Final Project

CS 5334

Dr. Shirley Moore

Oscar Galindo

***Project:  Scale a machine learning benchmark on multiple GPUs on a Bridges-2 node in the GPU partition.***

**Introduction**

CNNs or Convolutional Neural Networks are considered a benchmark on the field of Deep learning because of the Convolutional Neural Networks’ prevalence among many other implementations in the field. To name a few, all the networks of the family of RCNN, Faster R-CNN, Fast R-CNN, Mask RCNN, YOLO implementations like YOLOv3, YOLOv4, among others. Convolutional Neural Networks are also commonly used as a benchmark for its capacity to act a sort-of heuristic to assess what kind of performance could be obtained from other models. For example, if a Faster R-CNN model was to be used for detection of boundary coordinates and class detection on objects of images presented to the network it is normally a good idea to test the problem on a decreased complexity setup of the problem like classification. Rationally speaking, the idea is to ask a less complicated question like saying “is there any indication that something I consider important is on the image?” rather than asking “detect every single object of importance on the image as RoI or Regions of Interest and assign a probability ‘p’ for every class that exists and also report/obtain the coordinates of the bounding box of every region” which is a more complicated setup of the question.

Historically, Convolutional Neural Networks are also very relevant to the field, for example, challenges like that of MNIST were solved through the implementation of a CNN. Before CNN’s the best approaches were algorithms that are part of Machine Learning, like Decision trees, SVMs, Dense networks, etc. With the introduction of Convolutional Neural Network challenges that used to be difficult for Machine Learning algorithms became “easier”.

**Dataset**

The dataset used for the scaling-up experimentation is a dataset called COVIDx CXR-2. It is a dataset composed of thousands of images of X-ray scans made to people. The challenge in this dataset was to identify the lung scans that did show a COVID case and those that did not. The dataset does not contain binary data in the sense that either an X-ray scan could have Covid or else the rest of the scans were normal-looking lungs, instead the dataset contained images of either lungs that are affected by some type of disease that is not COVID, perfectly normal lungs, and lungs that are affected by COVID.

The dataset is basically subdivided among two different lists of images, training which has exactly 15951 images of which 13793 are negative images and 2158 are positive, in addition there is also testing which has 400 images of which 200 are positive and 200 are negative. The implicit meaning of an image being tagged as “positive” means to indeed show COVID affecting the lungs shown in the scan.

The complete description of the dataset can be found here: <https://www.kaggle.com/andyczhao/covidx-cxr2?select=test>

**Architecture**

In order to test the scaling up in the number of accelerators for this project I decided to construct a custom-made network that complied with three basic requirements. First, the network should be able to solve the problem (i.e., at least having a decent score when the accuracy on the validation/test set was fed into the network), second, the network was supposed to have a decent number of parameters – if possible- so that the process of communication between the different accelerators was not trivial, and lastly the network was to be implemented on TensorFlow.

With the three previous requirements in mind, I decided to implement the following network:

Table

Description automatically generated

Something that is worth noting is that this implementation tries to avoid having mostly all parameters at the beginning of the network -which is a bad design- and instead try to have good spreads between the layers.

**Platforms & Tools Used**

For this project it was needed to develop an algorithm in order to move the files of the dataset files to directories that described their class, for example Testing->Positive or Training->Negative. The reason why moving the files was needed was that the platform used, TensorFlow needs to receive a main directory were objects for either testing or training are contained within subdirectories that have as name their class. In other experiments it could be possible to describe the classes of the dataset as something different like “Testing -> Frog, Testing-> Car, etc.” The algorithm used to move files “around” needs to be placed in the directory were the dataset is located in bridges-2, such directory is /ocean/projects/cis21002p/galindo, and it is attached to the submission of this work named as ***moving.py.*** It is important to point out that ***moving.py*** moves the files by reading the description files of the dataset which are called “test.txt” and “train.txt”.

To perform the experimentations needed for this project the environment that is loaded in bridges-2 by running the command “module load AI” was copied dependency by dependency to create a new environment called “lab5”. Copying the environment provided as part of the module meant that the “lab5” environment now contained the following dependencies:

\* astropy 4.2

\* blas 1.0

\* bokeh 2.2.3

\* cudatoolkit 10.0.130

\* cudnn 7.6.5

\* h5py 2.8.0

\* hdf5 1.10.2

\* ipython 7.19.0

\* jupyter 1.0.0

\* jupyterlab 2.2.6

\* keras-gpu 2.3.1

\* matplotlib 3.3.2

\* mkl 2019.4

\* nccl 1.3.5

\* networkx 2.5

\* ninja 1.10.2

\* nltk 3.5

\* notebook 6.1.6

\* numba 0.51.2

\* numpy 1.17.0

\* opencv 3.4.2

\* pandas 1.2.0

\* pillow 8.1.0

\* pip 20.3.3

\* python 3.7.9

\* pytorch 1.5.0

\* scikit-learn 0.23.2

\* scipy 1.5.2

\* seaborn 0.11.1

\* tensorboard 2.0.0

\* tensorflow-gpu 2.0.0

\* theano 1.0.4

\*Important dependencies are highlighted in red

From the dependencies highlighted in the previous list the TensorFlow-gpu dependency is the most important. With TensorFlow it is possible to import Keras like constructs like those of the layers and the optimization algorithm that is needed. It is also possible to implement the backbone code that provides for the parallelization of the work.

**Implementation and Parallelization**

As it was previously mentioned, TensorFlow and Keras are the main dependencies that provide for the successful implementation of the neural network model. In particular, TensorFlow provides for the Keras-like invocation of layers that build up a network, compilation of the network using an optimizer, and the framework needed for the scaling-up of the neural network training among multiple GPUs.

What follows is the code that implements the custom-made neural network presented before in this work.

Implementation Code:

def cnn\_model(input\_shape=(28,28,1)):

model = Sequential()

model.add(Conv2D(16, (3, 3), input\_shape=input\_shape))

model.add(Conv2D(16, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))

model.add(Conv2D(32, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2),padding='same'))

model.add(Conv2D(64, (3, 3)))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2),padding='same'))

model.add(Conv2D(128, (3, 3)))

model.add(Conv2D(128, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2),padding='same'))

model.add(Conv2D(256, (3, 3)))

model.add(Conv2D(256, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2),padding='same'))

model.add(Flatten())

model.add(Dense(64, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

return model

Compilation Code:

model = cnn\_model(input\_shape=(250,250,3))

model.summary()

opt = RMSprop(lr=0.001)

model.compile(loss="binary\_crossentropy", optimizer=opt, metrics=["accuracy"])

In order to parallelize the training of the neural among “n” number of GPUs the TensorFlow framework provides a very simple set of instructions. First it is needed to define a “MirroredStrategy” construction by invoking such construction from the TensorFlow implementation. And secondly, it is needed to compile the model when the parallel strategy is open, so that the model can be loaded into all the GPUs allocated training and then at every GPU the training process could start.

Following is the implementation of the steps needed to parallelize the training of the network.

mirrored\_strategy = tf.distribute.MirroredStrategy()

devices = tf.config.experimental.list\_physical\_devices("GPU")

print(devices)

with mirrored\_strategy.scope():

model = cnn\_model(input\_shape=(250,250,3))

model.summary()

opt = RMSprop(lr=0.001)

model.compile(loss="binary\_crossentropy", optimizer=opt, metrics=["accuracy"])

**Experimentation Results**

To assess how the performance of the training phase improved by adding more GPUs a baseline time when training on a CPU configuration is introduced. Then, the time taken when 1,2, and 4 GPUs were introduced to the task are included.

CPU Results:

Table

Description automatically generated

Single GPU (Tesla V100) Results:

Table

Description automatically generated

Dual GPU (Tesla V100) Configuration Results:

Table

Description automatically generated with medium confidence

Interconnection Proof:

Graphical user interface

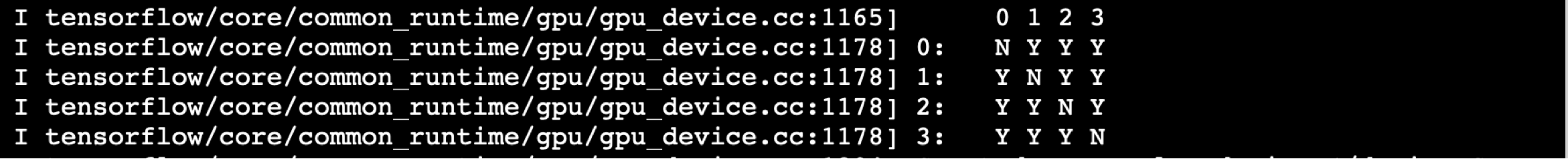
Description automatically generated with medium confidence

Four GPU (Tesla V100) Configuration Results:

Table

Description automatically generated

Interconnection Proof:



**Comments on Communication**

According to the documentation of the TensorFlow’s “MirroredStrategy” construction, the implementation of the communication between the coordinating CPU and all other GPUs is done through the use of NCCL or the NVIDIA Collective Communication Library. This library implements multiple algorithms that utilize all-reduce operations to communicate results between the GPUs and the controlling CPU of the execution. The documentation also mentions that anybody can decide to implement his or her own communication algorithm or choose among other options which are not the default NCCL.

For more details check the use of the parallelization scheme here: <https://www.tensorflow.org/guide/distributed_training>

**Conclusions**

Definitely after performing the experimentations multiple things can be mentioned about the results. First, it is important to mention that the communication cost of the experiments seems to be high, to the point that there is not a gain in terms of time taken by adding more accelerators/GPUs. Second, most likely having thousands of images is not enough to “hide” the communication cost of the GPUs, probably having a dataset of, say, more than 1,000,000 images the communication would be negligible compared to the gains in terms of computation done in parallel. Thirdly, since communication cost seems to be high for any operation two other points could be of interest, first see if the communication scheme used is optimal, and second check if the load used is not too small for a single GPU, if it is, then most likely, even by increasing the batch size – therefore increasing the use of memory per step of training- and then increasing the number of GPUs utilized would not show a speedup of the computations.