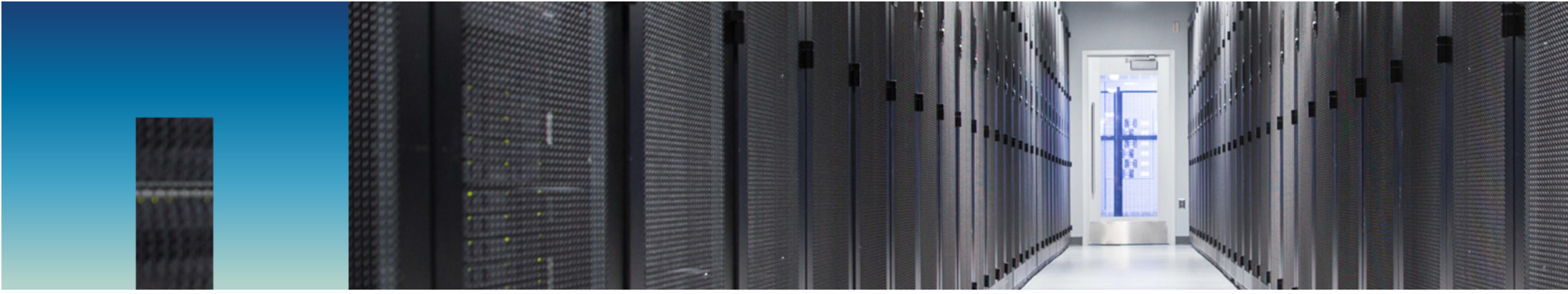


Technical Report

AI Deployment with NetApp E-Series and BeeGFS

Nagalakshmi Raju, Daniel Landes, Nathan Swartz, Amine Bennani, NetApp
July 2019 | TR-4785

Abstract

Artificial intelligence (AI), machine learning (ML), and deep learning (DL) applications involve large datasets and high computations. To run these workloads successfully, you need an agile infrastructure that allows you to scale out both storage and compute nodes seamlessly. This report includes the steps for running an AI training model in a distributed mode, which allows seamless scale-out of compute and storage nodes. The report also includes various performance metrics to show how a solution combining NetApp® E-Series storage with the BeeGFS parallel file system provides a flexible, cost-effective, and simple solution for AI workloads.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

Deep learning (DL) is a subset of artificial intelligence (AI) that is widely used in many popular applications such as image recognition, healthcare, self-driving cars, and voice search. DL produces accurate results by training large volumes of data, which is storage intensive, compute intensive, and time consuming. As datasets continue to grow, it's challenging to scale the DL training from a single node, improve training speed, and reduce time to value.

The distributed training approach eases all the DL training challenges by distributing the training on various machines with high-compute devices like graphic processing units (GPUs) for parallel computations.

As the datasets increase, it can be harder to add a storage node without disturbing the training. NetApp® E-Series storage with the BeeGFS file system solves this problem in DL workloads, because it enables customers to scale out storage nodes seamlessly.

DL applications' success depends on the training speed. To improve the training speed, GPU utilization must be close to 100% throughout the training; GPUs should not experience idle cycles while waiting for the training data. To achieve a low-latency and high-throughput infrastructure, a high-performing file system is needed. From our experiments, we know that BeeGFS with an NetApp EF570 array for metadata and a NetApp E5700 array for data facilitates low latency to maintain the GPU's utilization close to 100% throughout the training.

In our experiments at NetApp, we focused on the training phase of the DL workflow, because this phase involves complex computations on large volumes of datasets. Our tests demonstrated that if E-Series with BeeGFS can run the training phase seamlessly, we can infer that our storage can easily support other DL workflow phases (preprocessing and inference). We chose image recognition as the DL application with an ImageNet dataset, because it is widely used and recognized. Because Google's TensorFlow framework for DL provides the performance benchmarks on various platforms for image recognition, we used TensorFlow as a reference for comparison. The training framework we used was Horovod, an open-source framework that makes distributed training fast and easy. Our training model was data parallelism, which makes it easy to add more storage nodes as data grows.

2 Deep Learning and Convolutional Neural Networks Training Models

This section discusses the training models used for DL and convolutional neural networks (CNN) training.

2.1 Deep Learning

DL is a subset of machine learning (ML) that uses deep artificial neural networks as models and does not require feature engineering. In DL, interconnected layers of software-based calculators, known as "neurons," form a neural network. The network can ingest vast amounts of input data and process it through multiple layers that learn increasingly complex features of the data at each layer. The network can then decide about the data, learn if its determination is correct, and use what it has learned to make determinations about new data. For example, after the network learns what an object looks like, it can recognize the object in a new image.

Here are a few DL training models:

- CNN for image recognition and processing
- Sequence-to-sequence (Seq2Seq) models for human language–related features
- Large-scale linear models for data analysis and simple behavioral predictions

2.2 CNNs

CNN training models are used for image recognition and processing. In general, in a deep CNN, several layers are stacked and are trained on a task. The network learns several low-, middle-, and high-level features at the end of its layers. CNN training models with more layers need more computations, making the model more accurate. The time to train and the throughput (number of images processed per second) depend on the number of layers in the model.

CNNs are often compute intensive, which makes them similar to traditional supercomputing (high-performance computing, or HPC) applications. Thus, large learning workloads perform well on accelerated systems such as general-purpose GPUs.

2.3 CNN Training Models

ResNet

ResNet, short for “residual network,” introduces the concept of residual learning.

ResNet is trained on more than a million images from the ImageNet database. ResNet can have a very deep network of up to 152 layers. (For example, ResNet50 is a 50-layer residual network; ResNet101 and ResNet152 are other variants.) With increased depth, the model can increase accuracy and reduce training errors. The time needed to train the ResNet model is longer than for the other models, such as AlexNet and VGG-16.

AlexNet

AlexNet is a CNN that is trained on more than a million images from the ImageNet database. The network is eight layers deep and can classify images into 1,000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images.

VGG-16

The VGG-16 CNN has capabilities similar to those of AlexNet, but it is 16 layers deep and has an image input size of 224 x 224.

3 Distributed Training

With DL, datasets increase in size and models increase in complexity. Accelerating the training is challenging and demands greater computational resources. The idea behind distributed training is to run the tasks in parallel on multiple devices such as GPUs, instead of running them serially as you would in a single machine.

There are two types of parallelism: data parallelism and model parallelism.

The term “performance” in these systems can refer either to the computational speed of the process or the accuracy of the model.

Computation speed depends on the platform on which the model is deployed; the metrics are speed, efficiency, and scalability. Accuracy of the model is independent of the platform; it depends on the models used for the training.

3.1 Data Parallelism

Data parallelism is used for speeding up the computation of CNNs with large datasets. In data parallelism, the training data is divided into multiple subsets, each of which runs on the same replicated model in a different node. Each device independently computes the errors between its predictions for its training samples and the labeled outputs. At the end of the batch computation, the nodes must synchronize the model parameters (or its “gradients”) to ensure that they are training a consistent model (as if the algorithm runs on a single processor).

The training model scales with the amount of data available and speeds up the rate at which the entire dataset contributes to the optimization. The training model requires less communication between nodes, benefiting from the high number of computations per weight.

3.2 Model Parallelism

Model parallelism training is segmented into different parts that can run concurrently on the same data in different nodes. This model is more complex to implement than data parallelism.

Because model parallelism needs to synchronize only the shared parameters, communication between the GPUs is reduced.

Model parallelism is used with larger models, because hardware constraints per node are not a limitation.

4 Distributed Training with BeeGFS

To be successful, a distributed system must meet the following requirements:

- An agile infrastructure that enables seamless scale-out of compute and storage nodes
- A high-performing file system to feed the large volumes of datasets to various GPUs to support parallel computations
- A file system with low latency that keeps the GPUs at a high level of utilization and speeds up the training period

NetApp E-Series storage with BeeGFS meets all the requirements and fits into the distributed architecture.

BeeGFS is a simple file system that requires no kernel changes and runs in user space. You can set up your BeeGFS system in minutes.

You can expand your BeeGFS environment with ease, requiring only the IP of your management node and a service to be spun up (metadata, storage, or client service).

With the data parallelism model of distributed training, you can easily add storage nodes as the datasets grow. The BeeGFS file system presents the data to the clients as a single mount point.

With a high price-to-performance ratio, NetApp E-Series storage combined with the open-source BeeGFS parallel file system is a cost-effective, simple, and scalable solution suited for distributed training for DL.

For more information about BeeGFS with E-Series storage, see TR-4755: BeeGFS with NetApp E-Series Solution Deployment and TR-4782: BeeGFS with NetApp E-Series Reference Architecture.

5 TensorFlow

TensorFlow is an open-source library developed by Google for numerical computation and large-scale ML. TensorFlow bundles a large amount of ML and DL models and algorithms.

TensorFlow supports all the CNN training models and provides benchmarks of various models on different platforms (NVIDIA GPUs).

The flexible architecture allows you to use a single API to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device.

The TensorFlow libraries make it far easier to incorporate self-learning elements and AI features such as speech recognition, computer vision, or natural language processing into applications.

Some of the companies currently using TensorFlow are Google, Airbnb, eBay, Intel, Dropbox, DeepMind, Airbus, CEVA, Snapchat, SAP, Uber, Twitter, and IBM.

5.1 TensorFlow Dependencies

Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA for general computing on its own GPUs.

CUDA enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation.

The CUDA Toolkit includes libraries, debugging and optimization tools, a compiler, documentation, and a run-time library to deploy your applications.

The toolkit has components that support DL, linear algebra, signal processing, and parallel algorithms.

In general, CUDA libraries support all families of NVIDIA GPUs but perform optimally on the latest generation, such as the V100, which can be three times faster than the P100 for DL training workloads.

Using one or more libraries is the easiest way to take advantage of GPUs, as long as the algorithms you need have been implemented in the appropriate library.

CUDA Deep Neural Network Library

The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. The cuDNN library provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.

DL researchers and framework developers worldwide rely on cuDNN for high-performance GPU acceleration.

The cuDNN library accelerates widely used DL frameworks, including Caffe, Caffe2, Chainer, Keras, MATLAB, MXNet, TensorFlow, and PyTorch.

6 Horovod

Horovod is an open-source distributed training framework for TensorFlow, and its goal is to make distributed DL fast and easy.

Horovod requires the prior installation of Open Message Passing Interface (Open MPI) and NVIDIA's NCCL-2 (NVIDIA Collective Communications Library) to support inter-GPU communication. For averaging gradients and communicating those gradients to all nodes, Horovod uses the ring-allreduce decentralized scheme. Horovod uses NCCL-2, which provides a highly optimized version of ring-allreduce across multiple machines. The allreduce algorithm allows the worker nodes to average gradients and disperse them to all nodes without the need for a centralized scheme with a parameter server.

6.1 Data Parallelism in Horovod

To run distributed training with Horovod, each compute node in the distributed architecture must have a copy of the training model script. Horovod runs the training script on various compute nodes by running the Horovod command from a single compute node.

Horovod runs the training in many iterations, with each iteration involving training of a batch of data within a dataset.

The following steps describe how data parallelism distributed training is run (for each batch):

1. Horovod runs the training script on each compute node in parallel. At each node, the training script completes the following actions:
   – Reads a chunk of the dataset.
   – Runs the dataset through the training model.
   – Computes the model gradients.
2. Horovod computes the average model gradients among all the training models on various nodes.
3. Horovod updates the training model with the average gradients to ensure that the model is consistent across all the compute nodes.
4. Horovod repeats steps 1 through 3 with the next batch of data.

7 Configuration Details

Figure 1 shows a BeeGFS with E-Series storage building block. Each client consists of GPUs.

For more details on this building block architecture, refer to TR-4755: BeeGFS with NetApp E-Series Solution Deployment.

Figure 1) BeeGFS with E-Series building block.

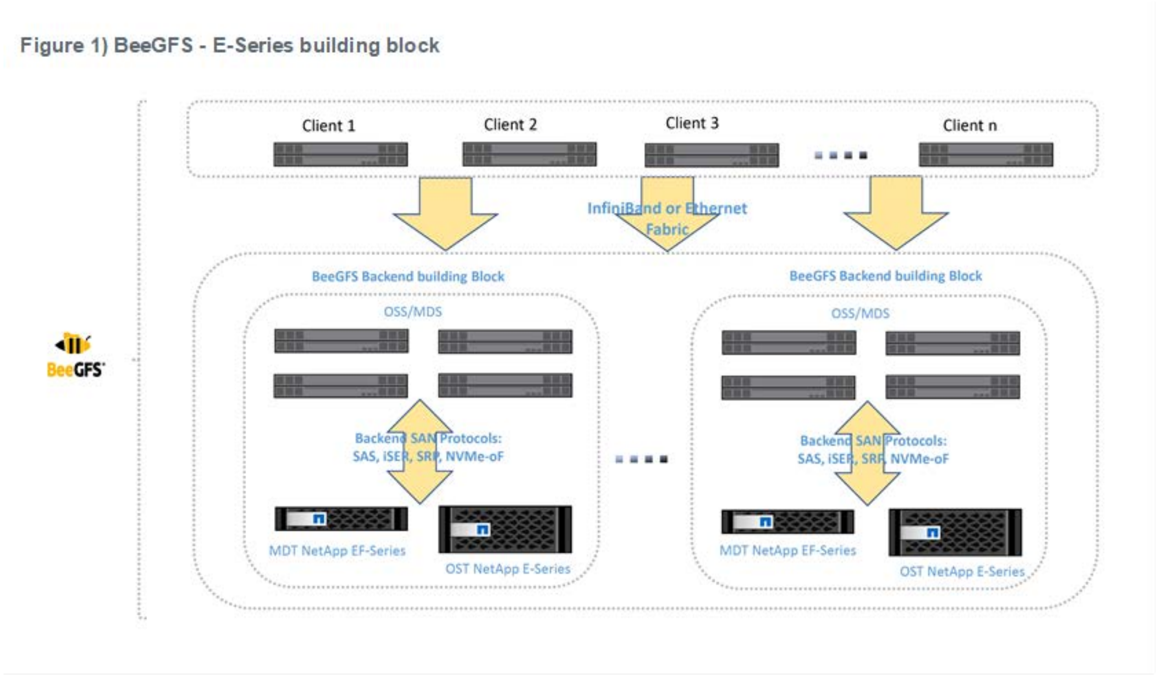


Figure 1) BeeGFS - E-Series building block
Client 1
Client 2
Client 3
Client n
InfiniBand or Ethernet Fabric
BeeGFS
BeeGFS Backend building Block
OSS/MDS
Backend SAN Protocols: SAS, iSER, SRP, NVMe-oF
MDT NetApp EF-Series
OST NetApp E-Series
BeeGFS Backend building Block
OSS/MDS
Backend SAN Protocols: SAS, iSER, SRP, NVMe-oF
MDT NetApp EF-Series
OST NetApp E-Series

Table 1 shows the configuration details.

Table 1) Configuration details.

Component	Details
Clients x 2	CPU: 2.6GHz, 24 cores 1 GPU per client: NVIDIA Titan X Pascal RAM: 128GB Ubuntu 18.04 LTS Python 3.6.7

BeeGFS storage server/OSS x 2	CPU: 3.3GHz, 16 cores, RAM: 128GB Red Hat Enterprise Linux 7.5
BeeGFS metadata/management server x 2	CPU: 3.3GHz, 16 cores, RAM: 128GB Red Hat Enterprise Linux 7.5
Storage array	NetApp E5760 Hybrid with 20 SSDs and 100 HDDs NetApp SANtricity® OS 11.50
Connections	32Gbps FC and 100Gbps InfiniBand (IB)
Network switch	32Gbps FC switch and 100Gbps IB switch

7.1 Software Stack

Table 2 shows a list of the necessary software components on the clients and their versions for this configuration.

Table 2) Software stack.

Software	Version
CUDA	9.0
cuDNN	7.1.4
Horovod	0.16.1
NCCL	2.3.7
Open MPI	4.0.0
Tensorflow GPU	1.12.0
TensorFlow benchmark scripts	1.12

7.2 BeeGFS Setup

To set up the BeeGFS parallel file system, see, BeeGFS with NetApp E-Series.

7.3 Volume Configuration and Mounting the Dataset

For different BeeGFS services, we provisioned the storage array as shown in Table 3.

Table 3) Volume configuration details.

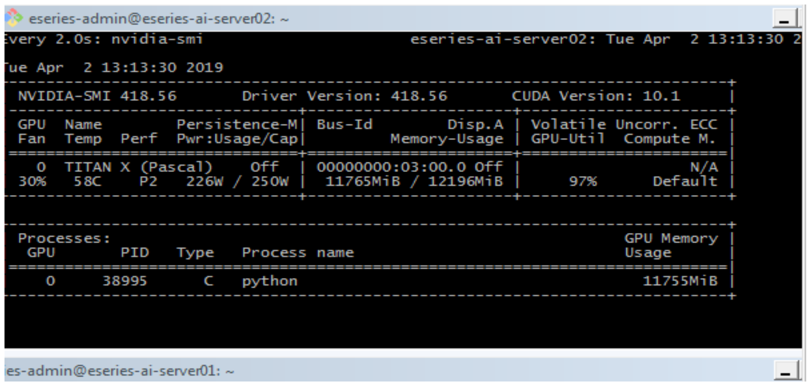
# Volume Groups	# Volumes	RAID Type	Media Type	Data Type
1	11	2+2 RAID 1	SSDs	Metadata
1	4	16 DDP	SSDs	Data
10	10	8+2 RAID 6	HDDs	Data

For more information about volume configuration, see TR-4782: BeeGFS with NetApp E-Series Reference Architecture.

7.4 TensorFlow Setup

Verify that the NVIDIA drivers are installed by running the command `nvidia-smi` on your system. If the drivers are installed, you should get output similar to the output shown in .

Figure 2) Example of nvidia-smi output.



eseries-admin@eseries-ai-server02: ~

```
very 2.0s: nvidia-smi                          eseries-ai-server02: Tue Apr  2 13:13:30 2

ue Apr  2 13:13:30 2019
-----------------------------------------------------------------------------+
 NVIDIA-SMI 418.56       Driver Version: 418.56       CUDA Version: 10.1     |
-------------------------------+----------------------+----------------------+
 GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
 Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
===============================+======================+======================|
   0  TITAN X (Pascal)    Off  | 00000000:03:00.0 Off |                  N/A |
 30%   58C    P2   226W / 250W |  11765MiB / 12196MiB |     97%      Default |
-------------------------------+----------------------+----------------------+

-----------------------------------------------------------------------------+
 Processes:                                                       GPU Memory |
  GPU       PID   Type   Process name                             Usage      |
=============================================================================|
    0     38995      C   python                                     11755MiB |
-----------------------------------------------------------------------------+
```

es-admin@eseries-ai-server01: ~

Follow the links in Table 4 to ensure that the dependencies are met for TensorFlow. Install them in the order shown in the table. (For the complete installation guide, see https://medium.com/@taylordenouden/installing-tensorflow-gpu-on-ubuntu-18-04-89a142325138.)

Table 4) Links for various pieces of software needed by TensorFlow.

TensorFlow Installations	Links
NVIDIA drivers	https://www.nvidia.com/Download/index.aspx?lang=en-us
CUDA Toolkit	https://developer.nvidia.com/cuda-downloads
cuDNN	https://developer.nvidia.com/cudnn
TensorFlow GPU	https://www.tensorflow.org/install/gpu

7.5 Horovod Setup

Table 5 shows links to the software required for the Horovod installation. Install the packages in the order shown in the table.

Table 5) Required Horovod software.

Horovod Stack	Links
Open MPI	https://www.open-mpi.org/faq/?category=building#easy-build
NCCL	https://developer.nvidia.com/nccl/nccl-legacy-downloads
Horovod	Install Horovod pip package Installation steps: https://github.com/horovod/horovod

8 Test Methodology

The main objective of this project was to demonstrate that NetApp E-Series storage with BeeGFS can support AI workloads. To prove that this architecture can meet all the AI workload demands, such as throughput, IOPS, and capacity, we experimented with the TensorFlow benchmark scripts. TensorFlow benchmark scripts contain several high-performing convolution models that can be run on a single machine or in distributed mode. Also, TensorFlow provides the benchmark results of various models tested on different platforms that can be used as a point of reference for the test results.

We conducted various tests with different training models (ResNet50, ResNet152, AlexNet, VGG-16) on various computing platforms (NVIDIA's Titan X Pascal and Tesla M10). We used images per second as the performance measurement, because the success of an AI application depends on the total duration for training the model (for instance, how many images can be processed in a second). Because AI workloads contain large volumes of data, training the model with these huge datasets should be fast. The speed of the training depends on the platform (GPU model) on which the training is run and the training model used (ResNet50, AlexNet). For instance, training on high-compute-powered GPUs is much faster than on low-end GPUs. Also, more complex training models (training models with many layers in the convolution network) take longer to train than simpler training models.

This report shows the results for the AlexNet training model because it requires high throughput and minimum latency. Because we were evaluating our storage with BeeGFS for AI workloads, we wanted to stress our storage nodes with the high throughput that AlexNet needs.

To prove that the storage and compute nodes can be scaled out easily, we ran the training in a distributed mode using Horovod.

8.1 Setting Up TensorFlow Benchmark Scripts

To set up the TensorFlow benchmark scripts, we followed these steps:

1. Clone the benchmark scripts from https://github.com/tensorflow/benchmarks/tree/cnn_tf_v1.12_compatible. (The TensorFlow installed version should match the benchmark scripts version, so we cloned benchmark scripts version 1.12.)
2. Download the ImageNet dataset from http://image-net.org/download-images and preprocess using the TFRecord preprocessing script from https://github.com/tensorflow/models/blob/master/research/inception/inception/data/download_and_preprocess_imagenet.sh. Mount the preprocessed ImageNet TFRecords on the volume.
3. Navigate to the benchmark scripts directory and use `cd` to change to `scripts/tf_cnn_benchmarks`.
4. Run the benchmark scripts according to these instructions: https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks

8.2 Setting Up Distributed Training with Horovod

To run the benchmark scripts in a distributed mode, we installed all compute nodes with the software stack and verified that all compute node system requirements matched.

To run the benchmark scripts in a distributed mode, we followed the instructions at https://github.com/horovod/horovod.

9 Test Results

This section describes the test results for distributed training that used Horovod with the AlexNet training model. We ran distributed training across two GPUs on two compute nodes (one GPU per node). In this scenario, users can then run several copies of the training script across multiple servers by using the `mpirun` command. The `mpirun` command distributes `tf_cnn_benchmarks.py` to two nodes, and runs it on one GPU per node.

In this test, we executed the `mpirun` command. This command must be executed from the directory where the TensorFlow benchmark scripts reside.

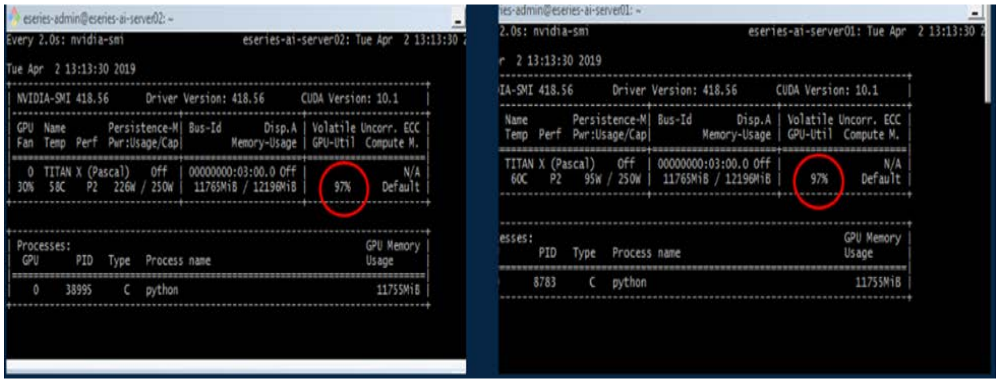
```
mpirun -np 2 \
           -H localhost:1,host_ip_2:1 \
           -bind-to none \
           -map-by slot \
           -x NCCL_DEBUG=INFO \
           -x LD_LIBRARY_PATH \
           -x PATH \
           -mca pml ob1 \
          -mca btl ^openib \
python tf_cnn_benchmarks.py --model alexnet  \
                                                --batch_size 768  \
                                                 --variable_update horovod  \
                                                --data_dir=path_to_Imagenet_tfrecords \
                                                --data_name=imagenet \
                                                --nodistortions \
                                                --num_batches=200 \
                                                --horovod_device=gpu \
                                                --use_fp16=True \
                                               --data_format=NCHW
```

9.1 GPU Performance

GPU utilization was almost 100% throughout the training. From this utilization, we can infer that BeeGFS feeds the training data to the GPUs with minimum latency, which keeps the GPUs busy without any idle cycles while they wait for the training data.

Figure 3 shows an example of the GPU utilization displayed on two servers.

Figure 3) GPU utilization shown on two servers.



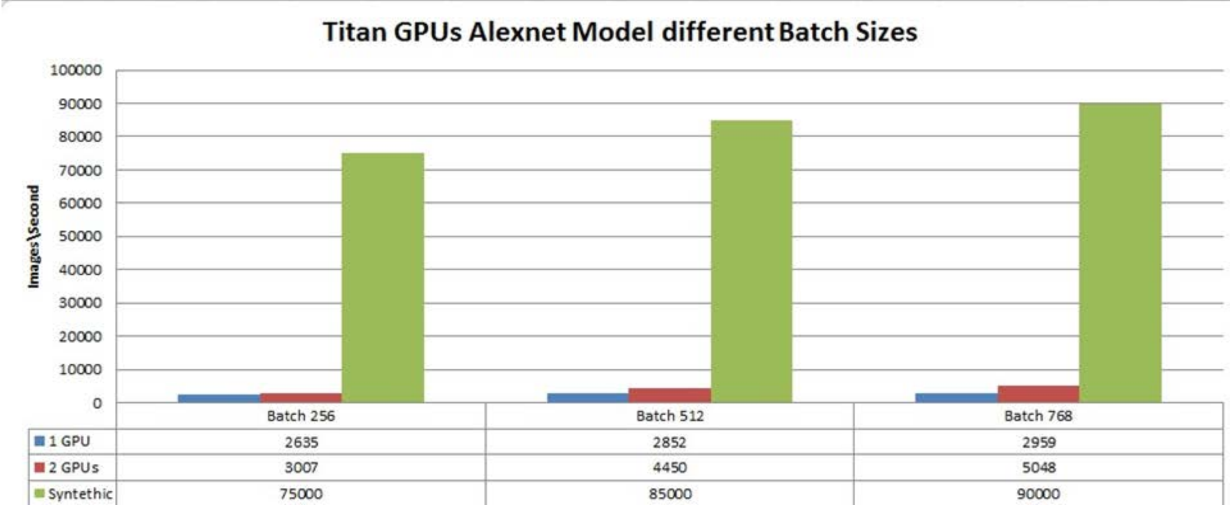
```
eseries-admin@eseries-ai-server02: ~
Every 2.0s: nvidia-smi                          eseries-ai-server02: Tue Apr  2 13:13:30

Tue Apr  2 13:13:30 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.56       Driver Version: 418.56       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  TITAN X (Pascal)    Off  | 00000000:03:00.0 Off |                  N/A |
| 30%   58C    P2   226W / 250W |  11765MiB / 12196MiB |     97%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     38995      C   python                                     11755MiB |
+-----------------------------------------------------------------------------+
```

```
ies-admin@eseries-ai-server01: ~
2.0s: nvidia-smi                          eseries-ai-server01: Tue Apr  2 13:13:30

r  2 13:13:30 2019
------------------------------------------------------------------------+
IA-SMI 418.56       Driver Version: 418.56       CUDA Version: 10.1     |
--------------------------+----------------------+----------------------+
 Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
 Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
==========================+======================+======================|
 TITAN X (Pascal)    Off  | 00000000:03:00.0 Off |                  N/A |
  60C    P2    95W / 250W |  11765MiB / 12196MiB |     97%      Default |
--------------------------+----------------------+----------------------+

------------------------------------------------------------------------+
esses:                                                       GPU Memory |
      PID   Type   Process name                             Usage      |
========================================================================|
     8783      C   python                                     11755MiB |
------------------------------------------------------------------------+
```

9.2 Performance—Images per Second

We used AlexNet as the model of choice for stressing our storage system. Figure 4 shows an example of the performance (in images per second) at varying batch sizes for 1, 2, and a theoretical maximum number of GPUs.

Figure 4) Images per second with different batch sizes.



	Batch 256	Batch 512	Batch 768
🟦 1 GPU	2635	2852	2959
🟥 2 GPUs	3007	4450	5048
🟩 Syntethic	75000	85000	90000

From our experiments, we can estimate that our configuration can support up to 30 Titan X GPUs for AlexNet before saturation and up to 130 Titan X GPUs for the ResNet50 model before saturation.

shows that as we increased the batch size, we saw an overall improvement in throughput. We observed similar behavior when we increased the number of GPUs from one to two. At a batch size of 768, we achieved 2959 images per second for one GPU and 5048 images per second for two GPUs. This improvement did not increase linearly with the number of GPUs as it would with distributed training. There is an overhead where each GPU needs to update the gradients and share with other GPUs in the training to update the training model after each batch. We profiled the workload at different batch sizes, and then we used a synthetic I/O generator, Vdbench, to measure the maximum throughput that we could achieve with the profiled workload. The green bar displays the estimated synthetic throughput images per second from Vdbench.

9.3 Workload Summary

Table 6 shows the differences in performance between SSDs and HDDs.

The workload was mostly 1MB I/O 100% read operations and 30% random.

Table 6) SSD and HDD performance.

	HDD	SSD
Average response time (milliseconds)	4.2	1.68
I/O size	1MB 95% of the time	1MB 95% of the time
Throughput (MBps)	520	520
Completion time (minutes)	1.53	1.45
Queue depth (QD)	6 QD 55% of the time 45% idle	4 QD 60% of the time 40% idle
% randomness	30%	30%

9.4 GPU Comparison (Titan X Pascal Versus Tesla M10)

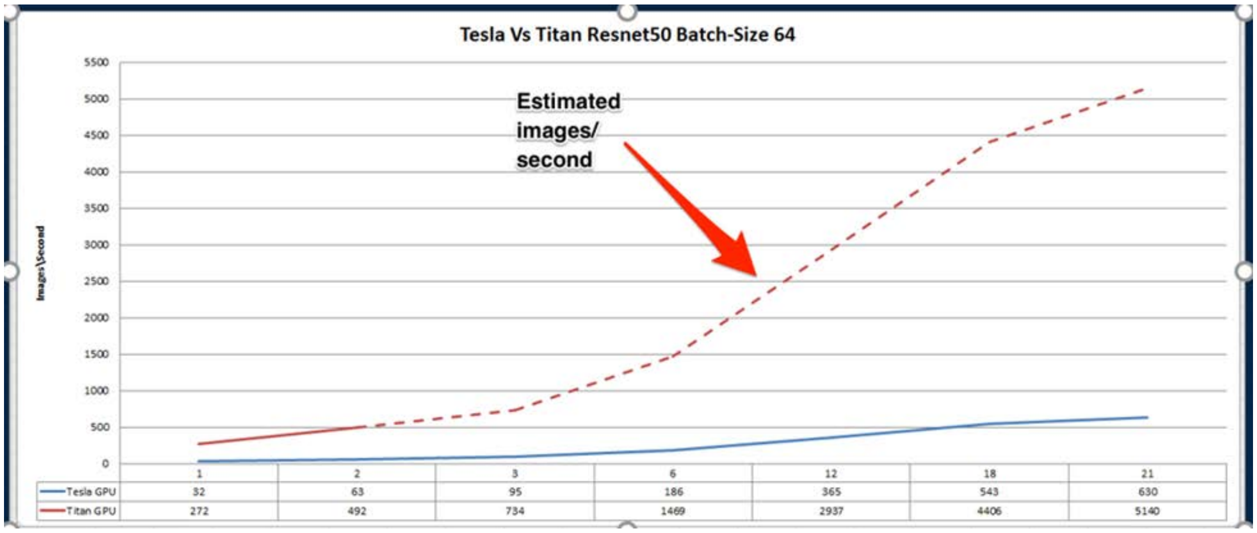
The purpose of this experiment was to show that the performance of an AI application depends on the compute power.

With high-compute-powered GPUs, training is faster, but it demands high throughput and low latency from the storage and file system. With lower-end GPUs, training is slow and storage is less of a bottleneck for the performance of the AI application.

We used Titan X Pascal GPUs, which have higher compute power compared with Tesla M10 GPUs. We used the ResNet50 model to saturate the compute nodes, because ResNet50 is a more complex training model than AlexNet.

In, Figure 5 the x-axis represents the number of GPUs. The number of GPUs did not scale linearly (1, 2, 3, 6, 12, 18, 21).

Figure 5) Tesla versus Titan on ResNet50.



	1	2	3	6	12	18	21
Tesla GPU	32	63	95	186	365	543	630
Titan GPU	272	492	734	1469	2937	4406	5140

10 Conclusion

From the experiments, we can conclude that NetApp E-Series storage with BeeGFS provides a flexible, cost-effective, and simple solution for AI workloads.

NetApp EF570 and E5700 systems are proven to be among the better-value high-performing arrays on the market. BeeGFS is open-source and free to get started. The combination of these products is a cost-efficient solution.

By using BeeGFS with E-Series systems and implementing a building-block architecture, you can scale out to meet your performance needs, making sizing much simpler.

Distributed training requires a very agile infrastructure. The BeeGFS file system with E-Series storage is easy to set up, which makes the combination an attractive solution for AI workloads with distributed training. Expanding your BeeGFS environment is easy; it requires only the IP of your management node and a service to be spun up (metadata, storage, or client service). In addition, E-Series on-box NetApp SANtricity management simplifies deployment.

Where to Find Additional Information

To learn more about the information that is described in this document, review the following documents and websites:

- TR-4782: BeeGFS with NetApp E-Series Reference Architecture
  www.netapp.com/us/media/tr-4782.pdf
- TR-4755: BeeGFS with NetApp E-Series Solution Deployment
  www.netapp.com/us/media/tr-4755.pdf
- Installing TensorFlow GPU on Ubuntu 18.04 LTS
  https://medium.com/@taylordenouden/installing-tensorflow-gpu-on-ubuntu-18-04-89a142325138
- NetApp product documentation
  https://docs.netapp.com/
- BeeGFS documentation
  http://www.beegfs.io

Version History

Version	Date	Document Version History
Version 1.0	July 2019	First Release