Deep Learning Inference on PowerEdge R7425

Abstract

This whitepaper looks at the performance and efficiency of Deep learning inference when using the Dell EMC PowerEdge R7425 server with NVIDIA T4-16GB GPU. 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.

Mar-19

Revisions

Date	Description
Mar-19	Initial release

Acknowledgements

This paper was produced by the following persons:

Authors:

Bhavesh Patel, Dell EMC Server Advanced Engineering

Vilmara Sanchez, Dell EMC Server Advanced Engineering

Contributor: Josh Anderson, Dell EMC Server System Engineering

Table of Contents

1 Overview 4
2 Introduction 4
2.1 Deep Learning 4
2.2 Deep Learning Inferencing 5
2.3 What is TensorRT™? 6
2.4 Dell EMC PowerEdge R7425 Server 7
3 TensorRT Implementations 8
3.1 TensorFlow-TensorRT 5 Integration (TF-TRT) 8
3.2 TensorRT™ Python API 9
3.3 TensorRT™ C++ API 10
4 Test Methodology 10
4.1 Criteria 10
4.2 Test Design 11
4.3 Environment Setup 12
5 Performance Results 13
5.1 Phase 1: Inference with TF-TRT 5 Integration 13
5.1.1 ResNet-50 Inference performance: Throughput vs Batch size 13
5.1.2 All Models: Images/sec vs batch size vs Neural models 15
5.1.3 All Models - R7425-T4-16GB versus Other servers and NVIDIA GPU 16
5.2 Phase 2: Inference Image Classification with TensorRT™ 5 C++ API 17
5.2.1 ResNet50 – TensorFlow 17
5.2.2 ResNet-50 – TensorFlow vs Caffe 19
5.2.3 Resnet-50 - R7425-T4-16GB versus Other Servers 20
5.3 Phase 3: Inference Performance using ResNet50 – TF-TRT versus TRT 5 C++ API 21
5.4 Phase 4: Inference Image Classification on Neural Models – Other Aspects 22

6 Hardware metrics 24
6.1 Percentage of GPU Utilization vs GPU Memory Utilization 25
6.2 Core GPU Temperature 26
6.3 Power Draw 26
7 Conclusion and Future Work 27
8 References 28
9 Annex A - PowerEdge R7425-T4-16GB Server – GPU Features 29

Acknowledgements

We would like to acknowledge the following individuals, Jaime Edwards (Director of PowerEdge Advanced Engineering) for setting the direction of this project, April Berman (PowerEdge Acceleration Product Manager), Pallavi Arora (PowerEdge R7425 Product Manager) for providing us the resources for this paper.

We would also like to acknowledge:

❖ Nvidia TensorRT™ support team & Nvidia account team for their expedited support.

1 Overview

This paper talks about Deep Learning Inference using NVIDIA T4-16GB GPU and TensorRT™. The NVIDIA T4-16GB GPU is based on their latest Turing architecture which significantly boosts graphic performance using a new GPU processor (streaming multiprocessor) with improved shader execution efficiency and new memory system architecture that supports GDDR6 memory technology. Turing's Tensor cores provide higher throughput and lower latency for AI Inference applications.

Dell EMC PowerEdge R7425 is based on AMD's EPYC™ architecture and since EPYC™ architecture supports higher number of PCIe Gen3 x16 lanes, it allows the server to be used as a scale-up inference server. It becomes a perfect solution when running large production-based AI workloads where both throughput and latency are important.

In this paper we tested the inference optimization tool Nvidia TensorRT™ 5 on the Dell EMC PowerEdge R7425 server to accelerate CNN image classification applications and demonstrate its capability to provide higher throughput & lower latency for neural models like ResNet50. During the tests, we ran inferences of image classification models in different precision modes on the server R7425 using NVDIA T4-16GB GPU, with both implementation of TensorRT™ i.e. the native TensorRT™ C++ API and the integrated TensorFlow-TensorRT™ integration library. TensorFlow was used as the primary framework for the pre-trained models to compare the optimized performance in terms of throughput (images/sec) and latency (milliseconds).

2 Introduction

2.1 Deep Learning

Deep Learning consists of two phases: Training and inference. As illustrated in ***Figure 1***, 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 [1].

The major difference between training and inference is training employs *forward propagation* and *backward propagation* (two classes of the deep learning process) whereas inference mostly consists of forward propagation. To generate models with good accuracy, the training phase involves several training iterations and substantial training data samples, thus requiring many-core CPUs or GPUs to accelerate performance.

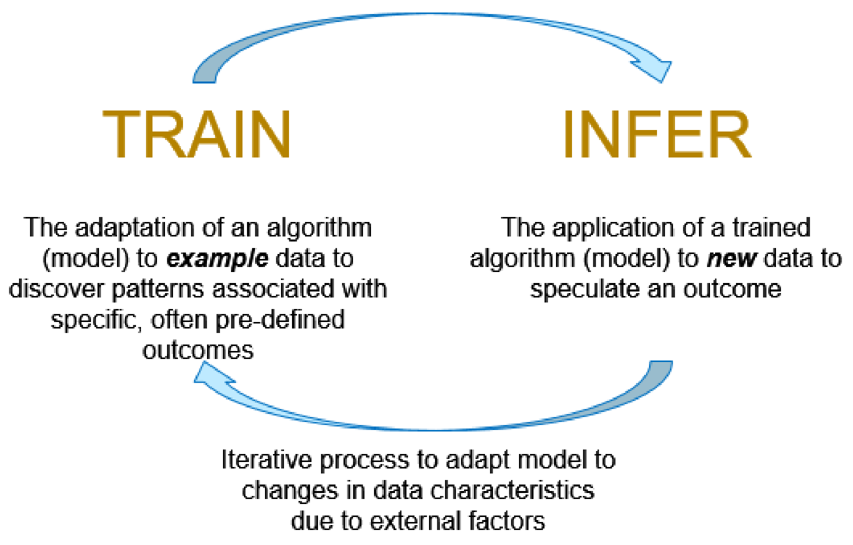

TRAIN
INFER
The adaptation of an algorithm (model) to ***example*** data to discover patterns associated with specific, often pre-defined outcomes
The application of a trained algorithm (model) to ***new*** data to speculate an outcome
Iterative process to adapt model to changes in data characteristics due to external factors

Figure 1. Deep Learning Phases

2.2 *Deep Learning Inferencing*

After a model is trained, the generated model may be *deployed* (forward propagation only) e.g., on FPGAs, CPUs or GPUs to perform a specific business-logic function or task such as identification, classification, recognition and segmentation. See ***Figure 2.***

1. The focus of this whitepaper will be on the power of Dell EMC PowerEdge R7425 using NVIDIA T4-16GB GPUs to accelerate image classification and deliver high-performance inference throughput and low latency using various implementations of TensorRT™ an excellent tool to speed up inference.

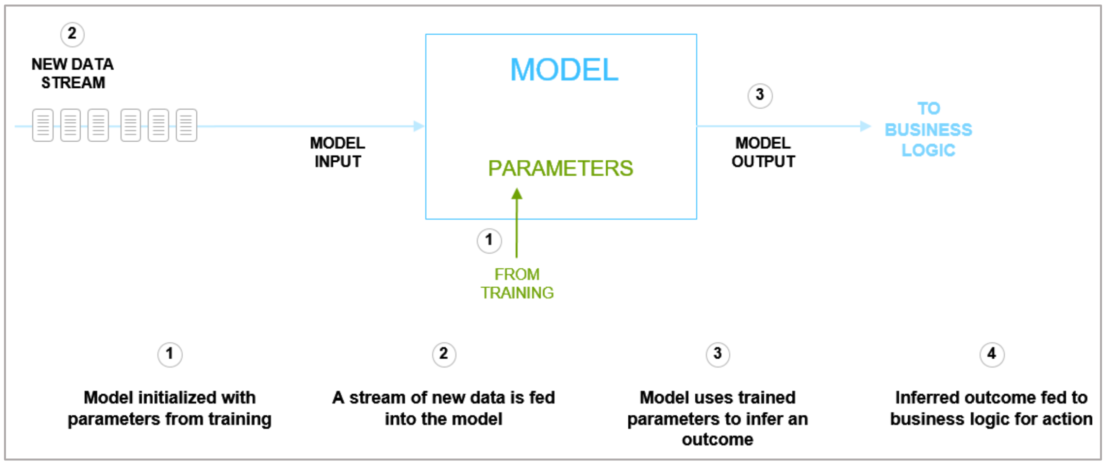

2
NEW DATA STREAM
MODEL INPUT
MODEL
PARAMETERS
1
FROM TRAINING
3
MODEL OUTPUT
TO BUSINESS LOGIC
1
Model initialized with parameters from training
2
A stream of new data is fed into the model
3
Model uses trained parameters to infer an outcome
4
Inferred outcome fed to business logic for action

Figure 2. Inference Workflow

2.3 What is TensorRT™?

The core of TensorRT™ is a C++ library that facilitates high performance inference on NVIDIA graphics processing units (GPUs). It is designed to work in a complementary fashion with training frameworks such as TensorFlow, Caffe, PyTorch, MXNet, etc. It focuses specifically on running an already trained network quickly and efficiently on a GPU for generating a result (a process that is referred to in various places as scoring, detecting, regression, or inference).

Some training frameworks such as TensorFlow have integrated TensorRT™ so that it can be used to accelerate inference within the framework. Alternatively, TensorRT™ can be used as a library within a user application. It includes parsers for importing existing models from Caffe, ONNX, or TensorFlow, and C++ and Python APIs for building models programmatically.

*Figure **3**. TensorRT Scheme. Source: Nvidia**Figure 3.*** TensorRT™ is a high performance neural network inference optimizer and runtime engine for production deployment.

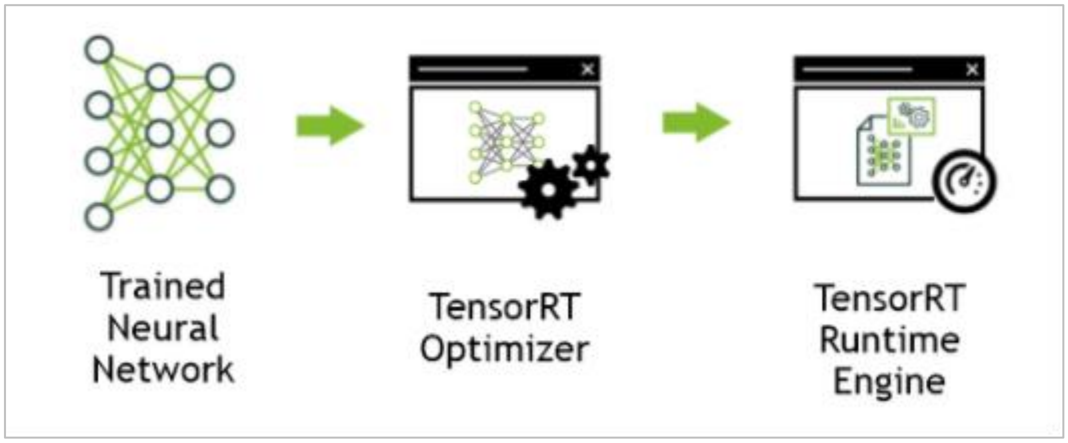


Figure 3. TensorRT Scheme. Source: Nvidia

TensorRT™ optimizes the network by combining layers and optimizing kernel selection for improved latency, throughput, power efficiency and memory consumption. If the application specifies, it will additionally optimize the network to run in lower precision, further increasing performance and reducing memory requirements.

The following figure shows TensorRT™ defined as part high-performance inference optimizer and part runtime engine. It can take in neural networks trained on these popular frameworks, optimize the neural network computation, generate a light-weight runtime engine (which is the only thing you need to deploy to your production environment), and it will then maximize the throughput, latency, and performance on these GPU platforms.

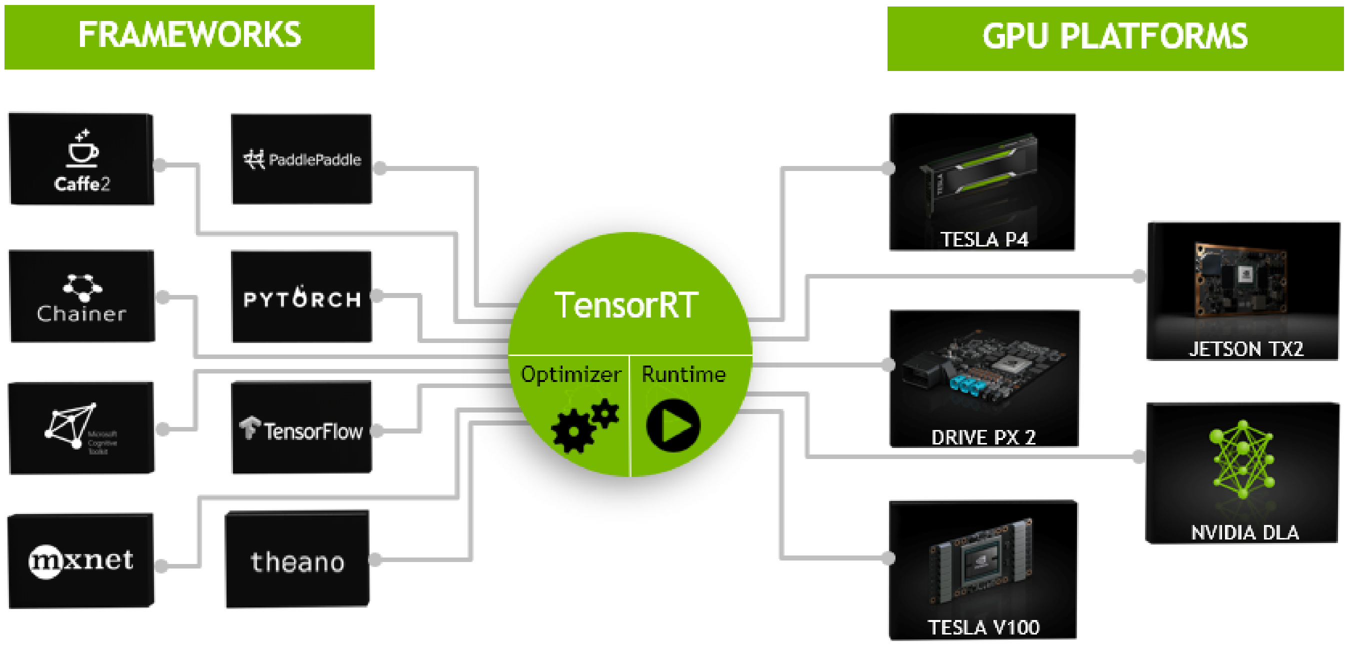


Figure 4. TensorRT™ is a Programmable Inference Accelerator

Nvidia TensorRT™ 5 is a high performance deep learning inference and run-time optimizer delivering low latency and high throughput for production deployment. TensorRT™ has been successfully used in a wide range of applications including autonomous vehicles, robotics, video analytics, automatic speech recognition among others. NVIDIA TensorRT™ 5 supports the newest NVIDIA Turing Tensor Core architecture and expands the set of neural network optimizations for a broader array of mixed-precision workloads [1].

2.4 Dell EMC PowerEdge R7425 Server

Dell EMC PowerEdge R7425-T4-16GB server supports the latest GPU accelerator 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 **5***.

Figure 5. DELL EMC PowerEdge R7425

The Dell™ PowerEdge™ R7425 is Dell's latest 2-socket, 2U rack server designed to run complex workloads using highly scalable memory, I/O, and network options The systems feature bases on AMD high performance processor, which named AMD SP3, supports 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.

3 TensorRT Implementations

TensorRT provides three tools to optimize the models for inference: TensorFlow-TensorRT Integration (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 APIs include methods for building models programmatically. Below is the brief description of each method, the assumption is that all of them start with a trained model.

3.1 TensorFlow-TensorRT 5 Integration (TF-TRT)

TensorRT™ works with training frameworks such as TensorFlow, Caffe, PyTorch, and MXNet. In this case, we used TensorFlow which has integrated TensorRT™ so that it can be used to accelerate inference within the framework. This is a contribution library that offers performance as well as robustness, built on top off. The command line to import TF-TRT integration into TensorFlow is:

```
# import tensorflow.contrib.tensorrt as trt
```

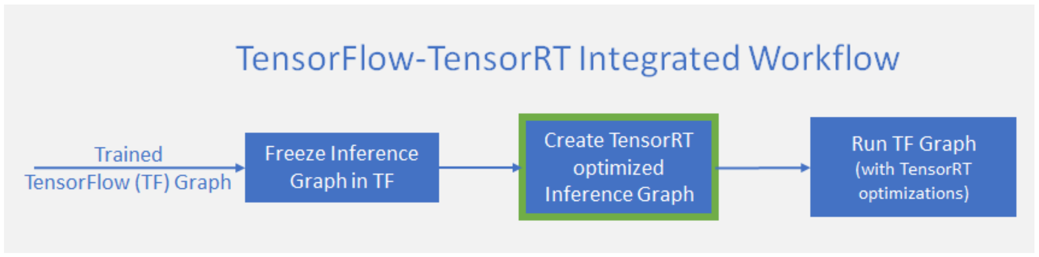

TensorFlow-TensorRT Integrated Workflow
Trained TensorFlow (TF) Graph
Freeze Inference Graph in TF
Create TensorRT optimized Inference Graph
Run TF Graph (with TensorRT optimizations)

Figure 6. Workflow working with TensorRT using TensorFlow-TensorRT Integration [2]

Figure 6 shows the workflow on how TF-TRT works: a trained model in TensorFlow is converted to a frozen graph; after freezing the TensorFlow graph for inference, TensorRT™ is used to create the TensorRT™ optimized inference graph, then the optimized graph is replaced as the default graph to run the inference.

The optimization graph process consists of parsing the model and applying optimization to portions of the graph as much as possible, the remaining portion of the graphs that are not subject to optimization are executed by TensorFlow. See Supported operations in TensorFlow for more information [3].

To run the inference using INT8 precision, it is required to calibrate the trained TensorFlow model first and then apply the TensorRT™ optimization, see ***Figure 7***. The Calibration process consists in calculating the weights of the model at the optimal scaling factor from FP32 to INT8:

- Run inference in FP32 on calibration dataset
- Collect required statistics
- Run calibration algorithm → optimal scaling factors
- Quantize FP32 weights → INT8
- Generate "Calibration Table" and INT8 execution engine

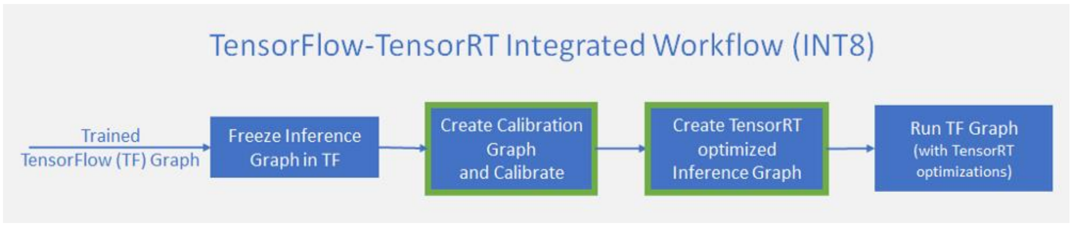

TensorFlow-TensorRT Integrated Workflow (INT8)
Trained TensorFlow (TF) Graph
Freeze Inference Graph in TF
Create Calibration Graph and Calibrate
Create TensorRT optimized Inference Graph
Run TF Graph (with TensorRT optimizations)

Figure 7. Workflow TensorFlow-TensorRT Integration using INT8 precision [1]

Using TF-TRT with precision modes lower than FP32, that is, FP16 and INT8, improves the performance of inference. The FP16 precision mode uses Tensor cores and half-precision hardware instructions, if possible, while the INT8 precision mode uses Tensor cores and integer hardware instructions [4].

3.2 TensorRT™ Python API

Different from TF-TRT, the Python API doesn't use any framework. TensorRT™ Python API is available for the x86_64 platform only. The command line to import the TensorRT™ Python API is:

```
# import tensorrt as trt
```

According to the technical documentation, the main benefit of the Python API is that data preprocessing and postprocessing is easy to use because it has a variety of libraries like NumPy and SciPy. For more information please see Deep Learning SDK Documentation [5].

3.3 TensorRT™ C++ API

TensorRT™ provides a C++ implementation on all supported platforms. C++ API includes similar steps to the Python API implementation and doesn't use any framework either. According to Nvidia's documentation, the C++ API should be used in any performance critical scenarios, as well as in situations where safety is important, for example, automotive or real time inferences [6].

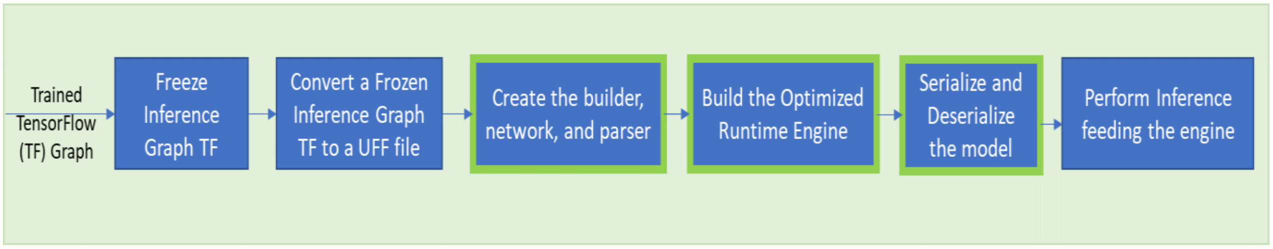

Trained TensorFlow (TF) Graph
Freeze Inference Graph TF
Convert a Frozen Inference Graph TF to a UFF file
Create the builder, network, and parser
Build the Optimized Runtime Engine
Serialize and Deserialize the model
Perform Inference feeding the engine

Figure 8. Workflow working with TensorRT Using the TensorRT C++ API

In ***Figure 8***, the workflow shows the steps to work directly with the TensorRT™ C++ API:

- The trained TensorFlow model needs to be frozen for inference and converted to UFF format
- Invoke the TensorRT™ builder to create an optimized runtime engine from the network
- Serialize and deserialize the engine to be recreated at runtime
- Feed the optimize engine with data to perform inference

4 Test Methodology

In this paper, we tested the inference performance of several pre-trained neural models using different precision settings. We compared throughput and latency and looked at the benefits of using Nvidia TensorRT™. As optimizer implementations, we used TF-TRT integration and TensorRT™ C++ API.

4.1 Criteria

1. In terms of server, we selected Dell EMC PowerEdge R7425 which includes the Nvidia Tesla T4-16GB GPU, the most advanced accelerator for AI inference workloads. According to Nvidia, T4-16GB's new Turing Tensor Cores accelerates INT8 precision more than 2x faster than the previous generation low-power offering [7].
2. For the framework and inference optimizer tools, we selected TensorFlow, TF-TRT integration and TensorRT™ C++ API since these have better technical support and a wide variety of pre-trained models readily available.
3. 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.

4.2 Test Design

The tests were conducted in 4 phases as shown in ***Figure 9***

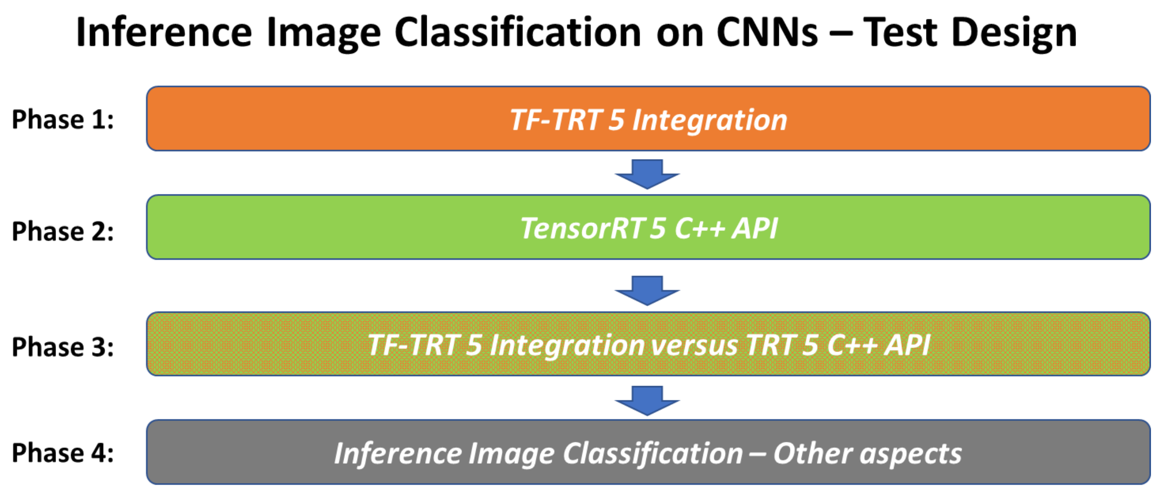

Inference Image Classification on CNNs – Test Design
Phase 1:
TF-TRT 5 Integration
Phase 2:
TensorRT 5 C++ API
Phase 3:
TF-TRT 5 Integration versus TRT 5 C++ API
Phase 4:
Inference Image Classification – Other aspects

Figure 9. TensorRT Test Design

Phase 1 the image classifications were run using TF-TRT 5.0 integrated. In this step we tested the inference with several pre-trained models in TensorFlow with ImageNet.

Phase 2, the image classifications were run using native TensorRT™ 5.0 C++ API without the framework, but still using pre-trained models.

Phase 3, we compared the results running the image classifications using TF-TRT Integration versus TRT 5.0 C++ API.

Phase 4 several inference modes were tested to highlight the benefit of running inference on GPU as well as native precision modes in TensorFlow versus optimized precision modes with TensorRT™.

Once the optimized models are generated for inference, they can be deployed in production environments such as datacenter, cloud and edge node like autonomous vehicles smart cameras. This paper does not focus on how these models are deployed or the optimal way to deploy them.

Here is a high-level summary of the steps involved when running inference workloads:

- **Use Case:** Optimized Image Classification on CNNs with TensorFlow and TensorRT™
- **Models**: The pre-trained models included vg116, vgg19, inception_v3, inception_v4, resnet_v1_50, and resnet_v2_50
- **Framework**: TensorFlow for pre-trained models, TensorRT version: TensorRT 5
- **TensorRT™ implementations**: TensorFlow-TensorRT integration and TensorRT C++ API
- **Server:** Dell EMC PowerEdge R7425

- **GPU:** NVIDIA T4-16GB
- **Performance**: The performance metrics used for comparison across inference precision modes were throughput (images per second) and the latency (ms)
- **Datasets** – IMAGENET (ILSVRC2012) for pre-trained models
- **Samples code**: TensorRT™ samples provided by Nvidia included in its container images
- **Software stack configuration**: The tests were conducted using the docker container environment
- **Precision Mode**: Inference with TensorRT in INT8

The below ***Table 1*** lists the tests conducted in different precision modes and TensorRT implementations. The script samples can be found in within the Nvidia container image.

Table 1. Tests Conducted

Inference Mode	TensorRT™ Implementation	Test script
TensorFlow CPU FP32	n/a	tensorrt_only_cpu.py
TensorFlow GPU FP32	n/a	tensorrt.py
Integration TF-TRT5 FP32	TF-TRT Integration	tensorrt.py
Integration TF-TRT5 INT8	TF-TRT Integration	tensorrt.py
Integration TF-TRT5 INT8	TF-TRT Integration	inference.py
Native TRT5 INT8 – C++ API	C++ API	trtexec.cpp

4.3 Environment Setup

The below ***Table 2*** shows the software stack configuration for server R7425 using NVIDIA T4-16GB.

Table 2. 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	410-72 - 410.79
CUDA	10.0
TensorFlow	TensorFlow 1.10
TensorRT	TensorRT™ 5.0

Software	Version
Docker Image for TensorFlow CPU only	tensorflow/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	December 2018

5 Performance Results

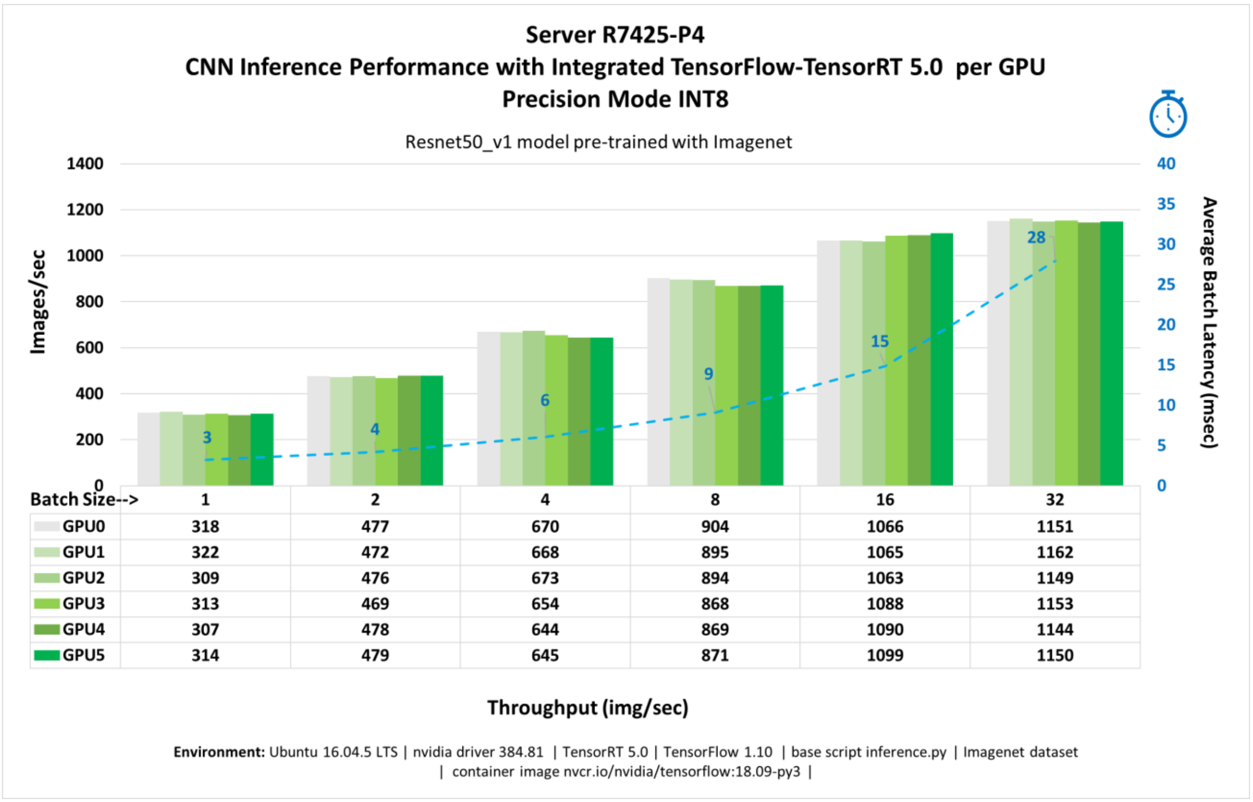
5.1 *Phase 1: Inference with TF-TRT 5 Integration*

In Phase 1 we ran inference tests using TF-TRT integrated. ResNet50 model was used to run inference on each GPU and we also tested other models to compare performance, finally, we ran comparative testing with other servers.

5.1.1 ResNet-50 Inference performance: Throughput vs Batch size

The throughput tests with pre-trained model ResNet50 was run on each GPU using different batch sizes (from 1 to 32).

Figure ***10*** shows the results for throughput (images/sec). The results were consistent across the GPUs as well as the latency (expressed in the right vertical axes).



Batch Size-->	1	2	4	8	16	32
GPU0	318	477	670	904	1066	1151
GPU1	322	472	668	895	1065	1162
GPU2	309	476	673	894	1063	1149
GPU3	313	469	654	868	1088	1153
GPU4	307	478	644	869	1090	1144
GPU5	314	479	645	871	1099	1150

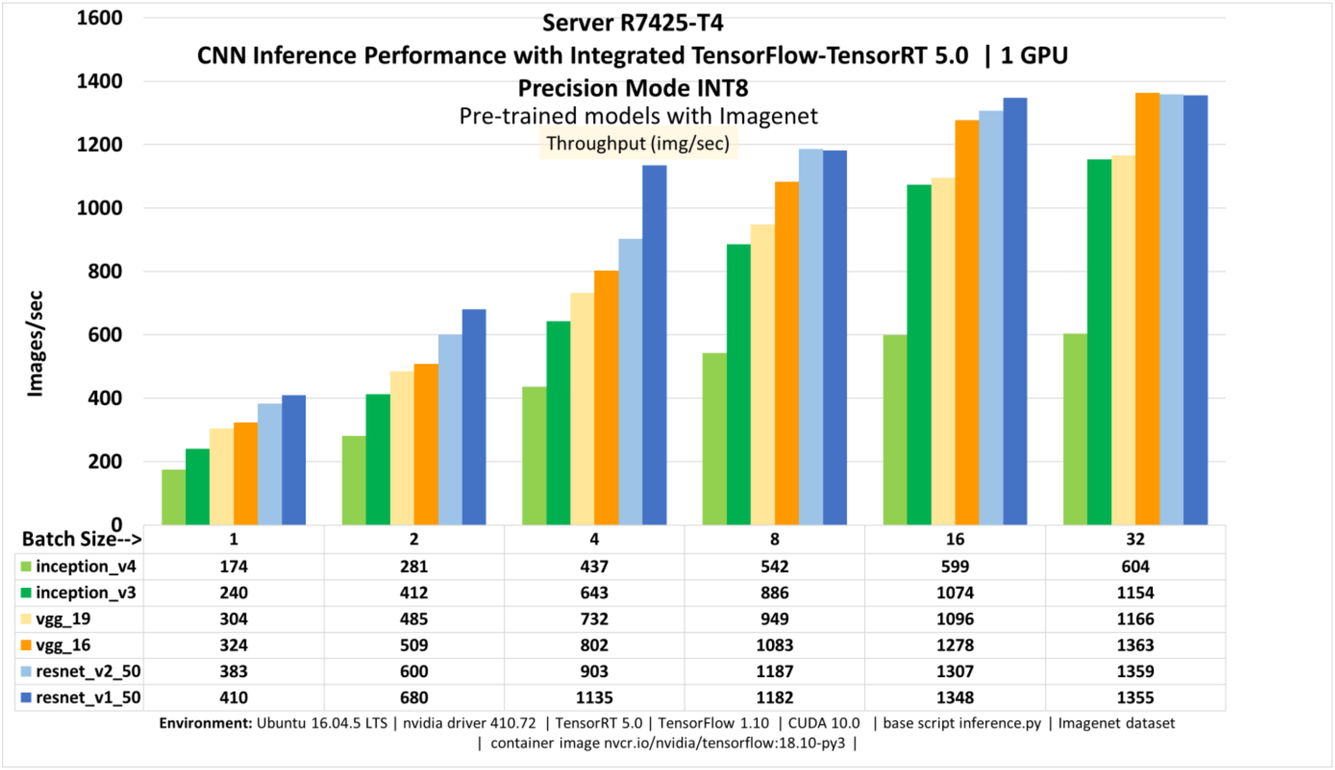
Figure 10. Figure 10. Resnet_50 Inference on each GPU. Server with 6 GPU's

The plots above show that we can achieve latency of 7ms using batch sizes 1-8 and looking at the y-axis, it shows that with batch size of 4 we get 670 images/sec within 7ms latency window.

When running the test, we realized the inference was conducted by default at the device 0, which means currently the TensorRT™ inference engine doesn't work with GPU-GPU communications to maximize the utilization of the GPUs available in the server. If the goal is to run the same graph in multiple GPU's to increase the throughput, Nvidia suggested that for now use the native TensorFlow.

On the other hand, TensorRT Inference Server (TRTIS) supports multiple GPU's but it does not support running a single inference distributed across multiple GPU's. TRTIS can run multiple models (and/or multiple instances of the same model) on multiple GPUs to increase throughput. We will be studying how to implement TRTIS in our next paper related to GPU inferencing.

5.1.2 All Models: Images/sec vs batch size vs Neural models



Batch Size-->	1	2	4	8	16	32
🟩 inception_v4	174	281	437	542	599	604
🟩 inception_v3	240	412	643	886	1074	1154
🟨 vgg_19	304	485	732	949	1096	1166
🟧 vgg_16	324	509	802	1083	1278	1363
🟦 resnet_v2_50	383	600	903	1187	1307	1359
🟦 resnet_v1_50	410	680	1135	1182	1348	1355

Environment: Ubuntu 16.04.5 LTS | nvidia driver 410.72 | TensorRT 5.0 | TensorFlow 1.10 | CUDA 10.0 | base script inference.py | Imagenet dataset | container image nvcr.io/nvidia/tensorflow:18.10-py3 |

Figure 11. Throughput Inference Performance with Several Neural Models and Batch Sizes

Here inference tests were conducted using different neural models in different batch sizes. *Figure* ***11*** and *Figure* ***12****Error! Reference source not found.* show the throughput and latency running inference in batch sizes 1, 2, 4, 8, 26, and 32. ResNet50 produced the highest throughput (images/sec) with the lowest latency.

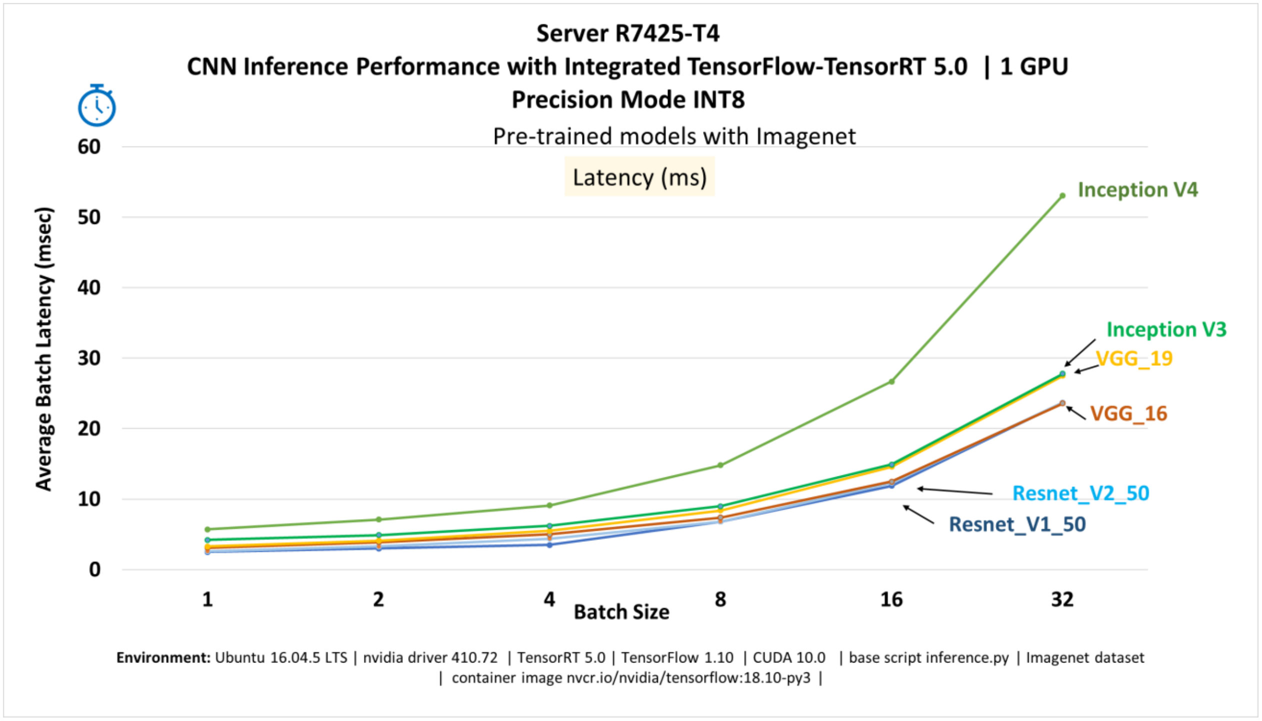


Figure 12. Latency Inference Performance with Several Neural Models and Batch Sizes

5.1.3 All Models - R7425-T4-16GB versus Other servers and NVIDIA GPU

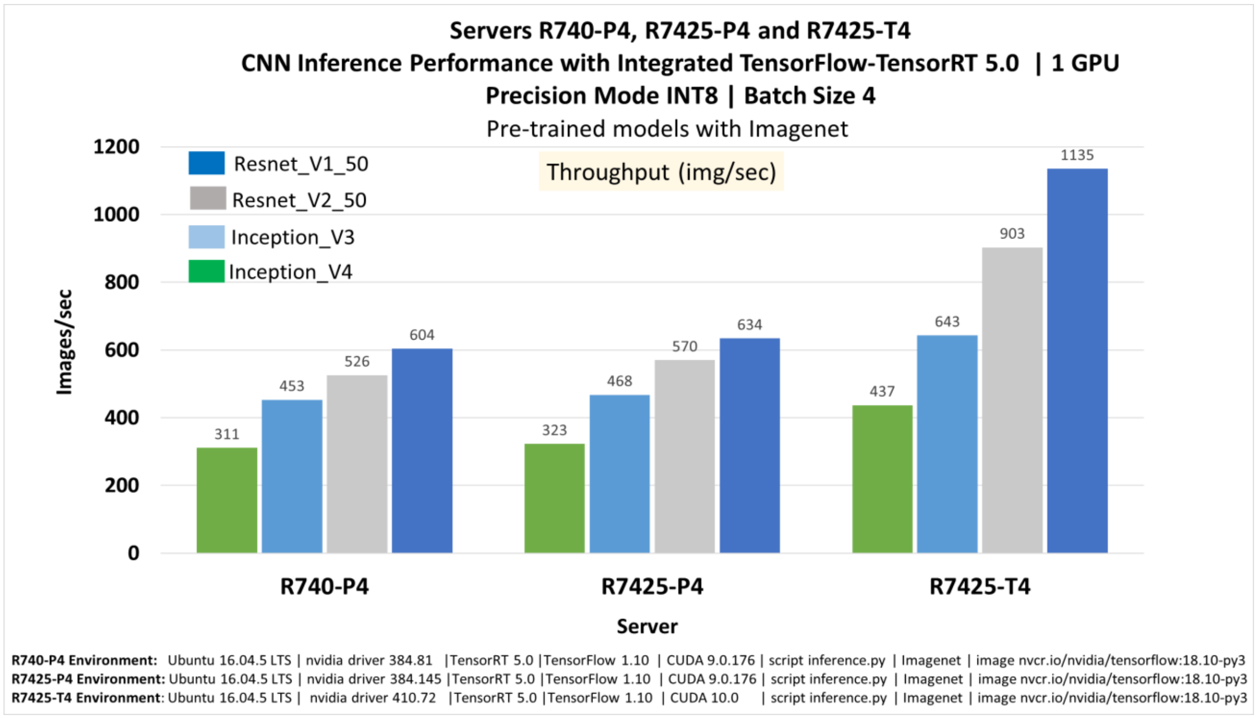


Figure 13. Throughput Inference Performance on R7425-T4-16GB Server versus Other Servers Using TF-TRT

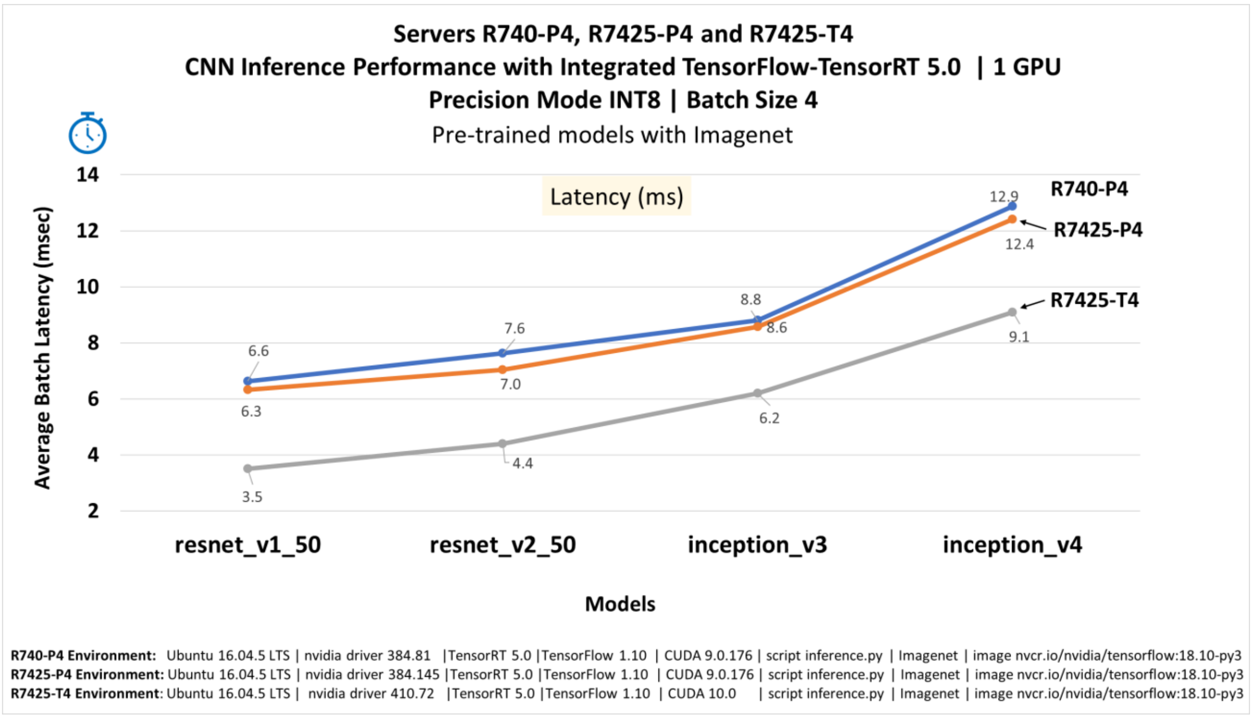


Figure 14. Latency Inference performance on R7425-T4-16GB Server versus other servers

In this section, the inference tests using several models were conducted on the servers R740-P4 and R7245-P4 and their results were compared against the R7425-T4-16GB results. The server R7425-T4-16GB performed around 1.8X faster versus the other servers on the model ResNet50 at half the latency. See ***Figure 13*** and *Figure **14***.

5.2 Phase 2: Inference Image Classification with TensorRT™ 5 C++ API

Due the highest inference performance shown by the model ResNet50 in Phase 1, we used ResNet50 in Phase 2.

In Phase 2, ResNet50 was tested with TensorRT C++ API with different batch sizes. We also ran a competitive test using Caffe pre-trained models to see if there are any performance differences. Finally, we ran a comparison with pre-trained models with other servers.

5.2.1 ResNet50 – TensorFlow

When running tests with TensorRT C++ API, ResNet50 optimized inference engine produces higher throughput at a lower latency.

For instance, the highest throughput of 4,000 images/sec for batch size 64 and the lowest latency of 0.95 milliseconds for batch size 1. This could be useful in different application areas where throughput is more important e.g. image classification vs applications which require low latency like fraud prevention or theft monitoring. *Figure **15*** and *Figure **16*** shows the performance of ResNet50 for PowerEdge R7425 using NVIDIA T4 GPU using different batch sizes with C++ API implementation.

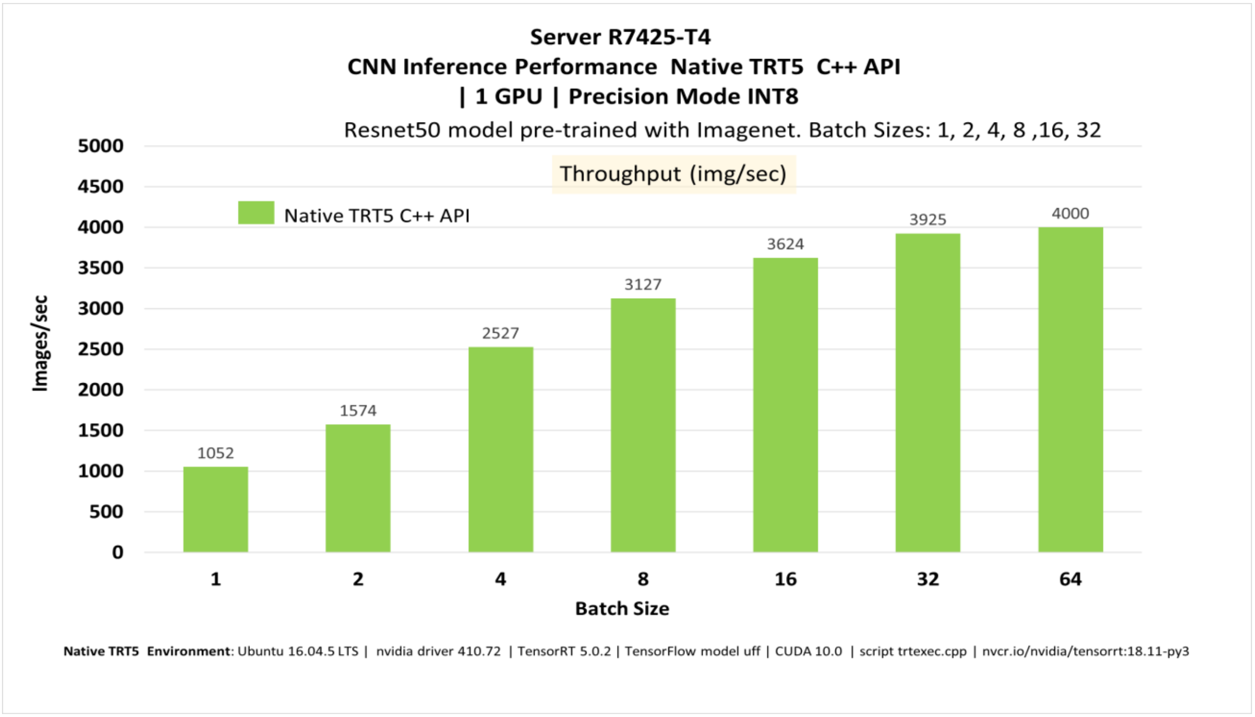


Figure 15. Resnet_50 Throughput Inference Performance Using TensorRT C++ API

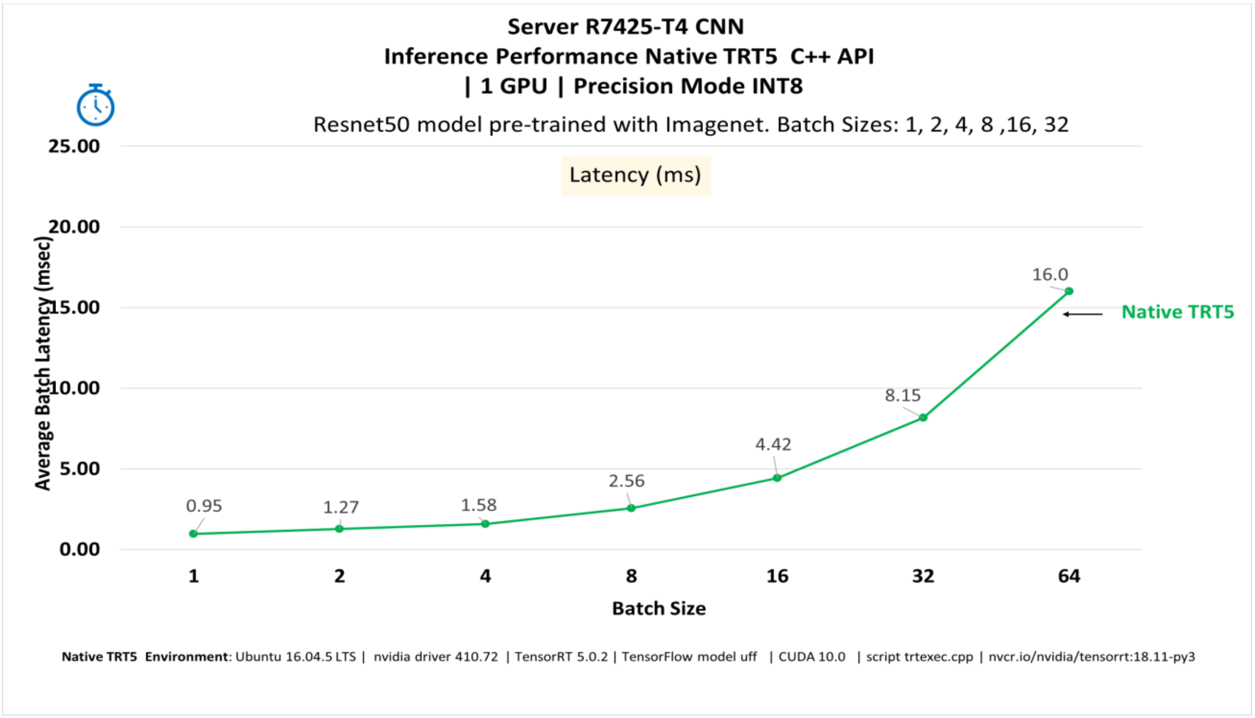


Figure 16. Resnet_50 Latency Inference Performance Using TensorRT C++ API

*Figure **16*** shows also that the target latency ~8ms can be reached with batch sizes below 32.

5.2.2 ResNet-50 – TensorFlow vs Caffe

Another test was comparing the optimized pre-trained models in TensorFlow and Caffe frameworks, and there was no difference in performance using both the models.

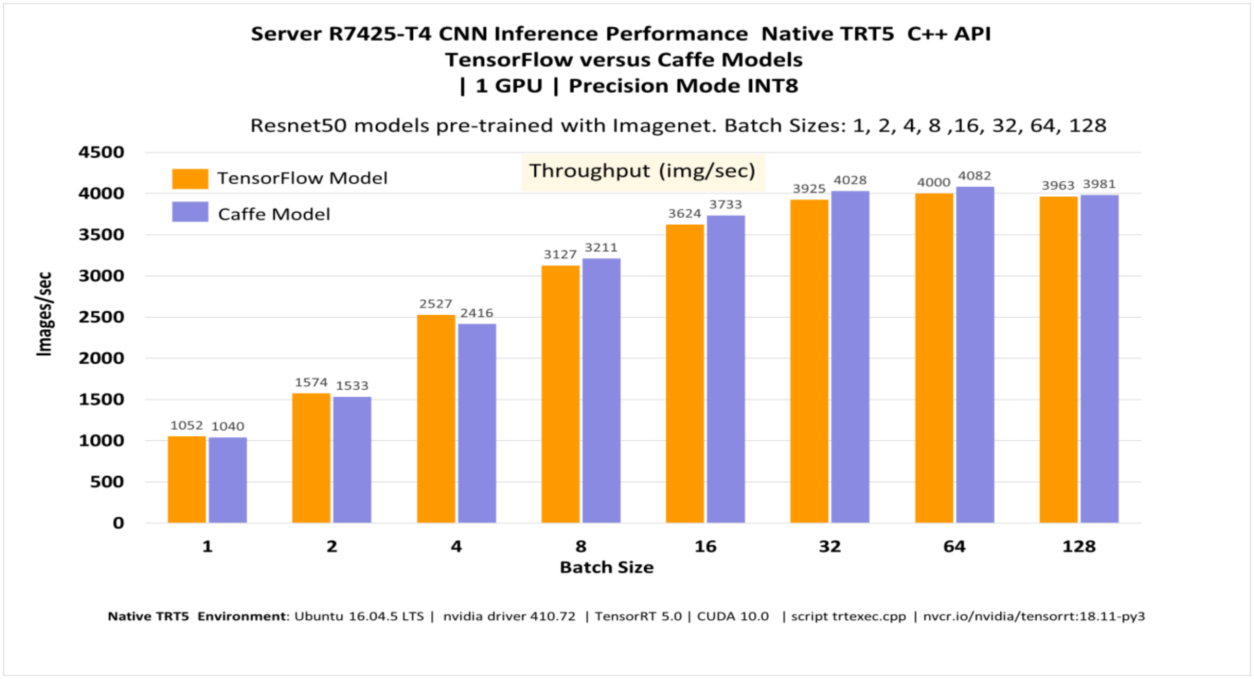


Figure 17. Throughput TensorFlow versus Caffe Optimized Models

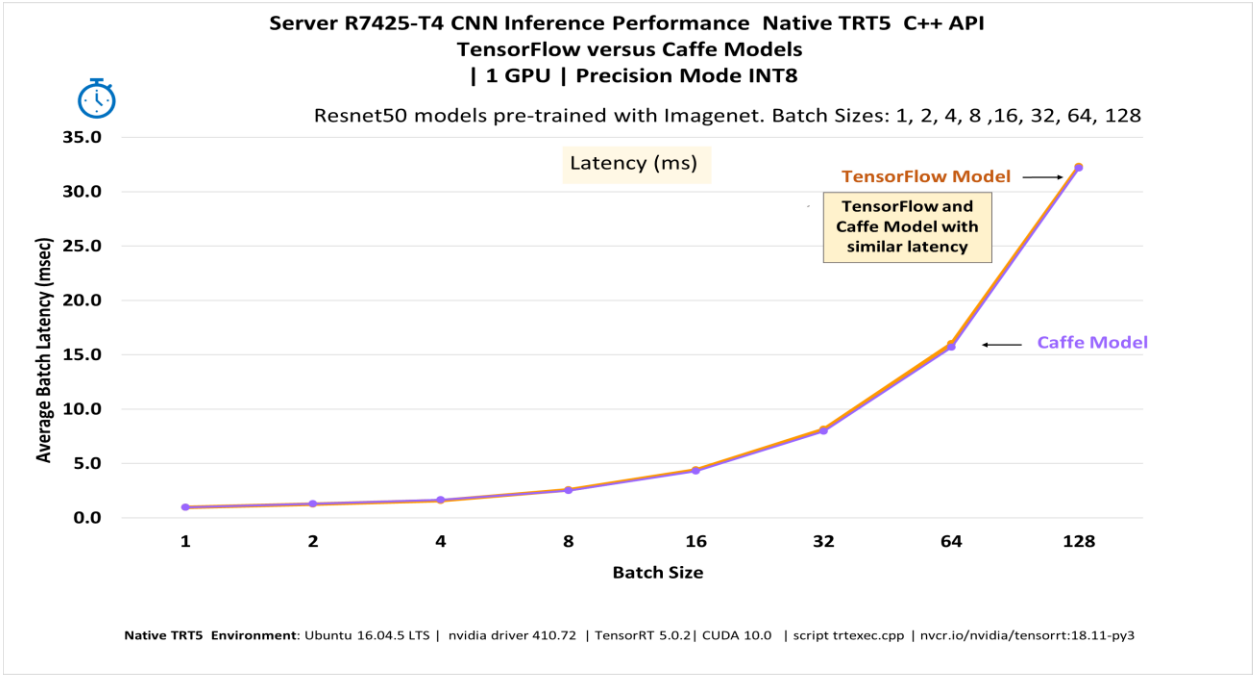


Figure 18. Latency TensorFlow versus Caffe Optimized Models

5.2.3 Resnet-50 - R7425-T4-16GB versus Other Servers

The ResNet50 inference engines with TensorRT C++ API were generated and tested on other servers and compared the results against R7425-T4-16GB. The results show that R7425-T4-16GB is around 2.2x better over servers with NVIDIA P4 GPU. See *Figure **19*** and *Figure **20***.

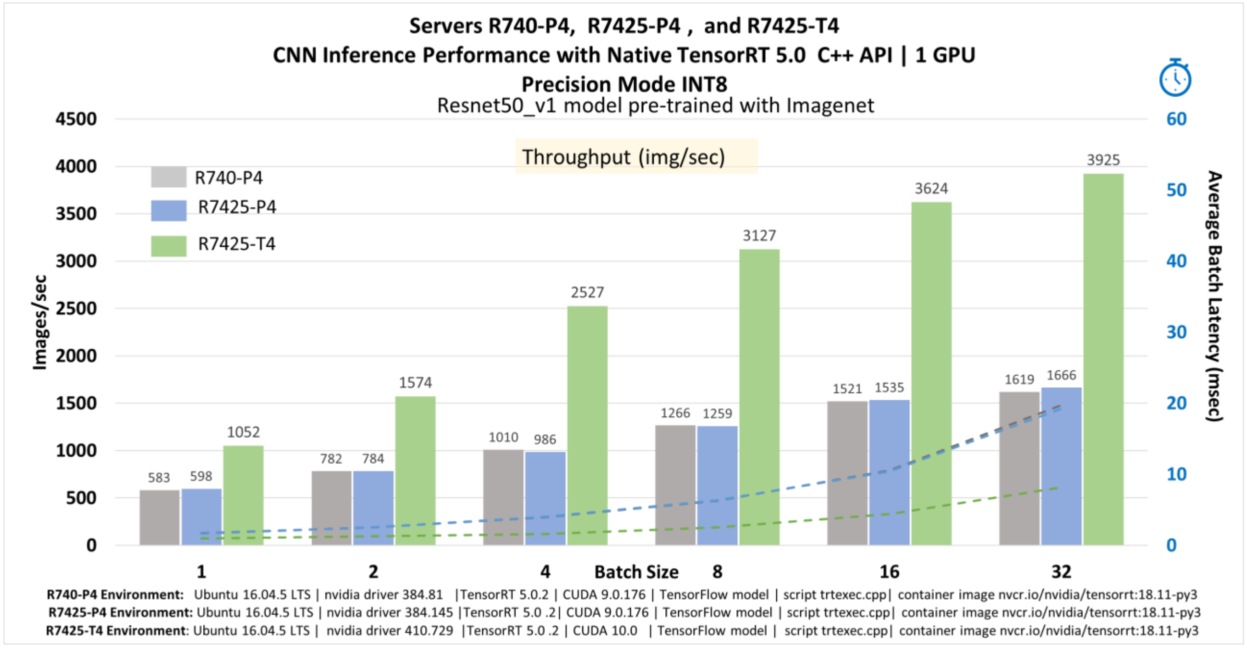


Figure 19. Inference Performance on R7425-T4-16GB vs P4 GPU Servers using TensorRT C++ API

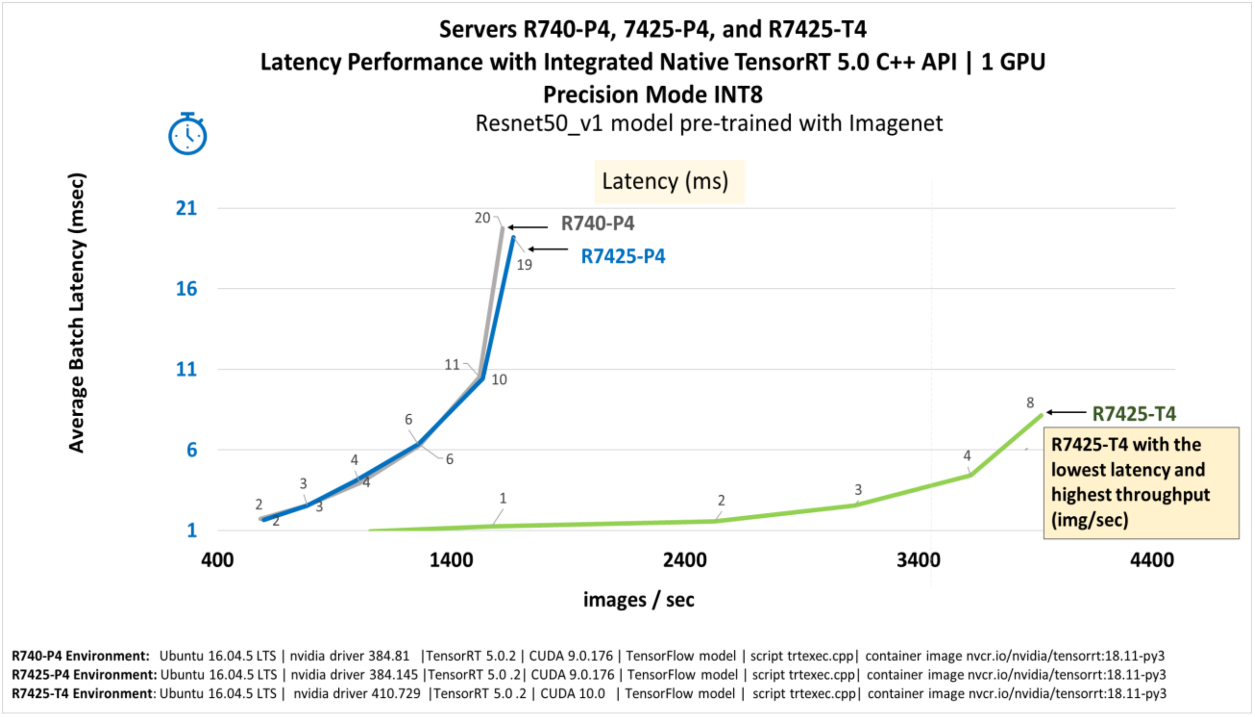


Figure 20. Inference Performance on R7425-T4-16GB vs P4 GPU Servers using TensorRT C++ API

In *Figure **20***, we can see that the server R7425 produces the higher throughput (images/sec) at lower latency for several batch sizes.

5.3 Phase 3: Inference Performance using ResNet50 – TF-TRT versus TRT 5 C++ API

When comparing the results of the optimized inference using TF-TRT integrated vs TensorRT C++ API across the different batch sizes, we observed that the C++ API performance 2.7X faster than the TF-TRT integration at lower latency **. See *Figure **21*** and *Figure **22***

** We are working both with NVIDIA and TensorFlow team to look closely into this performance difference between integrated TensorFlow with TensorRT vs native TensorRT.

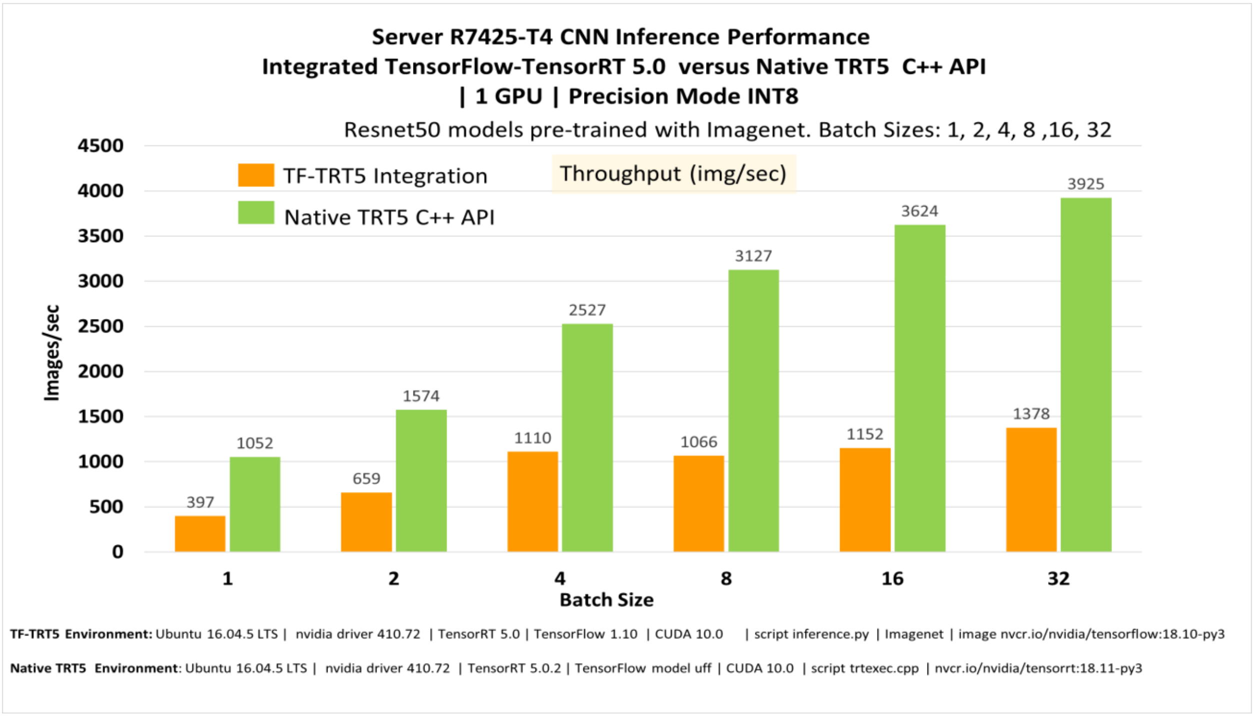


Figure 21. Performance with TensorFlow-TensorRT Integrated vs TensorRT C++ API

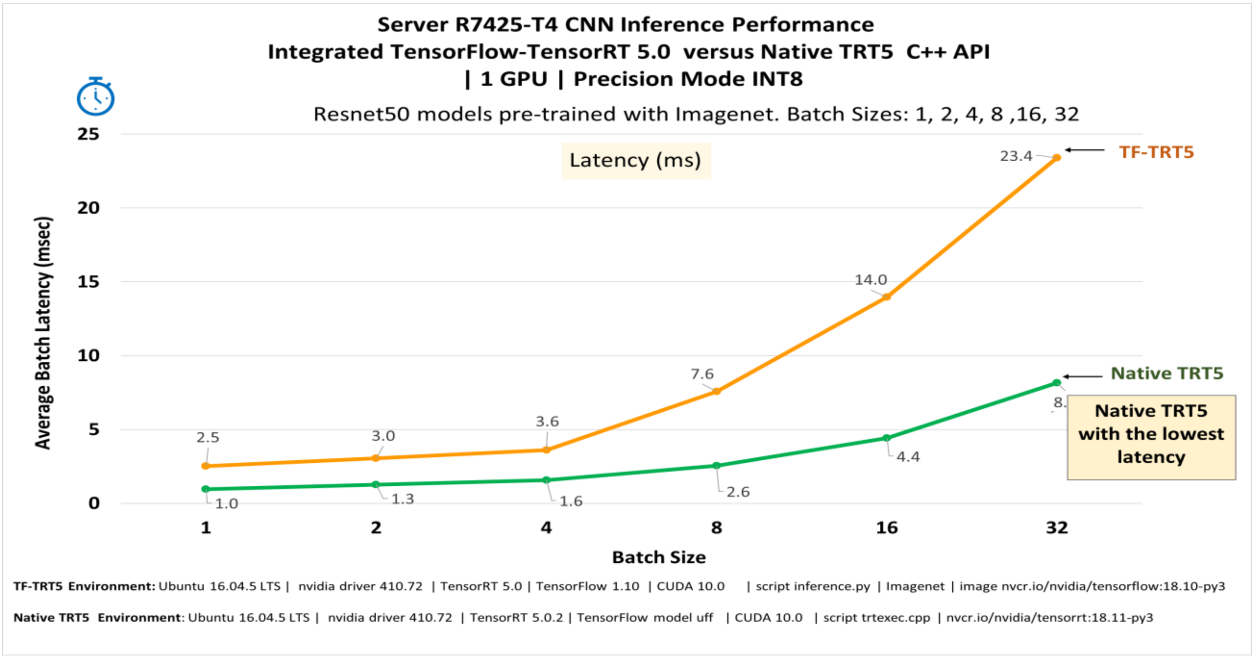


Figure 22. Latency performance with TensorFlow-TensorRT Integrated vs TensorRT C++ API

5.4 Phase 4: Inference Image Classification on Neural Models – Other Aspects

To look at the benefits of running inference with TensorRT™ using lower precision INT8, we compared it with other precision methods. Below are the configurations used to look at the performance:

- Native TensorFlow-FP32 on CPU-only: test conducted using the TensorFlow with GPU disabled, to do so we used the docker image with CPU-only support, the trace log showed the operations were executed at the device: CPU:0
- Native TensorFlow-FP32 on GPU: test conducted using the Nvidia docker image for TensorFlow with GPU enabled.
- TensorFlow-TRT5-FP32: test conducted using the TF-TRT integration and the Nvidia docker image for TensorFlow with GPU enabled.
- TensorFlow-TRT5-INT8: test conducted in precision mode INT8, using the TF-TRT integration, and the Nvidia docker image for TensorFlow with GPU enabled
- TensorRT™ Python API-FP32: test conducted in precision mode INT8, using the TensorRT™ C++ API and the Nvidia docker image for TensorFlow with GPU enabled.

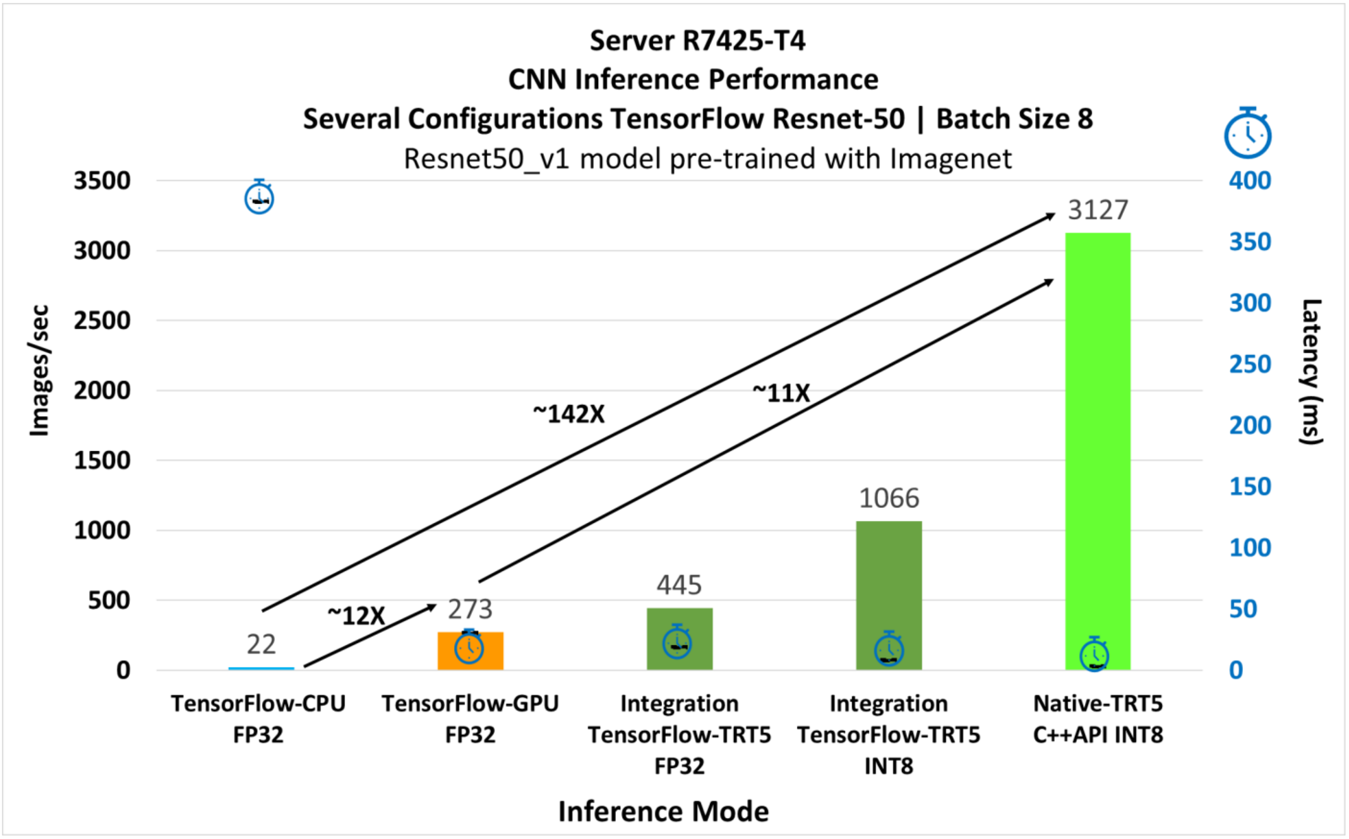


Figure 23. Throughput inference performance - Several Inference Modes

*Figure **23***, we observe that the native TensorFlow-FP32 inference on GPU is ~12X faster than the same inference run on AMD CPU-only *** (273 vs 22). On the other hand, TensorRT C++ API INT8 inference was ~142X faster than the inference run with native TensorFlow-FP32 on CPU only (3127 vs 22), and ~11X faster than TensorFlow-FP32 on GPU only (3127 vs 273).

**** Note this AMD EPYC 7551 32-Core Processor is not using any CPU optimized libraries. Its only used as a baseline comparison.*

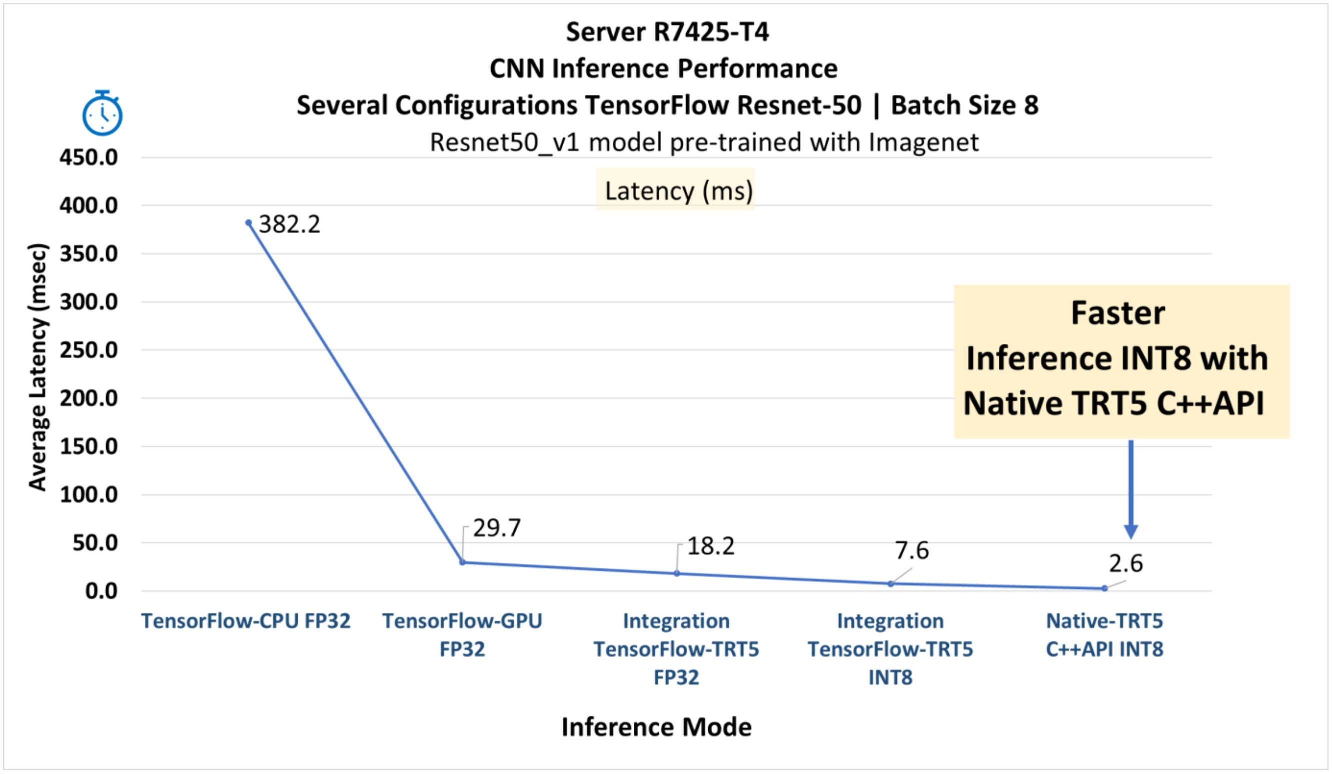


Figure 24. Latency performance comparison

Running the optimized inference of the model in native TensorFlow-FP32 on CPU-only took around 382 milliseconds, in contrast running the model with GPU enabled took around 30 milliseconds. Finally, the lowest latency was obtained running the inference with the optimizer TensorRT C++ API implementation with 2.6 milliseconds. *Figure **24*** shows latency performance using different precision modes.

6 Hardware metrics

In this section, we demonstrate how to extract some GPU metrics while running the optimized inference in INT8 using the nvidia-smi command with query options provided by NVIDIA [8]. Some of the metrics monitored are listed and described below:

- GPU utilization: Percent of time over the past sample period during which one or more kernels was executing on the GPU.
- Memory Utilization: Percent of time over the past sample period during which global (device) memory was being read or written.
- GPU temperature: Core GPU temperature in degrees C.
- Power Draw: Power consumption.

The following sections show the plots of each metric performance extracted with this nvidia-smi query:

```
$nvidia-smi -query
gpu=timestamp,index,pstate,utilization.gpu,utilization.memory,memory.total,memory.free,me
mory.used,power.draw,temperature.gpu,pcie.link.gen.max,pcie.link.gen.current, --format=csv -
l -f <filename>
```

6.1 Percentage of GPU Utilization vs GPU Memory Utilization

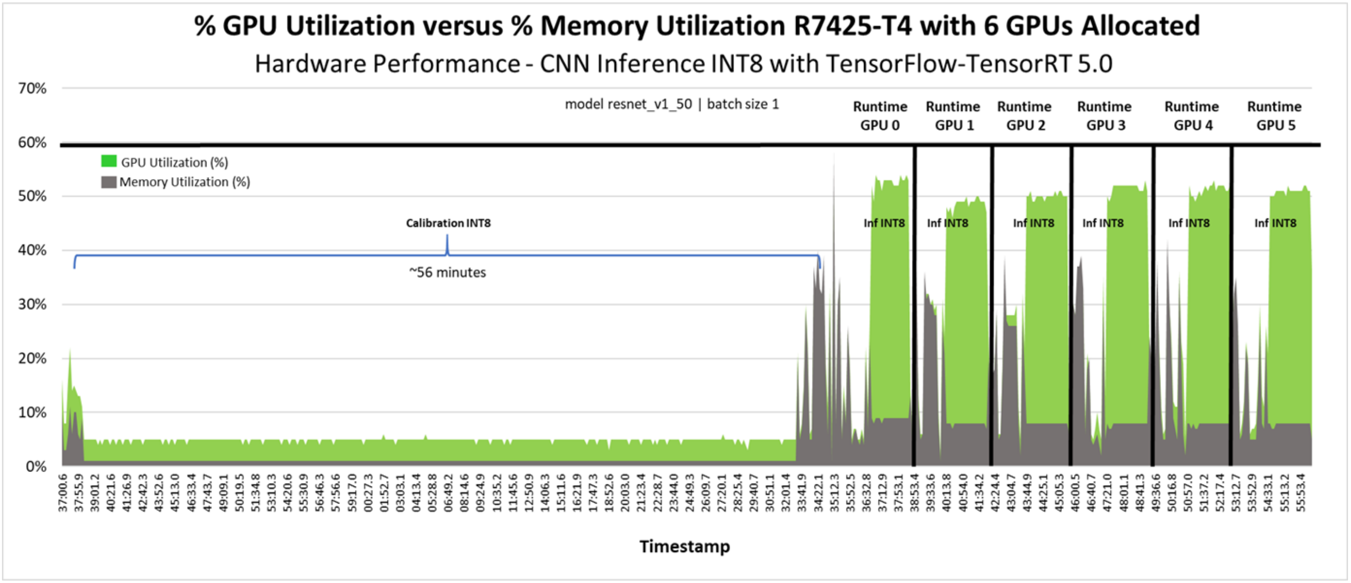


Figure 25. GPU Utilization versus Memory Utilization

The Calibration process consists in calculating the weights of the model at the optimal scaling factor from FP32 to INT8, this is a long process that could take around 1 hour, in the *Figure **25***Error! Reference source not found. we can see the percentage of the GPU (green color) and memory (gray color) consumption during the first hour corresponding to the calibration process; once the inference graph is calibrated and optimized, it is used to generate the inference. In this test, we wanted to show how the calibration process was conducted only one time at the GPU 0, then the optimized inference graph was saved at the cache and reused for inference in the rest of the GPU's.

6.2 Core GPU Temperature

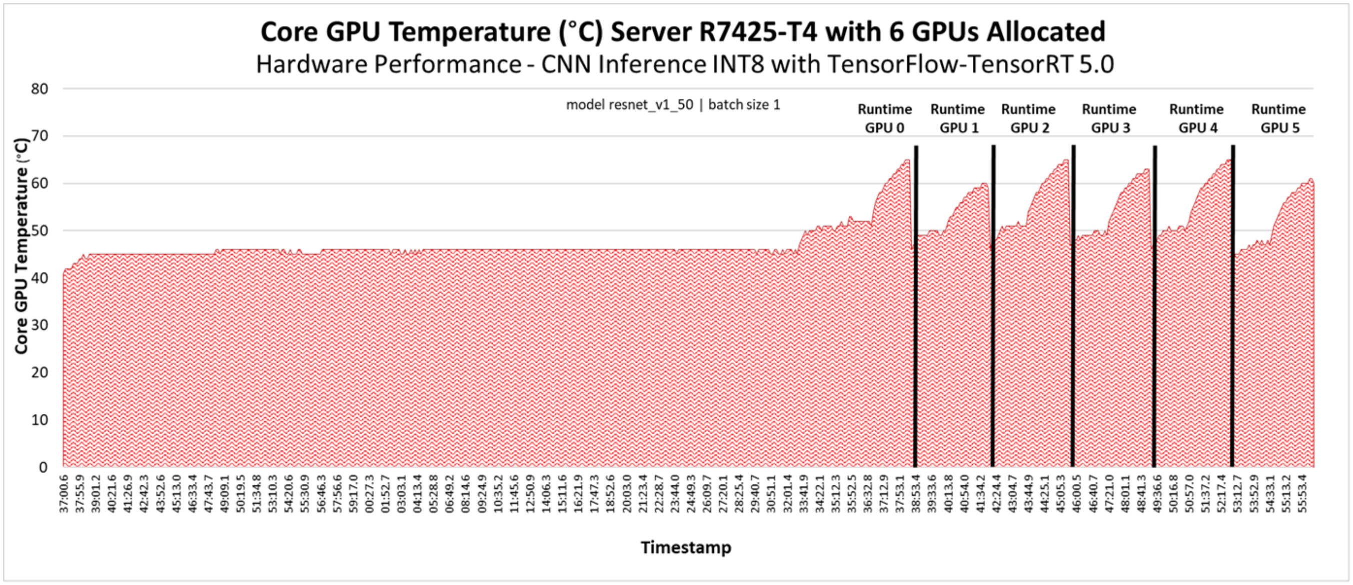


Figure 26. NVIDIA T4 GPU Temperature

In *Figure **26***, we can see the GPU temperature during the calibration process at the GPU 0, and later see how it increases gradually when running the inference, there were sleep time periods before initiating the inference in the next GPU, so we can appreciate how the temperature decreases during that period. The same pattern happens with the power drawn.

6.3 Power Draw

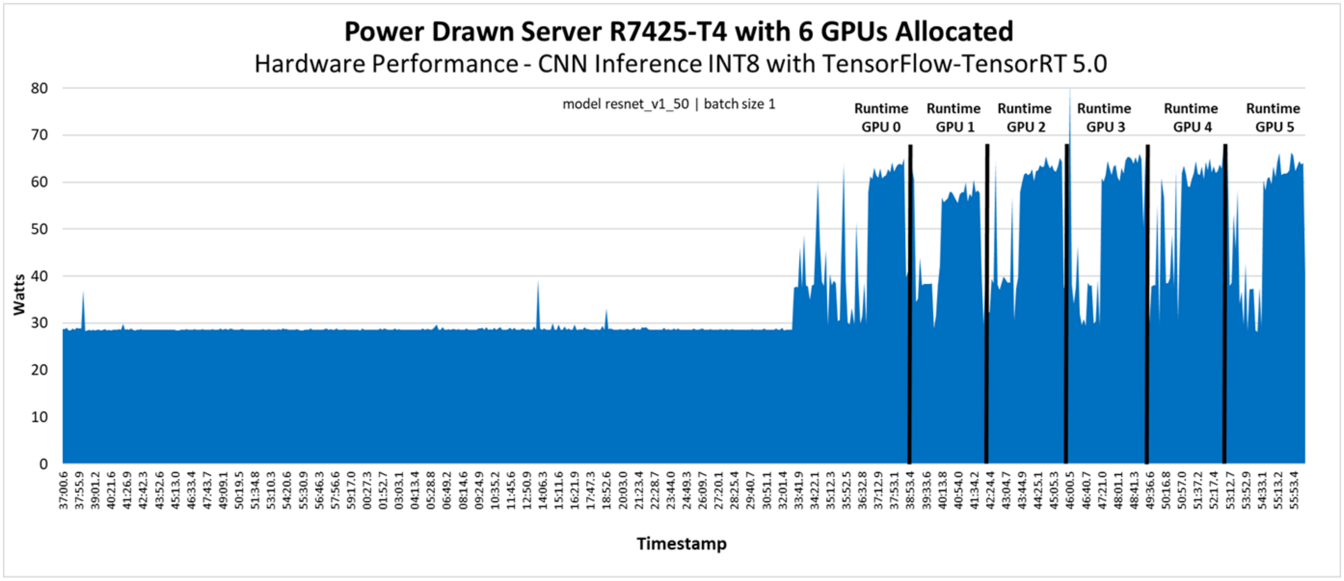


Figure 27. NVIDIA T4 GPU Power Consumption

7 Conclusion and Future Work

1. TensorRT™ is an excellent tool to speed up inference, and the results obtained in this project demonstrated the power of Dell EMC PowerEdge R7425 using T4-16GB GPU to accelerate image classification and deliver high-performance inference throughput and low latency production level environments.

2. Comparing TensorRT™ versus native TensorFlow (GPU enabled), TensorRT™ C++ API in INT8 speeds up ResNet50 inference to ~11X over the TensorFlow-FP32 inference on GPU.

3. When comparing the performance of different TensorRT™ implementations, the optimized inference using TensorRT™ C++ API produces around 2.7X more than the TF-TRT integration at lower latency. *Due to the better performance of native TensorRT™ C++ API implementation, its highly suitable for production environments where throughput & latency need to be considered*.

4. DELL EMC PowerEdge R7425 with NVIDIA T4-16GB GPU performed ~1.8X faster when comparing it to NVIDIA P4-8GB GPU.

5. After the optimized inference model is generated, it can be deployed into the production environment. For deployment of models in production environment we are exploring TensorRT Inference Server (TRTIS), which can run multiple models (and/or multiple instances of the same model) on multiple GPUs.

8 References

- [1] Nvidia Developer, “NVIDIA TensorRT™ Programmable Inference Accelerator” [Online]. Available: https://developer.nvidia.com/TensorRT™
- [2] Speed up TensorFlow Inference on GPUs with TensorRT . [Online]. Available: https://medium.com/tensorflow/speed-up-tensorflow-inference-on-gpus-with-tensorrt-13b49f3db3fa
- [3] TensorRT™ Developer Guide, “Supported Operations By Framework” [Online]. Available: https://docs.nvidia.com/deeplearning/dgx/integrate-tf-trt/index.html#usingtftrt
- [4] Nvidia, “Accelerating Inference In TensorFlow With TensorRT™ User Guide” [Online]. Available: https://docs.nvidia.com/deeplearning/dgx/integrate-tf-trt/index.html
- [5] Nvidia TensorRT™ Developer Guide, “Working with TensorRT™ Using The Python API” [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/TensorRT-developer-guide/index.html#python_topics
- [6] Nvidia TensorRT™ Developer Guide, “Working with TensorRT™ Using the C++ API” [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/TensorRT-developer-guide/index.html#c_topics
- [7] 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/
- [8] Nvidia Support, “Useful nvidia-smi Queries” [Online]. Available: https://nvidia.custhelp.com/app/answers/detail/a_id/3751/~/useful-nvidia-smi-queries

9 Annex A- PowerEdge R7425-T4-16GB Server – 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
	Integer 8 - INT8 (Tera Operations /s)	130
	Integer 4 – INT4-16GB (Tera Operations /s)	260
Power		
	Min Power Limit (W)	60
	Max Power Limit (W)	70