# CheXNet – Inference with Nvidia T4 on Dell EMC PowerEdge R7425

Abstract

This whitepaper looks at how to implement inferencing using GPUs. This work is based on CheXNet model developed by Stanford University to detect pneumonia. This paper describes the utilization of trained model and TensorRT™ to perform inferencing using Nvidia T4 GPUs.

June 2019

# Revisions

| Date | Description |
|---|---|
| June 2019 | Initial release |
| | |

# Acknowledgements

This paper was produced by the following members of the Dell EMC SIS team:

Authors: Bhavesh Patel, Vilmara Sanchez [Dell EMC Advanced Engineering]

Support: Josh Anderson [Dell EMC System Engineer]

Others:

Nvidia account team for their expedited support

Nvidia Developer forum

TensorFlow -TensorRT Integration Forum

Dell EMC HPC Engineering team {Lucas A. Wilson, Srinivas Varadharajan, Alex Filby and Quy Ta}

# Table of contents

# Executive summary

The Healthcare industry has been one of the leading-edge industries to adopt techniques related to machine learning and deep learning to improve diagnosis, provide higher level of accuracies in term of detection and reduce overall cost related to mis-diagnosis. Deep Learning consists of two phases: training and inference. Training involves learning a neural network model from a given training dataset over a certain number of training iterations and loss function. The output of this phase, the learned model, is then used in the inference phase to speculate on new data. For the training phase, we leveraged the CheXNet model developed by Stanford University ML Group to detect pneumonia which outperformed a panel of radiologists [1]. We used National Institutes of Health (NIH) Chest X-ray dataset which consist of 112,120 images labeled with 14 different thoracic diseases including pneumonia. All images are labeled with either single or multiple pathologies, making it a multi-label classification problem. Images in the Chest X-ray dataset are 3 channel (RGB) with dimensions 1024x1024.

We trained CheXNet model on (NIH) Chest X-ray dataset using Dell EMC PowerEdge C4140 with NVIDIA V100-SXM2 GPU server. For inference we used Nvidia TensorRT™, a high-performance deep learning inference optimizer and runtime that delivers low latency and high-throughput. In this project, we have used the CheXNet model as reference to train a custom model from scratch and classify 14 different thoracic deceases, and the TensorRT™ tool to optimize the model and accelerate its inference.

The objective is to show how PowerEdge R7425 can be used as a scale-up inferencing server to run production-level deep learning inference workloads. Here we will show how to train a CheXNet model and run optimized inference with Nvidia TensorRT™ on Dell EMC PowerEdge R7425 server.

The topics explained here are presented from development perspective, explaining the different TensorRT™ implementation tools at the coding level to optimize the inference CheXNet model. During the tests, we ran inference workloads on PowerEdge R7425 with several configurations. TensorFlow was used as the primary framework to train the model and run the inferences, the performance was measured in terms of throughput (images/sec) and latency (milliseconds).

# 1 Background & Definitions

Deploying AI applications into production sometimes requires high throughput at the lowest latency. The models generally are trained in 32-bit floating point (fp32) precision mode but need to be deployed for inference at lower precision mode without losing significant accuracy. Using lower bit precision like 8-bit integer (int8) gives higher throughput because of low memory requirements. As a solution, Nvidia has developed the TensorRT™ Inference optimization tool, it minimizes loss of accuracy when quantizing trained model weights to int8 and during int8 computation of activations it generates inference graphs with optimal scaling factor from fp32 to int8. We will walk through the inference optimization process with a custom model, covering the key components involved in this project and described in the sections below. See **Figure** 1

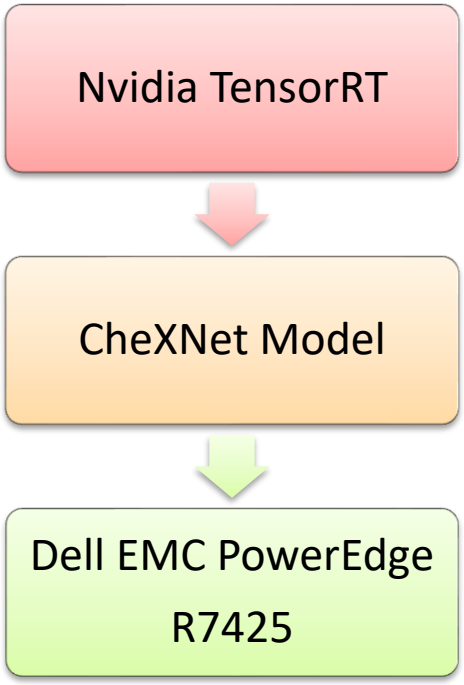


Figure 1:Inference Implementation

| Deep learning |
| --- |
| Deep Learning (DL) is a subfield of Artificial Intelligent and Machine Learning (ML), based on methods to learn data representations; deep learning architectures like convolutional neural networks (CNN) and Recurrent Neural Networks (RNN) among others have been successfully applied to applications such as computer vision, speech recognition, and machine language translation producing results comparable to human experts. |
| TensorFlow |

TensorFlow™ is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning libraries and the flexible numerical computation core is used across many other scientific domains.

## Transfer Learning

Transfer Learning is a technique that shortcuts the training process by taking portion of a model and reusing it in a new neural model. The pre-trained model is used to initialize a training process and start from there. Transfer Learning is useful when training on small datasets.

## TensorRT™

Nvidia TensorRT™ is a high-performance deep learning inference and run-time optimizer delivering low latency and high throughput for production model deployment. TensorRT™ has been successfully used in a wide range of applications including autonomous vehicles, robotics, video analytics, automatic speech recognition among others. TensorRT™ supports Turing Tensor Cores and expands the set of neural network optimizations for multi-precision workloads. With the TensorRT™ 5, DL applications can be optimized and calibrated for lower precision with high throughout and accuracy for production deployment.

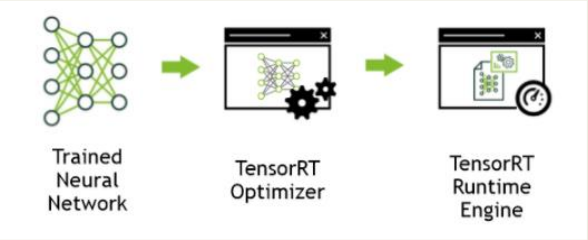


Figure 2:TensorRT™ scheme. Source: Nvidia

In **Figure** 2 we present the general scheme of how TensorRT™ works. TensorRT™ optimizes an already trained neural network by combining layers, fusing tensors, and optimizing kernel selection for improved latency, throughput, power efficiency and memory consumption. It also optimizes the network and generate runtime engines in lower precision to increase performance.

## CheXNet

CheXNet is a Deep Learning based model for Radiologist-Level Pneumonia Detection on Chest X-Rays, developed by the Stanford University ML Group and trained on the Chest X-Ray dataset. For the pneumonia detection, the ML group have labeled the images that have pneumonia as the positive examples and labeled all other images with other pathologies as negative examples.

NIH Chest X-ray Dataset

The National Institutes of Health released the NIH Chest X-ray Dataset, which includes 112,120 X-ray images from 30,805 unique patients, and labeled with 14 different thoracic deceases through the application of Natural Language Processing algorithm to text-mine disease classification from the original radiological reports.

## 1.1 Dell EMC PowerEdge R7425

Dell EMC PowerEdge R7425 server supports the latest GPU accelerators to speed results in data analytics and AI applications. It enables fast workload performance on more cores for cutting edge applications such Artificial Intelligence (AI), High Performance Computing (HPC), and scale up software defined deployments. See **Figure** 3


Figure 3:DELL EMC PowerEdge R7425

The Dell™ PowerEdge™ R7425 is Dell EMC's 2-socket, 2U rack server designed to run complex workloads using highly scalable memory, I/O, and network options The systems feature AMD high performance processors, named AMD SP3, which support up to 32 AMD "Zen" x86 cores (AMD Naples Zeppelin SP3), up to 16 DIMMs, PCI Express® (PCIe) 3.0 enabled expansion slots, and a choice of OCP technologies.

The PowerEdge R7425 is a general-purpose platform capable of handling demanding workloads and applications, such as VDI cloud client computing, database/in-line analytics, scale up software defined environments, and high-performance computing (HPC).

The PowerEdge R7425 adds extraordinary storage capacity options, making it well-suited for data intensive applications that require greater storage, while not sacrificing I/O performance.

# 2 Test Methodology

In this project we ran image classification inference for the custom model CheXNet on the PowerEdge R7425 server in different precision modes and software configurations: with TensorFlow-CPU support only, TensorFlow-GPU support, TensorFlow with TensorRT™, and native TensorRT™. Using different settings, we were able to compare the throughput and latency and expose the capacity of PowerEdge R7425 server when running inference with Nvidia TensorRT™. See **Figure 4**

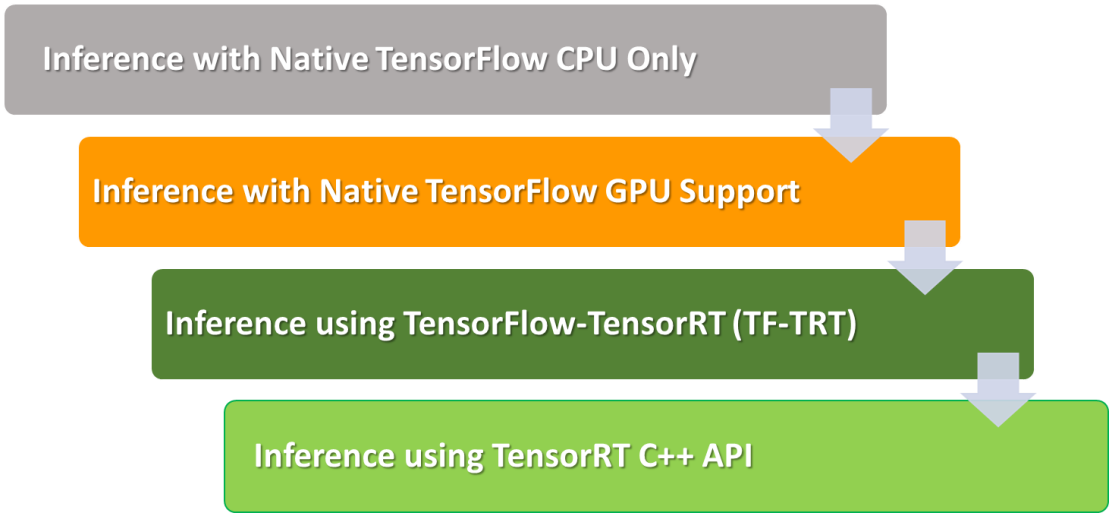


Figure 4:Test Methodology for Inference

## 2.1 Test Design

The workflow pipeline started with the training of the custom model from scratch until running the optimized inference graphs in multi-precision modes and configurations. To do so, we followed the below the steps:

a) Building the CheXNet model with TensorFlow, transfer learning & estimator.
b) Training the Model for Inference
c) Saving Trained Model with TensorFlow Serving for Inference
d) Freezing the Saved Model
e) Running the Inference with Native TensorFlow CPU Only
f) Running the Inference with Native TensorFlow GPU Support
g) Converting the Custom Model to Run Inference with TensorRT™
h) Running Inference using TensorFlow-TensorRT (TF-TRT) Integration
i) Running Inference using TensorRT™ C++ API
j) Comparing Inferences in multi-mode configurations

**Table 1** shows the summary of the project design below:

Table 1:Project Design Summary

| Element | Description |
|---|---|
| **Use Case:** | Optimized Inference Image Classification with TensorFlow and TensorRT™ |
| **Models:** | Custom Model CheXNet and base model ResnetV2_50 |
| **Framework:** | TensorFlow 1.0 |
| **TensorRT™ version:** | TensorRT™ 5.0 |
| **TensorRT™ implementations:** | TensorFlow-TensorRT Integration (TF-TRT) and TensorRT C++ API (TRT) |
| **Precision Modes:** | ▪ Native TensorFlow FP32 – CPU Only<br>▪ Native TensorFlow FP32 - GPU<br>▪ TF-TRT-FP32<br>▪ TF-TRT-FP16<br>▪ TF-TRT-INT8<br>▪ TRT-INT8 |
| **Performance:** | Throughput (images per second) and the Latency (msec) |
| **Dataset:** | NIH Chest X-ray Dataset from the National Institutes of Health |
| **Samples code:** | TensorRT™ samples provided by Nvidia included on its container images, and adapted to run the optimized inference of the custom model |
| **Software stack configuration:** | Tests conducted using the docker container environment |
| **Server:** | Dell EMC PowerEdge R7425 |

**Table 2** lists the tests conducted to train the model, and inferences in different precision modes with the TensorRT™ implementations. The script samples can be found within the Nvidia container image.

Table 2. Tests Conducted

| Model/Inference Mode | TensorRT™ Implementation | Test script |
|---|---|---|
| Custom Model | n/a | chexnet.py |
| Native TensorFlow CPU FP32 | n/a | tensorrt_chexnet.py |
| Native TensorFlow GPU FP32 | n/a | tensorrt_chexnet.py |
| Integration TF-TRT5 FP32 | TF-TRT Integration | tensorrt_chexnet.py |
| Integration TF-TRT5 FP16 | TF-TRT Integration | tensorrt_chexnet.py |
| Integration TF-TRT5 INT8 | TF-TRT Integration | tensorrt_chexnet.py |
| Native TRT5 INT8 – C++ API | C++ API | trtexec.cpp |

## 2.2 Test Setup

a) For the hardware, we selected PowerEdge 7425 which includes the Nvidia Tesla T4 GPU, the most advanced accelerator for AI inference workloads. According to Nvidia, T4's new Turing Tensor cores accelerate int8 precision more than 2x faster than the previous generation low-power offering [2].
b) For the framework and inference optimizer tools, we selected TensorFlow, TF-TRT integrated and TensorRT C++ API, since they have better technical support and a wide variety of pre-trained models are readily available.
c) Most of the tests were run in int8 precision mode, since it has significantly lower precision and dynamic range than fp32, as well as lower memory requirements; therefore, it allows higher throughput at lower latency.

**Table 3** shows the software stack configuration on PowerEdge R7425

Table 3. OS and Software Stack Configuration

| Software | Version |
|---|---|
| OS | Ubuntu 16.04.5 LTS |
| Kernel | GNU/Linux 4.4.0-133-generic x86_64 |
| Nvidia-driver | 410.79 |
| CUDA™ | 10.0 |
| TensorFlow version | 1.10 |
| TensorRT™ version | 5.0 |
| Docker Image for TensorFlow CPU only | tensorflow:1.10.0-py3 |
| Docker Image for TensorFlow GPU only | nvcr.io/nvidia/tensorflow:18.10-py3 |
| Docker Image for TF-TRT integration | nvcr.io/nvidia/tensorflow:18.10-py3 |
| Docker Image for TensorRT™ C++ API | nvcr.io/nvidia/tensorrt:18.11-py3 |
| Script samples source | Samples included within the docker images |
| Test Date | February 2019 |

# 3 Development Methodology

In this section we explain the general instructions on how we trained the custom model CheXNet from scratch with TensorFlow framework using transfer Learning, and how the trained model was optimized then with TensorRT™ to run accelerated inferencing.

## 3.1 Build a CheXNet Model with TensorFlow Framework

The CheXNet model was developed using transfer Learning based on resnet_v2_50, it means we built the model using the TensorFlow official pre-trained resnetV2_50 checkpoints downloaded from its website. The model was trained with 14 output classes representing the thoracic deceases.

In the next paragraphs and snippet codes we will explain the steps and the APIs used to build the model. **Figure 5** shows the general workflow pipeline followed:

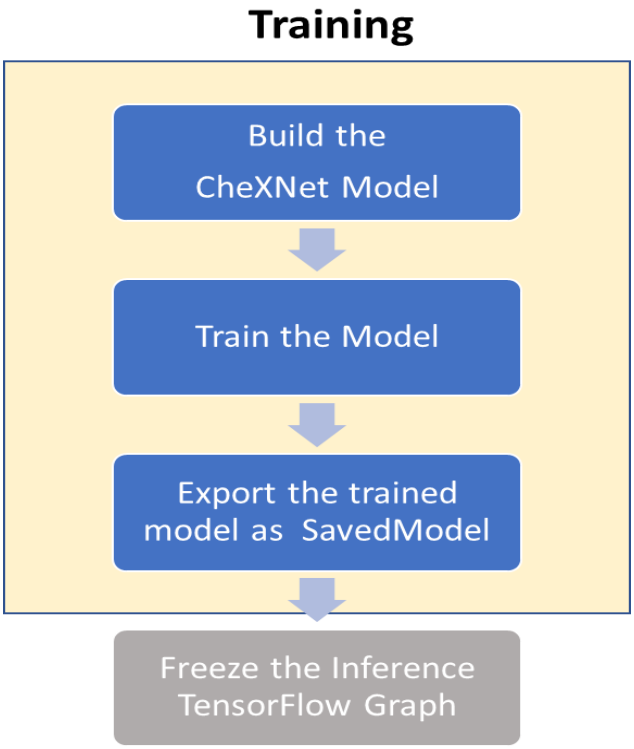


Figure 5: Training workload of the custom model CheXNet

**Define the Classes:**

Below is listed the 14 distinct categories of thoracic diseases to be predicted for the multiclass classification model

```
classes = ['Cardiomegaly',

           'Emphysema',
```

```
        'Effussion',
        'Hernia',
        'Nodule',
        'Pneumonia',
        'Atelectasis',
        'PT',
        'Mass',
        'Edema',
        'Consolidation',
        'Infiltration',
        'Fibrosis',
        'Pneumothorax']
```

**Build a Convolutional Neural Network using Estimators:**

Here we describe the building process of the CheXNet model with Transfer Learning using Custom Estimator. We used the high-level TensorFlow API tf.estimator and its class Estimator to build the model, it handles the high-level model training, evaluation, and inference of our model much easier than with the low-level TensorFlow APIs; it builds the graph for us and simplifies sharing the implementation of the model on a distributed multi-server environment, among other advantages.[3].

There are pre-made estimators and custom estimators [4], in our case we used the last one since it allows to customize our model through the model_fn function. Also, we defined the input_fn function which provides batches for training, evaluation, and prediction. When the tf. estimator class is called, it returns an initialized estimator, that at the same time calls the. train, eval, and predict functions, handling graphs and sessions for us.

See **Figure 6** with the overview of the estimator.

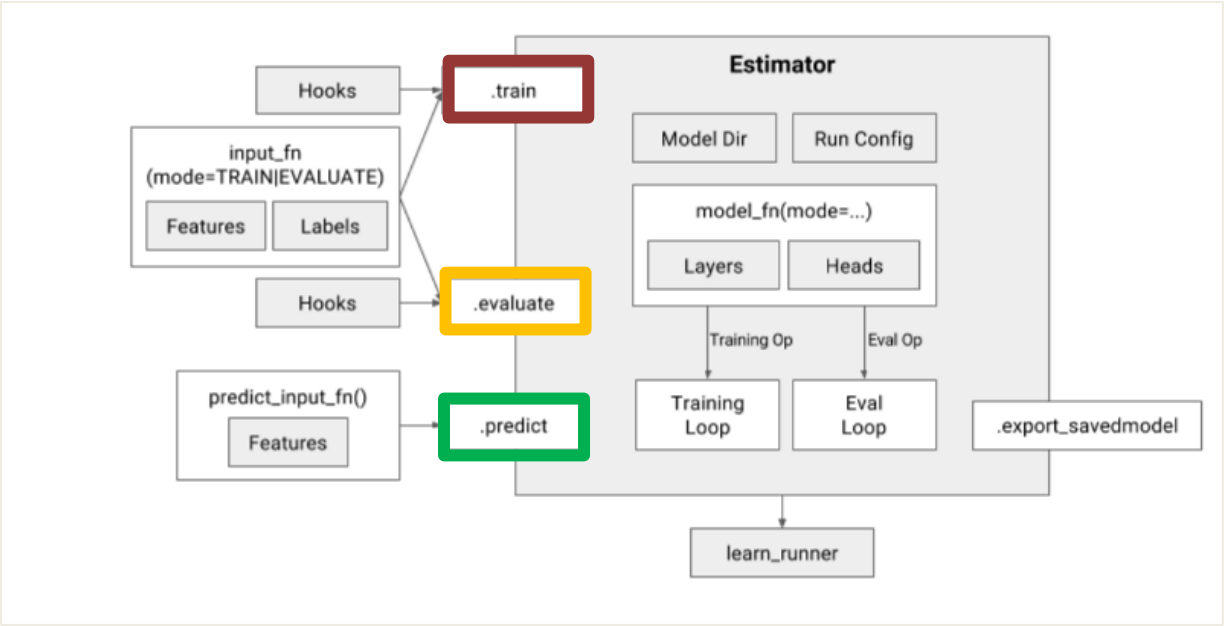


Figure 6: Overview of the Estimator Interface [5]

See the Table 4 with the Estimator's methods and modes to call train, evaluate, or predict. The Estimator framework invokes the model function with the mode parameter set as follows:

Table 4. Implement training, evaluation, and prediction. Source [4]

| Estimator method | Estimator Mode |
|---|---|
| tf.estimator.Estimator.train | tf.estimator.ModeKeys.TRAIN |
| tf.estimator.Estimator.evaluate | tf.estimator.ModeKeys.EVAL |
| tf.estimator.Estimator.predict | tf.estimator.ModeKeys.PREDICT |

**Create the Estimator:**

```
Chexnet_classifier = tf.estimator.Estimator(
    model_fn=model_function, model_dir=FLAGS.model_dir, config=run_config,
    params={
        'densenet_depth': FLAGS.densenet_depth,
        'data_format': FLAGS.data_format,
        'batch_size': FLAGS.batch_size,
        'lr': lr})
```

**Define the model function for training using transfer Learning:**

In this case, the architecture of an existing official network was used as base model (resnet_v2_50). The output of the model is defined by a layer with 14 neurons to predict each class. Since X-ray images can show more than one pathology, the model should also detect multiple classifications; to do so, we used the sigmoid activation function. See the snippet code below:

```
def model_fn(features, labels, mode, params):
    tf.summary.image('images', features, max_outputs=6)
    model = resnet_model.imagenet_resnet_v2(50, _NUM_CLASSES, params['data_format'])
    logits = model(features, mode == tf.estimator.ModeKeys.TRAIN)
    probs = tf.sigmoid(logits)
     predictions = tf.argmax(logits, axis=1)
```

**Restoring checkpoints from pre-trained model:**

The variable checkpoint file holds the os.path with the directory where the pretrained model with the ImageNet dataset ResNet-50_v2 (fp32) was stored, which was previously downloaded from the official TensorFlow repository to the local host [6]. The model was downloaded in the form of checkpoints produced by estimator during official training, then the estimator initializes the weights from there.

```
if not tf.train.latest_checkpoint(FLAGS.model_dir):

    vars_to_restore = [var for var in tf.global_variables() if 'dense' not in var.name]

    checkpoint_file = os.path.join(FLAGS.pretrained_model_dir,

                                   tf.train.latest_checkpoint(FLAGS.pretrained_model_dir))

    latest_ckp = tf.train.latest_checkpoint(checkpoint_file)

    tf.train.init_from_checkpoint(checkpoint_file,

                                  {var.name.split(':')[0]: var for var in vars_to_restore})
```

Each subsequent call to the Estimator's train, evaluate, or predict method causes TensorFlow rebuilds the model. See the **Figure 7**

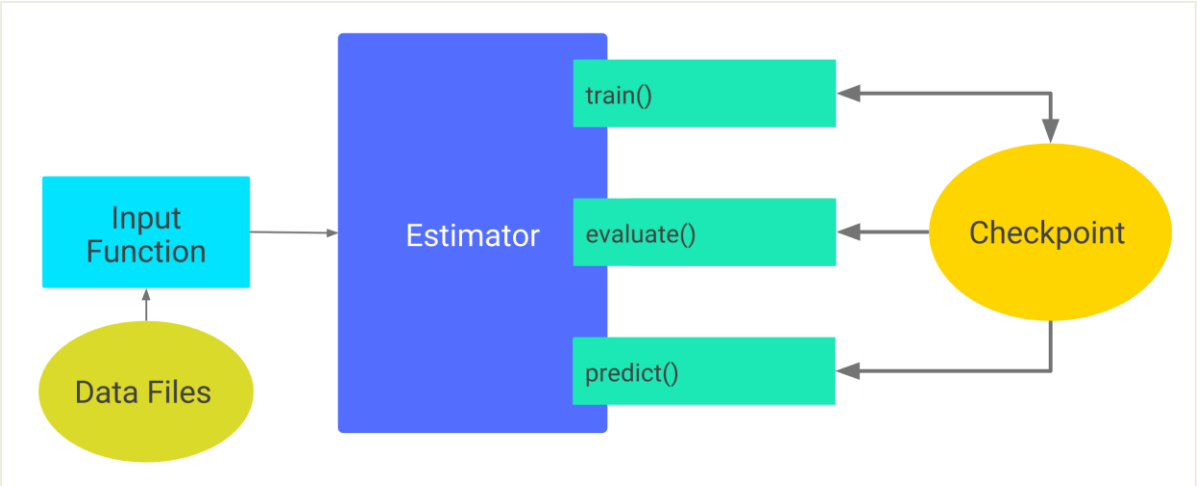


Figure 7. Subsequent calls to train(), evaluate(), or predict(). Source [7]

Variable Scope: When building the custom model, it's important to create it placing the variables under the same variable scope as the checkpoints; otherwise, the system will output errors similar to "tensorbatch_normalization/beta is not found in resnet_v2_imagenet_checkpoint". Variable scopes allow you to control variable reuse when calling functions which implicitly create and use variables. They also allow to name the variables in a hierarchical and understandable way [8].

**For evaluation mode:**

```
if mode == tf.estimator.ModeKeys.EVAL:

    for i in range(14):

        metrics.update({classes[i]: tf.metrics.auc(labels[: i], probs[:, i])})



return tf.estimator.EstimatorSpec(

    mode=mode,

    loss=loss,

    predictions=predictions,

    train_op=train_op,

    eval_metric_ops=metrics)
```

**For predict mode:**

We need to provide the export_output argument to the EstimatorSpec, it defines signatures for TensorFlow serving

```
prediction = {

    'categories': tf.argmax(logits, axis=1, name='categories'),

    'scores': tf.sigmoid(logits, name='chexnet_sigmoid_tensor')

    }



if mode == tf.estimator.ModeKeys.PREDICT:

# Return the predictions and the specification for serving a SavedModel

return tf.estimator.EstimatorSpec(

    mode=mode,

    predictions=prediction,

    export_outputs={

        'predict': tf.estimator.export.PredictOutput(prediction)
```

## 3.2 Train the model for Inference with Estimator

**Load training and evaluation data (part omitted) and Create the Custom CheXNet Estimator**

```
Chexnet_classifier = tf.estimator.Estimator(

            model_fn=model_function, model_dir=FLAGS.model_dir, config=run_config,

            params={

                'densenet_depth': FLAGS.densenet_depth,

                'data_format': FLAGS.data_format,

                'batch_size': FLAGS.batch_size,

                'lr': lr

            })
```

**Train the model:**

```
Chexnet_classifier.train(
```

```
    input_fn=lambda: input_fn(

    True, FLAGS.data_dir, FLAGS.batch_size, FLAGS.epochs_per_eval))
```

**Evaluate the model and print results:**

```
eval_results = chexnet_classifier.evaluate(

    input_fn=lambda: input_fn(False, FLAGS.data_dir, FLAGS.batch_size))

    lr = reduce_lr_hook.update_lr(eval_results['loss'])

print (eval_results)
```

## 3.3 Save the Trained Model with TensorFlow Serving for Inference

**Export the trained model as SavedModel with the Estimator function export_savedmodel**

Exports inference graph as a SavedModel into the given directory [9][10]

```
def export_saved_model(chexnet_classifier):

    shape=[_DEFAULT_IMAGE_SIZE, _DEFAULT_IMAGE_SIZE, _NUM_CHANNELS]

    input_receiver_fn = export.build_tensor_serving_input_receiver_fn(shape,

    batch_size=FLAGS.batch_size)

    Chexnet_classifier.export_savedmodel(FLAGS.export_dir, input_receiver_fn)
```

## 3.4 Freeze the Saved Model (optional)

**Convert Saved Model to a Frozen Graph:**

```
def convert_savedmodel_to_frozen_graph(savedmodel_dir, output_dir):

    meta_graph = get_serving_meta_graph_def(savedmodel_dir)

        signature_def = tf.contrib.saved_model.get_signature_def_by_key(

            output=return_tensors[0].outputs[0]

    with tf.Session(graph=g, config=get_gpu_config()) as sess:

       result = sess.run([output])

            meta_graph, tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY)
```

```
    graph = tf.Graph()
    with tf.Session(graph=graph) as sess:
            tf.saved_model.loader.load(sess, meta_graph.meta_info_def.tags,
            savedmodel_dir)
            frozen_graph_def = tf.graph_util.convert_variables_to_constants(sess,
            graph.as_graph_def(),
            output_node_names= ["chexnet_sigmoid_tensor", "categories"])
    #remove the unnecessary training nodes
    cleaned_frozen_graph = tf.graph_util.remove_training_nodes(frozen_graph_def)
    write_graph_to_file(_GRAPH_FILE, cleaned_frozen_graph, output_dir)
    return cleaned_frozen_graph
```

### Command line example to execute the chexnet.py file:

```
python3 chexnet.py \
    --train_epochs=15 \
    --learning_rate=0.001 \
    --batch_size=128 \
    --data_dir='/home/chexnet_tfrecords’ \
    --pretrained_model_dir='/home/resnet_v2_imagenet_checkpoint/ \
    --model_dir='/home/chest-x-ray/chexnet_checkpoints' \
    --export_dir='/home/chest-x-ray/chexnet_saved_model/ \
    --frozen_graph_dir='/home/chest-x-ray/chexnet_frozen_graph/
```

### Files used for development:

| | |
|---|---|
| **Script**: | chexnet.py |
| **Base model script**: | TensorFlow official ResnetV2_50 resnet_model.py |

# 4 Inference with TensorRT™

NVIDIA TensorRT™ is a C++ based library aimed to perform high performance inference on GPUs. After a model is trained and saved, TensorRT™ transforms it by applying graph optimization and layer fusion for a faster implementation, so it can be deployed in an inference context.

TensorRT™ provides three tools to optimize the models for inference: TensorFlow-TensorRT integrated (TF-TRT), TensorRT C++ API, and TensorRT Python API. The last two tools include parsers for importing existing models from Caffe, ONNX, or TensorFlow. Also, C++ and Python API's include methods for building models programmatically. It is important to note that TF-TRT is the Nvidia's preferred method for importing TensorFlow models to TensorRT™ [11].

In this project we will show how to implement using TensorRT™ C++ API, as well as TF-TRT integrated with the parser (UFF for TensorFlow). Below is a brief description of each method applied to CheXNet model.

## 4.1 TensorRT™ using TensorFlow-TensorRT (TF-TRT) Integrated

With TF-TRT integrated, TensorRT™ will parse the model and apply optimizations to the portions of the graph wherever possible, allowing TensorFlow to execute the remaining graph that couldn't be optimized. TF-TRT integration workflow includes importing a TensorFlow model, creating an optimized graph with TensorRT™, importing it back as the default graph, and running inference. After importing the model TensorRT™ optimizes the TensorFlow's subgraphs, then replaces each supported subgraph with a TensorRT™ optimized node, producing a frozen graph that runs in TensorFlow for inference. TensorFlow executes the graph for all supported areas and calls TensorRT™ to execute TensorRT™ optimized nodes. In this section we present the general steps to work with the custom model CheXNet and TF-TRT integration. For step-by-step instructions on how to use TensorRT™ with the TensorFlow framework, see "Accelerating Inference In TensorFlow With TensorRT™-User Guide"[11].

### 4.1.1 TF-TRT Workflow with a Frozen Graph

There are three workflows for creating a TensorRT™ inference graph from a TensorFlow model depending of the format: for saved model, frozen graph, and separate MetaGraph with checkpoint files.

In this project we will focus on the workflow using a frozen graph file. **Figure 8**. shows the specific workflow for creating a TensorRT™ inference graph from a TensorFlow model in frozen graph format file as an input. For more information about the other two methods, refer to the following Nvidia documentation: "Accelerating Inference in TensorFlow With TensorRT™ - User Guide" [12].

Further, the model needs to be built with supported operations by TF-TRT integrated, otherwise the system will output errors for unsupported operations. See the reference list for further description [13].

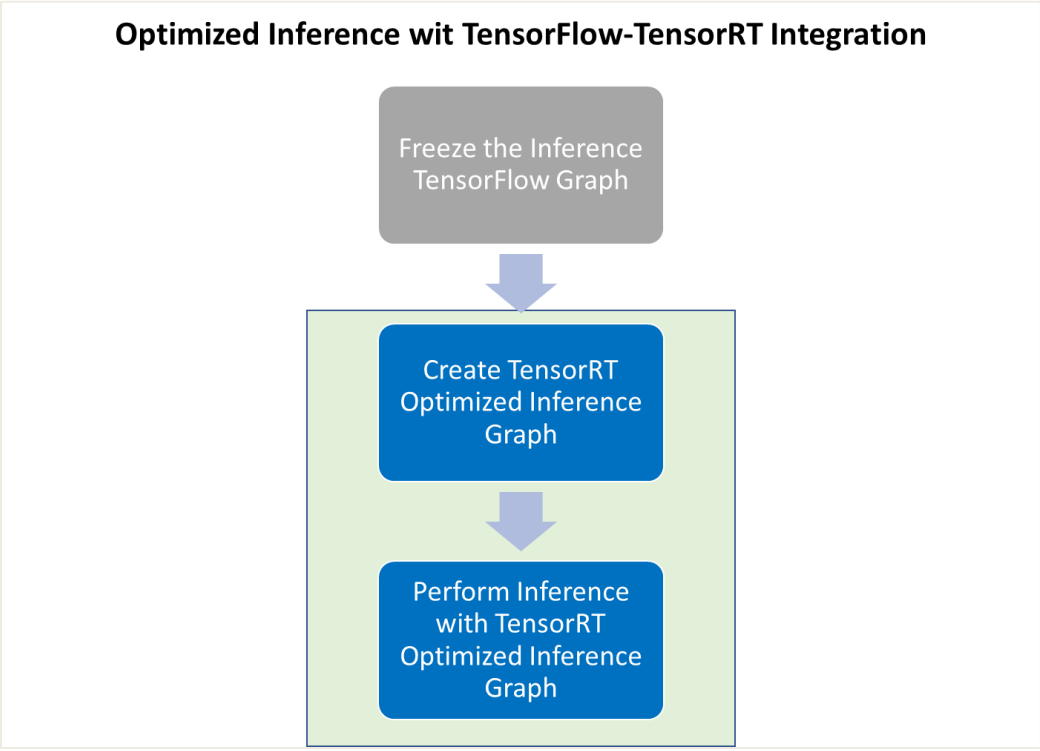


Figure 8: Workflow for Creating a TensorRT Inference Graph from a TensorFlow Model in Frozen Graph Format

**Import the library TensorFlow-TensorRT Integration:**

```
import tensorflow.contrib.TensorRT as trt
```

**Convert a SavedModel to a Frozen Graph and save it in the disk:**

If not converted already, the trained model needs to be frozen before use TensorRT™ through the frozen graph method, below is the function to do the conversion

```
def convert_savedmodel_to_frozen_graph(savedmodel_dir, output_dir):

  meta_graph = get_serving_meta_graph_def(savedmodel_dir)

  signature_def = tf.contrib.saved_model.get_signature_def_by_key(

      meta_graph,

      tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY)


  outputs = [v.name for v in signature_def.outputs.values()]
```

```
  output_names = [node.split(":")[0] for node in outputs]



  graph = tf.Graph()

  with tf.Session(graph=graph) as sess:

    tf.saved_model.loader.load(

        sess, meta_graph.meta_info_def.tags, savedmodel_dir)

    frozen_graph_def = tf.graph_util.convert_variables_to_constants(

        sess, graph.as_graph_def(), output_names)



  write_graph_to_file(_GRAPH_FILE, frozen_graph_def, output_dir)



  return frozen_graph_def
```

Freezing a model means pulling the values for all the variables from the latest model file, and then replace each variable op with a constant that has the numerical data for the weights stored in its attributes. It then strips away all the extraneous nodes that aren't used for forward inference, and saves out the resulting GraphDef into a just single output file, which is easily deployable for production[14].

**Load the frozen graph file from disk:**

```
def get_frozen_graph(graph_file):

     with tf.gfile.FastGFile(graph_file, "rb") as f:

          graph_def = tf.GraphDef()

          graph_def.ParseFromString(f.read())
```

**Create and save GraphDef for the TensorRT™ inference using TensorRT™ library:**

```
def get_trt_graph(graph_name, graph_def, precision_mode, output_dir,

             output_node, batch_size=128, workspace_size=2<<10):

      trt_graph = trt.create_inference_graph(

           input_graph_def=graph_def,

           outputs=[output_node],
```

```
            max_batch_size=batch_size,
            max_workspace_size_bytes=workspace_size<<20,
            precision_mode=precision_mode)
       write_graph_to_file(graph_name, trt_graph, output_dir)
       return trt_graph
```

### Create and save GraphDef for the TensorRT™ inference using TensorRT™ library (optional INT8 ):

"Convert a TensorRT™ graph used for calibration to an inference graph "

```
def get_trt_graph_from_calib(graph_name, calib_graph_def, output_dir):
       trt_graph = trt.calib_graph_to_infer_graph(calib_graph_def)
       write_graph_to_file(graph_name, trt_graph, output_dir)
       return trt_graph
```

### Import the TensorRT™ graph into a new graph:

```
output_node = tf.import_graph_def(
    trt_graph,
    return_elements=["chexnet_sigmoid_tensor"])
```

### Run the Optimized Inference in all desired modes:

```
output = return_tensors[0].outputs[0]
with tf.Session(graph=g, config=get_gpu_config()) as sess:
      result = sess.run([output])
```

### Command line example to execute the tensorrt_chexnet.py file

To evaluate the inference with TF-TRT integration using the trained CheXNet model:

```
python3 tensorrt_chexnet.py \
    --savedmodel_dir=/home/chest-x-ray/chexnet_saved_model/1541777429/ \
```

```
    --image_file=image.jpg \

    --int8 \

    --output_dir=/home/chest-x-ray/output_tensorrt_chexnet_1541777429/ \

    --batch_size=1 \

    --input_node="input_tensor” \

    --output_node="chexnet_sigmoid_tensor"
```

Where:

```
--savedmodel_dir: The location of a saved model directory to be converted into a Frozen Graph

--image_file: The location of a JPEG image that will be passed in for inference

--int8: Benchmark the model with TensorRT™ using int8 precision

--output_dir: The location where output files will be saved

--batch_size: Batch size for inference

--input_node: The name of the graph input node where the float image array should be fed for
   prediction

--output_node: The names of the graph output node
```

**Script Output sample:**

On completion, the script prints overall metrics and timing information over the inference session

```
==========================

network: tftrt_int8_frozen_graph.pb,     batchsize 1, steps 100

  fps   median: 284.6, mean: 304.3,    uncertainty: 5.5,   jitter: 4.4

  latency   median: 0.00351,    mean: 0.00337, 99th_p: 0.00383,    99th_uncertainty: 0.00053

==========================
```

- Throughput (images/sec): 304
- Latency (sec): 3.37

**Files used for development:**

| | |
|---|---|
| **Script**: | tensorrt_chexnet.py |
| **Base model script**: | tensorrt.py |
| **Labels file** | labellist_chest_x_ray.json |

## 4.2 TensorRT™ using TensorRT C++ API

In this section, we present how to run optimized inferences with an existing TensorFlow model using TensorRT C++ API. The first step is to convert the frozen graph model to uff file format with the C++ UFF parser API which supports TensorFlow models, then follow the workflow in the **Figure 9** to create the TensorRT™ engine for optimized inferences:

- Create a TensorRT™ network definition from the existing trained model
- Invoke the TensorRT™ builder to create an optimized runtime engine from the network
- Serialize and deserialize the engine so that it can be rapidly recreated at runtime
- Feed the engine with data to perform inference

For the current implementation, we are using Nvidia script trtexec.cpp and referenced the TensorRT™ Developer Guide to document the steps described below [15].

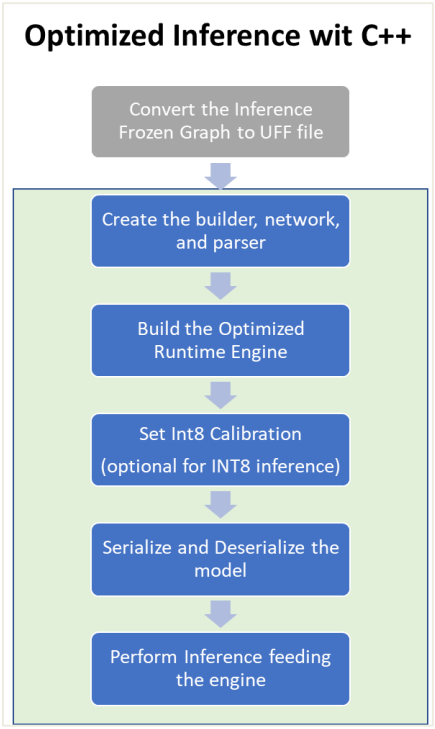


Figure 9: Workflow for Creating a TensorRT Inference Graph using the TensorRT C++ API

### Converting A Frozen Graph To UFF:

An existing model built with TensorFlow can be used to build a TensorRT™ engine. Importing from the TensorFlow framework requires to convert the TensorFlow model into the intermediate format UFF file. To do so, we used the tool convert_to_uff.py located at the directory /usr/lib/python3.5/dist-packages/uff/bin, which uses as an input a frozen model, below the command to convert .pb TensorFlow frozen graph to .uff format file:

```
    python3 convert_to_uff.py \

         input_file /home/chest-x-ray/chexnet_frozen_graph_1541777429.pb
```

### Create the builder, network, and UFF parser

```
//1-Create the builder and network:

IBuilder* builder = createInferBuilder(gLogger);

INetworkDefinition* network = builder->createNetwork();



//2-Create the UFF parser:

IUFFParser* parser = createUffParser();



//3-Declare the network inputs and outputs to the UFF parser:

parser->registerInput("input_tensor", DimsCHW(3,256,256), UffInputOrder::kNCHW);

parser->registerOutput("chexnet_sigmoid_tensor");



//Parse the imported model to populate the network:

parser->parse(uffFile, *network, nvinfer1::DataType::kFLOAT);
```

For the network definition, it is important to directly specify to TensorRT™ which tensors are inputs and their dimensions, as well as specify which tensors are outputs for inference (inputs and output tensors must also be given names); the rest of the tensors are transient values that may be optimized by the builder.

UFF Parser is used to parse a network in UFF format. For more details on the C++ UFF Parser, see NvUffParser or the Python UFF Parser [16].

TensorRT™ C++ API expects the input tensor to be in channel first order (CHW). When importing from TensorFlow, the input tensor is required to be in this format in order to achieve

the best possible performance; and if not, it is recommended to convert it to CHW. Overall, CHW is generally better for GPUs, while HWC is generally better for CPUs. [6]

**Build the Optimized Runtime Engine in fp16 or iInt8 mode (calibration optional for INT8int8 inference) [15]:**

```
//Configure the builder

builder->setMaxBatchSize(gParams.batchSize);

builder->setMaxWorkspaceSize(gParams.workspaceSize << 20);



//To run in fp16 mode

if (gParams.fp16)

{

    builder->setFp16Mode(gParams.fp16);

}



//To run in Int8 mod (calibration optional for int8 inference)

if (gParams.int8)

{

    builder->setInt8Mode(true);

    builder->setInt8Calibrator(&calibrator);

}



//Build the engine

ICudaEngine* engine = builder->buildCudaEngine(*network);
```

Highlights:

- After the network has been built, it can be used as default in FP32fp32 precision mode, for example, inputs and outputs remain in 32-bit floating point.
- Setting the builder's fp16 mode flag enables 16-bit precision inference mode

Setting the builder flag to int8 enables int8 precision inference mode. Calibration is an additional step required when building networks for int8. The application must provide TensorRT™ with

sample input. TensorRT™ will then perform inference in fp32 and gather statistics about intermediate activation layers that it will use to build the reduce precision int8 engine. When the engine is built, TensorRT™ makes copies of the weights. The TensorRT™ network definition contains pointers to model weights, the builder copies the weights into the optimized engine, and the parser will own the memory occupied by the weights; the parser object is then deleted after the builder has run for inference.

### Serialize and Deserialize the model

```
//1-Run the builder as a prior offline step and then serialize:

HostMemory *serializedModel = engine->serialize();



//Store model to disk

assert(serializedModel);

p.write(reinterpret_cast<const char*>(serializedModel->data()), serializedModel->size());

serializedModel->destroy();



//2-Create a runtime object to deserialize:

IRuntime* runtime = createInferRuntime(gLogger);

ICudaEngine* engine = runtime->deserializeCudaEngine(modelData, modelSize, nullptr);
```

It is not mandatory to serialize and deserialize a model before using it for inference, if desirable, the engine object can be used for inference directly. Since creating an engine from the network definition can be time consuming, we can avoid rebuilding the engine every time the application reruns by serializing it once and deserializing it while inferencing. Therefore, after the engine is built, it is common to serialize it for later use [17].

### Perform Inference feeding the engine

```
//1-Create the execution context to hold the network definition, trained parameters, necessary space:

IExecutionContext *context = engine->createExecutionContext();



//2-Use the input and output tensor names to get the corresponding input and output index:

int inputIndex = engine.getBindingIndex(“input_tensor”);
```

```
int outputIndex = engine.getBindingIndex(“CheXNet_sigmoid_tensor”);



//3-Set up a buffer array pointing to the input and output buffers on the GPU, using the indexes:

void* buffers[2];

buffers[inputIndex] = inputbuffer;

buffers[outputIndex] = outputBuffer;



//4-TensorRT™ execution is typically asynchronous, so enqueue the kernels on a CUDA stream:

context.enqueue(batchSize, buffers, stream, nullptr):
```

### Command line to execute the trtexec file:

```
./trtexec

    --uff=/home/chest-x-ray/output_convert_to_uff/chexnet_frozen_graph_1541777429.uff \

    --output=chexnet_sigmoid_tensor \

    --uffInput=input_tensor,3,256,256 \

    --iterations=40 \

    --int8 \

    --batch=1 \

    --device=0 \

    --avgRuns=100
```

*Docker image used for native TRT: nvcr.io/nvidia/tensorrt:18.11-py3*

Where:

```
--uff=: UFF file location
--output: output tensor name
--uffInput: Input tensor name and its dimensions for UFF parser (in CHW format)
--iterations: Run N iterations
--int8: Run in int8 precision mode
--batch: Set batch size
--device: Set specific cuda device to N
--avgRuns: Set avgRuns to N - perf is measured as an average of avgRuns
```

**Script Output sample:**

On completion, the script prints overall metrics and timing information over the inference session

```
Average over 100 runs is 1.44041 ms (host walltime is 1.56217 ms, 99% percentile time is 1.52326).
Average over 100 runs is 1.43143 ms (host walltime is 1.54826 ms, 99% percentile time is 1.50819).
Average over 100 runs is 1.44583 ms (host walltime is 1.56766 ms, 99% percentile time is 1.54211).
Average over 100 runs is 1.43773 ms (host walltime is 1.55612 ms, 99% percentile time is 1.53363).
Average over 100 runs is 1.44332 ms (host walltime is 1.55968 ms, 99% percentile time is 1.51658).
Average over 100 runs is 1.43861 ms (host walltime is 1.56039 ms, 99% percentile time is 1.50253).
Average over 100 runs is 1.43901 ms (host walltime is 1.56038 ms, 99% percentile time is 1.55898).
Average over 100 runs is 1.43517 ms (host walltime is 1.55967 ms, 99% percentile time is 1.51555).
Average over 100 runs is 1.45124 ms (host walltime is 1.57128 ms, 99% percentile time is 1.57366).
Average over 100 runs is 1.4332 ms (host walltime is 1.55241 ms, 99% percentile time is 1.51955).
Average over 100 runs is 1.43537 ms (host walltime is 1.55512 ms, 99% percentile time is 1.50966).
```

- Throughput $\left(\frac{\text{imgs}}{\text{sec}}\right) = \left(\frac{Batch\ Size}{Latency(ms)}\right) * 1000 = \left(\frac{1}{1.43537}\right) * 1000 = 697$
- Latency (msec): 1.43537

**Description of files and parameters used for development:**

| | | **Description** |
|---|---|---|
| **Script**: | trtexec.cpp | Nvidia sample code showing the optimized inference using TensorRT C++ API |
| **TensorFlow Frozen Graph:** | chexnet_frozen_graph_1541777429.pb | existing TensorFlow model |
| **TensorFlow UFF file:** | chexnet_frozen_graph_1541777429.uff | existing TensorFlow model converted to uff format |
| **Input tensor name:** | “input_tensor” | Input tensor name |
| **Input tensor dimension (NCHW):** | (3,256,256) | input tensor dimensions for UFF parser |
| **Output tensor name:** | “chexnet_sigmoid_tensor” | Output tensor name for inference |

# 5 Results

## 5.1 CheXNet Inference - Native TensorFlow FP32fp32 with CPU Only

Benchmarks ran with batch sizes 1-32 using native TensorFlow FP32fp32 with CPU-Only (AMD EPYC 7551 32-Core Processor). Tests conducted using the docker image TensorFlow:1.10.0-py3

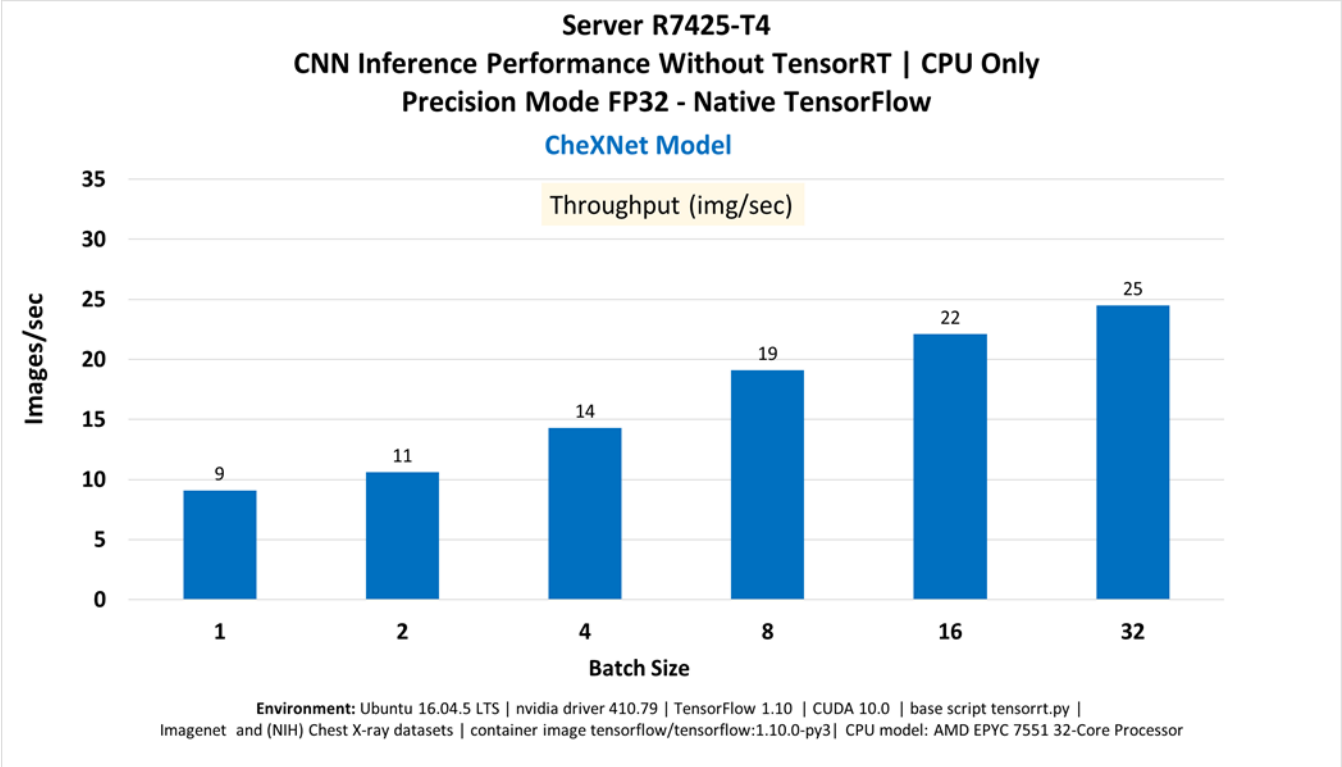


Figure 10: CheXNet Inference - Native TensorFlow FP32 with CPU-Only. AMD EPYC 7551 32-Core

**Command line to execute the benchmark:**

```
python3 tensorrt_chest.py

    --savedmodel_dir=/home/dell/chest-x-ray/chexnet_saved_model/1541777429/ \

    --image_file=image.jpg \

    --native \

    --output_dir=/home/dell/chest-x-ray/output_tensorrt_chest_only_cpu/ \

    --batch_size=1
```

*Docker image for TensorFlow-CPU Only: tensorflow/tensorflow:1.10.0-py3*

Where: --native: Benchmark model with it's native precision FP32 and without TensorRT™.
**Script Output sample:**

```
==========================

network: native_frozen_graph.pb, batchsize 1, steps 100

  fps       median: 9.2,     mean: 9.1, uncertainty: 0.1, jitter: 0.3

  latency   median: 0.10912, mean: 0.11459,  99th_p: 0.23157, 99th_uncertainty: 0.18079

==========================
```

- Throughput (images/sec): ~9
- Latency (sec): 0.11459*1000 = ~115

## 5.2 CheXNet Inference - Native TensorFlow fp32 with GPU

Benchmarks ran with batch sizes 1-32 using native TensorFlow FP32 GPU without TensorRT™. We ran the benchmarks within the docker image nvcr.io/nvidia/tensorflow:18.10-py3, which supports TensorFlow with GPU support.

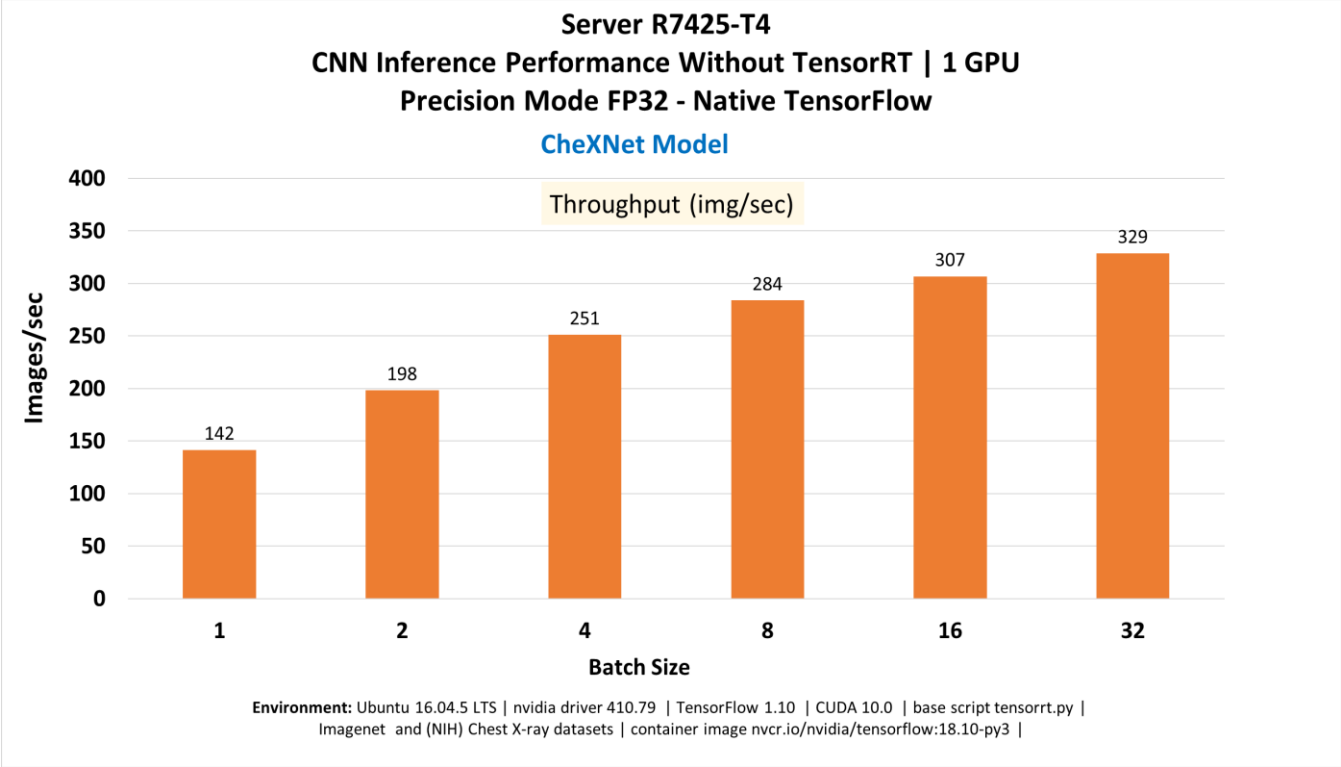


Figure 11. CheXNet Inference - Native TensorFlow FP32 with GPU

**Command line to execute the benchmark:**

```
python3 tensorrt_chest.py

    --savedmodel_dir=/home/dell/chest-x-ray/chexnet_saved_model/1541777429/ \

    --image_file=image.jpg \

    --native \

    --output_dir=/home/dell/chest-x-ray/output_tensorrt_chexnet_1541777429/

    --batch_size=1
```

*Docker image for TensorFlow-GPU: nvcr.io/nvidia/tensorflow:18.10-py3*

**Where:** `--native: Benchmark model with it's native precision FP32 and without TensorRT™.`

**Script Output sample:**

```
==========================

network: native_frozen_graph.pb,      batchsize 1, steps 100

  fps       median: 141.8,   mean: 142.1, uncertainty: 0.3, jitter: 2.3

  latency   median: 0.00705, mean: 0.00704, 99th_p: 0.00740, 99th_uncertainty: 0.00010

==========================
```

- Throughput (images/sec): ~142
- Latency (sec): 0.00704*1000 = ~7

## 5.3 CheXNet Inference –TF-TRT 5.0 Integration in INT8int8 Precision Mode

Benchmarks ran with batch sizes 1-32 using native TensorFlow FP32fp32 TensorRT™. We ran the benchmarks within the docker image nvcr.io/nvidia/tensorflow:18.10-py3, which supports TensorFlow with GPU as well as TensorRT™ 5.0.

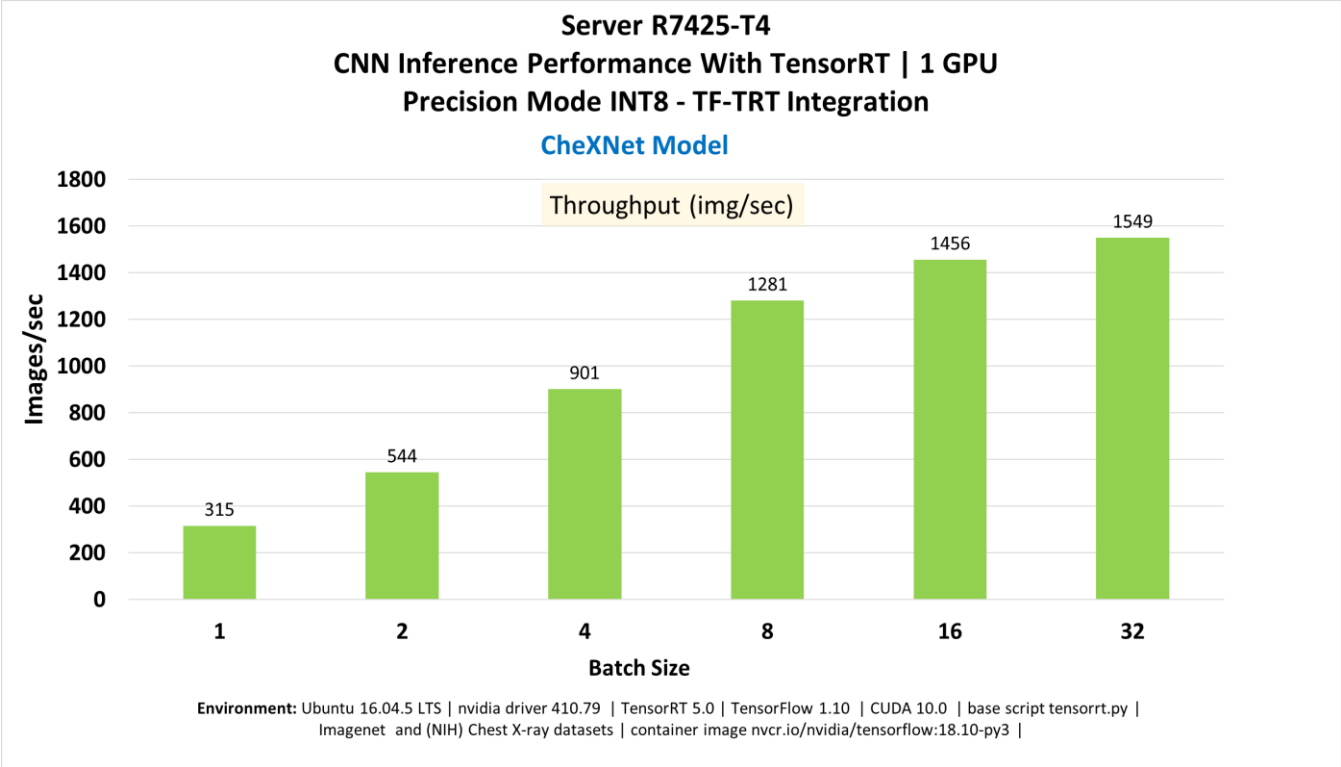


Figure 12. CheXNet Inference –TF-TRT 5.0 Integration in INT8int8 Precision Mode

**Command line to execute the benchmark:**

```
python3 tensorrt_chest.py

    --savedmodel_dir=/home/dell/chest-x-ray/chexnet_saved_model/1541777429/ \

    --image_file=image.jpg \

    --int8 \

    --output_dir=/home/dell/chest-x-ray/output_tensorrt_chexnet_1541777429/

    --batch_size=1
```

*Docker image for TensorFlow-GPU: nvcr.io/nvidia/tensorflow:18.10-py3*

**Where:** `--int8: Benchmark the model with TensorRT™ using int8 precision`

**Script Output sample:**

```
==========================

network: tftrt_int8_frozen_graph.pb,  batchsize 1, steps 100

  fps       median: 282.2,   mean: 315.2, uncertainty: 6.8, jitter: 5.6

  latency   median: 0.00354, mean: 0.00329, 99th_p: 0.00371, 99th_uncertainty: 0.00008

==========================
```

- Throughput (images/sec): ~315
- Latency (sec): 0.00704*1000 = ~3

## 5.4 Benchmarking CheXNet Model Inference with Official ResnetV2_50

To benchmark our custom model CheXNet with a well-known model, we replicated the same inference tests TF-TRT-INT8 Integration using the official pre-trained version of the ResNet-50 v2 model (fp32, Accuracy 76.47%) [6]. The model was downloaded as SavedModel format produced with Estimator during the training in FP32 precision mode, this version also accepts input tensors with channel first format (CHW). See the TensorFlow performance guide for more details[18].

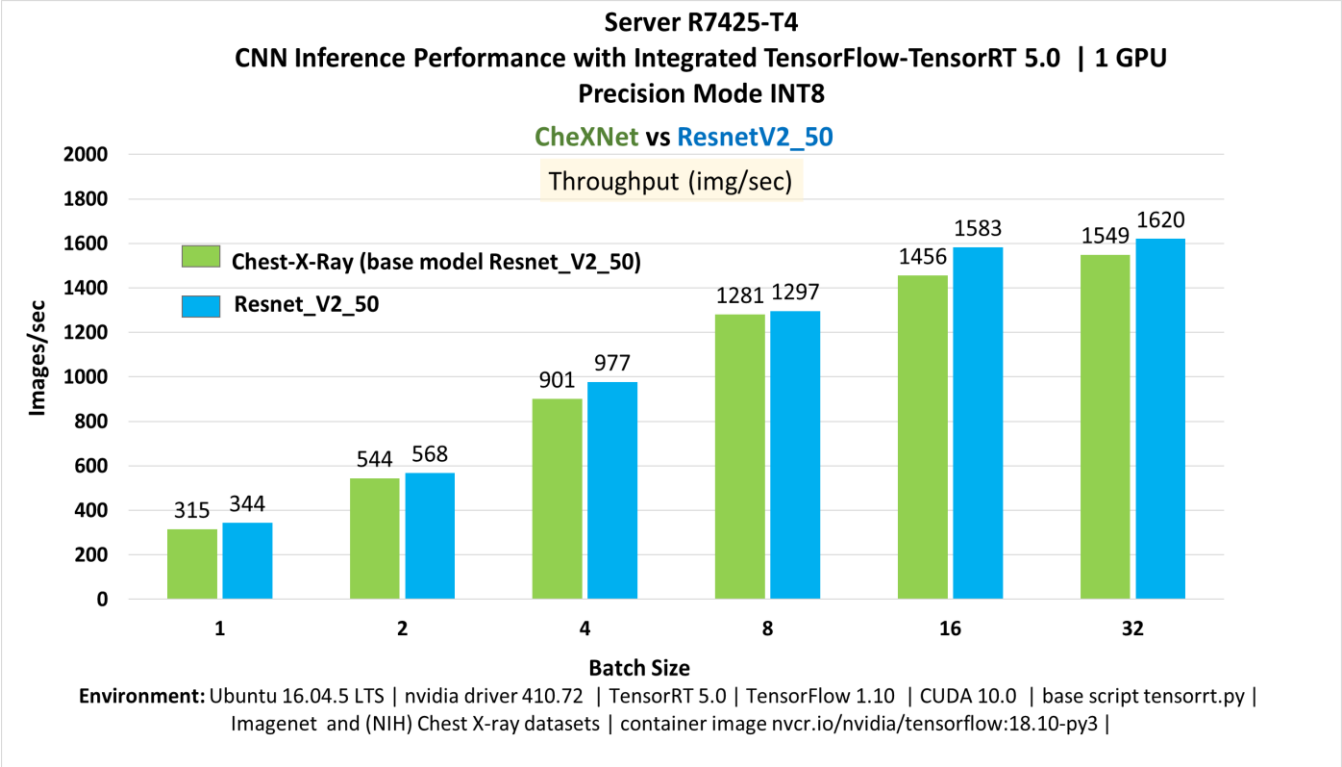


Figure 13. Throughput CheXNet TF-TRT-INT8int8 versus ResnetV2_50 TF-TRT-INT8int8 Inference

In the **Figure 13** we can appreciate that our custom model CheXNet and the official model ResnetV2_50 performed closely when running optimized inferences with TF-TRT INT8int8 integration. It is a good practice to benchmark our custom models with official models, so we can decide whether going back and retrain it or move forward with the optimized model.

We see also in **Figure** 14 that the latency of both models was similar too across different batch sizes. Lower latency is better, mainly for critical real time applications where milliseconds matter.

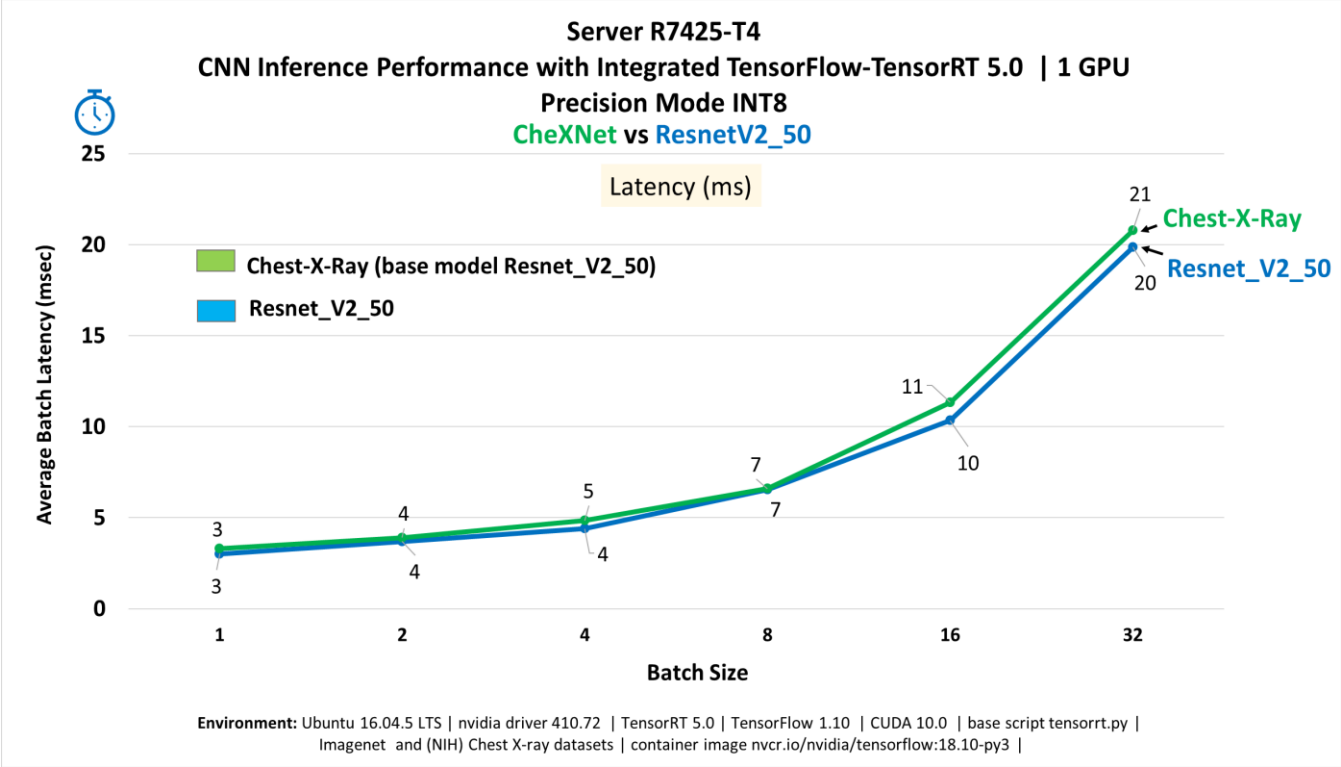


Figure 14. Latency CheXNet TF-TRT-INT8int8 versus ResnetV2_50 TF-TRT-INT8int8 Inference

## 5.5 CheXNet Inference - Native TensorFlow FP32fp32 with GPU versus TF-TRT 5.0 INT8

After confirming that our custom model performed well compared to the optimized inference TF-TRT of an official model, we proceeded in this section to compare the CheXNet inference model itself in different configurations. In the *Error! Reference source not found.* we have gathered the previous results obtained when we ran the inference in three modes:

a) Native TensorFlow fpFP32-CPU Only (CPU)
b) Native TensorFlow fpFP32-GPU (GPU)
c) TF-TRT Integration in INT8int8 (GPU)

**Figure** 15 shows the CheXNet inference throughput (img/sec) ran in different configuration modes and batch sizes. As we can appreciate the TF-TRT_INT8 precision mode outperformed the two other configurations consistently across several batch sizes. In the next sections we analyzed in detail this performance improvement.

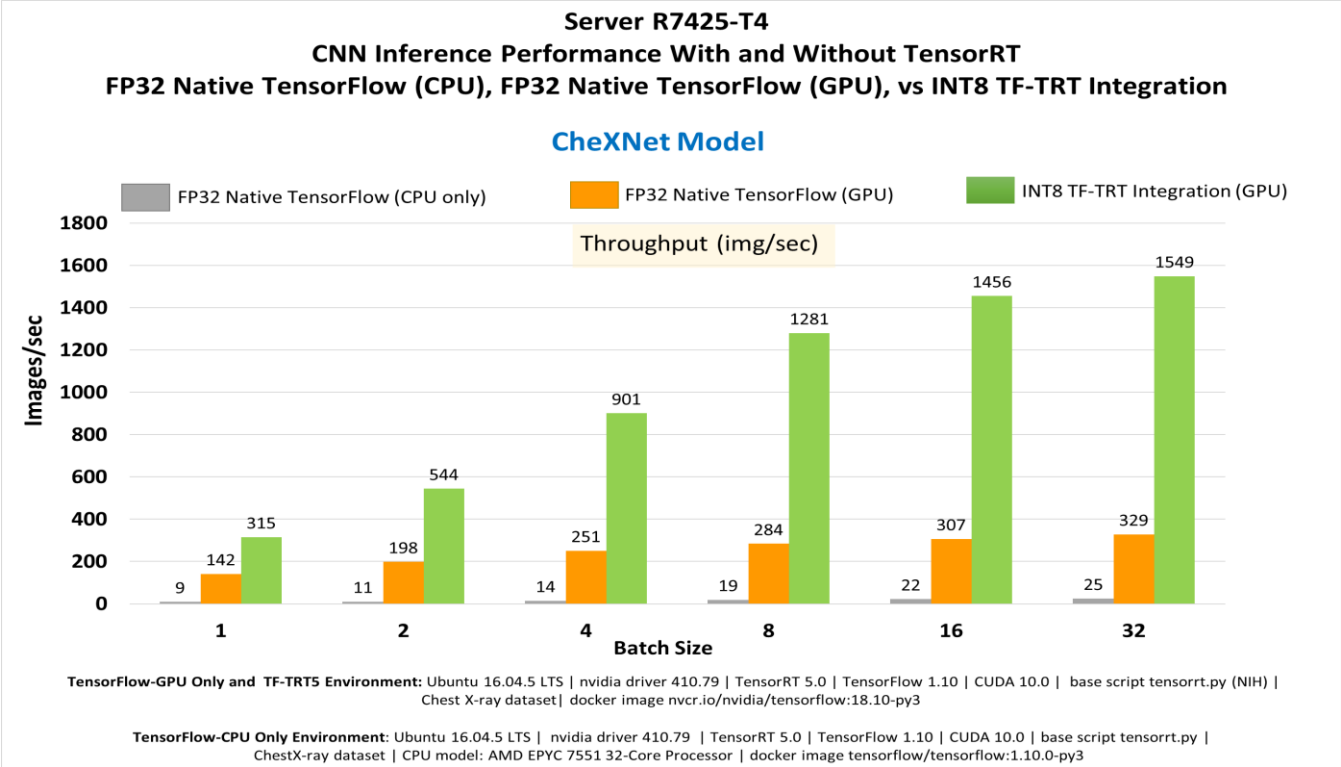


Figure 15. Throughput Native TensorFlow FP32 versus TF-TRT 5.0 Integration INT8

**Figure 16** shows the latency curve for each inference configuration, the lower is the latency better is the performance, and in this case TF-TRT-INT8 implementation reached the lowest inference time for all the batch sizes.

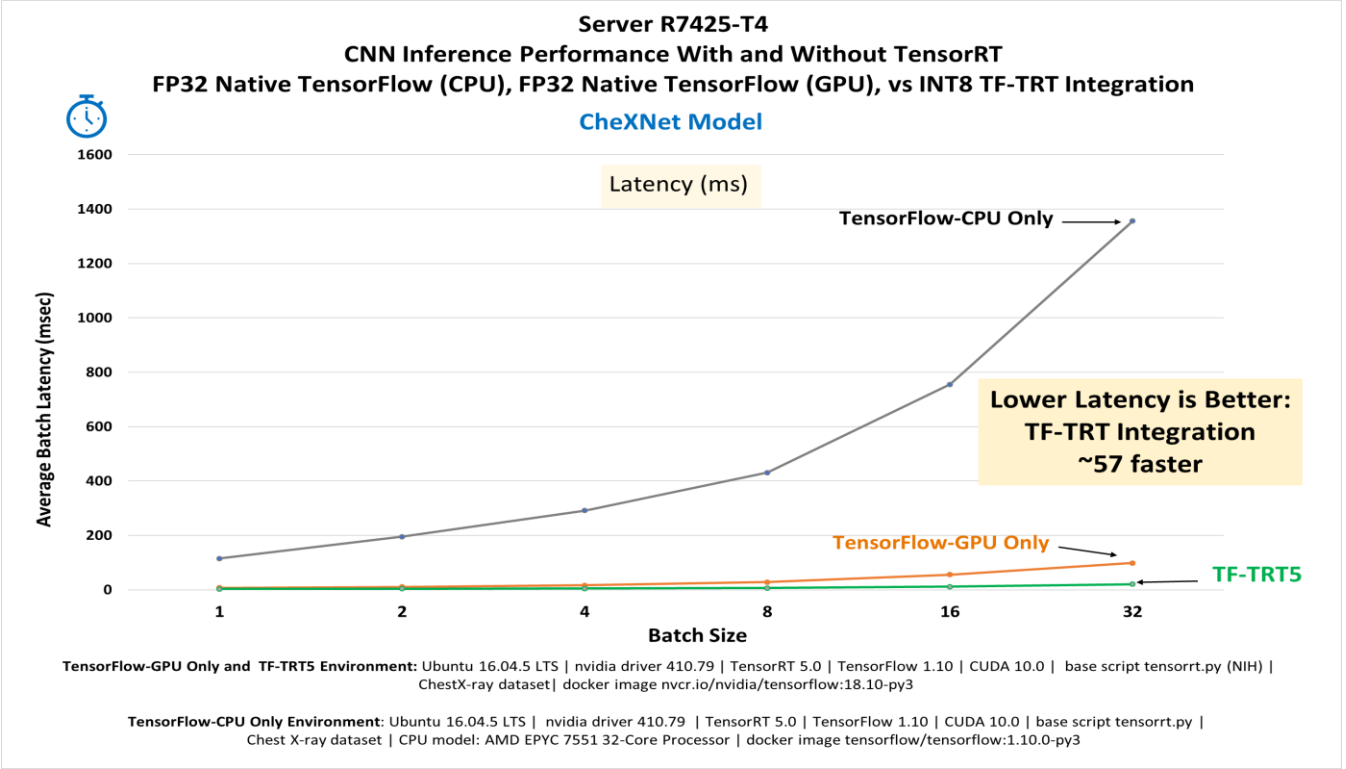


Figure 16. Latency Native TensorFlow FP32fp32 (CPU / GPU) versus TF-TRT 5.0 Integration INT8int8

See the **Table 5** with the consolidated results of the CheXNet Inference in Native TensorFlow FP32 mode versus TF-TRT 5.0 Integration INT8int8, in terms of throughput and latency. We observed the huge different when running the test in different configurations. For speedup factors see the next tables.

Table 5. Throughput and Latency Native TensorFlow FP32 versus TF-TRT 5.0 Integration INT8

| Batch Size | TF-TRT INT8 | | Native TensorFlow FP32-GPU | | Native TensorFlow FP32- CPU Only | |
|---|---|---|---|---|---|---|
| | Throughput (img/sec) | Latency (ms) | Throughput (img/sec) | Latency (ms) | Throughput (img/sec) | Latency (ms) |
| **1** | 315 | 3 | 142 | 7 | 9 | 115 |
| **2** | 544 | 4 | 198 | 10 | 11 | 195 |
| **4** | 901 | 5 | 251 | 16 | 14 | 292 |
| **8** | 1281 | 7 | 284 | 28 | 19 | 431 |
| **16** | 1456 | 11 | 307 | 55 | 22 | 755 |
| **32** | 1549 | 21 | 329 | 98 | 25 | 1356 |

In **Table 6** we have calculated the speedup factor of TF-TRT 5.0 Integration INT8 versus Native TensorFlow FP32-GPU. The server PowerEdge R7425-T4 performed in average up to 4X faster than native TensorFlow-GPU when accelerating the workloads with TF-TRT Integration.

Table 6. PowerEdge R7425-T4 Speedup Factor with TF-TRT versus native TensorFlow-GPU

| Batch Size | TF-TRT INT8 | Native TensorFlow FP32-GPU | Speedup Factor X |
|---|---|---|---|
| | Throughput (img/sec) | Throughput (img/sec) | |
| **1** | 315 | 142 | 2X |
| **2** | 544 | 198 | 3X |
| **4** | 901 | 251 | 4X |
| **8** | 1281 | 284 | 5X |
| **16** | 1456 | 307 | 5X |
| **32** | 1549 | 329 | 5X |
| | | **Average** | **4X** |

In **Table 7** we have calculated the speedup factor of TF-TRT 5.0 Integration INT8 versus Native TensorFlow FP32-CPU Only. The server PowerEdge R7425-T4 performed in average up to 58X faster than native TensorFlow-CPU Only when accelerating the workloads with TF-TRT Integration

Table 7. PowerEdge R7425-T4 Speedup Factor with TF-TRT versus native TensorFlow-CPU Only

| Batch Size | TF-TRT INT8 | Native TensorFlow FP32-CPU Only | Speedup Factor X |
|---|---|---|---|
| | Throughput (img/sec) | Throughput (img/sec) | |
| **1** | 315 | 9 | 35X |
| **2** | 544 | 11 | 51X |
| **4** | 901 | 14 | 63X |
| **8** | 1281 | 19 | 67X |
| **16** | 1456 | 22 | 66X |
| **32** | 1549 | 25 | 63X |
| | | **Average** | **58X** |

See **Figure 17** the R7425-T4-16GB speedup Factor with TF-TRT versus Native TensorFlow

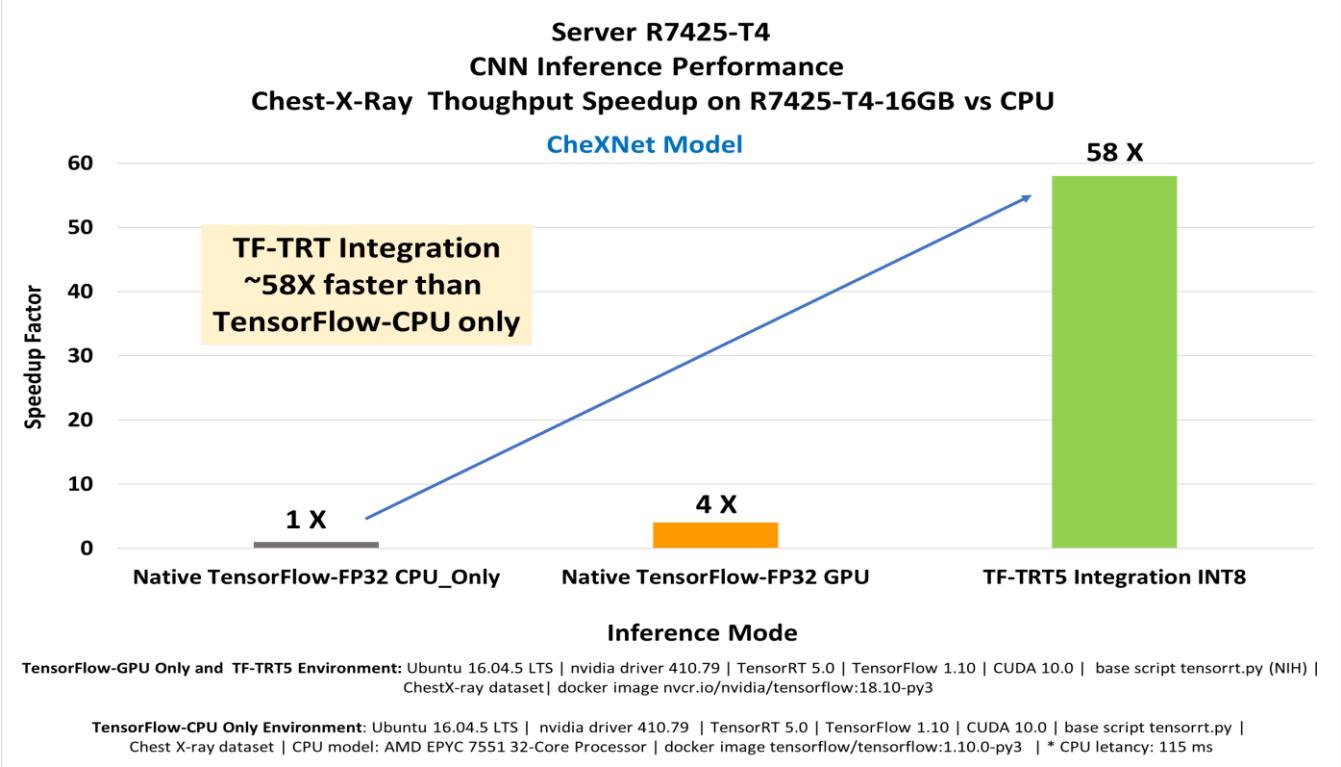


Figure 17: Speedup Factor with TF-TRT versus Native TensorFlow

## 5.6 CheXNet Inference - TF-TRT 5.0 Integration vs Native TRT5 C++ API

We wanted to explore further and optimized the CheXNet inference using the TensorRT C++ API with the sample tool **trtexec** provided by Nvidia. This sample is very useful for generating serialized engines and can be used as a template to work with our custom models.

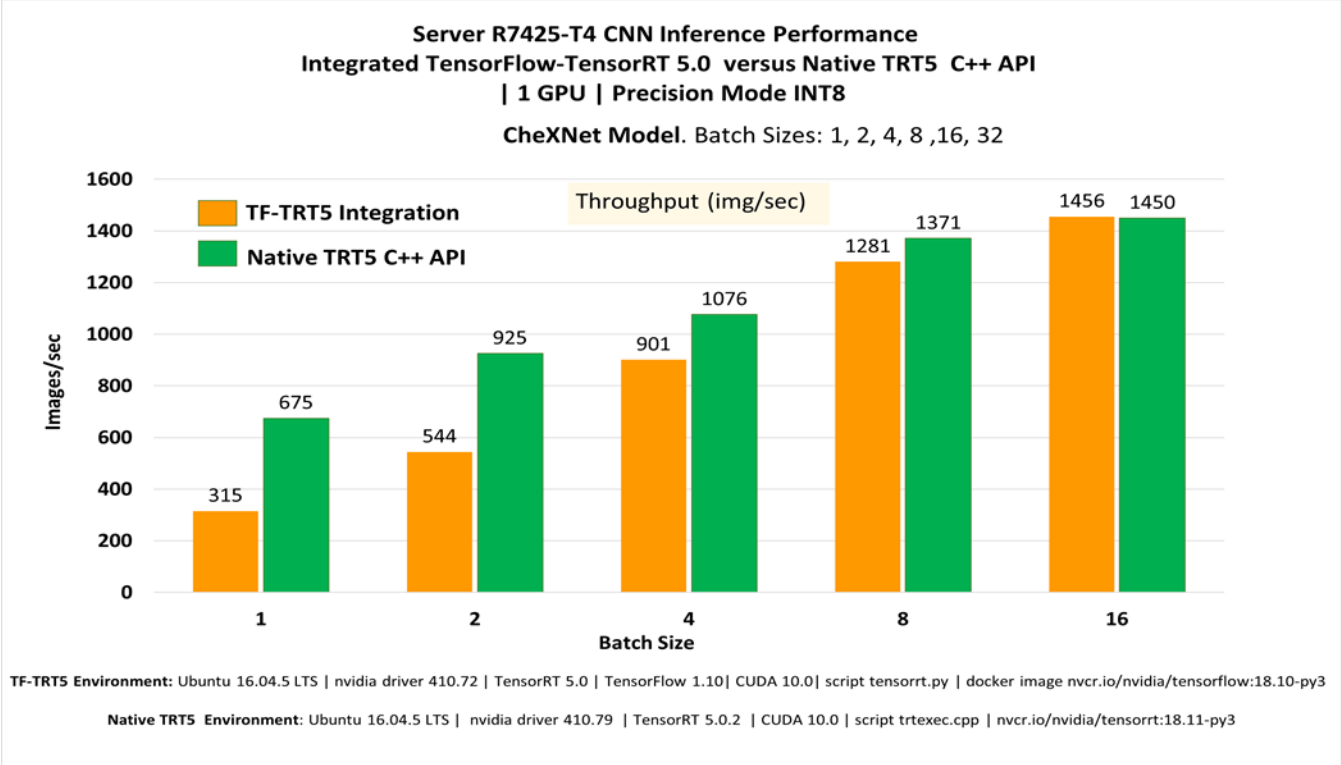


Figure 18:Throughput TF-TRT 5.0 Integration vs Native TRT5 C++ API

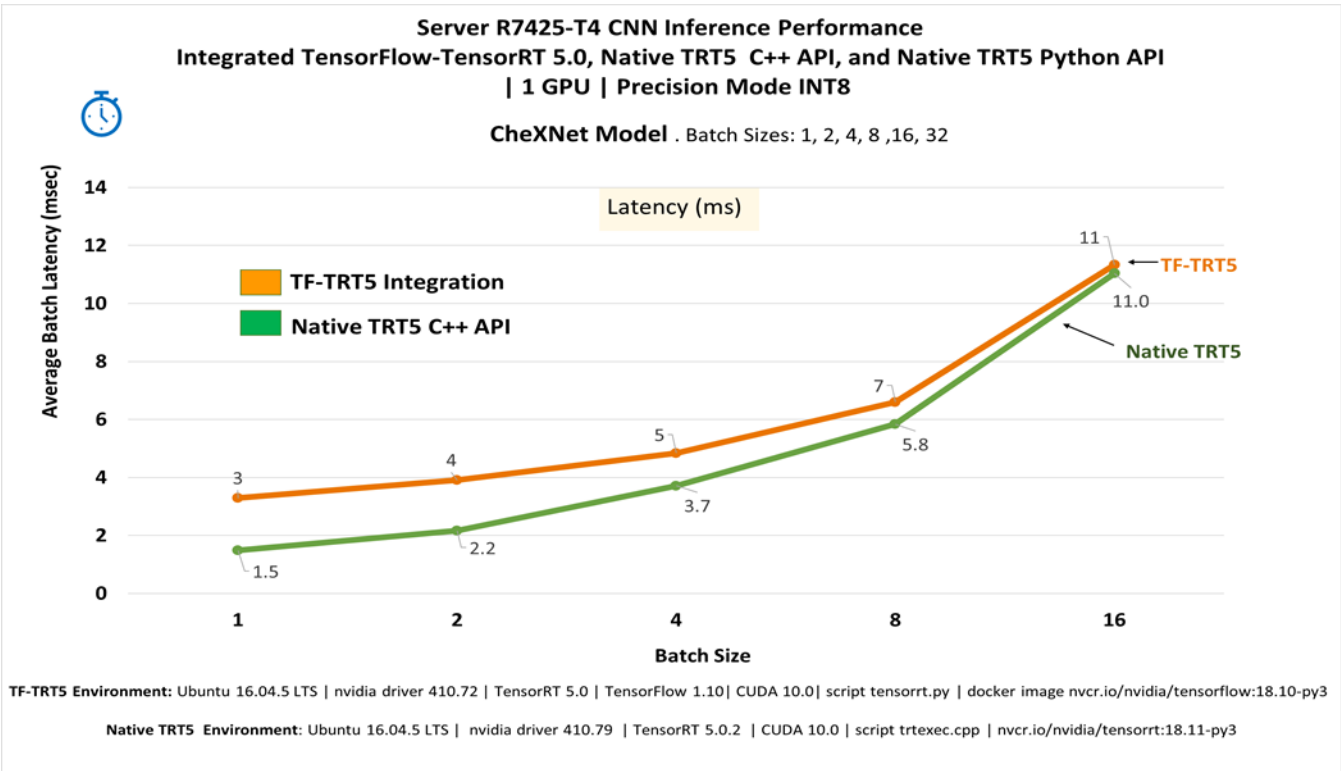


Figure 19: Latency TF-TRT 5.0 Integration vs Native TRT5 C++ API

**Command line to execute the Native TensorRT™ C++ API benchmark:**

```
./trtexec
   --uff=/home/dell/chest-x-ray/output_convert_to_uff/chexnet_frozen_graph_1541777429.uff
   --output= chexnet_sigmoid_tensor
   --uffInput=input_tensor,3,256,256
   --iterations=40 --int8 --batch=1
   --device=0
   --avgRuns=100
```

**Where:**

```
--uff=: UFF file location
--output: output tensor name
--uffInput: Input tensor name and its dimensions for UFF parser (in CHW format)
--iterations: Run N iterations
--int8: Run in int8 precision mode
--batch: Set batch size
--device: Set specific cuda device to N
--avgRuns: Set avgRuns to N - perf is measured as an average of avgRuns
```

**Script Output sample:**

```
Average over 100 runs is 1.4675 ms (host walltime is 1.57855 ms, 99% percentile time is 1.54624).
Average over 100 runs is 1.48153 ms (host walltime is 1.59364 ms, 99% percentile time is 1.5831).
Average over 100 runs is 1.4899 ms (host walltime is 1.6021 ms, 99% percentile time is 1.58061).
Average over 100 runs is 1.47487 ms (host walltime is 1.58658 ms, 99% percentile time is 1.56506).
Average over 100 runs is 1.47848 ms (host walltime is 1.59125 ms, 99% percentile time is 1.56266).
Average over 100 runs is 1.48204 ms (host walltime is 1.59392 ms, 99% percentile time is 1.57078).
Average over 100 runs is 1.48219 ms (host walltime is 1.59398 ms, 99% percentile time is 1.5673).
```

- Throughput $\left(\frac{\text{imgs}}{\text{sec}}\right) = \left(\frac{Batch\ Size}{Latency(ms)}\right) * 1000 = \left(\frac{1}{1.48219}\right) * 1000 = 675$
- Latency (msec): 1.48219

In **Figure 18** we observed that CheXNet inference optimized with Native TRT5 C++ API performed ~2X faster than with TF-TRT Integration API optimization, this factor was exposed only with batch size 1 and 2; the outperform of TRT5 C++ API over TF-TRT API gradually decreased in the way the batch size increases. We are still working with the Nvidia Developer group to find out what should be the performance of both APIs implementations.

Further, in the **Figure 19** we showed the latency curves of TRT5 C++ API versus TF-TRT API, lower latency is better, as shown by the TRT5 C++ API.

## 5.7 CheXNet Inference – Throughput with TensorRT™ at ~7ms Latency Target

The ~7ms Latency Target is critical, mainly for real time applications. In this section we have selected all those configurations that performed at that latency target, see below **Table 8** with the selected tests we have included the inference TensorFlow-FP32-CPU Only as reference since its latency was ~115ms.

Table 8. Throughput with TensorRT™ at ~7ms Latency Target

| Inference Mode | Batch Size | Thoughput (img/sec) | Latency (ms) |
|---|---|---|---|
| TensorFlow-FP32-CPU Only | 1 | 9 | 114.9* |
| TensorFlow-FP32-GPU | 1 | 142 | 7.1 |
| TF-TRT5 Integration FP32 | 2 | 272 | 7.6 |
| TF-TRT5 Integration FP16 | 4 | 656 | 6.3 |
| TF-TRT5 Integration INT8 | 8 | 1281 | 6.6 |
| TensorRT™ C++ API INT8 | 8 | 1371 | 5.8 |

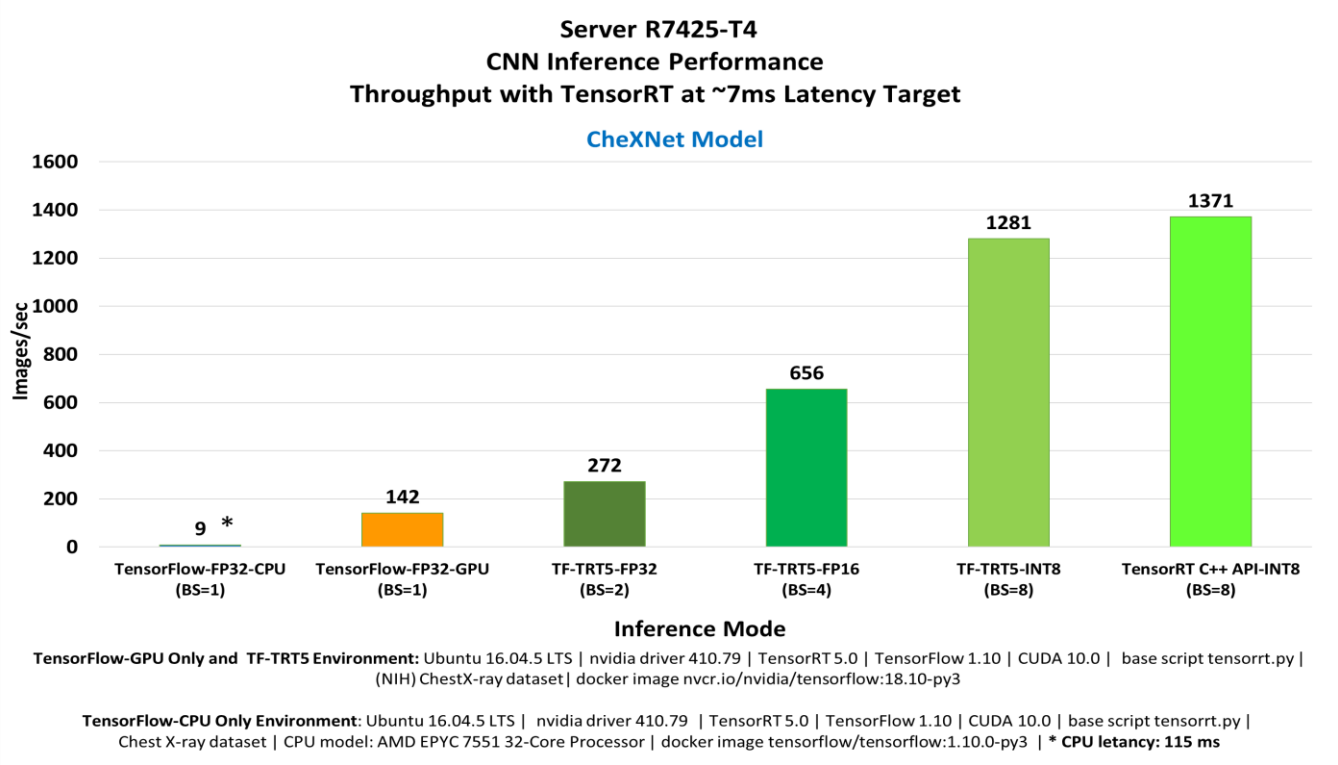


Figure 20. Throughput with TensorRT™ at ~7ms Latency Target

From **Table 8** and **Figure 20** above, we can observe:

- **Native TensorFlow FP32 without TensorRT™ (batch size=1) inference ran on CPU-Only** (AMD EPYC 7551 32-Core Processor) performed 9 img/sec with the minimal latency of ~115 ms. It is a referenceable measurement that shows the different using CPU Only based systems versus GPU based systems.
- The same **Native TensorFlow FP32 without TensorRT™ (batch size=1) inference ran on** GPU performed 142 img/sec at ~7ms latency target. It means ~16X faster than CPU Only (142 vs 9). Now let us use this configuration as a landmark to benchmark the optimized inferences with TensorRT™.

- Using **TF-TRT-FP32** with TensorRT™ (batch size=2) instead of Native TensorFlow FP32 without TensorRT™, improved throughput by ~92% (272 vs 142) at ~7ms latency target.
- Using **TF-TRT-FP16** with TensorRT™ (batch size=4) improved throughput by ~362% (656 vs 142). Also, it decreases latency by ~11%, making it in 6.3ms versus 7.1ms.
- Now, when using **TF-TRT-INT8** (batch size=8) we can appreciate a huge improvement in terms of throughput keeping the ~7ms latency target, we observed a speedup of ~802% (from 1281 vs 142). This is a significant boost in performance.
- On the other hand, comparing **TF-TRT-INT8 Integration** versus **Native TensorRT-INT8 C++ API** (batch size=8) we found that there was a slightly improvement of 7% (1371 vs 1281).

It is important to highlight that there are other implementation factors that could affect the end to end inference's speed when deploying these models into production, so model optimization is just one of those factors and we have demonstrated here how to do it.

# 6 Conclusion and Future Work

- Dell EMC offers an excellent solution with its **PowerEdge R7425 server** based on Nvidia T4 GPU to accelerate Artificial Intelligent workloads, including high-performance Deep learning inference boosted with the Nvidia TensorRT™ library.

- The Native TensorFlow fp32 (without TensorRT™) inference on PowerEdge R7425-T4-16GB server speedup **~16X faster than CPU Only** (AMD EPYC 7551 32-Core Processor). It is a referenceable measurement that shows the benefit of using GPU based systems versus CPU only based systems.

- When accelerating the custom model CheXNet with TensorFlow-TensorRT Integration, the PowerEdge R7425-T4-16GB server performed on average up to **58X faster than native TensorFlow-CPU Only**.

- When accelerating the custom model CheXNet with TensorFlow-TensorRT Integration, the PowerEdge R7425-T4-16GB server performed on average up to **4X faster than native TensorFlow-GPU**.

- The CheXNet inference using **TF-TRT-INT8** precision mode **speedup of ~802%** versus Native TensorFlow FP32 on GPU, at a ~7ms latency target.

- CheXNet inference optimized with Native TRT5 C++ API performed ~2X faster than with TF-TRT Integration API optimization, this factor was exposed only with batch size 1 and 2; the outperform of TRT5 C++ API over TF-TRT API gradually decreased in the way the batch size was bigger. We are still working with the Nvidia Developer group to find how out what should be the performance of both APIs implementations.

- Optimized models with Nvidia TensorRT™ 5 can be deployed in several environments depending of the target application such as scale-out data centers, embedded systems, or automotive product platforms. There are other implementation factors that could affect the end to end inference's speed when deploying the optimized models into these production environments, so model optimization is just one of factors and we have demonstrated in these projects some methods on how to approach it.

# A Troubleshooting

In this section we describe the main issues we faced implementing the custom model CheXNet with Nvidia TensorRT™ and how we solved these:

- **TensorRT™ installation.** For TF-TRT integration, recommended to work with the docker image nvcr.io/nvidia/tensorflow:<tag version>-py3. For Native TRT, recommended to work with the docker image nvcr.io/nvidia/TensorRT™:<tag version>-py3.

- **Python path to TF models**. If using TensorFlow official model as based model, and working within the docker environment, make sure to include the python path to official models once inside the docker: **export PYTHONPATH="$PYTHONPATH:/home/models/".**

- **ImageNet TFRecords**. If using TensorFlow official model as based model, make sure that there are not missing tfrecords in the dataset. If this is the case, update the file **/home/models/official/resnet/imagenet_main.py.**

- **Non-supported Layer Error**. Before building the custom model, double check that the selected framework supports operations by TensorRT™; otherwise, the network subgraph conversion will fail. In our case, we started with Keras-TensorFlow backend framework and the TensorRT™ script failed converting most of the nodes. Then, we switched the model to TensorFlow framework version and resolved the issues. See Supported operations for TF-TRT Integration [13].

- **Unimplemented: Not supported constant type at Const_1/Const_5 Error**. Error related with the same issue above. By the time the tests were conducted, it looks like some Keras layers were not supported by TF-TRT Integration.

- **Not conversion function registered for layer IteratortoGetNet Error**. This error was thrown by the system because the input function was not configurated in the model. When building the custom model, make sure to define the **input_function** properly, and when exporting the model with **export_savedmodel** make sure assure to configure the **input_receiver_fn** for serving as *input_receiver_fn=export.build_tensor_serving_input_receiver_fn(shape, batch_size=FLAGS.batch_size)*

- **Cuda Error in allocate:2. Subgraph conversion error for subgraph_index 1 due to: "Internal: Engine building failure" SKIPPING (437 nodes)**. Sometimes this error is related to the GPU memory capacity; so, try to run the tests with lower batch size and one precision mode at the time.

- **Tensor batch_normalization/beta is not found in resnet_v2_imagenet_checkpoint error**. In our case we built the custom model CheXNet using transfer learning and the TensorFlow official pre-trained ResnetV2_50 checkpoints downloaded from its website. This error was produced because by the time the model was trained we didn't place our variables in the same

variable scope as it was in the restored checkpoints. Solution: we customized official TensorFlow base script **resnet_model.py** and placed the variables in the same variable scope name **"resnet_model"** as it was in the official checkpoints downloaded previously. So, we added this code line in the model function *with tf.variable_scope("resnet_model"):*. For more information see What's the difference of name scope and a variable scope in TensorFlow [8].

- **TensorFlow Serving for Inference**. When building and training the custom model, save the trained model with TensorFlow Serving for Inference. To do so, export the trained model as **SavedModel** with the Estimator function **export_savedmodel.** Also include the PREDICT Estimator's method to enable the inferences mode. For predict mode, it is required to provide the **export_output** argument to the **EstimatorSpec**, it defines signatures for tensorflow serving when Serving a SavedModel. Specify the inputs and outputs node manes, which will be needed later on by the TensorRT™ library. See Serving Pre-Modeled and Custom TensorFlow Estimator with Tensorflow Serving [10].

- **ValueError: Negative dimension size caused by subtracting 8 from 7 for 'import/resnet_model/average_pooling2d/AvgPool' (op: 'AvgPool') with input shapes: [128,7,7,2048]**. Problem solved updating the base model script **resnet_model.py** , in the model function section, changing from padding='VALID' to **padding='SAME'** : *inputs = tf.layers.average_pooling2d( inputs=inputs, pool_size=pool_size, strides=1, padding='SAME', data_format=data_format)*

# B References


[1] P. Rajpurkar, J. Irvin, K. Zhu, B. Yang, H. Mehta, T. Duan, D. Ding, A. Bagul, C. Langlotz, K. Shpanskaya et al., “CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning,” arXiv preprint arXiv:1711.05225, 2017 [Online]. Available: https://arxiv.org/abs/1711.05225

[2] Nvidia News Center, ”NVIDIA AI Inference Performance Milestones: Delivering Leading Throughput, Latency and Efficiency” [Online]. Available: https://news.developer.nvidia.com/nvidia-ai-inference-performance-milestones-delivering-leading-throughput-latency-and-efficiency/

[3] TensorFlow Guide “tf.estimator.Estimator, Estimator”[Online]. Available: https://www.tensorflow.org/guide/estimators

[4] TensorFlow Guide, “Creating Custom Estimators” [Online]. Available: https://www.tensorflow.org/guide/custom_estimators

[5] “Multithreaded predictions with TensorFlow Estimators.” [Online]. Available: https://medium.com/element-ai-research-lab/multithreaded-predictions-with-tensorflow-estimators-eb041861da07

[6] TensorFlow Official Models, “ResNet in TensorFlow“ [Online]. Available: https://github.com/tensorflow/models/tree/master/official/resnet

[7] TensorFlow Guide, “Checkpoints. Restoring your model”. [Online]. Available: https://www.tensorflow.org/guide/checkpoints#checkpointing_frequency

[8] Stackoverflow, “Variables. What's the difference of name scope and a variable scope in tensorflow?” [Online]. Available: https://stackoverflow.com/questions/35919020/whats-the-difference-of-name-scope-and-a-variable-scope-in-tensorflow . TensorFlow Guide “Sharing variables” [Online]. Available: https://www.tensorflow.org/guide/variables#sharing_variables

[9] TensorFlow API , “Exports inference graph as a SavedModel” . [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator#export_savedmodel

[10] Medium, “Serving Pre-Modeled and Custom Tensorflow Estimator with Tensorflow Serving“. [Online]. Available: https://medium.com/@yuu.ishikawa/serving-pre-modeled-and-custom-tensorflow-estimator-with-tensorflow-serving-12833b4be421

[11] Nvidia, “Accelerating Inference In TensorFlow With TensorRT™ User Guide”. [Online]. Available: https://docs.nvidia.com/deeplearning/dgx/integrate-tf-trt/index.html

[12] Nvidia, “Using TF-TRT. Accelerating Inference In TensorFlow With TensorRT™ User Guide” . [Online]. Available: https://docs.nvidia.com/deeplearning/dgx/integrate-tf-trt/index.html#usingtftrt

[13] Nvidia, “Accelerating Inference In TensorFlow With TensorRT™ User Guide”, “Supported Ops” [Online]. Available: https://docs.nvidia.com/deeplearning/dgx/integrate-tf-trt/index.html#support-ops . “Working With Deep Learning Frameworks” [Online].Available: https://docs.nvidia.com/deeplearning/sdk/TensorRT™-developer-guide/index.html#build_model

[14] TensorFlow Guide, “A Tool Developer's Guide to TensorFlow Model Files”. [Online]. Available: https://www.tensorflow.org/guide/extend/model_files#freezing

[15] Nvidia, “Working with TensorRT™ Using The C++ API, TensorRT™ Developer Guide”. [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/TensorRT™-developer-guide/index.html#c_topics

[16] Nvidia, “Importing a TensorFlow Model Using The C++ UFF Parser API”. [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/TensorRT™-developer-guide/index.html#import_tf_c

[17] Nvidia, “Serializing A Model In C++. [Online]”. Available: https://docs.nvidia.com/deeplearning/sdk/TensorRT™-developer-guide/index.html#serial_model_c

[18] TensorFlow Guide , “TensorFlow Data Formats”. [Online]. Available: https://www.tensorflow.org/guide/performance/overview#data_formats

# C Appendix - PowerEdge R7425 – GPU Features

| Server | | | R7425-T4 |
|---|---|---|---|
| CPU | | | |
| | | CPU model | AMD EPYC 7551 32-Core Processor |
| GPU | | | |
| | | GPU model | Tesla T4-16GB |
| | | GPU Architecture | NVIDIA Turing |
| | | Attached GPUs | 6 |
| Features per GPU | | | |
| | | Driver Version | 410.79 |
| | | Compute Capability | 7.5 |
| Multiprocessor | | | |
| | | Multiprocessors (MP) | 40 |
| | | CUDA Cores/MP | 64 |
| | | CUDA Cores | 2,560 |
| | | Clock Rate (GHz) | 1.59 |
| Memory | | | |
| | | Global Memory Bandwidth (GB/s) | 300 |
| | | Global Memory Size (GB) | 16 |
| | | Constant Memory Size (KB) | 65 |
| | | L2 Cache Size (MB) | 4 |
| Bus Interface PCIe | | | |
| | | Generation | 3 |
| | | Link Width | 16 |
| Peak Performance Floating Point Operations (FLOP) and TOPS | | | |
| | | Single-Precision - FP32 (Teraflop/s) | 8.1 |
| | | Mixed Precision - FP16/FP32 (TeraFLOP/s) | 65 |