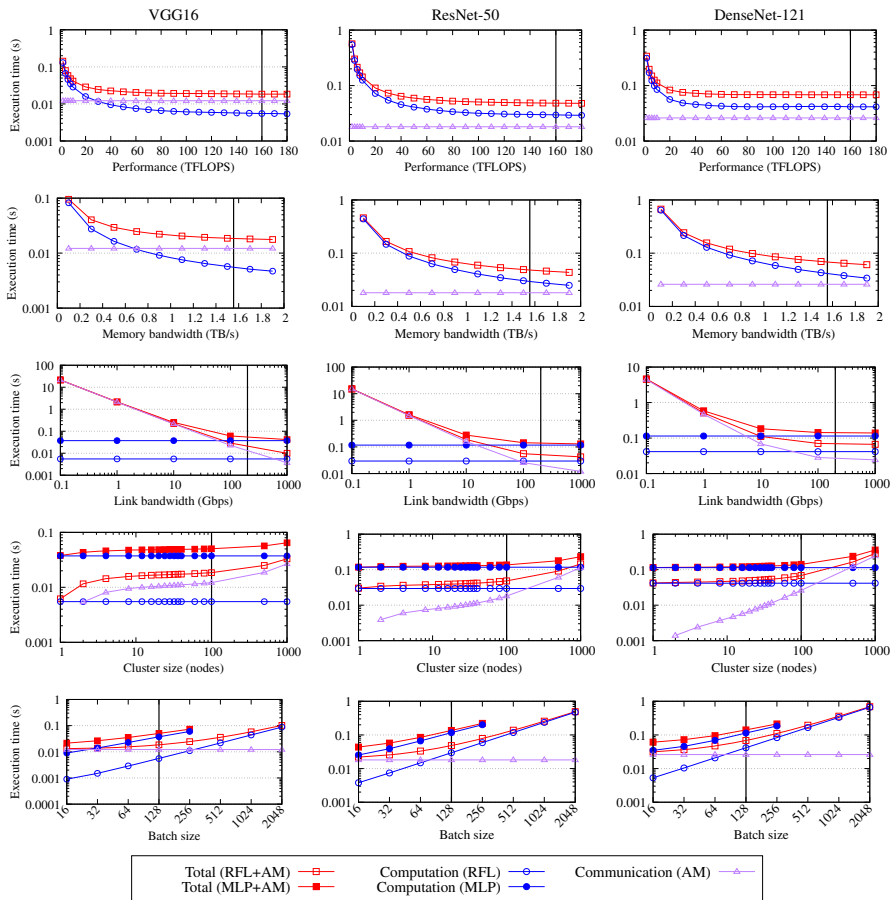


Fig. 4 CNN models execution time (FP + BP + WU) per batch on varying batch sizes, performance, memory bandwidth, link bandwidth and cluster sizes (rows 1–5, respectively) for the CIFAR-10 dataset

the memory bandwidth considerably reduces the costs, showing that all the CNNs for ImageNet, and ResNet-50/DenseNet-121 for CIFAR-10, are mainly memory-bound.

Cluster configuration The experiments at this level (see third and fourth rows of plots) show the relative importance of network communication in the training process. In particular, even for an ideal topology such as a star, a reduction from the nominal $\beta = 200$ Gbps to 10 Gbps transforms the link bandwidth into a bottleneck. This effect is more exacerbated on the CIFAR-10 dataset. This offers little hope for networks slower than Infiniband EDR/HDR such as, e.g. Gigabit Ethernet. Also, increasing the cluster dimension (#nodes, P) and, as a consequence the global batch size, the total costs remain constant up to roughly $P = 100$, where the communications become the bottleneck. This is due to the adoption of the Ring algorithm for the Allreduce exchanges, which entails a communication cost that is linear on $(P - 1)/P$.

Algorithmic batch In the DP scheme, the dimension of the local “problem” is proportional to the ratio between the batch size and the number of nodes. A low value

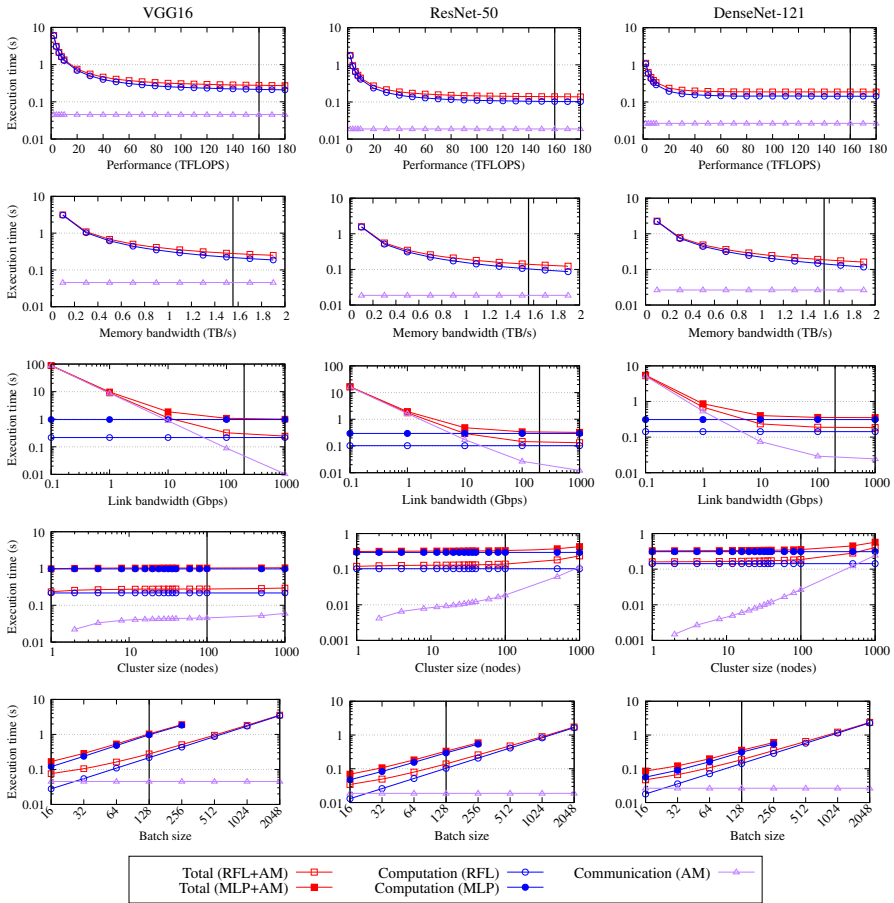


Fig. 5 CNN models execution time (FP + BP + WU) per batch on varying batch sizes, performance, memory bandwidth, link bandwidth and cluster sizes (rows 1–5, respectively) for the ImageNet dataset

for b/P can turn the problem into memory-bound at the node level or communication-bound at the cluster level. The last row of plots shows the memory/communication boundness effect for the smallest batch size, $b = 16$, and CIFAR-10. Scaling further the batch size, shifts the bottleneck to the peak GPU performance. Note that the costs predicted by the MLPs are only represented until $b = 256$, as the correctness of the predictions above that value has not been validated because we could not generate training data above that limit, due to the kernels workspace exceeding the available GPU memory.

As a general remark, we observe that the costs obtained using the MLP models compared with those obtained using the Roofline are slightly different, but follow a similar trend which, according to our previous validation, guarantees a higher estimation accuracy.

5 Related work

There are a few recent theoretical analyses on the performance of distributed data parallelism and model parallelism. In [10], the authors propose hybrid data-model-domain strategies to parallelize the training process of DNNs and study their performance via analytical models. Similarly, in [14] the authors present a DL system that automatically finds parallelization strategies for DNNs across different dimensions via hybrid (data+model) parallelism.

There also exist other works proposing analytical models for DL. For instance, Qi et al. [19] present PALEO, a tool for building analytical performance models. This tool considers a number of representative DL workloads that run operations on the target platform in order to estimate relative inefficiencies compared to the theoretical platform peak. Alternatively, Justus et al. [15] proposed an analytical model based on measured data utilizing several DL models on different target platforms. Afterward, the analytical models are constructed upon a combination of the measured individual layers, though they do not consider distributed DL training on clusters.

6 Concluding remarks

In this work, we have proposed accurate performance models that reproduce the main computation and communication stages appearing in the distributed training of CNNs via DP. In particular, the general performance model is composed of two parts: *i*) a collection of MLPs in charge of individually modeling the related cuDNN/cuBLAS, memory copies, SGD kernels involved in a training step; and *ii*) an analytical model to estimate the cost of the Allreduce collective performed via the NCCL library using the Ring algorithm. Our analysis showed that the relative error of the MLP models on the testing datasets is, at maximum, 15%, while the validation of the models against PyDTNN shows that RE and the WRE are, on average, 5% and 3%, respectively.

Next, we leveraged the models to assess the performance of three representative CNNs on the two datasets along five dimensions. Concerning the GPU performance, we observed that, due to the selected baseline batch size of 128, the layer processing for the CNNs is in general memory-bound. From the cluster point of view, we can conclude that the link bandwidth plays an important role and becomes a limiting factor when it is below 10–30 Gbps. Also, the cost of the Allreduce collective on a large number of GPUs starts exerting some pressure on the total execution time when $P > 100$. Finally, we detect that the execution time is mostly proportional to the batch size (b). Note, however, that increasing b by a certain factor leads to a reduction in the number of FP+BP passes required to complete an epoch in the same factor. Also increasing the b entails a larger local problem per node, improving the arithmetic intensity, and potentially eliminating some of the memory access constraints.

Declarations

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This research was partially sponsored by projects TIN2017-82972-R of *Ministerio de Ciencia, Innovación y Universidades* and Prometeo/2019/109 of the *Generalitat Valenciana*. Manuel F. Dolz was also supported by the Plan GenT project CDEIGENT/2018/014 of the *Generalitat Valenciana*.

Conflicts of interest Not applicable

Availability of data and material Not applicable

Code availability Not applicable

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Awan AA, Manian KV, Chu CH, Subramoni H, Panda DK (2019) Optimized large-message broadcast for deep learning workloads: MPI, MPI+ NCCL, or NCCL2? *Parallel Comput* 85:141–152
2. Barrachina S, Castelló A, Catalán M, Dolz MF, Mestre J (2021) A flexible research-oriented framework for distributed training of deep neural networks. In: 2021 IEEE international symposium on parallel distributed processing, workshops and Phd Forum, pp 730–739
3. Barrachina S, Castelló A, Catalán M, Dolz MF, Mestre J (2021) Pydtnn: a user-friendly and extensible framework for distributed deep learning. *J Supercomput* 77(9):9971–9987
4. Ben-Nun T, Hoefler T (2019) Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. *ACM Comput Surv* 52(4):65:1–65:43
5. Castelló A, Dolz MF, Quintana-Ortí ES, Duato J (2019) Analysis of model parallelism for distributed neural networks. In: EuroMPI '19. Association for Computing Machinery, New York, NY, Article 7, pp 1–10
6. Castelló A, Catalán M, Dolz MF, Mestre JI, Quintana-Ortí ES, Duato J (2021) Performance modeling for distributed training of convolutional neural networks. In: 2021 29th Euromicro international conference on parallel, distributed and network-based processing (PDP), pp 99–108
7. Castelló A, Dolz MF, Quintana-Ortí ES, Duato J (2019) Theoretical scalability analysis of distributed deep convolutional neural networks. In: 2019 19th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID), pp 534–541
8. Chan E, Heimlich M, Purkayastha A, van de Geijn R (2007) Collective communication: theory, practice, and experience: research articles. *Concurr Comput Pract Exper* 19(13):1749–1783
9. Chellapilla K, Puri S, Simard P (2006) High performance convolutional neural networks for document processing. In: International workshop on frontiers in handwriting recognition
10. Gholami A, Azad A, Jin P, Keutzer K, Buluc A (2018) Integrated model, batch, and domain parallelism in training neural networks, pp 77–86
11. Google Inc. Tensorflow benchmarks
12. Hasanov K, Lastovetsky A (2017) Hierarchical redesign of classic MPI reduction algorithms. *J Supercomput* 73(2):713–725
13. Higham CF, Higham DJ (2018) Deep learning: an introduction for applied mathematicians. [arXiv:1801.05894](https://arxiv.org/abs/1801.05894)
14. Jia Z, Zaharia M, Aiken A (2018) Beyond data and model parallelism for deep neural networks. CoRR, [arXiv:1807.05358](https://arxiv.org/abs/1807.05358)

15. Justus D, Brennan J, Bonner S, McGough AS (2018) Predicting the computational cost of deep learning models. In: IEEE international conference on big data, Big Data 2018, Seattle, WA, USA, December 10–13. pp 3873–3882. IEEE
16. NVIDIA (2021) NCCL Tests. <https://github.com/NVIDIA/nccl-tests>
17. NVIDIA (2021) The NVIDIA Collective Communication Library (NCCL). <https://developer.nvidia.com/nccl>
18. Pouyanfar S, Sadiq S, Yan Y, Tian H, Tao Y, Reyes MP, Shyu M-L, Chen S-C, Iyengar SS (2018) A survey on deep learning: algorithms, techniques, and applications. *ACM Comput Surv* 51(5):92:1–92:36
19. Qi H, Sparks ER, Talwalkar A (2017) Paleo: a performance model for deep neural networks. In: Proceedings of the international conference on learning representations
20. Simonyan K, Zisserman A (2014) Very deep convolutional networks for large-scale image recognition
21. Sze V, Chen Y-H, Yang T-J, Emer JS (2017) Efficient processing of deep neural networks: a tutorial and survey. *Proc IEEE* 105(12):2295–2329
22. Thakur R, Rabenseifner R, Gropp W (2005) Optimization of collective communication operations in MPI. *Int J High Perform Comput Appl* 19(1):49–66
23. Williams S, Patterson D, Oliker L, Shalf J, Yelick K (2008) The roofline model: a pedagogical tool for program analysis and optimization. In: 2008 IEEE hot chips 20 symposium (HCS), pp 1–71
24. You Y, Demmel J, Keutzer K, Hsieh C-J, Ying C, Hseu J (2018) Large-batch training for LSTM and beyond. Technical Report UCB/EECS-2018-138, Electrical Engineering and Computer Sciences, University of California at Berkeley
25. You Y, Gitman I, Ginsburg B (2017) Scaling SGD batch size to 32k for ImageNet training. [arXiv:1708.03888](https://arxiv.org/abs/1708.03888)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.