IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-2021)

Performance Comparison of TPU, GPU, CPU on Google Colaboratory over Distributed Deep Learning

Haklin Kimm, Incheon Paik, Hanke Kimm,
Computer Science Department
East Stroudsburg University, USA
University of Aizu, Japan
Stony Brook University USA

December 20-23, 2021

What is Deep Learning ?

- Deep Learning is a subfield of machine learning based on algorithms inspired by artificial neural networks.
- Deep Learning models
  - need massive amount of compute powers and
  - tend to improve performance running on special purpose processors **accelerators** *designed to speed up compute-intensive applications.*
  - The accelerators like Tensor Processing Units (**TPU**s) and Graphics Processing Units (**GPU**s) are widely used.

In this paper…

- Distributed Bidirectional Recurrent Neural Network (RNN) where each layer is filled with Long Short-Term Memory (LSTM) units are introduced.
- Comparison of the hardware platforms on **Google Colaboratory** using distributed Bidirectional Long Short-Term Memory (**dBLSTM**) models
  - upon the number of layers, the number of units each layer, and the numbers of input and output units.

Google Colaboratory

- CPU, TPU, and GPU are available in Google cloud.
- The maximum lifetime of a VM on Google Colab is **12 hours with 90-min** idle time.
- **Free CPU** for Google Colab is equipped with 2-core Intel Xeon @2.0GHz and 13GB of RAM and 33GB HDD.
- **Free GPU** on Google Colab is Tesla K80, dual-chip graphics card, having **2496 CUDA** cores and 12GB GDDR5 VRAM and base clock runs at 560MHz.
  - A single CPU, hyperthreaded Xeon Processors @2.3Ghz, is provided to work with the CUDA cores.

Google Colaboratory

- The TPU is a custom ASIC developed by Google.
  - Consisting of the computational resources of Matrix Multipliers Unit (MXU): 65536 8-bit multiply-and-add units, Unified Buffer (UB): 24MB of SRAM, Activation Unit (AU): Hardwired activation functions.
- TPU v2 delivers a peak of **180** TFLOPS on a single board with 64GB of memory per board
- TPU v3 provides a peak performance up to **420** TFLOPs.
- The TPU Matrix Multiplication Unit has a systolic array mechanism that contains 256 × 256 = total 65,536 ALUs.
  - TPU can process 65,536 multiply-and-adds for 8-bit integers every cycle.
  - Because a TPU runs at 700MHz, a TPU can compute 65,536 × 700,000,000 = 46 × 1012 multiply-and-add operations or **92 TFLOPs** per second (92 × 1012).

Related Works

- DL models that have been developed and used widely are
  - Fully Connected Neural Networks (FC),
  - Convolutional Neural Networks (CNN),
  - Recurrent Neural Networks (RNN),
  - Combined neural networks of the known ones.

Convolutional Neural Network

- Deep Learning HAR research works
  - have been chiefly focused on building efficient Convolutional Neural Networks (CNN) for various applications,
  - are used to process the stacked sequences of vision data mostly obtained by fixed sensors, or to implement motion activity datasets acquired by mobile sensors.

Recurrent Neural Network

- The long-range temporal information cannot be well captured in CNN.
- RNNs are distinguished by their "**memory**" that takes in **previous inputs** to influence the current input and output; it works with the **present and immediate past inputs** such a way that information cycles through a loop.
- LSTM (long short-term memory) or GRU (Gated Recurrent Unit) cells are applied to process the sequence information in RNNs

Recurrent Neural Network

- RNN model that generates an output at each time step and provides recurrent connections only form the output at one-time step to the hidden units at the next-time step.
- Basic formula of RNN:

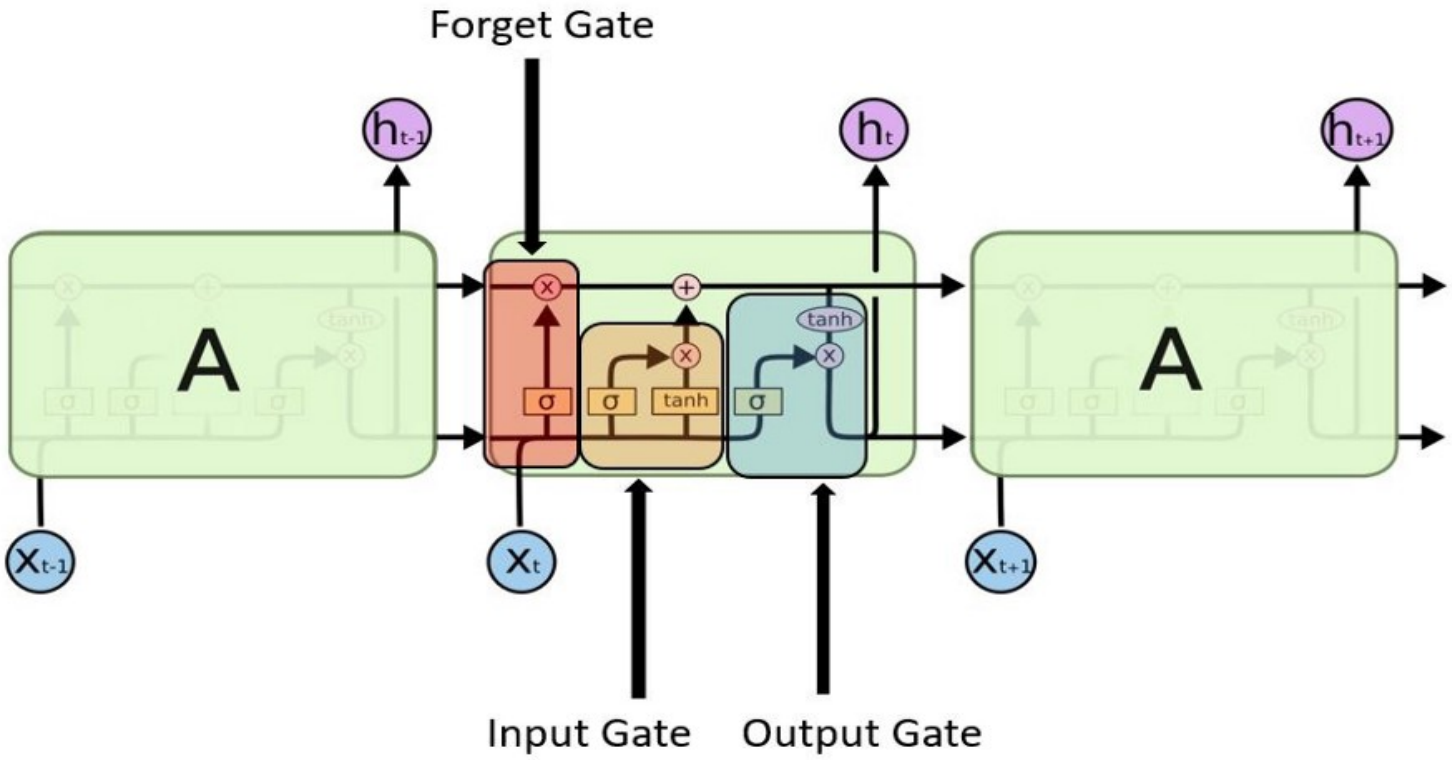
$$\boldsymbol{h(t) = f(h(t-1), x(t); \theta)}$$

- where ***h(t)*** is the current hidden state, ***h(t-1)*** is the previous hidden state, *x(t)* is the current input, and $\boldsymbol{\theta}$ is the parameters of the function $\boldsymbol{f}$.

What is Long Short Term Memory (LSTM) ?

- Long short-term memory (LSTM) is with:
  - input gate, forget gate, and output gate
  - information to be forgotten, remembered, and updated.
- Gated state or gated memory is incorporated into LSTMs and gated recurrent units (GRU).
- Bloomberg Business Week wrote: “These powers make LSTM arguably the most commercial AI achievement, used for everything from predicting diseases to composing music”.

LSTM Architecture



Distributed DL on Google Colab

- TensorFlow works well with Python, Java, Go, C++, etc.
- Supports both model parallelism and data parallelism.
- Most distributed DL applications work with data parallelism.
- At this time TensorFlow only supports data parallelism.

Distributed Training using TensorFlow

- ***MirroredStrategy*** supports synchronous distributed training on multiple on one machine, and all the variables in the DL model is mirrored across all the replicas.
- ***TPUStrategy*** provides their own implementation of efficient all-reduce and other collective operations across multiple TPU cores.
- ***MultiWorkerMirroredStrategy*** is very similar to *MirroredStrategy*, which also implements synchronous distributed training across multiple workers. The copies of all variables in the DL model on each device are created across all workers.
- ***ParameterServerStrategy*** is a common data-parallel method to scale up model training on multiple machines. A parameter server training cluster consists of workers and parameter servers.

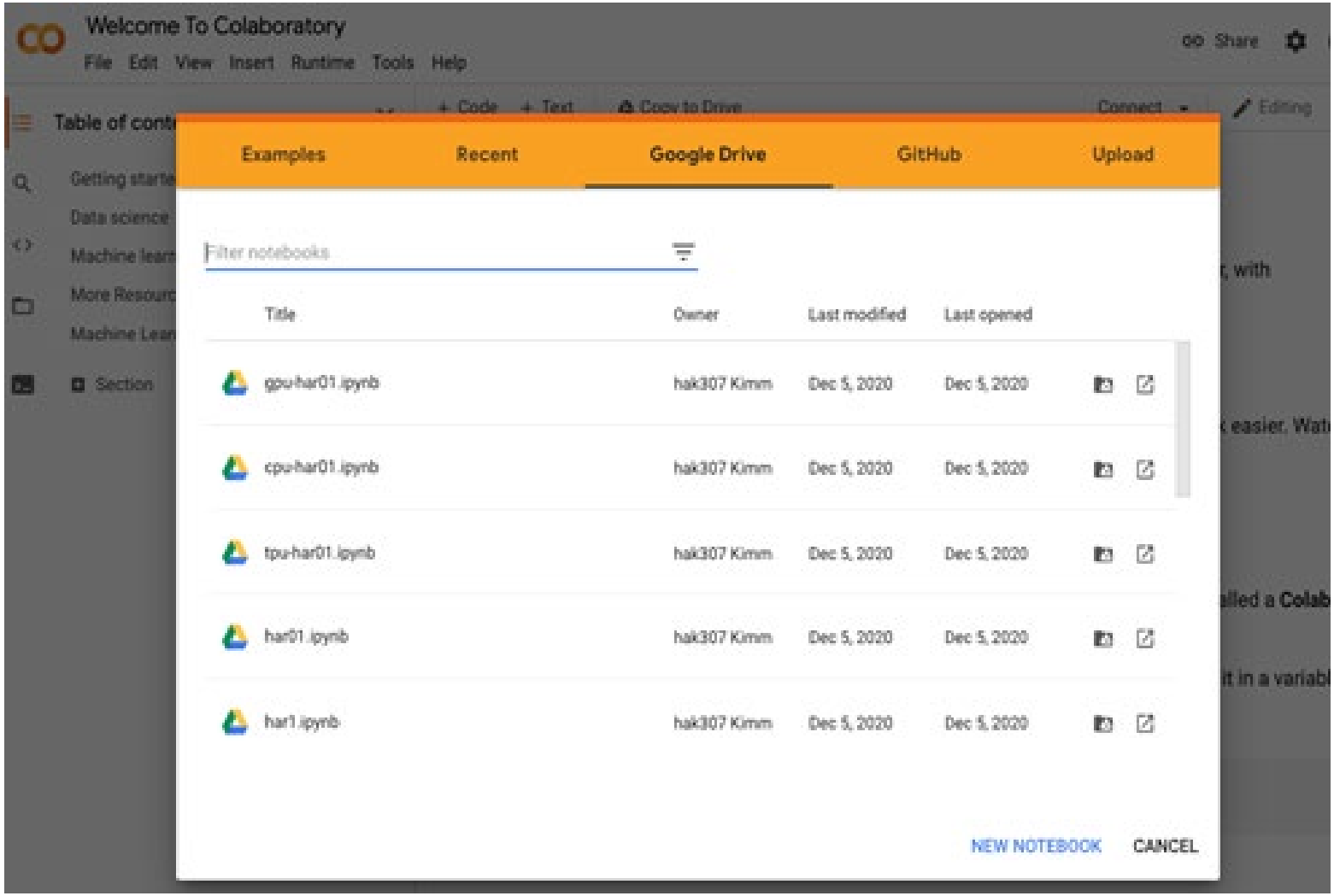
Benchmarking Colab Hardware Platforms

- Benchmarking TPU, GPU, and CPU platforms for DL has been well studied by Carneiro, et al. and Y. Wang, et al.
- Carneiro provided a performance analysis of Google Colab as tool running compute-intensive applications on Colab Tesla K80 and a local Tesla K40.
- Wang provided Parameterized deep learning benchmark suite (ParaDnn) was introduced that generates multi-layer models with thousands of parameters.
  - This benchmark suite has been tested upon local GPU platforms and Google cloud CPU and TPU platforms.

Distributed DL Implementation

- TensorFlow 2.X and Python 3 with Google Colab has been used, which is accessed by MacBook Pro equipped with 2.5 GHz Quad-Core Intel Core i7 and 16 GB 1600 MHz DDR3, working on macOS BigSur.
- Data Setup
  - Human Activity Recognition (HAR) Using Smartphones Dataset from UCI Repository.
  - 30 Volunteers each performing 6 activities WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING
  - Dataset randomly partitioned into two sets, *70% selected for generating test data and 30% selected for test data*.

Hardware Platforms on Colab



TPU on Google Colab

- **TPU** in this example, TPU is selected to run on Google Colab. The following code is implemented to run the proposed DL model on the Colab TPU.

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver
(tpu='grpc://'     + os.environ['COLAB_TPU_ADDR'])
tf.config.experimental_connect_to_cluster(resolver)

# This is the TPU initialization code that has to be at the
beginning.
tf.tpu.experimental.initialize_tpu_system(resolver)
print("All devices: ", tf.config.list_logical_devices('TPU'))

strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

GPU on Google Colab

- **GPU** in this example, GPU is selected to run on Google Colab from the dropdown *Hardware accelerator* selector in the *Runtime* tab in the menu bar. The following code is executed to run the proposed DL model on the Colab GPU.

```
device_name = tf.test.gpu_device_name()

if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')

print('Found GPU at: {}'.format(device_name))
print("Num GPUs Available: ",
      len(tf.config.experimental.list_physical_devices('GPU')))
```

CPU on Google Colab

- **CPU** as an example, none is selected to run on Google Colab from the dropdown *Hardware accelerator* selector in the *Runtime* tab in the menu bar.
- There is no need to add any extra code to implement the proposed DL in case of running on the Colab CPU.

Distributed DL with Distributed Strategy

```
def evaluate_model(trainX, trainy, testX, testy):

  with strategy.scope():
    verbose, epochs, batch_size = 0, 30, 64
    n_timesteps, n_features, n_outputs = trainX.shape[1], ...
    # build dBLSTM model
                        model = Sequential()
    model.add(Bidirectional(LSTM(128, return_sequences=True, ...)))
    model.add(Bidirectional(LSTM(128, return_sequences=True)))
    model.add(LSTM(64))
    model.add(Dense(n_outputs, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', ...)
    # fit network
    model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, ...)
    model.summary()
    # evaluate model
     _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, ...)
return accuracy
```

Distributed DL with Distributed Strategy

- Distributed *strategy.scope()* connected to the eight-core TPU as default.
- This model begins with batch size = 64 so that the DL is trained with 64 samples at a time, with step size = 128 matching with unit size = 128 of each LSTM layer.
- The final layer on our DL adapts ***activation = softmax*** that is used for predicting a corresponding activity; optimizer = *adam* is used with loss = *categorical_ crossentropy* to see whether the DL's prediction matches with one of six activities.

Distributed DL with Distributed Strategy

- Softmax activation function turns numbers into **probabilities that should sum to one**, and outputs a vector that represents the probability distributions of a list of potential outcomes.
- Categorical_crossentropy loss function produces a **one-hot array** containing the probable match for each category.
- In our DDL model, i.e., the one-hot target vector is assumed to be [0, 1, 0, 0, 0, 0] and the model to predict [.01, .91, .03, .01, .04, .01].

Distributed DL Model on TPU

```
# summarize scores
def summarize_results(scores):
        print(scores)
        m, s = mean(scores), std(scores)
        print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))

# run an experiment
def run_experiment(repeats):
  # load data
        trainX, trainy, testX, testy = load_dataset()
  # repeat experiment
        scores = list()
        for r in range(repeats):
                score = evaluate_model(trainX, trainy, testX, testy)
                score = score * 100.0
                print('>#%d: %.3f' % (r+1, score))
                scores.append(score)
# summarize results
        summarize_results(scores)
# run the experiment
        start = time.perf_counter()
        run_experiment()
        elapsed = time.perf_counter() - start
        print('Elapsed %.3f seconds.' % elapsed)
```

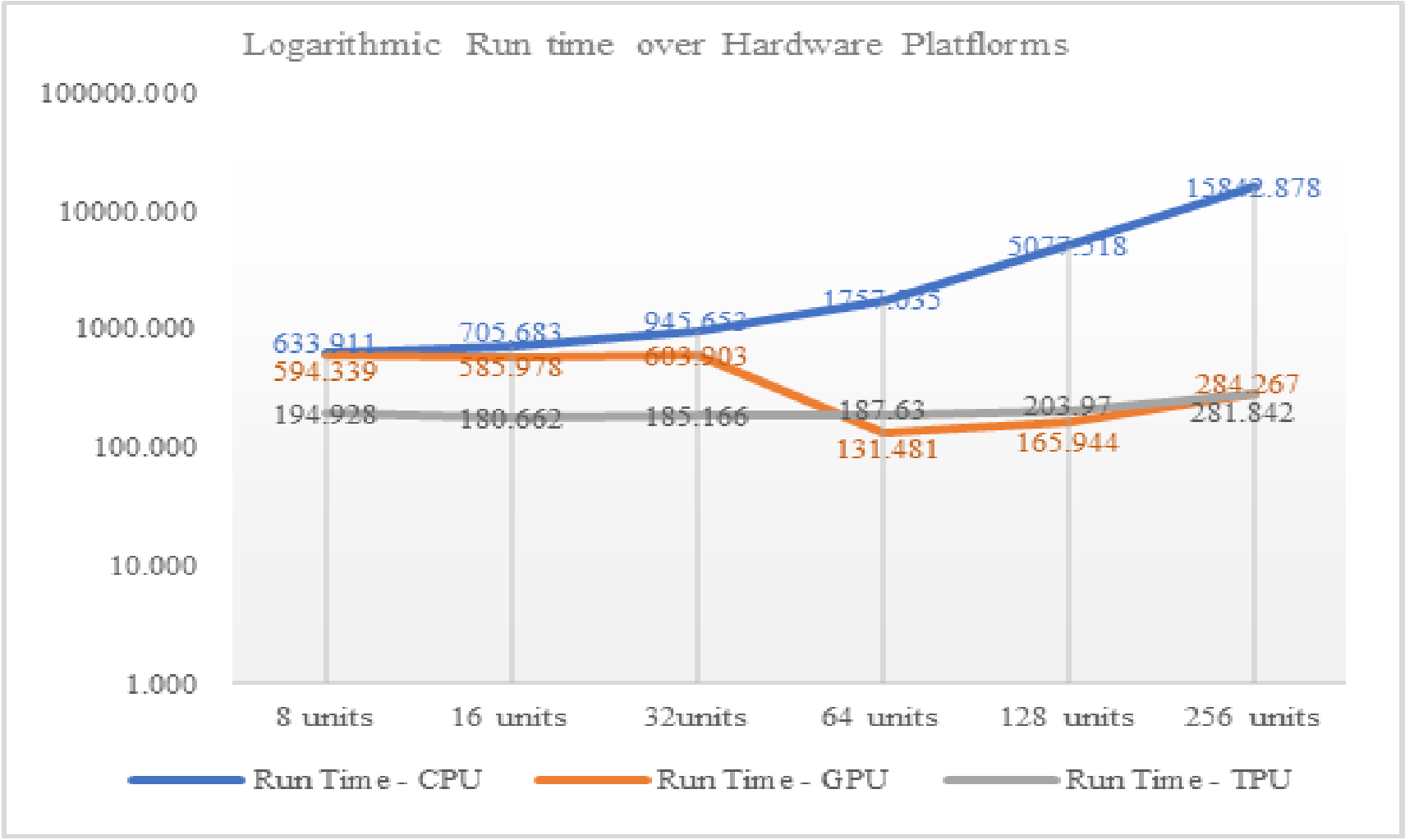
Distributed DL Model on Colab

```
def evaluate_model(trainX, trainy, testX, testy):
        ...
    model = Sequential()
        ...
    model.compile(
    loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy',tf.keras.metrics.Precision(),tf.keras.metrics.Recall()])
    # fit network
    history = model.fit (trainX, trainy, ..., verbose=0)
    plt.plot(history.history['loss'])
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['precision'])
    plt.plot(history.history['recall'])
        ...
    model.summary()
    # evaluate model
    loss, accuracy, precision, recall = model.evaluate(testX, testy, ...)
    return accuracy, precision, recall
```

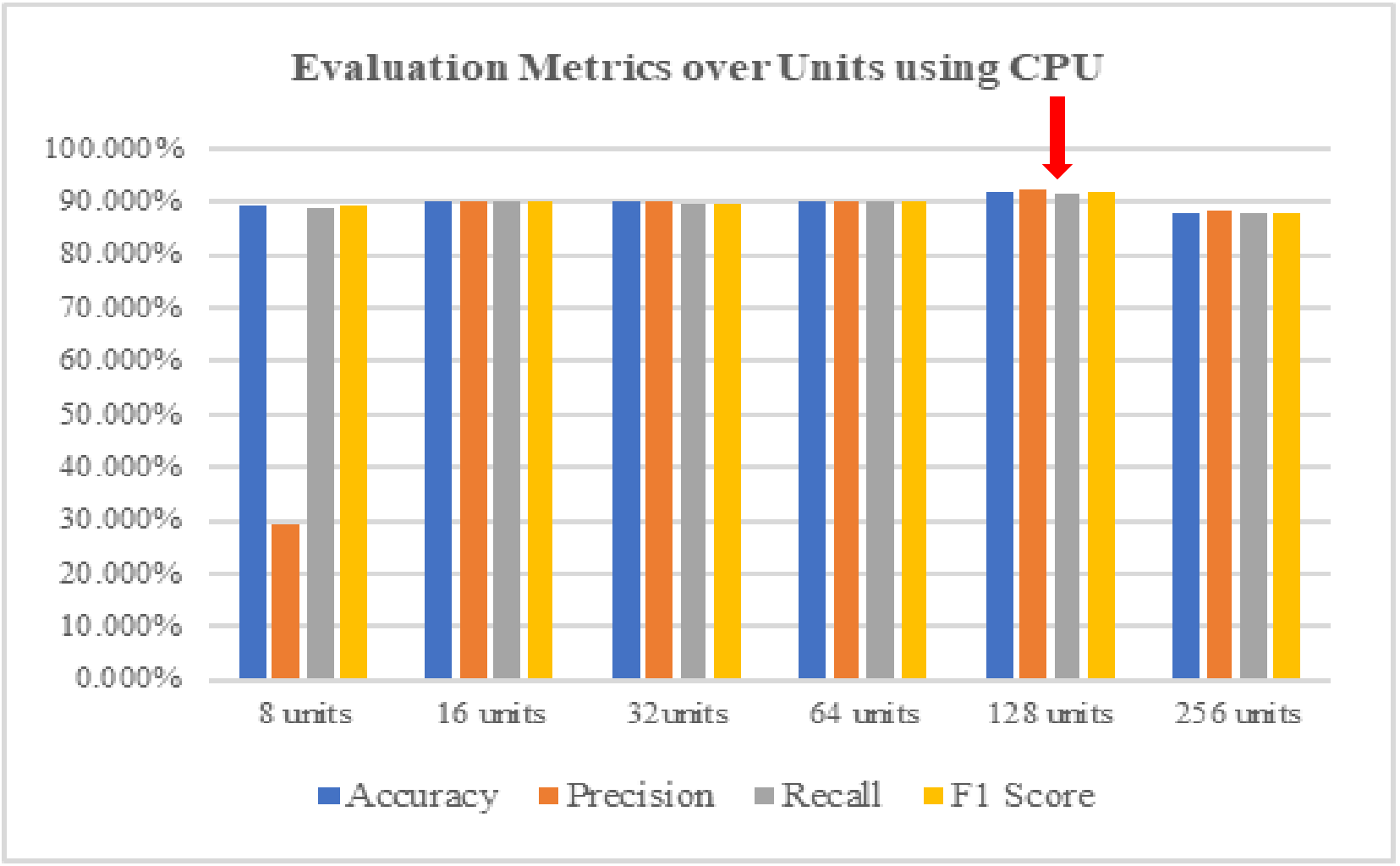
Performance Measures

- **Accuracy** is the most intuitive performance measure, and it is simply a ratio of **correctly predicted observation to the total observations**.
- **Precision** is the ratio of **correctly predicted positive observations to the total predicted positive observations**.
- **Recall** is the ratio of **correctly predicted positive observations to all observations** in actual class as true.
- **F1 Score** is the **weighted average of Precision and Recall**. Therefore, this score takes both false positives and false negatives into account.

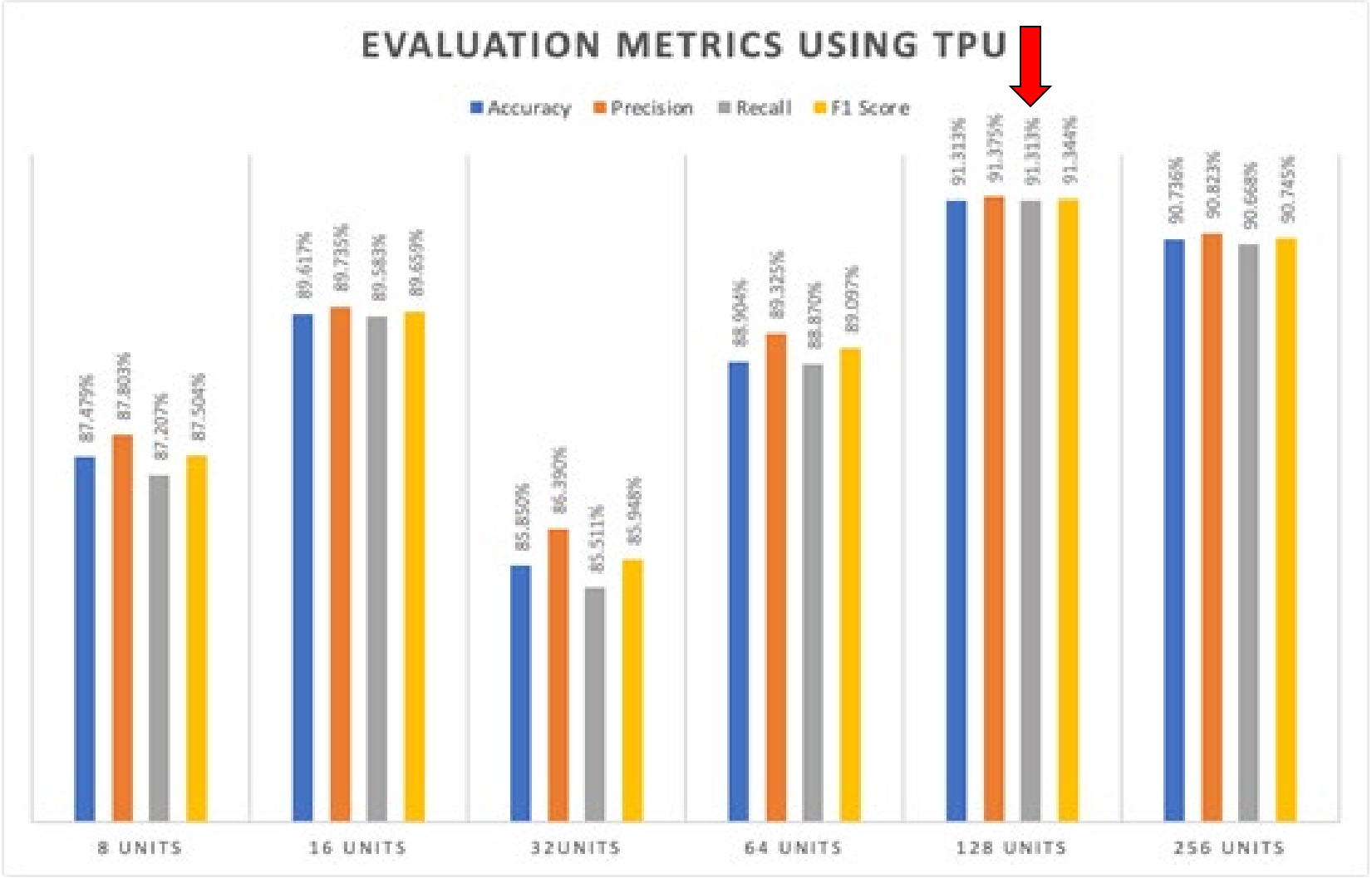
Runtime comparison



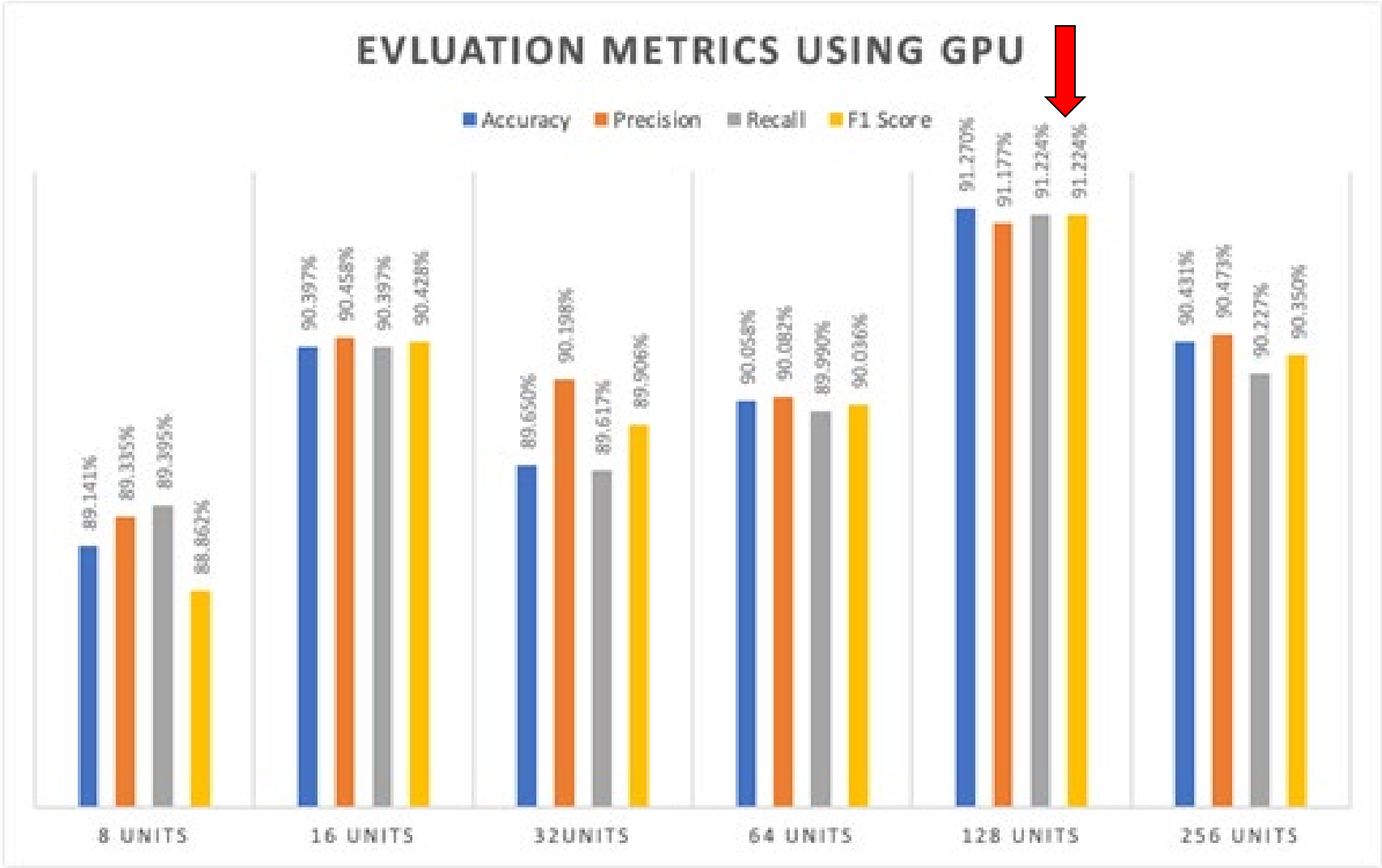
Evaluation Metrics over CPU



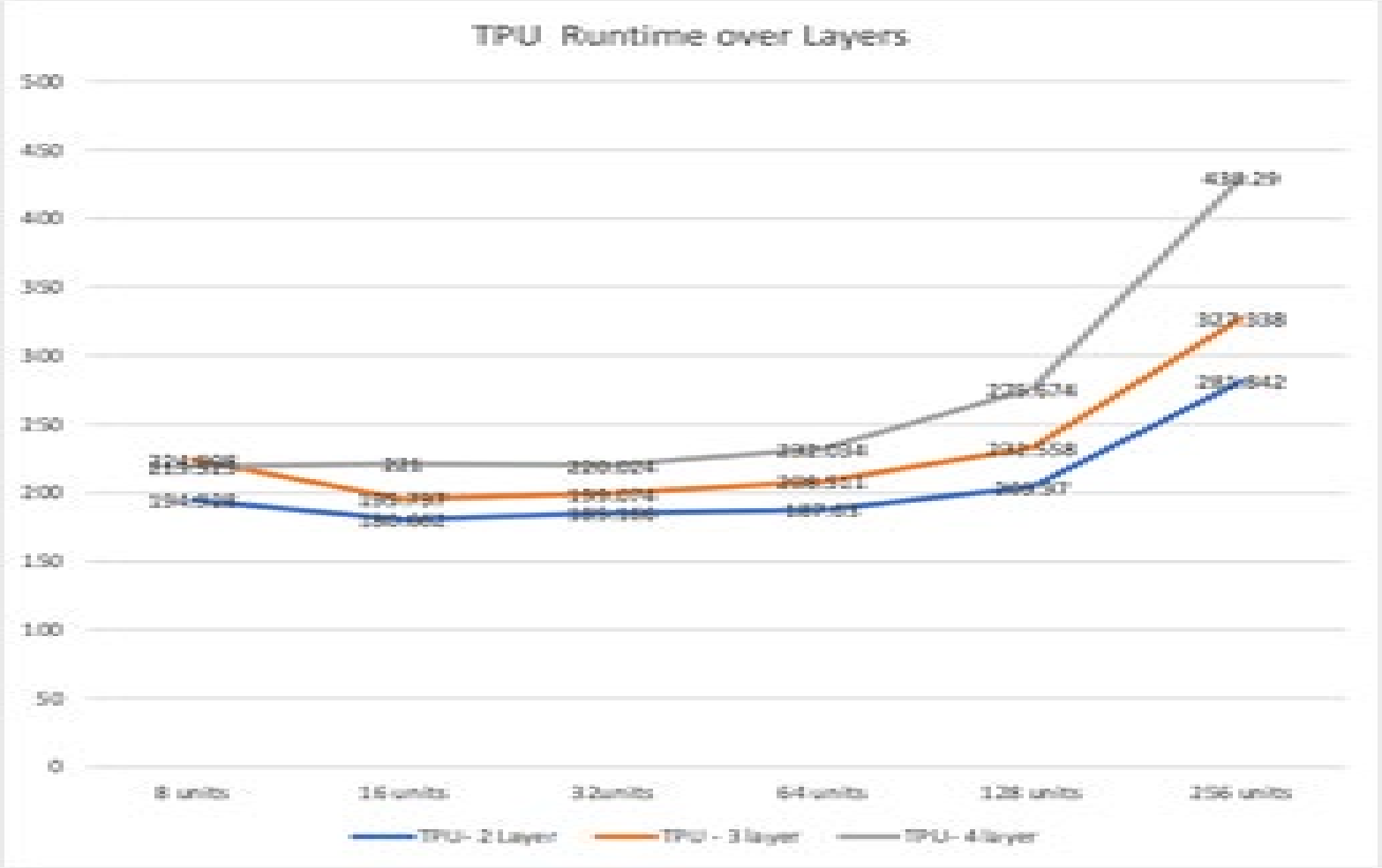
Evaluation Metrics over TPU



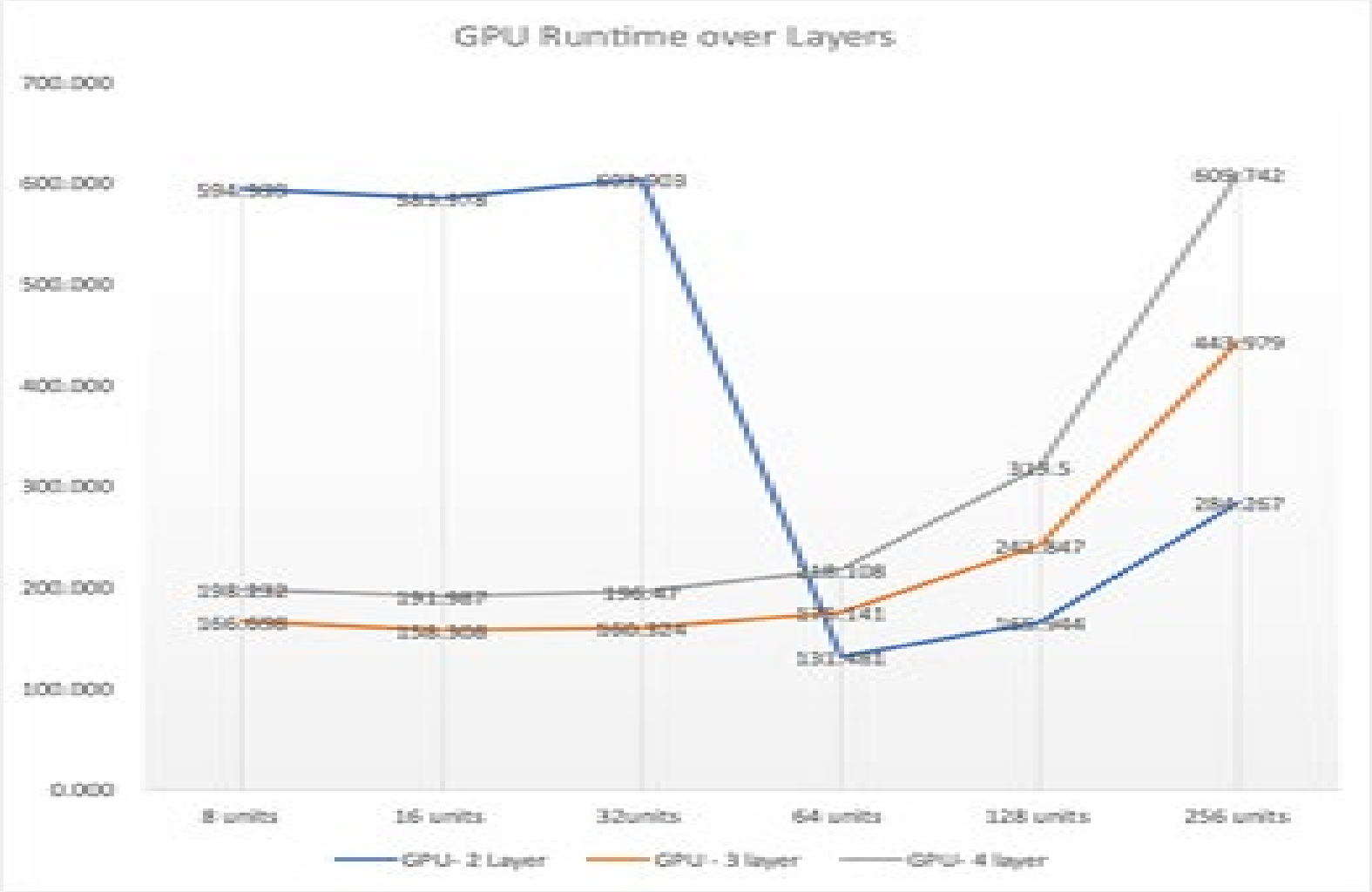
Evaluation Metrics over GPU



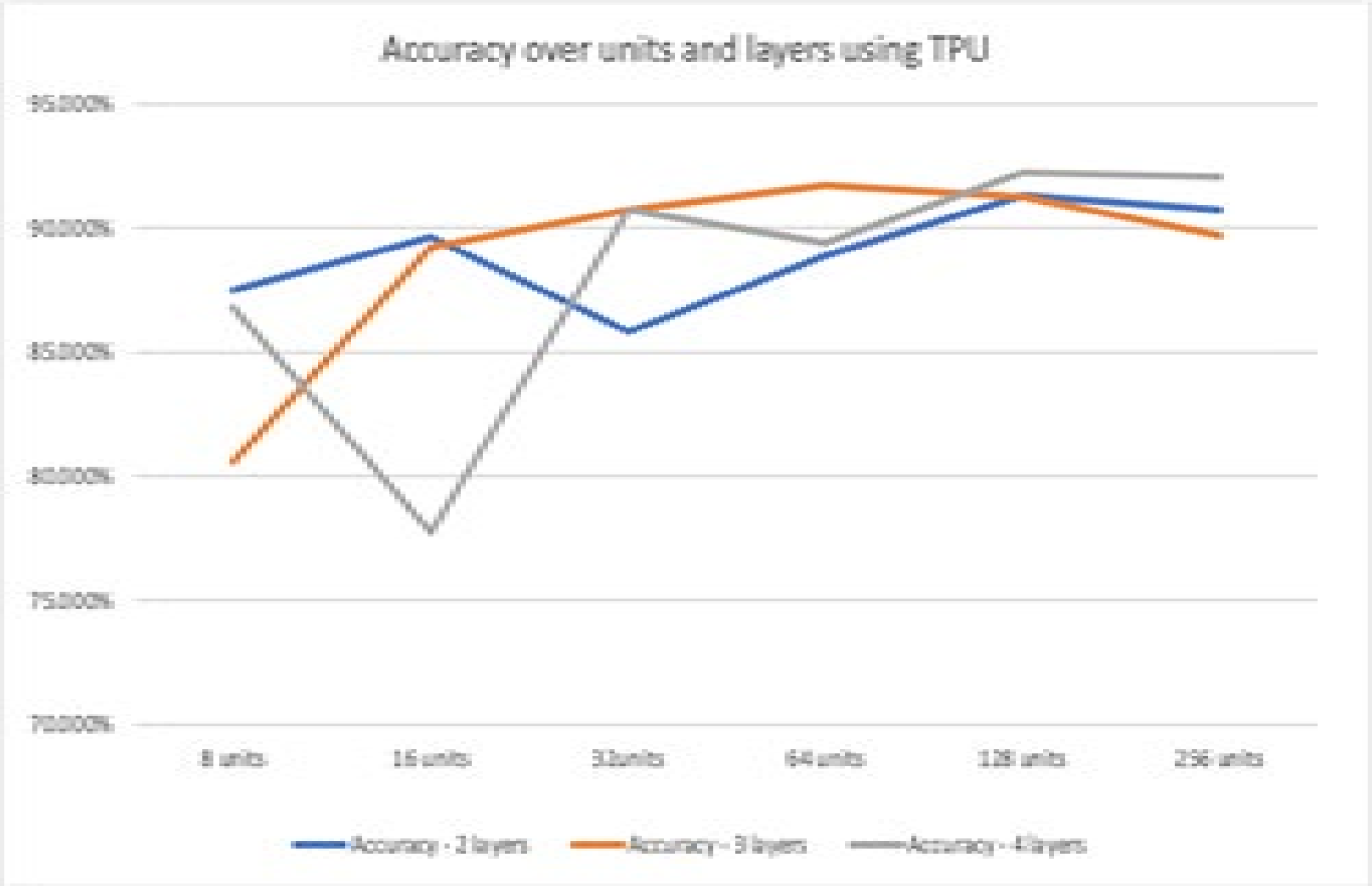
Runtime comparison over TPU layers



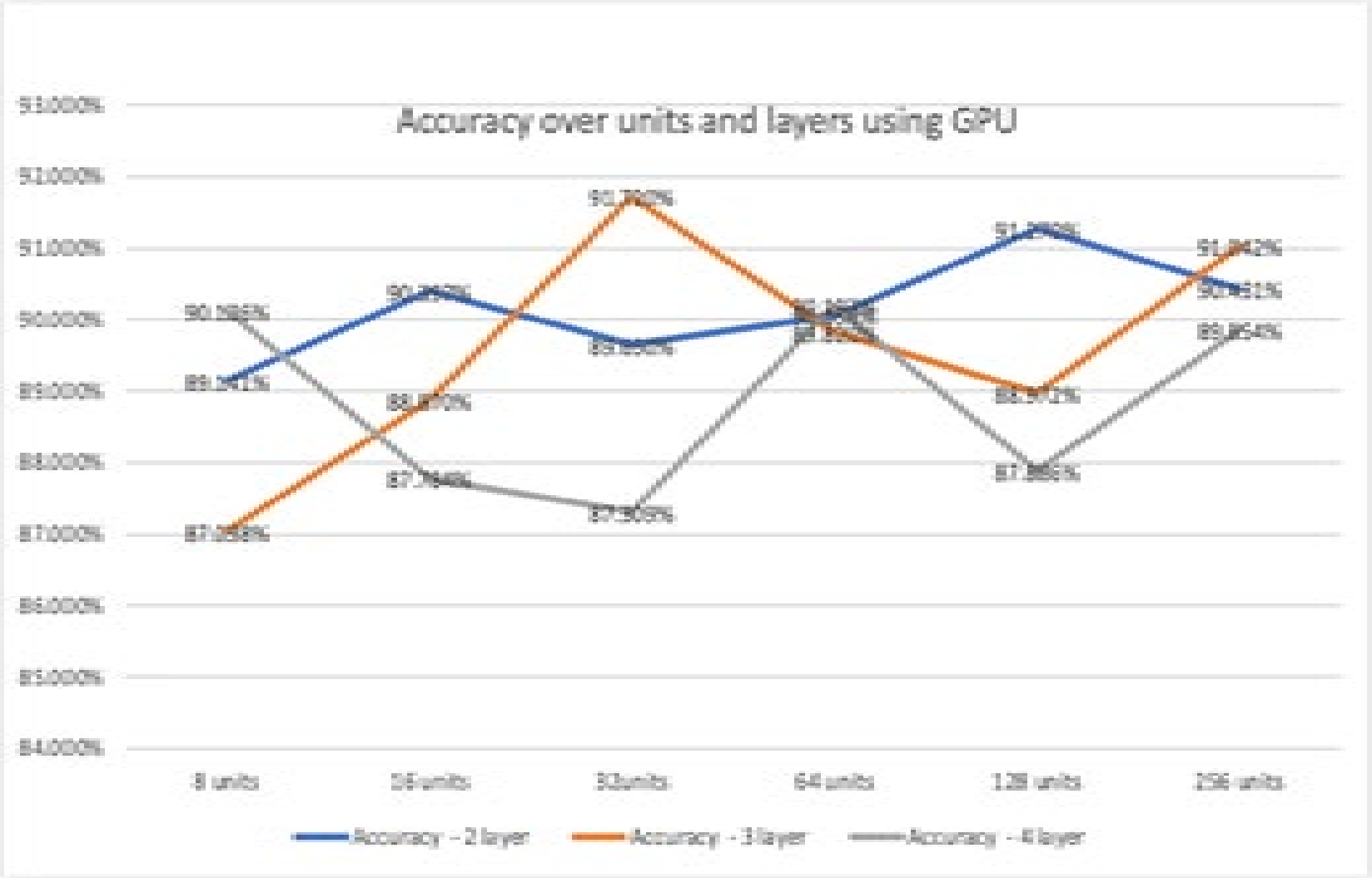
Runtime comparison over GPU layers



Accuracy comparison over TPU



Accuracy comparison over GPU



Conclusion

- The dBLSTM models with multiple units and layers have been applied to CPU, GPU, TPU with UCI-HAR dataset for benchmarking hardware platforms, where batch size was applied to 128 trying to test TPU and GPU evenly.
- DL models with large batch size prefer TPU, but with small batch size works prefer GPU.
- CPU has shown its runtime increases starting from the unit 64.
- In case of using 128 units, the runtime of CPU is slower 25 times than the runtimes of TPU and GPU.
- In case of using 256 units, and the CPU is slower 55 times than the TPU and GPU run

Conclusion

- TPU shows better runtimes as the number of units is increased than the GPU.
- However, GPU works better on the smaller number of units than TPU.
- TPU shows best accuracies on 128 units with all three layers and GPU shows the best accuracy on 32 units with 3 layers.

Summary

- Colab TPU works better with larger batches and hyperparameters,
- Colab GPU performs well with smaller batches,
- Colab CPU works well training the models but takes longer runtimes as the hyperparameters are increased.