CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Distributed TensorFlow
A performance evaluation

Emanuele Bugliarello
emanuele.bugliarello@gmail.com

Summer Internship Report

September 8, 2017

Supervisors:
Marcel Schöngens
Maxime Martinasso
Claudio Gheller

Contents

1 Introduction

In the past few years, deep neural networks have made breakthroughs in a wide variety of everyday technologies, such as speech-recognition on our smartphones, machine translation and in image recognition. The success of deep learning is built upon the availability of a vast volume of data and as their sizes grow larger, it can take weeks to train deeper neural networks to the desired accuracy. Fortunately, we are not restricted to a single machine and research has been conducted on enabling efficient distributed training of neural networks.

There are dozens of open source machine learning libraries that can be used to develop deep learning applications. Here, we focus on TensorFlow, Google's open source machine learning framework. There are two main reasons why we analyze TensorFlow: first, TensorFlow offers a flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. Second, most CSCS clients use TensorFlow as their deep learning framework.

In this report, we analyze the performance of distributed training in TensorFlow (in terms of number of images trained per second) in different systems and compare our results with the benchmarks available in TensorFlow's website.

The remainder of this report is organized as follows. We first give a brief overview of TensorFlow, present its architecture in distributed training and explain how to easily extend existing single-machine code to run on multiple nodes. We then introduce the systems on which we will run our benchmarks and give some pointers on how to set them up. Next, we describe the scripts we have written to easily run TensorFlow in a distributed environment, with a focus on Piz Daint which runs with Slurm Workload Manager. A case study on MNIST is presented to show how to extend a single-node TensorFlow application to run across multiple nodes. After that, we detail our methodology and discuss the results that we obtain when scaling out to 128 GPUs. Finally, we present directions for future work and conclude this report.

2 TensorFlow

TensorFlow [9] is an open source software library for numerical computation using data flow graphs. Nodes in these graphs represent mathematical operations, while multidimensional arrays (tensors) move across the edges between them; hence the name. An example of a computational graph is shown in Figure 1.

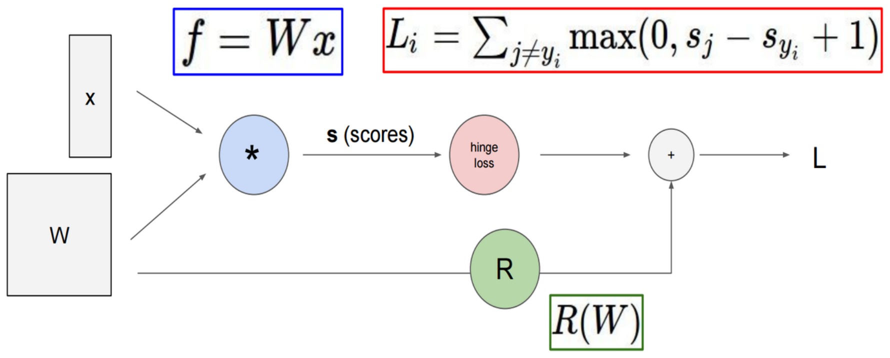

Figure 1: Computational graph for a regularized Multiclass SVM loss [3].

In TensorFlow, you firstly build the computational graph and then run instances of that graph. By doing so, the graph is created only once and the framework can apply some optimizations for you before it runs.

To make this more concrete, let's consider the linear regression example described in Figure 2. The corresponding TensorFlow code is shown in Listing 1.

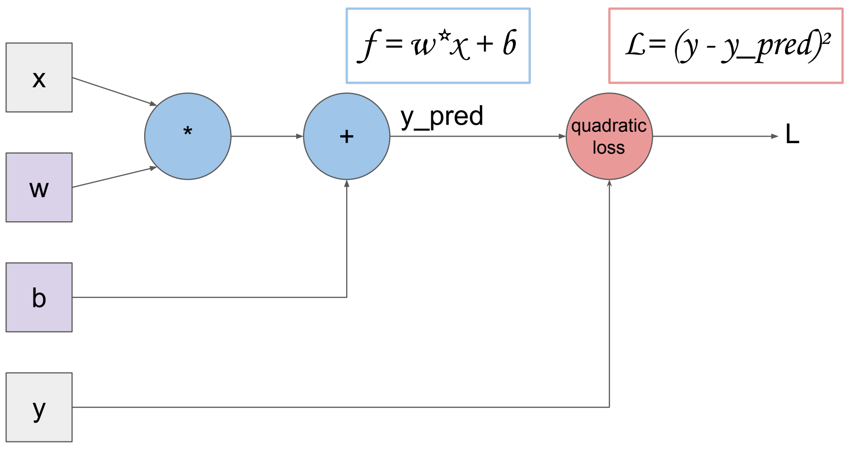

Figure 2: Linear regression computational graph.

```python
import numpy as np
import tensorflow as tf

# ================================================================== #
#                             LOAD DATA                              #
# ================================================================== #
# Generate some data as y=3*x + noise
N_SAMPLES = 10
x_in = np.arange(N_SAMPLES)
y_in = 3*x_in + np.random.randn(N_SAMPLES)
data = list(zip(x_in, y_in))

# ================================================================== #
#                            BUILD GRAPH                             #
# ================================================================== #
simple_graph = tf.Graph()
with simple_graph.as_default():
  # Generate placeholders for input x and output y
  x = tf.placeholder(tf.float32, name='x')
  y = tf.placeholder(tf.float32, name='y')

  # Create weight and bias, initialized to 0
  w = tf.Variable(0.0, name='weight')
  b = tf.Variable(0.0, name='bias')

  # Build model to predict y
  y_predicted = x * w + b

  # Use the square error as the loss function
  loss = tf.square(y - y_predicted, name='loss')

  # Use gradient descent to minimize loss
  optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
  train = optimizer.minimize(loss)

# ================================================================== #
#                           EXECUTE GRAPH                            #
# ================================================================== #
# Run training for N_EPOCHS epochs
N_EPOCHS = 5
```

```python
with tf.Session(graph=simple_graph) as sess:
  # Initialize the necessary variables (w and b here)
  sess.run(tf.global_variables_initializer())

  # Train the model
  for i in range(N_EPOCHS):
    total_loss = 0
    for x_,y_ in data:
      # Session runs train operation and fetches values of loss
      _, l_value = sess.run([train, loss], feed_dict={x: x_, y: y_})
      total_loss += l_value
    print('Epoch {0}: {1}'.format(i, total_loss/N_SAMPLES))
```

Listing 1: Linear regression in TensorFlow.

In the previous snippet, when we build the data flow graph, every variable (such as `x`, `w`, `loss`, and `train`) is not assigned any value but it is actually an *operation* that is added to the graph. Specifically, a `tf.placeholder` represents a container for future values that will be loaded at run time, a `tf.Variable` instead represents a tensor that will be modified by the learning algorithm during the optimization phase, while the other ones are mathematical operations, as shown in Figure 2.

We start an execution by opening a `tf.Session`, to which we pass the graph defined before. Here, we firstly initialize our `tf.Variables` by assigning them their initial value, and then train our model for `N_EPOCHS` epochs [1] by passing each time an input and an output sample via `feed_dict` in `sess.run()`. `sess.run()` evaluates the list of operations that are passed in its first argument. It does so by computing only the nodes in the graph these operations depend on and returns their values at the end of the evaluation.

The resulting linear model is shown in Figure 3.

[1] An epoch is one complete presentation of the training data set to a machine learning model.

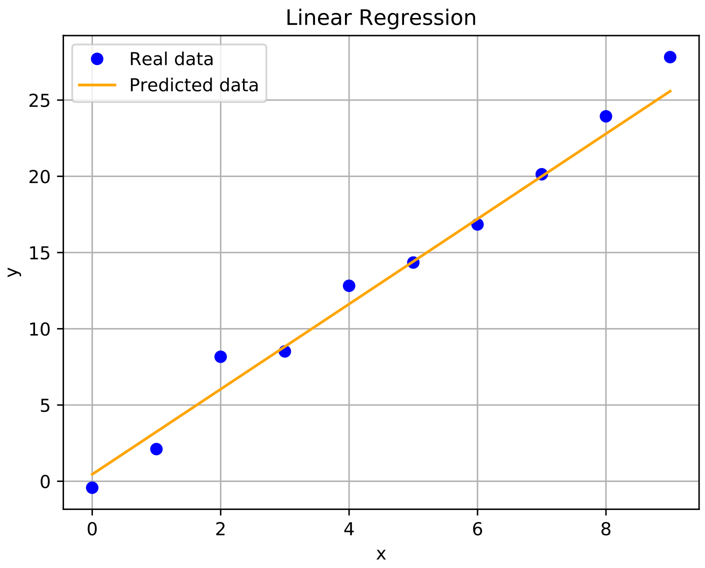

Figure 3: Linear model learned with the example code in Listing 1.

2.1 Distributed training

As neural networks become larger, it can take weeks to train one of them to achieve the desired accuracy. It is then of primary importance to distribute the training of these deep neural networks at a massive scale and reduce the training time to hours. TensorFlow offers a large degree of flexibility in the placement of graph operations, allowing easy implementations for parallel computation across multiple workers.

When splitting the training of a neural network across multiple nodes, the most common strategy is data parallelism, where each node has an instance of the model and reads different training samples.
When using TensorFlow, this is achieved with the so-called “between-graph replication” setting. In this context, processes have one of two roles: *Parameter Servers (PS)* or *Workers*. The former ones host the trainable variables and update them with the values sent by the Workers. Workers, on the other hand, run the model, send their local gradients to the PSs and receive the updated variables back.

In doing so, it is essential that all the Workers send their updates of each variable to the same PSs. To ensure correct device placement of each variable, TensorFlow offers `replica_device_setter`, which provides a deterministic method for variable allocation, ensuring that the variables reside on the same devices.

Given that each Worker runs the same model, the only high-level changes required in a parallel implementation are the definition of the cluster of nodes and the role of each of them (Parameter Server/Worker). The following code snippet (from [4]) shows how to specify such configuration in TensorFlow. Note that such a script would be executed on each machine in the cluster, but with different arguments.

```python
import sys
import tensorflow as tf

# Specify the cluster's architecture
cluster = tf.train.ClusterSpec({'ps': ['192.168.1.1:1111'],
                                'worker': ['192.168.1.2:1111',
                                           '192.168.1.3:1111']
                               })

# Parse command-line to specify machine
job_type = sys.argv[1]  # job type: "worker" or "ps"
task_idx = sys.argv[2]  # index job in the worker or ps list
                        # as defined in the ClusterSpec

# Create TensorFlow Server. This is how the machines communicate.
server = tf.train.Server(cluster, job_name=job_type, task_index=task_idx)

# Parameter server is updated by remote clients.
# Will not proceed beyond this if statement.
if job_type == 'ps':
  server.join()
else:
  # Workers only
  with tf.device(tf.train.replica_device_setter(
                      worker_device='/job:worker/task:'+task_idx,
                      cluster=cluster)):
    # Build your model here as if you only were using a single machine

  with tf.Session(server.target):
    # Train your model here
```

Listing 2: Distributed TensorFlow skeleton.

The first step in running distributed TensorFlow is to define the architecture of the cluster using `tf.train.ClusterSpec`, where the IP addresses and ports of all the processes for each role are provided.

Next, the script determines its job type (or role) and its index among all the processes with the same job type. This is typically achieved by passing command-line arguments to the script, which are then parsed. Here, `job_type` specifies whether the node is running a Parameter Server or a Worker task, whereas `task_idx` specifies the process's index into its task list. An important remark regarding `task_idx` is that the list of nodes per role is interpreted as a sorted array. That is, you cannot arbitrarily set the `task_idx` of a given process; instead, this must reflect the position of that process in the original PS or Worker list specified in `tf.train.ClusterSpec`. For instance, the script for Worker `192.168.1.2:1111` must be launched setting its `task_idx` to 0 as it is the first Worker in the list.

The next step is to use this information to create a TensorFlow Server, which allows this process to communicate with any other server in the same cluster and participate in distributed training.

If the node is a Parameter Server, it simply joins its threads and waits for them to terminate. While it may seem counterintuitive that there is no PS-specific code, the graph elements are actually pushed to it from the workers.

Conversely, if the device is a Worker, we use `replica_device_setter` to build our model, so that parameters are consistently allocated across our Parameter Servers. Finally, a `tf.Session` is created and the model is trained.

A valuable note is that Parameter Servers and Workers may coexist on the same machine. This is actually the recommended choice, especially when GPU-enabled nodes are available. In this case, Parameter Servers would run on CPUs and Workers on GPUs, as their workload is much heavier. By doing so, not only do we reduce the number of nodes required to run a given application, but also minimize the amount of traffic generated in the network, resulting in higher performance.

2.1.1 Load Balancing

In a distributed environment, it is of importance that each Worker has available the updates obtained by the other Workers in order to train faster. This leads to the need of tackling how to place the variables.

TensorFlow's `tf.device` function allows to specify where each operation is stored by means of a device string passed as its argument. The following snippet of code gives an example of how this is done.

```
with tf.device("/job:ps/task:0/cpu:0"):
  weights_1 = tf.get_variable('weights_1', [784, 100])
  biases_1 = tf.get_variable('biases_1', [100])
with tf.device("/job:ps/task:1/cpu:0"):
  weights_2 = tf.get_variable('weights_2', [100, 10])
  biases_2 = tf.get_variable('biases_2', [10])
with tf.device("/job:worker/task:0/gpu:0"):
  # Build your model here
```

Listing 3: Variable placement with device strings.

Here, we ask for `weights_1` and `biases_1` to be placed in the first PS, while `weights_2` and `biases_2` in the second one. Then, each worker designates itself in the third `with` block in the case of "between-graph replication".

However, it may be difficult to specify where each variable is hosted, especially if many Parameter Servers are desirable for an application in order to distribute the work of updating the variables or distribute the networking load for fetching them to the Workers. So, TensorFlow allows to pass a device function instead of a device string to `tf.device`, with the aim of setting a more sophisticated placement strategy.

Some of such functions are already embedded in TensorFlow. The simplest of them is called `tf.train.replica_device_setter`, which assigns variables to the Parameter Servers in a round-robin fashion as they are created. A nice property of this device function is that it allows to write all the code to build a model in a single `with` block. In fact, this only affects the variables, putting them in different Parameter Servers, while the rest of the operations in the graph go on Workers, simplifying "between-graph replication" parallelism.

The following snippet gives an example of using this function and the resulting variable placement is shown in Figure 4 under the round-robin case.

```
with tf.device(tf.train.replica_device_setter(ps_tasks=3)):
  weights_1 = tf.get_variable('weights_1', [784, 100])
  biases_1 = tf.get_variable('biases_1', [100])
  weights_2 = tf.get_variable('weights_2', [100, 10])
  biases_2 = tf.get_variable('biases_2', [10])
  # Build your model here
```

Listing 4: Default variable placement with `replica_device_setter`.

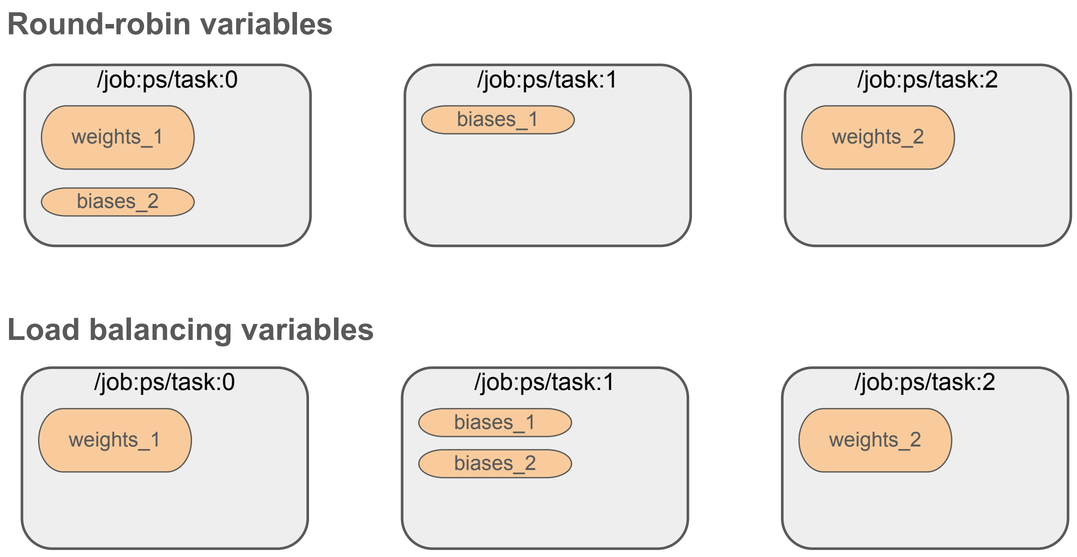

Figure 4: Round-robin (default) and greedy load balancing variable placement with replica_device_setter.

For this example, Figure 4 shows that `weights_1` would go to the first Parameter Server, `biases_1` would go to the second Parameter Server, `weights_2` would be put on the third Parameter Server and `biases_2` back on the first Parameter Server. This is obviously not a balance load for these variables, neither in terms of the memory usage nor in terms of the work to be done to update these variables.
Moreover, if only two Parameter Servers were used here, we would end up in an even worse case where all the weights would go on the first Parameter Server and all the biases on the second one, giving an even bigger imbalance between these tasks.

To achieve a more balance load, TensorFlow allows to specify a load balancing strategy in `tf.train.replica_device_setter` as an optional argument.
The only one currently available is a simple greedy strategy that does a kind of online bin packing based on the number of bytes of the parameters, giving a more balanced outcome as shown under load balancing variables in Figure 4 for our example.

```
greedy =  tf.contrib.training.GreedyLoadBalancingStrategy(...)
with tf.device(tf.train.replica_device_setter(ps_tasks=3,
                                              ps_strategy=greedy)):
  weights_1 = tf.get_variable('weights_1', [784, 100])
  biases_1 = tf.get_variable('biases_1', [100])
  weights_2 = tf.get_variable('weights_2', [100, 10])
  biases_2 = tf.get_variable('biases_2', [10])
```

Listing 5: Greedy load balancing variable placement with `replica_device_setter`.

3 Environments

In this section, we introduce all the systems which have been used to test and run TensorFlow applications and how to set them up.
The version of TensorFlow that we chose is 1.1.0 in order to compare our results with other benchmarks available online.

The code for this section can be found in the `environments_setup` folder of our repository.

3.1 Local workstation

With local workstation we mean a device, such as a laptop, which usually does not have much compute power. This can be used to just test whether an application works, even in a distribute setting if it possesses multiple CPUs and/or GPUs.

Follow the instructions on the GitHub page to install TensorFlow and create a virtual environment.

3.2 Piz Daint

Piz Daint is a hybrid Cray XC40/XC50 supercomputer at CSCS. The system has Aries routing and communications ASIC, with Dragonfly network topology.
At the time of writing, it is the third most powerful supercomputer in the world [12] and in the top ten of the most energy-efficient supercomputers [6].
Each node that we use in Piz Daint is equipped with an NVIDIA Tesla P100 [8].

We use the TensorFlow 1.1.0 module available in Piz Daint whenever we run an application.
The instructions in the GitHub page show how to create a virtual environment containing all the requirements needed to also run Jupyter notebooks (provided a local workstation has already been set up and its `pip` requirements are available).

3.3 AWS EC2

We also use Amazon EC2 instances [5] to compare the speedup achieved on Piz Daint with the virtual servers available in the cloud of one of the most popular web services.
There are many types of virtual servers, also known as compute instances, to choose from [1]. For our comparisons, we make use of P2 instances, intended for general-purpose GPU compute applications. In particular, we use *p2.xlarge* (1 GPU per node) and *p2.8xlarge* (8 GPUs per node) models.

`AWS.md` (in the repository folder) contains additional information on how to create EC2 instances, Amazon S3 [2] buckets (object storage) and how to transfer data from/to S3.
The instructions in the README file illustrate how to set up each instance to run TensorFlow 1.1.0. To do so, NVIDIA cuDNN [7] is required. In our case, we retrieve it from Piz Daint.
The only inputs required for the setup of all the machines are their IP addresses, both public and private ones [2]. Hence, you can simply launch compute instances via the AWS management console and copy their IP addresses, one per line, in `aws_public_ips.txt` and `aws_private_ips.txt` under the repository's root directory, without leaving any empty lines.

[2]We need the instances' private IP addresses in order to avoid sending each packet through an additional hop, which would considerably reduce performance.

4 Distributed TensorFlow

We now describe how to launch a script written to use distributed TensorFlow (see Section 2.1) in each of the environments introduced in the previous section.

The code for this section can be found in the `distributed_tensorflow_launchers` folder of our repository.

4.1 Local workstation

The setup script for the local workstation runs each task (PS or Worker) on a different terminal window. By default, Parameter Servers are launched starting at port 2230, while Workers at 2220. The script calls, for each task, `run_dist_tf_local.sh`.

This script then runs the (distributed TensorFlow) Python script defined herein, with the flags specified in this file as well, for its corresponding task.

4.2 Piz Daint

In the setup script for Piz Daint, we firstly set options for Slurm and load the TensorFlow module. Then, we define the (distributed TensorFlow) Python script to be executed and its flags. Finally, we set the number of Parameter Servers and Workers. These values must be consistent with the number of nodes requested for the job. In particular, if the Parameter Servers run in a (sub)set of the Worker nodes [3] (default behavior), then the number of Workers must not exceed the number of allocated nodes. On the other hand, if the PSs need to run on different nodes than the Workers, then the total number of tasks must not exceed the number of allocated nodes. In case the number of allocated nodes is not enough, an error message is returned. Other settings for a distribute run (commented in the setup script) can be tuned. This script then calls `run_dist_tf_daint.sh` with the settings declared, which runs the Python script in a distribute environment as described in the next paragraph.

`run_dist_tf_daint.sh` runs the Python script that exported in the setup file. Firstly, the script checks which configuration parameters have been set by the user in the setup file. The only necessary information needed is the name of the Python

[3] We assume that the number of Workers is always greater than or equal to the number of PSs.

script. The number of Parameter Servers defaults to 1, while the number of Workers defaults to the number of allocated nodes. If not set in the setup file, the script also assumes to run one Worker per node and at one Parameter Server. Note that it is not possible to run multiple Workers on a single node in Piz Daint if you use the GPU partition as multiple TensorFlow tasks cannot share the same device. As mentioned above, if multiple Parameter Servers are set, the script's default is to run them in a (sub)set of the nodes running a Worker task. This is possible because Worker's operations run in the GPU, while Parameter Servers run in the CPU.
The script then retrieves which nodes have been assigned to the job and creates two comma-separated lists: one indicating Parameter Server hosts and one indicating Worker hosts. For each node, PSs start at port 2230, while Workers at port 2220.
After that, for each node, the script determines how many PSs and Workers are to be run in that node, and creates a Bash script to launch Parameter Server and/or Worker processes. When creating these Bash scripts, if a Parameter Server is to be launched, then it is necessary to hide the GPU to avoid that the PS runs on it; which would result in the Worker running on the CPU.
The need of a Bash script is justified by the fact that you can only have a single `srun` execution per node. So, we just run each process in background (appending `&` at the end of the command) but the last one.

4.3 AWS EC2

The setup script for Amazon EC2 instances runs remotely; i.e. from a local workstation, for instance. It launches one or multiple tasks for each node, according to the number of PSs and Workers entered. Parameter Servers always run in nodes running Worker tasks as well. The only inputs to the setup script are the IP addresses of the instances, the path of the private key you use to log into them and the number of PSs and Workers. In detail, a `screen` session is started for each task.
Parameter Servers' ports start from 2230 at each node, while Workers' from 2220.
It is necessary that private and public IP addresses correspond to the same EC2 instance in the two IP files. That is, the private IP address in line 1 of the private IP addresses file must be the private address of the instance whose public IP address is in line 1 in the public IP addresses file. Private IP addresses are requested in order to reduce the number of hops between two nodes, achieving higher performance.

`run_dist_tf_aws.sh`, instead, has to be copied in each EC2 instance, along with the (distributed TensorFlow) Python script. When called from the setup file, this script firstly hides the GPUs from the Python application if the launched task is a Parameter Server (to the Workers to use them) and then runs the application.

4.4 Case Study: MNIST

The `MNIST` folder of our repository contains an application of the scripts described above for a local workstation and Piz Daint.

`DeepMNIST.ipynb` and `deepMNIST.py` contain the code of the original deep MNIST tutorial available in TensorFlow's website, which consists of a three-layer neural network (two convolutional layers followed by a fully-connected layer) to classify handwritten digits.

We then provide a GPU-enhanced version of this network (`deepMNIST_gpu.py`).
As described in TensorFlow's High-Performance Models page, one of the best practices to improve performance and increase flexibility of a model is to add the support for the data format. In fact, most TensorFlow operations used by a CNN support both NHWC and NCHW image data formats. Image data format refers to the representation of batches of images. TensorFlow supports NHWC (TensorFlow default) and NCHW (cuDNN default). N refers to the number of images in a batch, H refers to the number of pixels in the vertical dimension, W refers to the number of pixels in the horizontal dimension, and C refers to the channels (e.g. 1 for black and white, 3 for RGB, etc.). Although cuDNN can operate on both formats, it is faster to operate in its default format. So, NCHW should always be used when training with GPUs, while NHWC is sometimes faster on CPUs. By adding data formats to an application, it is then possible to train using NCHW on GPU, and then do inference with NHWC on CPU.
In order to make the existing application support NCHW data format, we introduce some if statements that allow to swap the order of the elements in the *kernel size* and *strides* arrays in the pooling layers. Moreover, we also use the optional `data format` argument of the `tf.nn.conv2d` function to let the specified image data format being used in convolutions.

Finally, we apply the template shown in Listing 2 to train this GPU-enhanced version of MNIST across multiple nodes in `dist_deepMNIST_gpu.py`. Here, only Worker 0 evaluates test accuracy, while each Worker evaluates their train accuracy. To launch this application, we used the setup and run_dist_tf scripts presented in this section.

5 Benchmarking Distributed Training

We now present the scalability results relative to training InceptionV3 [16], a deep neural network by Google, on GPU-enabled nodes in Piz Daint and in Amazon EC2. To do so, we use Google's script [10], which provides optimized implementations for multiple networks. The dataset used for training is ImageNet [14], one of the most common datasets used for classification in Computer Vision.

The code for this section can be found in the `google-benchmarks` folder of our repository.

5.1 Methodology

Google's script allows to set different parameters, such as the batch size, the number of warmup steps, the number of steps to be averaged, whether to use NVIDIA NCCL all-reduce primitives and the data layout format (NCHW or NHWC).

The main output of this script is the average number of images per second that have been trained in the system. In order to find a good ratio between the number of Workers and the number of Parameter Servers, we try, for each configuration of number of Workers and number of nodes, several values for the number of PSs ranging from 1 to the number of Workers. For each configuration, we then report the results achieving the largest number of images trained per second. In order to produce results that are as repeatable as possible, each test was run 5 times and then the times were averaged together, analogously to what Google did. GPUs are run in their default state on all the platforms.
For each test, 10 warmup steps are done and then the next 100 steps are averaged.

We ran our benchmarks using both real and synthetic data [4], so that we can evaluate both the compute and the input pipelines.

5.2 Systems

We run benchmarks on Piz Daint, as well as on *p2.xlarge* and *p2.8xlarge* Amazon EC2 instances. Whenever possible, we compare our results with the ones published

[4]By synthetic data we mean fake data that has almost the same properties as the real one.

by Google [11], obtained with *NVIDIA DGX-1* and Amazon *p2.8xlarge* systems. Piz Daint and NVIDIA DGX-1 both have NVIDIA Tesla P100 GPUs, even though the former only has one GPU per node, while the latter has 8 GPUs per node. Amazon *p2.xlarge* and *p2.8xlarge* EC2 instances, instead, are equipped with NVIDIA Tesla K80 GPUs. *p2.xlarge* instances have one GPU per node, while *p2.8xlarge* instances have eight GPUs per node (four K80).

5.3 Results

For all of the reported results, the following settings are used:

- **Model:** InceptionV3
- **Batch size per GPU:** 64
- **Data Format:** NCHW
- **Local Parameter Device:** CPU
- **Optimizer:** sgd
- **Piz Daint OS:** Suse 12/CLE 6.0.UP02
- **AWS OS:** Ubuntu 16.04 LTS
- **CUDA/cuDNN:** 8.0/5.1
- **TensorFlow:** 1.1.0
- **Piz Daint Parallel File System:** Lustre
- **AWS Disk:** Local SSD
- **DataSet:** ImageNet
- **Test Date:** August 2017

Moreover, nodes running Workers also run Parameter Servers as this leads to higher performance.

5.3.1 Training with NVIDIA Tesla P100

Google provides results only for a single NVIDIA DGX-1, hence allowing comparisons up to 8 GPUs. Results are shown in Figure 5 for synthetic data (no I/O). Here, we can see that the peak performance of Piz Daint is close to the one achieved by an NVIDIA DGX-1, even though multiple nodes are used in Piz Daint. Specifically,

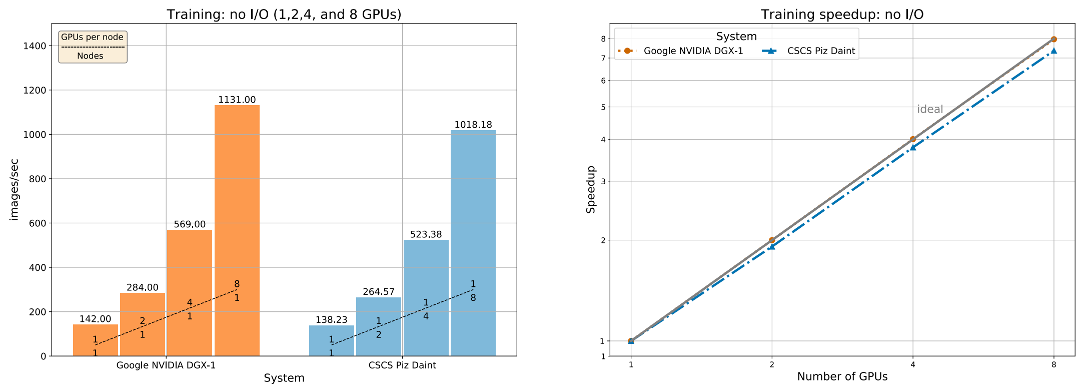

Figure 5: Training with NVIDIA Tesla P100 on synthetic data up to 8 GPUs.

with eight GPUs, while Google reports a speedup efficiency of 99.56%, we report a speedup efficiency of 92.07% on Piz Daint.

5.3.2 Training with NVIDIA Tesla K80

Figure 6 shows how Amazon EC2 *p2.xlarge* and *p2.8xlarge* compute instances scale out up to 8 GPUs.

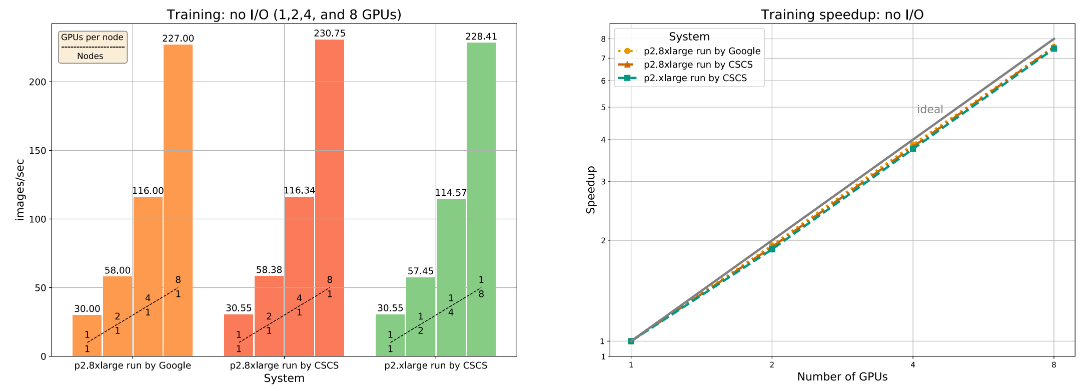

Figure 6: Training with NVIDIA Tesla K80 on synthetic data up to 8 GPUs.

Google provides results that achieve a scalability efficiency of 94.58% with 8 GPUs on a *p2.8xlarge*, and, similarly, our measurements show an efficiency of 94.44% on the same machine.
We also ran tests on *p2.xlarge* instances, showing that comparable performance (93.45% scalability efficiency) can be obtained with eight nodes (eight GPUs).
Hence, we can infer that the application is compute bounded when up to eight GPUs are used because we achieve the same performance with eight nodes as with a single node having eight GPUs regardless of the underlying network (Piz Daint or AWS).

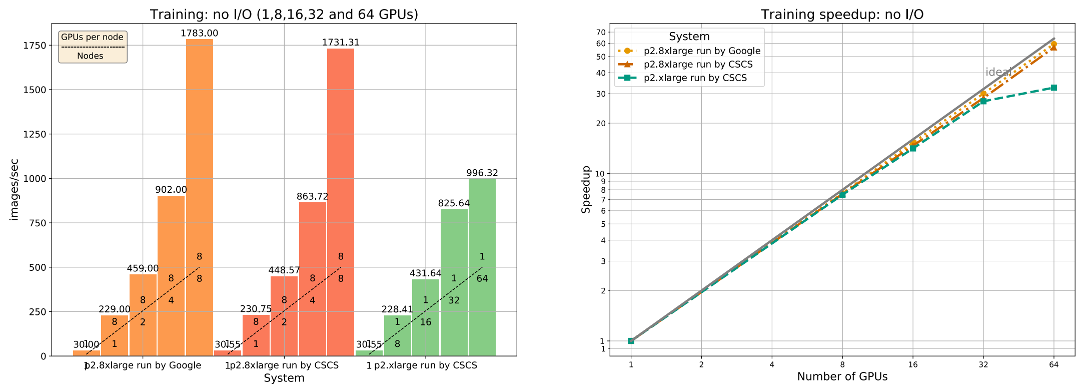

Figure 7: Training with NVIDIA Tesla K80 on synthetic data up to 64 GPUs.

Figure 7, instead, shows how the number of images trained per second in these systems scales when up to 64 GPUs are used. It is interesting to note that up to 16 GPUs, *p2.xlarge* and *p2.8xlarge* systems have close performance: We report 88.31% for the former and 91.77% for the latter.
Moreover, even though 32 nodes are required for a *p2.xlarge* system to use 32 GPUs, it still achieves a scalability efficiency greater than 80%. Specifically, we report 88.35% efficiency for a four-node *p2.8xlarge* system and 84.45% efficiency for a thirty-two-node *p2.xlarge* system.
However, once a cluster of sixty-four *p2.xlarge* nodes is employed, the scalability efficiency stops at 50.96%, while a cluster of eight *p2.8xlarge* still exhibits 88.55% efficiency from our measurements and 92.86% from Google's ones. This is probably due to the fact that the network capacity is not sufficient anymore for the amount of traffic generated by all the nodes in the *p2.xlarge* cluster.

5.3.3 Distributed training on Piz Daint

Figure 8 shows how the average number of images trained per second varies as the number of nodes (GPUs) increases both when using fake data and when reading data from the Parallel File System on Piz Daint. Each of these values represents the peak performance achieved with the corresponding number of GPUs, obtained by setting the parameters listed in Table 1. Here, we can see that Piz Daint's scalability efficiency drastically drops when 128 nodes are used. We think this is due to having reached the inter-node network capacity because of the largely increased amount of data sent between Workers and Parameter Servers.

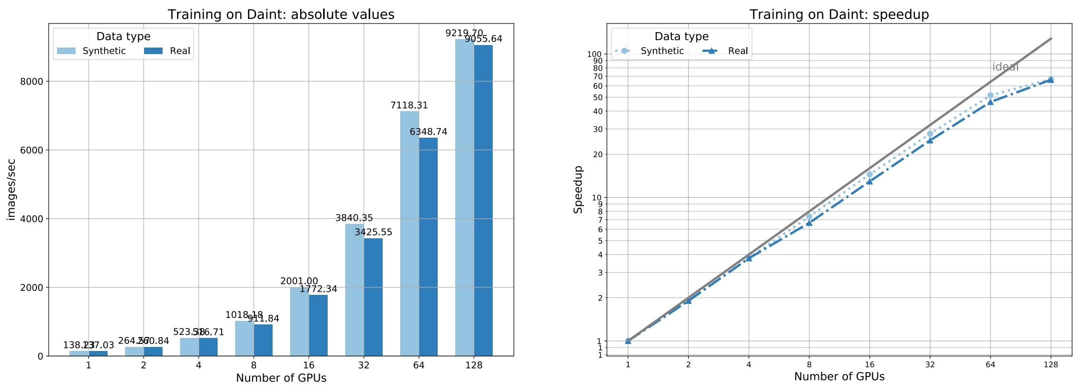

Figure 8: Training on Piz Daint with synthetic and real data up to 128 GPUs.

Num PSs	**Num GPUs**	**Variable Update**	**Real Data**	**Img/s**
1	1	parameter_server	FALSE	138.23
1	1	parameter_server	TRUE	137.03
1	2	parameter_server	FALSE	264.57
1	2	distributed_replicated	TRUE	260.84
3	4	distributed_replicated	FALSE	523.38
3	4	distributed_replicated	TRUE	516.71
2	8	parameter_server	FALSE	1018.18
2	8	parameter_server	TRUE	911.84
4	16	parameter_server	FALSE	2001.00
4	16	parameter_server	TRUE	1772.34
12	32	parameter_server	FALSE	3840.35
12	32	parameter_server	TRUE	3425.55
40	64	parameter_server	FALSE	7118.31
40	64	parameter_server	TRUE	6348.74
116	128	parameter_server	FALSE	9219.70
116	128	parameter_server	TRUE	9055.64

Table 1: Parameters achieving peak performance on Piz Daint.

5.3.4 Distributed training on Amazon p2.xlarge

Figure 9 displays the trend of average number of images trained per second as the number of nodes (GPUs) increases both when using fake data and when reading data from the local SSD at each node in a cluster of *p2.xlarge* machines. The parameters resulting in the peak performance for each different number of GPUs are listed in Table 2. In this plot, we can see that *p2.xlarge*'s scalability efficiency diminishes only when 64 nodes are benchmarked. This system has single-GPU nodes like Piz Daint but it stops scaling out efficiently for a smaller number of nodes. The main difference amongst them is their inter-node network (Piz Daint's being faster), providing additional support to our belief of inter-node network bottleneck for Piz Daint and *p2.xlarge* systems.

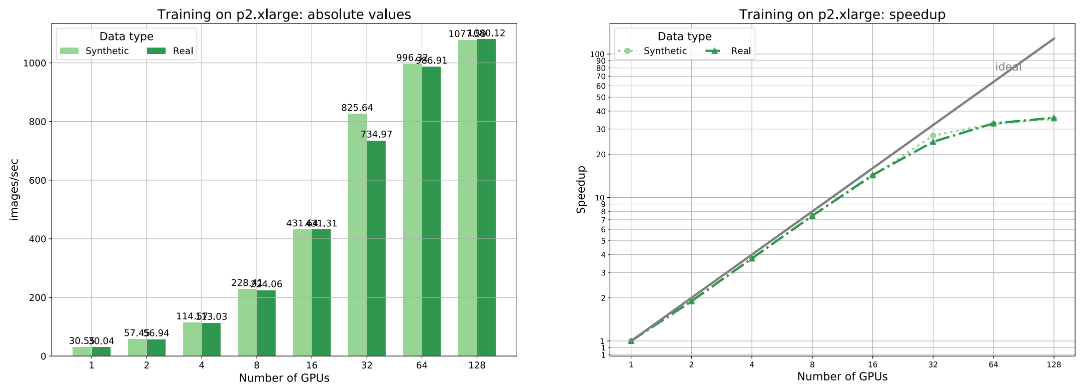

Figure 9: Training on *p2.xlarge* with synthetic and real data up to 128 GPUs.

Num PSs	Num GPUs	Variable Update	Real Data	Img/s
1	1	parameter_server	FALSE	30.55
1	1	parameter_server	TRUE	30.04
2	2	distributed_replicated	FALSE	57.45
2	2	distributed_replicated	TRUE	56.93
4	4	distributed_replicated	FALSE	114.57
4	4	distributed_replicated	TRUE	113.03
8	8	distributed_replicated	FALSE	228.41
8	8	distributed_replicated	TRUE	224.06
12	16	distributed_replicated	FALSE	431.64
12	16	distributed_replicated	TRUE	431.31
32	32	parameter_server	FALSE	825.64
32	32	parameter_server	TRUE	734.97
64	64	parameter_server	FALSE	996.32
64	64	parameter_server	TRUE	986.91
128	128	parameter_server	FALSE	1077.59
128	128	parameter_server	TRUE	1080.12

Table 2: Parameters achieving peak performance on *p2.xlarge* systems.

5.3.5 Distributed training on Amazon p2.8xlarge

Figure 10 presents the average number of images trained per second as a function of the number of GPUs both when using fake data and when reading data from local the SSD at each node in a *p2.8xlarge* cluster. The parameters giving the highest performance for the different numbers of GPUs are enlisted in Table 3. This plot does not show any evident reduction in scalability as the number of GPUs is increased. In such system, Workers aggregate their updates before sending them to the PSs. Hence, the traffic generated when 128 GPUs are used here is comparable to the one generated by a system of sixteen single-GPU nodes.

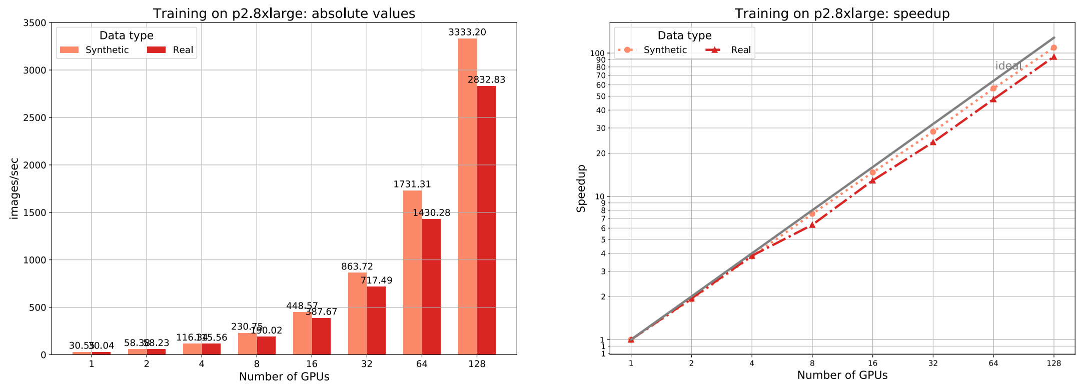

Figure 10: Training on *p2.8xlarge* with synthetic and real data up to 128 GPUs.

Num PSs	Num GPUs	Variable Update	Real Data	Img/s
1	1	parameter_server	FALSE	30.55
1	1	parameter_server	TRUE	30.04
1	2	distributed_replicated	FALSE	58.38
1	2	distributed_replicated	TRUE	58.23
1	4	distributed_replicated	FALSE	116.34
1	4	distributed_replicated	TRUE	115.56
1	8	distributed_replicated	FALSE	230.75
1	8	distributed_replicated	TRUE	190.02
1	16	distributed_replicated	FALSE	448.57
1	16	distributed_replicated	TRUE	387.67
3	32	distributed_replicated	FALSE	863.72
3	32	distributed_replicated	TRUE	717.49
8	64	distributed_replicated	FALSE	1731.31
8	64	distributed_replicated	TRUE	1430.28
16	128	distributed_replicated	FALSE	3333.20
16	128	distributed_replicated	TRUE	2832.83

Table 3: Parameters achieving peak performance on *p2.8xlarge* systems.

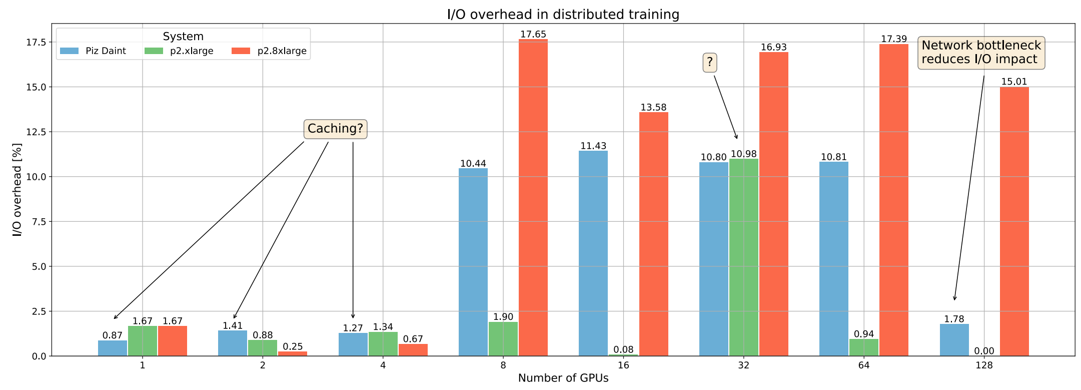

Figure 11: I/O overhead in distributed training in terms of number of GPUs for all the systems under study.

5.3.6 I/O overhead

Finally, Figure 11 plots the relative overhead (in percentage) due to I/O access. That is, for each setting, we obtain the relative I/O in percentage as:

$$I/O\ overhead\ {}^{N\_GPUs}_{System} = \frac{Img/s\_Synthetic^{N\_GPUs}_{System} - Img/s\_Real^{N\_GPUs}_{System}}{Img/s\_Synthetic^{N\_GPUs}_{System}} \times 100.$$

The first thing we observe from this plot is that when 8 GPUs per node are used in a *p2.8xlarge* cluster, where each node loads data from a local SSD, a constant I/O overhead of around 17% is present (due to PCIe traffic).

Looking at *p2.xlarge* clusters, instead, we see that I/O access does not add any overhead, apart when thirty-two nodes are used. However, this still comes at the expenses of replicating the data at each node.

Focusing on Piz Daint at last, we see that around 11% of I/O overhead is present when eight to sixty-four nodes are used.
On the other hand, this is not shown when less nodes are employed. The reason might be due to caching mechanisms in the system.
The I/O overhead drops down once more for one hundred and twenty-eight nodes. In this case, the reason of this reduction may be found in the predominance of the inter-node network bottleneck, which makes the impact of I/O access negligible.

TensorFlow communication patterns should be profiled to verify all our intuitions.

6 Conclusion and Future Work

Deep neural networks are being used to solve challenges that not long ago were believed to be infeasible to face. Deep learning thrives with large neural networks and large datasets, resulting in training times that can be impractical on a single node.

In this report, we show how to train a model in TensorFlow in a distributed setting and provide benchmarks for InceptionV3 on different systems.
The first outcome is that training on eight nodes in Piz Daint achieves close performance to an NVIDIA DGX-1, an integrated system for deep learning.
Looking at the scalability on Piz Daint for InceptionV3, we expect an average 11% overhead due to I/O access when compared to the corresponding performance with synthetic data. Moreover, we expect to detect an inter-node network bottleneck after 64 nodes for this application.
In multi-GPU systems, there is no strong dependence on the interconnect up to 16 8-GPU nodes thanks to the local aggregation performed at each node which reduces the inter-node traffic by the number of GPUs per node. Moreover, using local SSDs and eight GPUs per node adds a constant 17% I/O overhead due to the generated PCIe traffic.
Unfortunately, no benchmarks for multiple DGX-1 systems are available at the time of writing, making any direct comparison with Piz Daint impossible. However, for this application, we expect that it is possible for 64 nodes in Piz Daint to achieve performance close the one of a 8 DGX-1 systems.

As part of future work, we plan to profile TensorFlow communication patterns to verify our intuition of inter-node network bottleneck when the number of nodes in a systems becomes large.
A fundamental topic to be investigated is the resulting training accuracy when an application is trained in multiple single- and multi-GPU systems. Distributed deep learning is a currentt hot research area. Recently, Facebook showed that they trained, with no loss of accuracy, ImageNet in one hour in Caffe2 using ResNet-50 [15], while IBM trains ResNet-50 in fifty minutes [13] in Torch in their software-hardware co-optimized distributed deep learning system.
Another interesting aspect to look into is how the number of Parameter Servers required to achieve the highest performance for a given number of Workers and nodes depends on the underlying inter-node network capacity. In fact, both Piz Daint and *p2.xlarge* clusters have single-GPU nodes but when the number of Workers (nodes) becomes large, they require a different number of Parameter Servers to reach their peak performance. In particular, *p2.xlarge* cluster end up asking for as many Parameter Servers as Workers (we did not test whether more Parameter Servers than Workers might lead to better performance).

References

[1] Amazon ec2 instance types. `https://aws.amazon.com/ec2/instance-types/`. Accessed: 2017-08-29.

[2] Amazon simple storage service. `https://aws.amazon.com/s3/`. Accessed: 2017-08-29.

[3] Deep learning software. `http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture8.pdf`. Accessed: 2017-08-24.

[4] Distributed tensorflow. `https://clindatsci.com/blog/2017/5/31/distributed-tensorflow`. Accessed: 2017-07-20.

[5] Elastic compute cloud. `https://aws.amazon.com/ec2/`. Accessed: 2017-08-29.

[6] The green 500. `https://www.top500.org/green500/lists/2017/06/`.

[7] Nvidia cudnn. `https://developer.nvidia.com/cudnn`. Accessed: 2017-08-29.

[8] Nvidia tesla p100. `http://www.nvidia.com/object/tesla-p100.html`. Accessed: 2017-08-29.

[9] Tensorflow. `https://www.tensorflow.org/`. Accessed: 2017-08-24.

[10] Tensorflow benchmarks code. `https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks`. Accessed: 2017-07-18.

[11] Tensorflow benchmarks results. `https://www.tensorflow.org/performance/benchmarks`. Accessed: 2017-07-18.

[12] Top 500 list. `https://www.top500.org/lists/2017/06/`.

[13] Minsik Cho, Ulrich Finkler, Sameer Kumar, David Kung, Vaibhav Saxena, and Dheeraj Sreedhar. Powerai ddl. *ARXIV*, 08 2017.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[15] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *ARXIV*, 06 2017.

[16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.