

Dell EMC Isilon and NVIDIA DGX-1 servers for deep learning



This document demonstrates how the Dell EMC Isilon F800 All-Flash Scale-out NAS and NVIDIA® DGX-1™ servers with NVIDIA Tesla® V100 GPUs can be used to accelerate and scale deep learning and machine learning training and inference workloads. The results of industry-standard image classification benchmarks using TensorFlow are included.

August 2019

This document may contain certain words that are not consistent with Dell's current language guidelines. Dell plans to update the document over subsequent future releases to revise these words accordingly.

This document may contain language from third party content that is not under Dell's control and is not consistent with Dell's current guidelines for Dell's own content. When such third party content is updated by the relevant third parties, this document will be revised accordingly.

The information in this publication is provided "as is." DELL EMC Corporation makes no representations or warranties of any kind with respect to the information in this publication, and specifically disclaims implied warranties of merchantability or fitness for a purpose.

Use, copying, and distribution of any DELL EMC software described in this publication requires an applicable software license.

DELL EMC2, DELL EMC, the DELL EMC logo are registered trademarks or trademarks of DELL EMC Corporation in the United States and other countries. All other trademarks used herein are the property of their respective owners. © Copyright 2018 DELL EMC Corporation. All rights reserved. Published in the USA. 6/19.

DELL EMC believes the information in this document is accurate as of its publication date. The information is subject to change without notice. DELL EMC is now part of the Dell group of companies.

Table of Contents

Revisions.....	5
Executive summary.....	5
AUDIENCE	5
Introduction.....	5
Deep learning dataflow	6
Solution architecture	7
OVERVIEW	7
STORAGE: DELL EMC ISILON F800.....	8
Storage Tiering	9
OneFS Caching	9
File Reads	10
Locks and concurrency.....	10
COMPUTE: NVIDIA DGX-1 SERVER.....	11
NETWORKING: ARISTA 7060CX2-32S	11
BILL OF MATERIALS.....	12
SOFTWARE VERSIONS.....	12
Deep learning training performance and analysis.....	12
BENCHMARK METHODOLOGY	12
BENCHMARK RESULTS	15
TEST VARIATIONS.....	15
UNDERSTANDING FILE CACHING	17
UNDERSTANDING THE TRAINING PIPELINE	17
FLOATING POINT PRECISION (FP16 VS. FP32)	18
NVIDIA COLLECTIVE COMMUNICATION LIBRARY (NCCL)	19
Deep learning inference performance and analysis	20
BENCHMARK METHODOLOGY	20
BENCHMARK RESULTS	21
Storage-only performance.....	22
Machine Learning with RAPIDS.....	23
Solution sizing guidance	23
Conclusions.....	24
Appendix – System configuration.....	25
ISILON	25
Configuration.....	25
Configuring Automatic Storage Tiering.....	25
Testing Automatic Storage Tiering	27
NVIDIA DGX-1	28
ARISTA DATA SWITCHES	29
ISILON VOLUME MOUNTING	29
Appendix – Benchmark setup	30

CREATING THE IMAGENET TFRECORD DATASETS.....	30
OBTAIN THE TENSORFLOW BENCHMARKS	30
START TENSORFLOW CONTAINERS.....	30
Appendix – Monitoring Isilon performance.....	32
INSIGHTIQ	32
ISILON STATISTICS CLI	32
Appendix – Isilon performance testing with iPerf and FIO.....	33
IPERF	33
FIO.....	33
Appendix – Collecting system metrics with ELK	34
Appendix – Tips	35
References	35

Revisions

Date	Description	Author
October 2018	Initial release	Claudio Fahey
August 2019	Added sections about storage tiering with H500, RAPIDS	Claudio Fahey

Executive summary

Deep learning (DL) techniques have enabled great successes in many fields such as computer vision, natural language processing (NLP), gaming and autonomous driving by enabling a model to learn from existing data and then to make corresponding predictions. The success is due to a combination of improved algorithms, access to larger datasets, and increased computational power. To be effective at enterprise scale, the computational intensity of DL requires highly powerful and efficient parallel architectures. The choice and design of the system components, carefully selected and tuned for DL use-cases, can have a big impact on the speed, accuracy, and business value of implementing AI techniques.

In such a complex environment, it is critical that organizations be able to rely on vendors that they trust. Over the last few years, Dell EMC and NVIDIA have established a strong partnership to help organizations fast-track their AI initiatives. Our partnership is built on the philosophy of offering flexibility and informed choice across a broad portfolio which combines best of breed GPU accelerated compute, scale-out storage, and networking.

This paper focuses on how Dell EMC Isilon F800 All-Flash Scale-out NAS accelerates AI innovation by delivering the performance, scalability, and concurrency to complement the requirements of NVIDIA's Tesla V100 GPUs for high performance AI workloads.

AUDIENCE

This document is intended for organizations interested in simplifying and accelerating DL solutions with advanced computing and scale-out data management solutions. Solution architects, system administrators, and other interested readers within those organizations constitute the target audience.

Introduction

DL is an area of artificial intelligence which uses artificial neural networks to enable accurate pattern recognition of complex real-world patterns by computers. These new levels of innovation have applicability across nearly every industry vertical. Some of the early adopters include advanced research, precision medicine, high tech manufacturing, advanced driver assistance systems (ADAS) and autonomous driving. Building on these initial successes, AI initiatives are springing up in various business units, such as manufacturing, customer support, life sciences, marketing and sales. Gartner predicts that AI augmentation will generate \$2.9 Trillion in business value by 2021 alone. But, turning the promise of AI into real value isn't as easy as flipping a switch. Organizations are faced with a multitude of complex choices related to data, analytic skill-sets, software stacks, analytic toolkits and infrastructure components; each with significant implications on the time to market and the value associated with these initiatives.

In such a complex environment, it is critical that organizations be able to rely on vendors that they trust. Dell EMC and NVIDIA are at the forefront of AI providing the technology that makes tomorrow possible today. Over the last few years we have established a strong partnership to help organizations accelerate their AI initiatives. Our partnership is built on the philosophy of offering flexibility and informed choice across an extensive portfolio. Together our technologies provide the foundation for successful AI solutions which drive the development of advanced DL software frameworks, deliver massively parallel compute in the form of NVIDIA Graphic Processing Units (GPUs) for parallel model training, and scale-out file systems to support the concurrency, performance, and Petabyte scale of the unstructured image and video data sets.

This document focuses on the latest step in the Dell EMC and NVIDIA collaboration, a new AI reference architecture with Isilon F800 storage and DGX-1 servers for DL workloads. This new offer pairs DGX-1 servers with Isilon scale-out all-flash storage to give customers more flexibility in how they deploy scalable, high performance DL. The results of multiple industry standard image classification benchmarks using TensorFlow are included.



Deep learning dataflow

As visualized in Figure 1, DL usually consist of two distinct workflows, model development and inference.

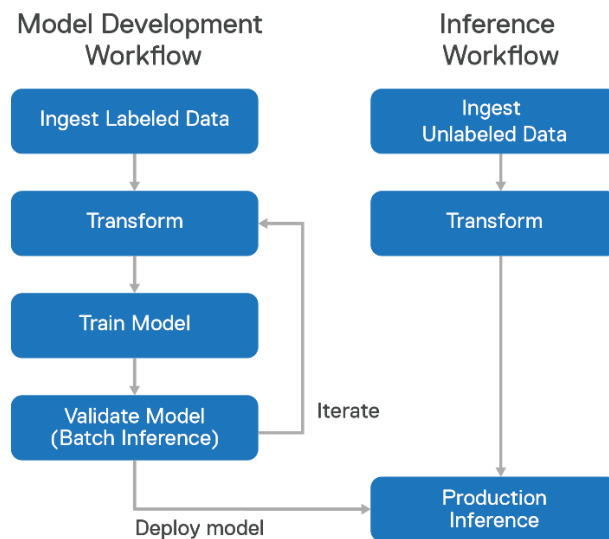


Figure 1: Common DL Workflows: Model development and inference

The workflow steps are defined and detailed below. Note that the Isilon storage and DGX-1 server architecture is optimized for the model development workflow which consists of the model training and the batch inference validation steps. It is not intended for and nor was it benchmarked for Production Inference.

1. **Ingest Labeled Data** - In this step, the labeled data (e.g. images and their labels which indicate whether the image contains a dog, cat, or horse) are ingested into the DL system.
2. **Transform** - Transformation includes all operations that are applied to the labeled data before they are passed to the DL algorithm. It is sometimes referred to as preprocessing. For images, this often includes file parsing, JPEG decoding, cropping, resizing, rotation, and color adjustments. Transformations can be performed on the entire dataset ahead of time, storing the transformed data on disk. Many transformations can also be applied in a training pipeline, avoiding the need to store the intermediate data.
3. **Train Model** - In this phase, the model parameters (edge weights) are learned from the labeled data using the stochastic gradient descent optimization method. In the case of image classification, there are several prebuilt structures of neural networks that have been proven to work well. To provide an example, Figure 2 shows the high-level workflow of the Inception-v3 model which contains nearly 25 million parameters that must be learned. In this diagram, images enter from the left and the probability of each class comes out on the right.

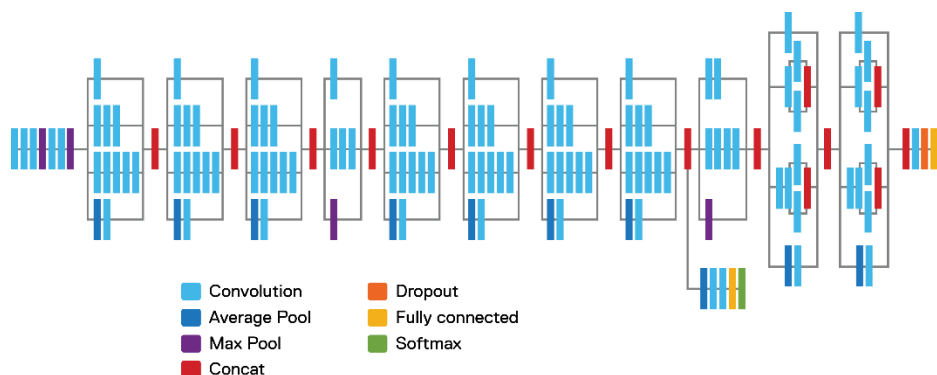


Figure 2: Inception-v3 model architecture.

4. **Validate Model** - Once the model training phase completes with a satisfactory accuracy, you'll want to measure the accuracy of it on validation data - data that the model training process has not seen. This is done by using the

trained model to make inferences from the validation data and comparing the result with the label. This is often referred to as inference but keep in mind that this is a distinct step from production inference.

5. **Production Inference** - The trained and validated model is then often deployed to a system that can perform real-time inference. It will accept as input a single image and output the predicted class (dog, cat, horse). In some cases, inputs are batched for higher throughput but higher latency.

Solution architecture

OVERVIEW

Figure 3 illustrates the reference architecture showing the key components that made up the solution as it was tested and benchmarked. Note that in a customer deployment, the number of DGX-1 servers and Isilon storage nodes will vary and can be scaled independently to meet the requirements of the specific DL workloads. Refer to the [Solution Sizing Guidance](#) section.

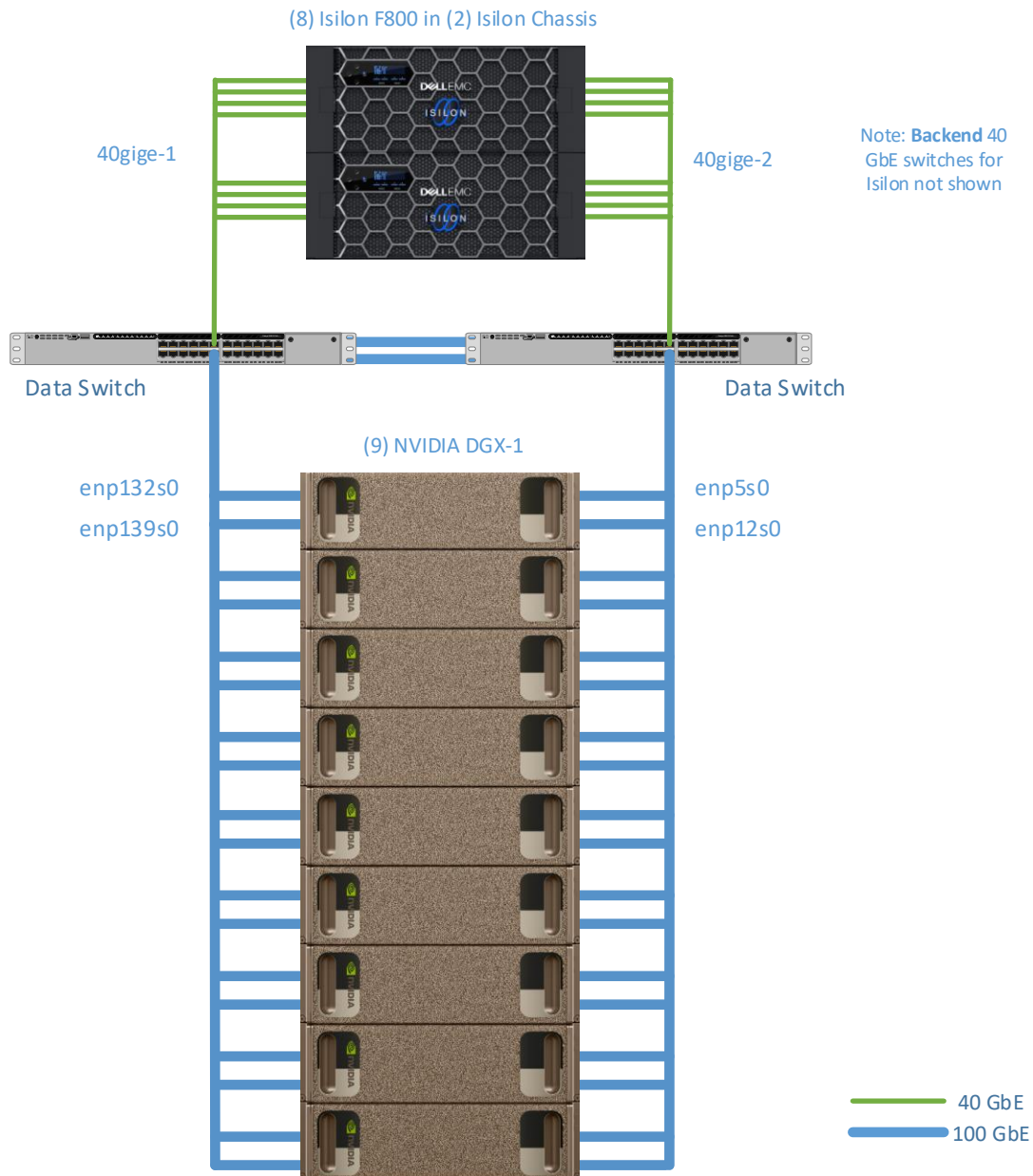


Figure 3: Reference Architecture

STORAGE: DELL EMC ISILON F800

Dell EMC Isilon F800 represents the 6th generation of hardware built to run the well-proven and massively scalable OneFS operating system. Each F800 chassis, shown in Figure 4, contains four storage nodes, 60 high performance Solid State Drives (SSDs), and eight 40 Gigabit Ethernet network connections. OneFS combines up to 144 nodes in 36 chassis into a single high-performance file system designed to handle the most intense I/O workloads such as DL. As performance and capacity demands increase, both can be scaled-out simply and non-disruptively, allowing applications and users to continue working.



Figure 4: Isilon F800 chassis, containing four storage nodes

In the solution tested in this document, eight F800 nodes, in two chassis, were used.

Dell EMC Isilon F800 has the following features.

Low latency, high throughput and massively parallel IO for AI.

- Up to 250,000 file IOPS per chassis, up to 15.75 million IOPS per cluster
- Up to 15 GB/s throughput per chassis, up to 945 GB/s per cluster
- 96 TB to 924 TB raw flash capacity per chassis; up to 58 PB per cluster (All-Flash)

This shortens time for training and testing analytical models for data sets from tens of TBs to tens of PBs on AI platforms such as TensorFlow, SparkML, Caffe, or proprietary AI platforms.

The ability to run AI in-place on data using multi-protocol access.

- Multi-protocol support such as SMB, NFS, HTTP, and native HDFS to maximize operational flexibility

This eliminates the need to migrate/copy data and results over to a separate AI stack. Organizations can perform DL and run other IT apps on same data already on Isilon by adding additional Isilon nodes to an existing cluster.

Enterprise grade features out-of-box.

- Enterprise data protection and resiliency
- Robust security options

This enables organizations to manage AI data lifecycle with minimal cost and risk, while protecting data and meeting regulatory requirements.

Extreme scale

- Seamlessly tier between All Flash, Hybrid, and Archive nodes via SmartPools.
- Grow-as-you-go scalability with up to 58 PB flash capacity per cluster
- Up to 63 chassis (252 nodes) may be connected to form a single cluster with a single namespace and a single coherent cache
- Up to 85% storage efficiency to reduce costs
- Optional data de-dup and compression enabling up to a 3:1 data reduction

Organizations can achieve AI at scale in a cost-effective manner, enabling them to handle multi-petabyte datasets with high resolution content without re-architecture and/or performance degradation.

There are several key features of Isilon OneFS that make it an excellent storage system for DL workloads that require performance, concurrency, and scale. These features are detailed below.

Storage Tiering

Dell EMC Isilon SmartPools software enables multiple levels of performance, protection and storage density to co-exist within the same file system, and unlocks the ability to aggregate and consolidate a wide range of applications within a single extensible, ubiquitous storage resource pool. This helps provide granular performance optimization, workflow isolation, higher utilization, and independent scalability – all with a single point of management.

SmartPools allows you to define the value of the data within your workflows based on policies, and automatically aligns data to the appropriate price/performance tier over time. Data movement is seamless, and with file-level granularity and control via automated policies, manual control, or API interface, you can tune performance and layout, storage tier alignment, and protection settings – all with minimal impact to your end-users.

Storage tiering has a very convincing value proposition, namely separating data according to its business value, and aligning it with the appropriate class of storage and levels of performance and protection. Information Lifecycle Management techniques have been around for a number of years, but have typically suffered from the following inefficiencies: complex to install and manage, involves changes to the file system, requires the use of stub files, etc.

Dell EMC Isilon SmartPools is a next generation approach to tiering that facilitates the management of heterogeneous clusters. The SmartPools capability is native to the Isilon OneFS scale-out file system, which allows for unprecedented flexibility, granularity, and ease of management. In order to achieve this, SmartPools leverages many of the components and attributes of OneFS, including data layout and mobility, protection, performance, scheduling, and impact management.

A typical Isilon cluster will store multiple datasets with different performance, protection, and price requirements. Generally, files that have been recently created and accessed should be stored in a hot tier while files that have not been accessed recently should be stored in a cold (or colder) tier. Because Isilon supports tiering based on a file's access time, this can be performed automatically. For storage administrators that want more control, complex rules can be defined to set the storage tier based on a file's path, size, or other attributes.

All files on Isilon are always immediately accessible (read and write) regardless of their storage tier and even while being moved between tiers. The file system path to a file is not changed by tiering. Storage tiering policies are applied and files are moved by the Isilon SmartPools job, which runs daily at 22:00 by default.

For more details, see [Storage Tiering with Dell EMC Isilon SmartPools](#).

OneFS Caching

The OneFS caching infrastructure design is predicated on aggregating the cache present on each node in a cluster into one globally accessible pool of memory. This allows all the memory cache in a node to be available to every node in the cluster. Remote memory is accessed over an internal interconnect and has much lower latency than accessing hard disk drives.

The OneFS caching subsystem is coherent across the cluster. This means that if the same content exists in the private caches of multiple nodes, this cached data is consistent across all instances.

OneFS uses up to three levels of read cache, plus an NVRAM-backed write cache, or coalescer. These, and their high-level interaction, are illustrated in Figure 5.

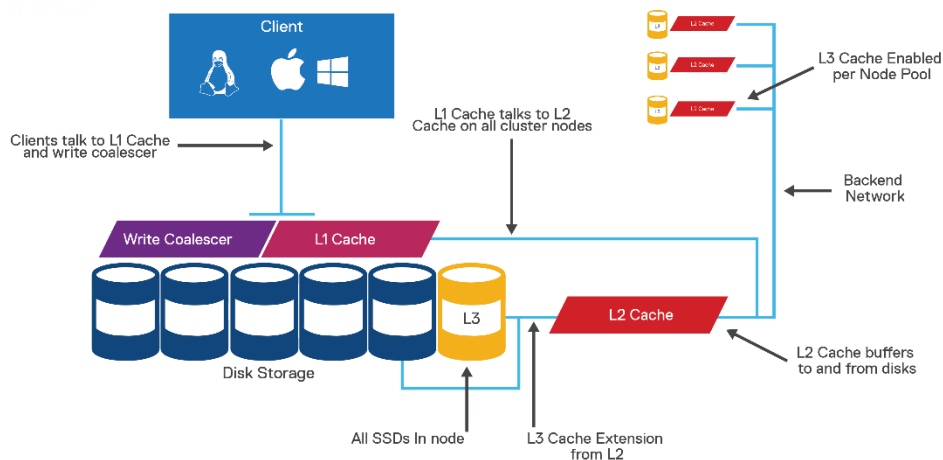


Figure 5: OneFS Caching Architecture

File Reads

For files marked with an access pattern of concurrent or streaming, OneFS can take advantage of prefetching of data based on heuristics used by the Isilon SmartRead component. SmartRead can create a data pipeline from L2 cache, prefetching into a local L1 cache on the captain node. This greatly improves sequential-read performance across all protocols and means that reads come directly from RAM within milliseconds. For high-sequential cases, SmartRead can very aggressively prefetch ahead, allowing reads of individual files at very high data rates.

Intelligent caching provided by SmartRead allows for very high read performance with high levels of concurrent access. Importantly, it is faster for Node 1 to get file data from the cache of Node 2 (over the low-latency cluster interconnect) than to access its own local disk. The SmartRead algorithms control how aggressive the pre-fetching is (disabling pre-fetch for random-access cases), how long data stays in the cache, and optimizes where data is cached. This optimized file read logic is visualized below in Figure 6.

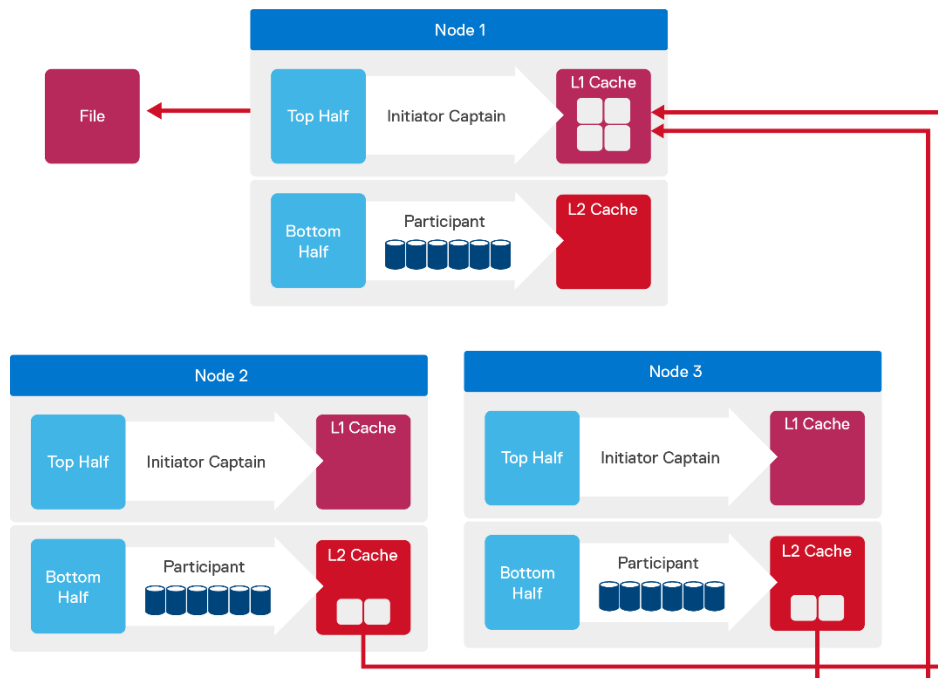


Figure 6: A File Read Operation on a 3-node Isilon Cluster

Locks and concurrency

OneFS has a fully distributed lock manager that coordinates locks on data across all nodes in a storage cluster. The locking manager is highly extensible and allows for multiple lock personalities to support both file system locks as well as

cluster-coherent protocol-level locks such as SMB share mode locks or NFS advisory-mode locks. OneFS also has support for delegated locks such as CIFS oplocks and NFSv4 delegations. Every node in a cluster is a coordinator for locking resources and a coordinator is assigned to lockable resources based upon an advanced hashing algorithm.

For more detailed technical overview of Isilon OneFS, refer to

<https://www.emc.com/collateral/hardware/white-papers/h10719-isilon-onefs-technical-overview-wp.pdf>.

Efficient locking is critical to support the efficient parallel IO profile demanded by many iterative DL workloads enabling concurrent file read access up into the millions.

COMPUTE: NVIDIA DGX-1 SERVER

The DGX-1 server is a fully integrated, turnkey hardware and software system that is purpose-built for DL workflows. Each DGX-1 server is powered by eight Tesla V100 GPUs that are configured in a hybrid cube mesh topology that uses NVIDIA NVLink™ technology, which provides an ultra-high bandwidth low-latency fabric for inter-GPU communication. This topology is essential for multi-GPU training, eliminating the bottleneck that is associated with PCIe-based interconnects that cannot deliver linearity of performance as GPU count increases. The DGX-1 server is also equipped with high-bandwidth, low-latency network interconnects for multi-node clustering over RDMA-capable fabrics.



Figure 7: NVIDIA DGX-1 server with eight Tesla V100 GPUs

The NGC container registry provides researchers, data scientists, and developers with simple access to a comprehensive catalog of GPU-accelerated software for AI, machine learning and HPC that take full advantage of NVIDIA DGX-1 GPUs on-premises and in the cloud. NGC provides containers for today's most popular DL frameworks such as Caffe2, TensorFlow, PyTorch, MXNet, and TensorRT, which are optimized for NVIDIA GPUs. The containers integrate the framework or application, necessary drivers, libraries, and communications primitives, and they are optimized across the stack by NVIDIA for maximum GPU-accelerated performance. NGC containers incorporate the NVIDIA CUDA Toolkit, which provides the NVIDIA CUDA Basic Linear Algebra Subroutines Library (cuBLAS), the NVIDIA CUDA Deep Neural Network Library (cuDNN), and much more. The NGC containers also include the NVIDIA Collective Communications Library (NCCL) for multi-GPU and multi-node collective communication primitives, enabling topology awareness for DL training. NCCL enables communication between GPUs inside a single DGX-1 server and across multiple DGX-1 servers.

NETWORKING: ARISTA 7060CX2-32S

The Arista 7060CX2 is a 1RU high performance 40 GbE and 100 GbE high density, fixed configuration, data center switch with wire speed layer 2 and layer 3 features, and advanced features for software driven cloud networking. It delivers a rich choice of interface speed and density allowing networks to seamlessly evolve from 10 GbE and 40 GbE to 25 GbE and 100 GbE. With support for advanced EOS features these switches are ideal for traditional or fully virtualized data centers. Arista 7060CX2-32S provides support for IEEE 25 GbE and support a larger shared packet buffer pool of 22 MB with the same low latency of 450 ns.



Figure 8: Arista 7060CX2-32S: 32 x 40/100GbE QSFP100ports, 2 SFP+ ports

BILL OF MATERIALS

Component	Purpose	Quantity
Dell EMC Isilon F800 96 TB SSD 1 TB RAM 4x1GE, 8x40GE Networking	Storage	2 4U Chassis (8 nodes)
Celestica D4040 851-0259	Isilon back-end Ethernet switch	2
NVIDIA DGX-1 server (8) Tesla V100-SXM2-32GB GPUs (2) 20-Core Intel Xeon E5-2698 v4 2.2 GHz 512 GB 2,133 MHz DDR4 RDIMM RAM	Compute server with 8 GPUs	9
Arista 7060CX2-32S	Data switch	2
Mellanox MCP1600-E02A Passive Copper Cable IB EDR up to 100Gb/s QSFP LSZH 2.5m 26AWG	100G cable - DGX-1 server to Arista	36
Optical cable, MPO, 038-004-218	100G cable - Arista cross connection	2
Optical cable, MPO, 038-004-218	40G cable – Isilon front-end	16
Optical cable, MPO, 038-004-218	40G cable – Isilon back-end	16

Table 1: Bill of materials

SOFTWARE VERSIONS

Table 2 shows the software versions that were tested for this document.

Component	Version
Isilon - OneFS	8.1.0.4 patches: 231017, 231838, 232711, 239173
NVIDIA GPU Cloud Image	nvidia.io/nvidia/tensorflow:18.09-py3
TensorFlow Benchmarks	https://github.com/claudiofahey/benchmarks/commit/6cdfc0c
TensorFlow Benchmark Util	https://github.com/claudiofahey/tensorflow-benchmark-util/commit/07c9a9c
Arista – System Image	4.19.8M
DGX-1 – Ubuntu	16.04.4 LTS
DGX-1 – Base OS	3.1.6
DGX-1 – BIOS	5.11
DGX-1 – NVIDIA Driver	384.125

Table 2: Software Versions

Deep learning training performance and analysis

BENCHMARK METHODOLOGY

In order to measure the performance of the solution, various benchmarks from the TensorFlow Benchmarks repository were carefully executed (see <https://www.tensorflow.org/performance/benchmarks#methodology>). This suite of benchmarks performs training of an image classification convolutional neural network (CNN) on labeled images. Essentially, the system learns whether an image contains a cat, dog, car, train, etc. The well-known [ILSVRC2012](#) image dataset (often referred to as ImageNet) was used. This dataset contains 1,281,167 training images in 144.8 GB¹. All images are grouped into 1000 categories or classes. This dataset is commonly used by DL researchers for benchmarking and comparison studies.

The individual JPEG images in the ImageNet dataset were converted to 1024 TFRecord files (see Appendix – Benchmark Setup). The TFRecord file format is a Protocol Buffers binary format that combines multiple JPEG image files together with their metadata (bounding box for cropping, and label) into one binary file. It maintains the image compression offered by the JPEG format and the total size of the dataset remained roughly the same (148 GB). The average image size was 115 KB.

¹ All unit prefixes use the SI standard (base 10) where 1 GB is 1 billion bytes.

When running the benchmarks on the 148 GB dataset, it was found that the storage I/O throughput gradually decreased and became virtually zero after a few minutes. This indicated that the entire dataset was cached in the Linux buffer cache on each DGX-1 server. Of course, this is not surprising since each DGX-1 server has 512 GB of RAM and this workload did not significantly use RAM for other purposes. As real datasets are often significantly larger than this, we wanted to determine the performance with datasets that are not only larger than the DGX-1 server RAM, but larger than the 2 TB of coherent shared cache available across the 8-node Isilon cluster. To accomplish this, we simply made 150 exact copies of each TFRecord file, creating a 22.2 TB dataset.

In our own testing, a parallel Python MPI script was created to quickly create the copies utilizing all DGX-1 and Isilon nodes. But to illustrate the copy process, it was basically as simple as this:

```
cp train-00000-of-01024 train-00000-of-01024-copy-000
cp train-00000-of-01024 train-00000-of-01024-copy-001
cp train-00000-of-01024 train-00000-of-01024-copy-002
cp train-00001-of-01024 train-00001-of-01024-copy-000
cp train-00001-of-01024 train-00001-of-01024-copy-001
cp train-00001-of-01024 train-00001-of-01024-copy-002
...
cp train-01023-of-01024 train-01023-of-01024-copy-002
```

Having 150 copies of the exact same images doesn't improve training accuracy or speed but it does produce the same I/O pattern for the storage, network, and GPUs. Having identical files did not provide an unfair advantage as Isilon deduplication was not enabled and all images are reordered randomly (shuffled) in the input pipeline.

For this workload, it is straightforward to quantify the effect of caching. Essentially there are many threads reading the TFRecord files. Each thread picks a TFRecord file at random, reads it sequentially and completely, and then it moves on to another TFRecord file at random. Sometimes, a thread will choose to read a file that another thread has recently read and that can be served from cache (either Isilon or Linux). The probability of this occurring, and therefore the fraction of data served by cache, is simply the cache size divided by the dataset size. Using this calculation, we expect a 2% cache hit rate from the Linux cache on the DGX-1 server and a 9% cache hit rate from Isilon. Conversely, we can say that 91% of the dataset is read from Isilon's SSDs. If the dataset were twice as large (44 TB), 95% of the dataset would need to be read from Isilon's SSDs, which is only a slightly faster rate than what we have measured with the 22 TB dataset.

One of the critical questions one has when trying to size a system is how fast the storage must be so that it is not a bottleneck. To answer this question, we take advantage of the fact that the Linux buffer cache can completely cache the entire 148 GB dataset. After performing several warm-up runs, the benchmark is executed and it is confirmed that there is virtually zero NFS network I/O to the storage system. The image rate (images/sec) measured in this way accounts for the significant preprocessing pipeline as well as the GPU computation. In the next section, results using this method are labeled Linux Cache.

To avoid confusion, the performance results using the synthetic benchmark mode are not reported in this document. In the synthetic mode of the benchmark, random images are generated directly in the GPU and training is based on these images. This mode is very useful to tune and understand parts of the training pipeline. However, it does not perform storage I/O, JPEG decoding, image resizing, or any other preprocessing.

There are a variety of ways to parallelize model training to take advantage of multiple GPUs across multiple servers. In our tests, we used MPI and Horovod.

Prior to each execution of the benchmark, the L1 and L2 caches on Isilon were flushed with the command "isi_for_array isi_flush". In addition, the Linux buffer cache was flushed on all DGX-1 servers by running "sync; echo 3 > /proc/sys/vm/drop_caches". However, note that the training process will read the same files repeatedly and after just several minutes, much of the data will be served from one of these caches.

The command below was used to perform the ResNet-50 training with 72 GPUs.

```
mpirun \
--n 72 \
--allow-run-as-root \
--host dgx1-1:8,dgx1-2:8,dgx1-3:8,dgx1-4:8,dgx1-5:8,dgx1-6:8,dgx1-7:8,dgx1-8:8,dgx1-9:8 \
--report-bindings \
--bind-to none \
--map-by slot \
-x LD_LIBRARY_PATH \
-x PATH \
-mca plm_rsh_agent ssh \
-mca plm_rsh_args "-p 2222" \
```

```

-mca pml ob1 \
-mca btl ^openib \
-mca btl_tcp_if_include enp132s0 \
-x NCCL_DEBUG=INFO \
-x NCCL_IB_HCA=mlx5 \
-x NCCL_IB_SL=4 \
-x NCCL_IB_GID_INDEX=3 \
-x NCCL_NET_GDR_READ=1 \
-x NCCL_SOCKET_IFNAME=docker0 \
./round_robin_mpi.py \
python \
-u \
/mnt/isilon/data/tensorflow-benchmarks/scripts/tf_cnn_benchmarks/\
tf_cnn_benchmarks.py \
--model=resnet50 \
--batch_size=256 \
--batch_group_size=20 \
--num_batches=500 \
--nodistortions \
--num_gpus=1 \
--device=gpu \
--force_gpu_compatible=True \
--data_format=NCHW \
--use_fp16=True \
--use_tf_layers=False \
--data_name=imagenet \
--use_datasets=True \
--num_intra_threads=1 \
--num_inter_threads=10 \
--datasets_prefetch_buffer_size=20 \
--datasets_num_private_threads=10 \
--train_dir=/mnt/isilon/data/train_dir/2018-10-11-19-36-39-resnet50 \
--sync_on_finish=True \
--summary_verbosity=1 \
--save_summaries_steps=100 \
--save_model_secs=600 \
--variable_update=horovod \
--horovod_device=gpu

```

The script `round_robin_mpi.py` was used to add the `--data_dir` parameter that distributed each process across four different mount points and thus to different Isilon nodes. Note that when testing the Linux Cache performance, only a single mount point was used.

For the other models, only the `--model` parameter was changed. For different numbers of GPUs, only the `--np` parameter was changed. Note that the `-map-by` slot setting causes MPI to use all eight GPUs (slots) on a DGX-1 server before it begins using the next DGX-1 server. For example, when testing 32 GPUs, only four DGX-1 servers are used.

The benchmark results in this section were obtained with eight Isilon F800 nodes in the cluster.

BENCHMARK RESULTS

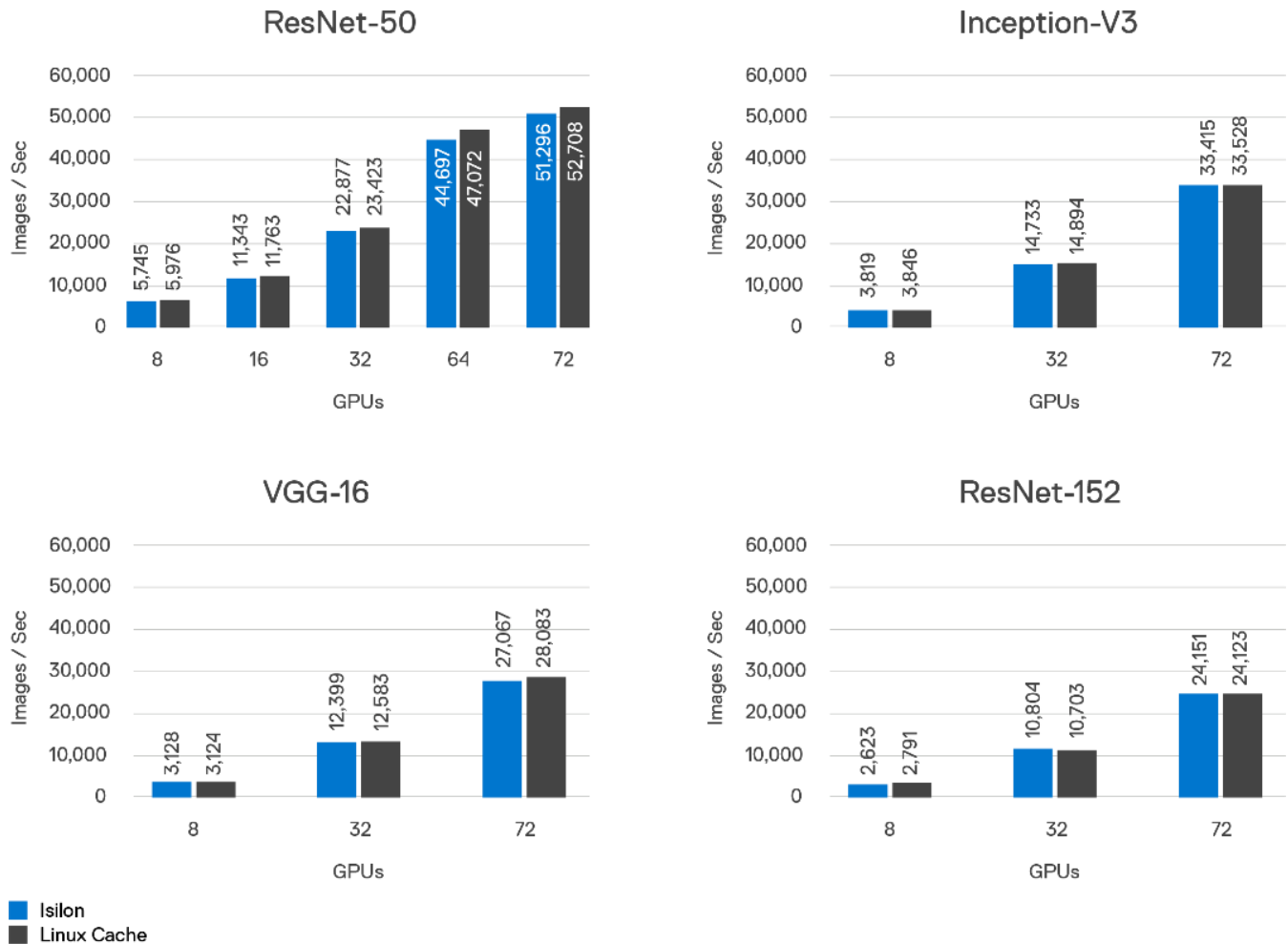


Figure 9: Model Development – Training Benchmark Results

There are a few conclusions that we can make from the benchmarks represented in Figure 9.

- Image throughput and therefore storage throughput scale linearly from 8 to 72 GPUs.
- The maximum throughput that was pulled from Isilon occurred with ResNet-50 and 72 GPUs. The total storage throughput was 5907 MB/sec.
- For all tests shown above, each GPU had 97% utilization or higher. This indicates that the GPU was the bottleneck.
- The maximum CPU utilization on the DGX-1 server was 46%. This occurred with ResNet-50.
- The difference between Linux Cache and Isilon averages only 2% and is at worst 6%. The reason for this minor decrease in performance is not known. Possible reasons include 1) storage network traffic increases network latency during gradient updates, 2) CPU handling NFS I/O slows other CPU tasks, and 3) preprocessing is sensitive to storage latency (this is not expected because there are large preprocessing buffers).

TEST VARIATIONS

Measurement of Network I/O between DGX-1 Servers

To get a sense of the network I/O between the DGX-1 servers during training, all Horovod traffic (i.e. gradient values) was forced to a single non-RoCE NIC and a Linux Cache test was performed. ResNet-50 with 72 GPUs was observed to generate 545 MB/sec (4.4 Gbps) per DGX-1 server for both send and receive with an average of 711 images/sec/GPU. Adding together this non-storage traffic across all nine DGX-1 servers in both directions gives a total of 9810 MB/sec.

Training with 4 Isilon F800 Nodes

In another variation, four of the Isilon F800 nodes were SmartFailed, leaving four nodes in the cluster. ResNet-50 training was performed on 72 GPUs and the performance result was identical to that with eight Isilon F800 nodes. This workload pulled only around 6 GB/sec.

Large image training

The benchmarks in the previous section used the original JPEG images from the ImageNet dataset, with an average size of 115 KB. Today it is common to perform DL on larger images. For this section, a new set of TFRecord files are generated by resizing all images to three times their original height and width. Each image is encoded as a JPEG with a quality of 100 to further increase the number of bytes. Finally, we make 13 copies of each of the 1024 new TFRecord files. This results in a new dataset that is 22.5 TB and has an average image size of 1.3 MB.

Because we are using larger images with the best JPEG quality, we want to match it with the most sophisticated model in the TensorFlow Benchmark suite, which is Inception-v4.

Note that regardless of the image height and width, all images must be cropped and/or scaled to be exactly 299 by 299 pixels to be used by Inception-v4. Thus, larger images place a larger load on the preprocessing pipeline (storage, network, CPU) but not on the GPU.

The benchmark results in Figure 10 were obtained with eight Isilon F800 nodes in the cluster.

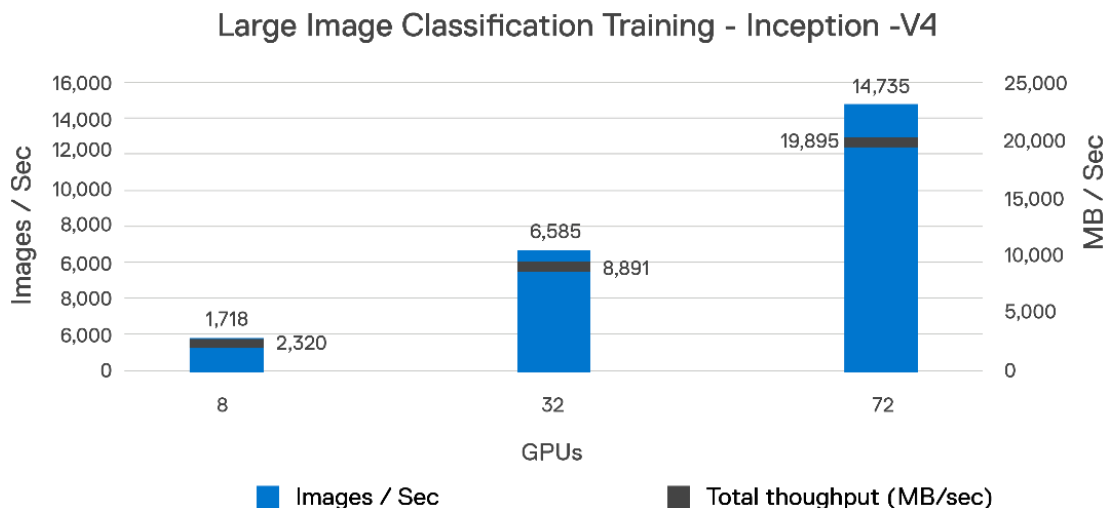


Figure 10: Training performance with large images

As before, we have linear scaling from 8 to 72 GPUs. The storage throughput with 72 GPUs is now 19,895 MB/sec. GPU utilization is at 98% and CPU utilization is at 84%.

This training benchmark was executed for 6 hours. Figure 11 and Figure 12 below show some of the key metrics collected by Isilon InsightIQ. The I/O throughput is constant. The disk throughput (reading from the SSDs on Isilon) is approximately 91% of the network throughput (NFS). This exactly matches our prediction of the cache hit rate.

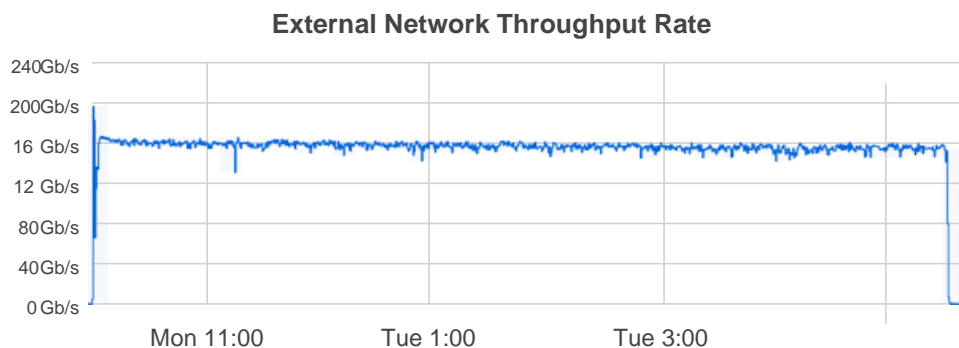


Figure 11: External network throughput rate (from Isilon to DGX-1 server) during training with large images. The rate is approximately 160 Gbps (20 GB/sec)

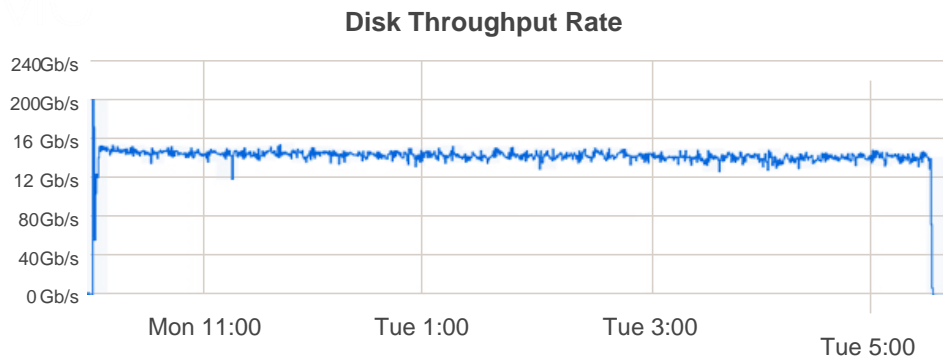


Figure 12: Isilon SSD throughput rate during training with large images. The rate is approximately 145 Gbps (18 GB/sec)

Training with 4 Isilon H500 Hybrid Nodes

In this variation, a single DGX-1 server was connected to an Isilon cluster composed of four Isilon F800 all-flash nodes and four Isilon H500 hybrid nodes. First a ResNet50 training benchmark was executed with the data on the F800 tier. Then a SmartPools job was executed to move the ImageNet data to the H500 tier and the training benchmark was executed again. The images per second rate was the same in both benchmark runs. This is not surprising since there were only 8 GPUs which demanded a total of 700 MB/sec. An Isilon chassis with four H500 nodes has 60 SATA drives and can deliver up to 5 GB/sec and saturate up to 56 GPUs (14 C4140 nodes with 4-GPUs each or 7 DGX-1 nodes).

Bottom-line: an H500 is a perfectly viable option for large scale deep learning, despite the general perception in the industry that all-flash storage is required for Deep Learning.

UNDERSTANDING FILE CACHING

There are several caches related to NFS and Isilon storage.

Isilon Cache

An Isilon cluster has L1 and L2 cache which utilize the RAM on each Isilon node. Some Isilon models also have an L3 cache. The L3 cache system will store frequently-used data on the SSDs within each Isilon node, avoiding I/O to slower HDDs. The F800 all-flash array does not have L3 cache as all data is on SSDs.

Linux Buffer Cache

A DGX-1 server, like other Linux-based hosts, has a buffer cache. The Linux buffer cache uses RAM that is otherwise unused to store recently used file blocks that were read from, or written to, local disks or NFS filesystems such as Isilon. There is generally no need to tune the Linux buffer cache and it is enabled by default on all modern versions of Linux. It will automatically grow as more data is read from disks (local and NFS) and it will be released when applications need additional RAM.

Linux NFS File System Cache

When enabled, the Linux NFS File System Cache uses the local disk to cache NFS files. In the solution described in this document (including all benchmarking), File System Cache is not used. In some cases, such as when using a slow NFS server, it may help to enable the File System Cache and point it to a local SSD. It is disabled by default. It can be enabled by settings the `fsc` mount option and starting `cachefilesd`.

UNDERSTANDING THE TRAINING PIPELINE

To tune the performance of the training pipeline, it is helpful to understand the key parts of the pipeline shown in Figure 13.

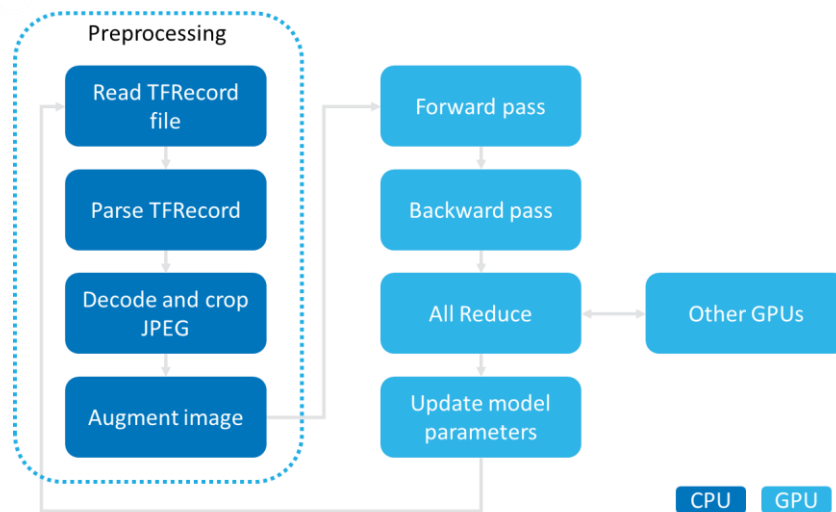


Figure 13: Training pipeline

1. Preprocessing

- (CPU) Read TFRecord files from disk. This may require NFS I/O to the storage system or it may be served by the Linux buffer cache in RAM.
- (CPU) Parse the TFRecord file into individual records and fields. The JPEG-encoded image is in one of these fields.
- (CPU) The JPEG is cropped and decoded. We now have an uncompressed RGB image. This is generally the most CPU-intensive step and can become a bottleneck in certain cases.
- (CPU) The cropped image is resized to 299x299 (Inception only) or 224x224 (all other models).
- (CPU) The image is randomly flipped horizontally or vertically.
- (CPU) If distortions are enabled, the image brightness is randomly adjusted. Note that distortions were disabled for all benchmarks described in this document.
- (CPU and GPU) The image is copied from the CPU RAM to the GPU RAM.

2. Forward and Backward Pass

- (GPU) Run the image through the model's forward pass (evaluate loss function) and backward pass (back propagation) to calculate the gradient.

3. Optimization

- (GPU) All GPUs across all nodes exchange and combine their gradients through the network using the All Reduce algorithm. In this solution, the communication is accelerated using NCCL and NVLink technology, allowing the GPUs to communicate through the Ethernet network, bypassing the CPU and PCIe buses.
- (GPU) The model parameters are updated based on the combined gradients and the optimization algorithm (gradient descent).
- Repeat until the desired accuracy (or another metric) is achieved.

All the steps above are done concurrently. For example, one CPU thread is reading a TFRecord file, while another is performing JPEG decoding on a file that was read several milliseconds prior, and this occurs while the GPU is calculating gradients on images that were resized several milliseconds ago. Additionally, there is batching and buffering at various stages to optimize the performance. As you can see, the preprocessing pipeline is completely handled by the CPUs on the DGX-1 server.

FLOATING POINT PRECISION (FP16 VS. FP32)

The NVIDIA Tesla V100 GPU contains a new type of processing core called Tensor Core which supports mixed precision training. Although many high-performance computing (HPC) applications require high precision computation with FP32

(32-bit floating point) or FP64 (64-bit floating point), DL researchers have found they are able to achieve the same inference accuracy with FP16 (16-bit floating point) as can be had with FP32. In this document, mixed precision training which includes FP16 and FP32 representations is denoted as “FP16” training.

Although FP16 makes training faster, it requires extra work in the neural network model implementation to match the accuracy achieved with FP32. This is because some neural networks require their gradient values to be shifted into FP16 representable range and may do some scaling and normalization to use FP16 during training. For more details, please refer to NVIDIA [mixed precision training](#).

All benchmark results in this document were obtained using FP16.

NVIDIA COLLECTIVE COMMUNICATION LIBRARY (NCCL)

There are two significant types of network traffic when performing training. First, there is the NFS traffic required to read the images (TFRecord files). This uses standard TCP/IP on Ethernet. Second, the gradient (the derivative of the loss function with respect to each parameter) calculated on each GPU must be averaged with the gradient from all other GPUs and the resulting average gradient must be distributed to all GPUs. This is performed optimally using the [MPI All Reduce](#) algorithm. See Figure 14 below, which optimally calculates the sum of values from all nodes (top) and stores the sum on all nodes (bottom).

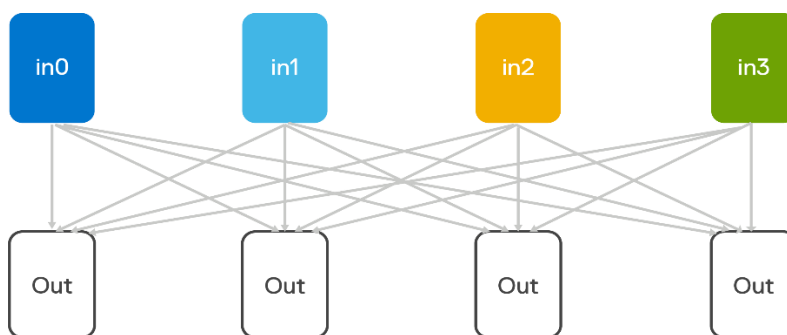


Figure 14: The All Reduce algorithm.

NCCL provides fast collectives (such as All Reduce) over multiple GPUs both within and across nodes. It supports a variety of interconnect technologies including PCIe, NVLink technology, InfiniBand Verbs, and IP sockets. NCCL also automatically patterns its communication strategy to match the system's underlying GPU interconnect topology. When configured properly on the DGX-1 server, NCCL allows GPUs in different nodes to communicate with each other through the PCIe switch, NIC, and the Ethernet switch, bypassing the CPU and the main system memory. This logic is visualized below in Figure 15. Note: The corners of the mesh-connected faces of the cube are connected to the PCIe tree network, which also connects to the CPUs and NICs.

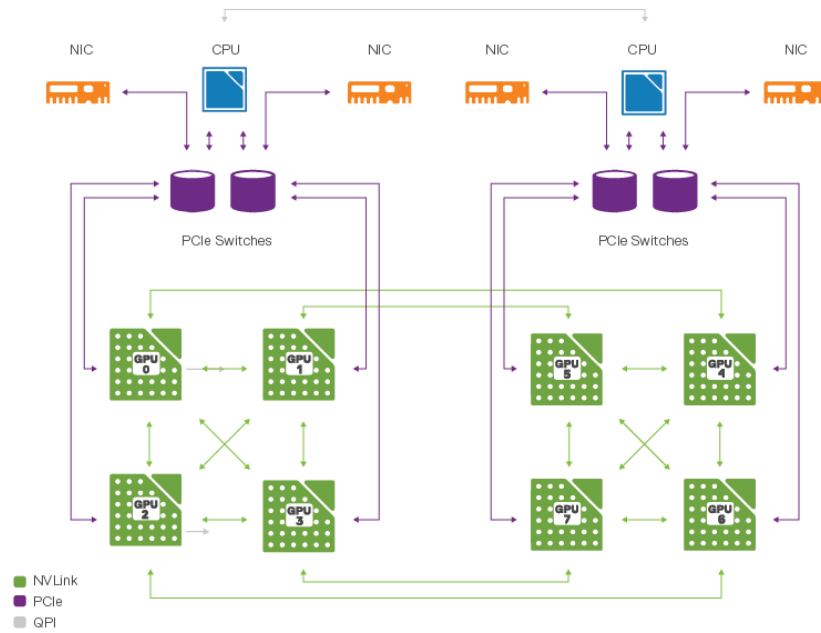


Figure 15: DGX-1 server uses an 8-GPU hybrid cube-mesh interconnection network topology.

The mpirun parameters below will configure NCCL to use the four RoCE-enabled NICs (mlx5_0 through mlx5_3).

```
-x NCCL_DEBUG=INFO \
-x NCCL_IB_HCA=mlx5 \
-x NCCL_IB_SL=4 \
-x NCCL_IB_GID_INDEX=3 \
-x NCCL_NET_GDR_READ=1 \
-x NCCL_SOCKET_IFNAME=^docker0 \
```

To confirm that NCCL is optimally configured, carefully review the log messages of the training job and confirm that there are lines with “x -> x via NET/IB/x/GDRDMA”. For example:

```
DGX1-1:1626:1760 [0] NCCL INFO CUDA Dev 0, IB Ports : mlx5_3/1(SOC) mlx5_2/1(SOC)
mlx5_1/1(PHB) mlx5_0/1(PIX)
DGX1-1:1627:1763 [1] NCCL INFO comm 0x7f7db8539180 rank 1 nrank 4
DGX1-1:1627:1763 [1] NCCL INFO NET : Using interface enp132s0:10.55.66. 231<0>
DGX1-1:1627:1763 [1] NCCL INFO NET/Socket : 1 interfaces found
DGX1-1:1627:1763 [1] NCCL INFO CUDA Dev 1, IB Ports : mlx5_3/1(SOC) mlx5_2/1(SOC)
mlx5_1/1(PHB) mlx5_0/1(PIX)
DGX1-1:1626:1760 [0] NCCL INFO 3 -> 0 via NET/IB/3/GDRDMA
DGX1-1:1626:1760 [0] NCCL INFO Ring 00 : 0[0] -> 1[1] via P2P/IPC
DGX1-1:1627:1763 [1] NCCL INFO Net: enabling net device 3 to read from rank 1
DGX1-1:1627:1763 [1] NCCL INFO NCCL_IB_GID_INDEX set by environment to 3.
DGX1-1:1627:1763 [1] NCCL INFO NET/IB: Dev 3 Port 1 qpn 687 mtu 5 GID 3 (0/
E705370AFFFFF0000)
```

Deep learning inference performance and analysis

BENCHMARK METHODOLOGY

In this section, the same methodology is used as in the previous training section except that the **forward_only** parameter is added. This configures the benchmark to perform the same preprocessing and forward pass (inference) as during training. However, the forward-only mode skips the backward pass (back propagation). Forward-only mode avoids the calculation and communication of the gradient. For each image, there is less work for the GPU and network to perform. Therefore, forward-only mode can always run at a faster image rate than training mode.

Another important difference between the two modes is that in training mode, all threads must synchronize after each batch because they must communicate their gradients. This is why the optimization method is sometimes called synchronous stochastic gradient descent. (Although there are multiple asynchronous optimization methods they are not used as a part this document). In forward-only mode, synchronization does not occur. This generally results in some threads going consistently faster than others because there is no mechanism to ensure a uniform speed.

The CNN benchmark in the TensorFlow Benchmarks suite runs the forward-only mode using Horovod (which uses MPI) to launch an inference process for each GPU on each server. Once launched however, there is no synchronization of the processes. When a process completes a pre-determined number of steps, it prints out the average rate obtained by just that process. For example, when running with 72 GPUs, there would be 72 reports for the average image/sec rate. The first reported rate would always be the fastest because it completed the same number of images in the shortest time. However, this value is not the average rate of all processes because the other 71 processes are still running. The last reported rate would always be the slowest. However, this is also not an accurate measurement because the concurrency will have varied from 72 to 1. In fact, because of the variance in the concurrency, none of the reported rates, nor an aggregation of them, can provide an estimate of the expected long-term images/sec rate.

To work around this limitation, a slightly different test was performed. The num_batches parameter was set to 50,000 preventing even the fastest thread from completing. After 15 minutes, the average storage throughput was measured to be 11,825 MB/sec (for 72 GPUs). From this, the average images/sec was calculated by dividing by the average image size (115 KB/image). Unfortunately, this methodology does not easily allow one to measure the image/sec rate when the data is in the Linux Cache. However, it was observed that the GPU utilization (68%) and CPU utilization (78%) were identical whether the data came from Isilon or from the Linux cache, suggesting that there was no measurable slow down caused by using Isilon.

The benchmark results in this section were obtained with only four Isilon F800 nodes in the cluster. The other four nodes were SmartFailed.

BENCHMARK RESULTS

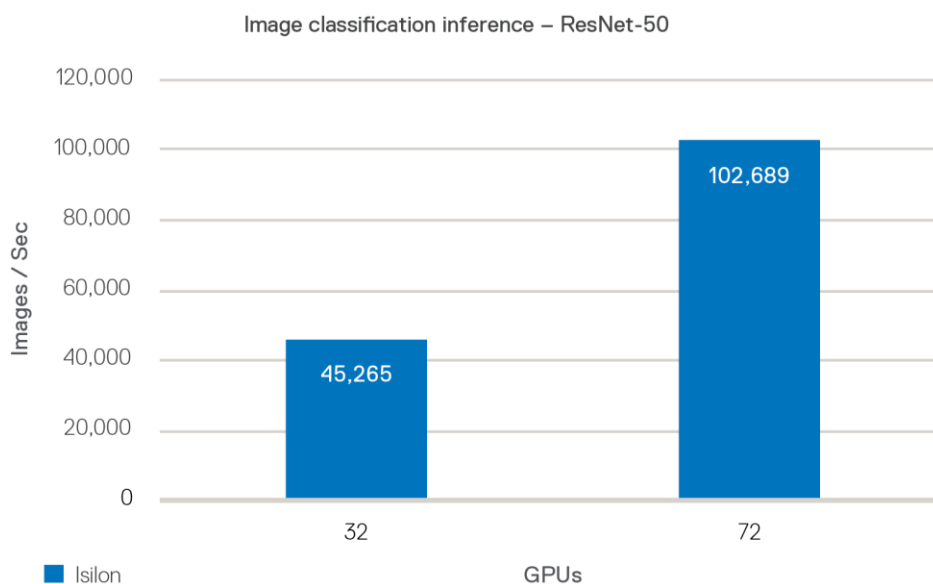


Figure 16: Inference using four Isilon F800 nodes

There are a few conclusions that we can make from the benchmarks represented above in Figure 18.

- Image throughput and therefore storage throughput scale linearly from 32 to 72 GPUs.
- The maximum throughput that was pulled from Isilon occurred with ResNet-50 and 72 GPUs. The total storage throughput was 11,825 MB/sec.
- GPU utilization was 68% and CPU utilization was 78%.
- Despite the reported CPU utilization of only 78%, it is believed that the bottleneck is related to the CPU. Despite the reported CPU utilization of only 78%, it is believed that the bottleneck is related to the CPU on the DGX-1 server. This is based on the observations that the GPU utilization is only 68%, the same performance is observed when

using Linux Cache instead of Isilon, and when tests are run with only four GPUs/node, the images/sec/GPU rate increases significantly.

Storage-only performance

To understand the limits of Isilon when used with TensorFlow, a TensorFlow application was created (*storage_benchmark_tensorflow.py*) that only reads the TFRecord files (the same ones that were used for training and inference). No preprocessing nor GPU computation is performed. The only work performed is counting the number of bytes in each TFRecord. There is no synchronization between processes during this testing, so some processes may read faster than others. The result of this benchmark is shown below.

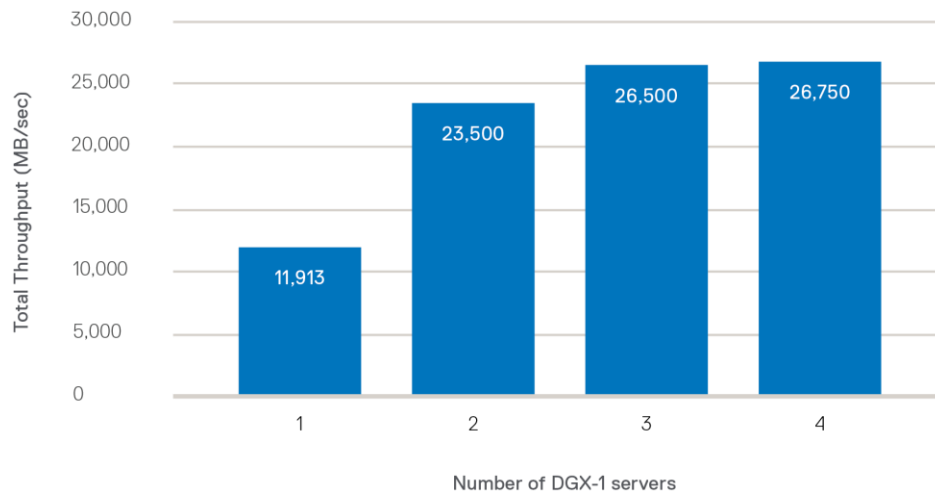


Figure 17: Maximum read throughput from eight Isilon F800 nodes with a trivial asynchronous workload.

There is linear scaling from one to two DGX-1 servers, but it clearly tops out at 26.5 GB/sec at three nodes. In this solution consisting of eight Isilon F800 nodes, this is the upper limit of an asynchronous TensorFlow application such as one performing inference.

Synchronous reading, in which all processes synchronize with each other after each step (as when training), are generally slower than the asynchronous reading shown above. The storage benchmark used above was not used for synchronous reading benchmarking. However, for a lower limit, we can use the throughput of 20 GB/sec obtained in the Large Image Training section.

As Isilon has been demonstrated to scale to 144 nodes, additional throughput can be obtained simply by adding Isilon nodes.

Machine Learning with RAPIDS

The [RAPIDS](#) suite of software libraries can accelerate some machine learning algorithms and analytics pipelines. Like TensorFlow, it uses NVIDIA GPUs for acceleration. RAPIDS requires CUDA 9.2 or 10.0 so you must ensure that the NVIDIA driver on the host supports this.

Below shows the steps to run a simple RAPIDS benchmark using the [mortgage demo](#), which uses a 195 GB dataset (uncompressed)

```
mkdir -p /mnt/isilon/data/mortgage
cd /mnt/isilon/data/mortgage

wget http://rapidsai-data.s3-website.us-east-2.amazonaws.com/notebook-mortgage-
data/mortgage_2000-2016.tgz
tar -xzf mortgage_2000-2016.tgz

docker run --runtime=nvidia \
-it \
-p 8888:8888 \
-p 8787:8787 \
-p 8786:8786 \
-v /mnt:/mnt \
--name rapidsai \
nvcr.io/nvidia/rapidsai/rapidsai:0.5-cuda10.0-runtime-ubuntu18.04-gcc7-py3.7

jupyter@12b8e880eef:/rapids/notebooks$
source activate rapids

(rapids) jupyter@12b8e880eef:/rapids/notebooks$
bash utils/start-jupyter.sh
```

Open your browser to <http://host:8888/>.

Open the notebook *mortgage/E2E.ipynb*.

Change *acq_data_path* to *"/mnt/isilon/data/mortgage/acq"*. Change other paths in same way.

For a quick test, change *end_year* to 2000 and *part_count* to 1. For a benchmark, leave these at the default of 2016 and 16.

From the menu, click Kernel -> Restart Kernel and Run All Cells.

Record the time elapsed for following cells:

- ETL (cell 17)
- Load into GPUs (cell 21)
- Training (cell 22)

To provide an idea of how long several key phases took to run, the following elapsed times were obtained on a single DGX-1 server connected to a 4-node Isilon F800 cluster.

Phase	Notebook Cell	Elapsed Time
ETL	17	1min 39s
Load into GPUs	21	2min 3s
Training	22	1min 35s

Solution sizing guidance

DL workloads vary significantly with respect to the demand for compute, memory, disk and IO profiles, often by orders of magnitude. Sizing guidance for the GPU quantity and configuration of Isilon nodes can only be provided when these resource requirements are known in advance. That said, it's usually beneficial to have a few data points on the ratio of GPUs per Isilon node for common image classification benchmarks.

The following results in Table 3 show a few such data points:

Storage Performance Demanded	Benchmark	Required Storage Throughput per V100 GPU (MB/sec/GPU)	V100 GPUs per Isilon Node	
			F800	H500
Low	Training, Small Images, ResNet-50	90	27	7

Medium	Inference, Small Images, ResNet-50	160	20	5
High	Training, Large Images, Inception-v4	300	9	2

Table 3: Sizing consideration based on benchmarked workloads

To illustrate, training of ResNet-50 with small images and 72 GPUs would need a storage system that can read $72 * 80 = 5760$ MB/sec which can be handled by 2.4 Isilon nodes. As another example, training of Inception-v4 with large images and 72 GPUs would need $72 * 300 = 21,600$ MB/sec which can be handled by eight Isilon nodes.

For DL workloads and datasets that require a greater ratio of capacity to throughput or capacity to price than what the F800 provides, the Isilon H500 hybrid storage platform may be a good choice. Per node, the H500 delivers about 25-33% of the throughput of the F800 for sequential reads. A cluster may contain exclusively H500 nodes, two tiers including F800 and H500 nodes, or a variety of other combinations to suit your specific workload and datasets. In Table 3, the H500 sizing information is based on general NFS benchmarks. The H500 has not been tested at scale with the specific DL workloads described in this document.

The above data points do not account for node failure, drive failure, network failure, administrative jobs, or any other concurrent workloads supported by Isilon. The ideal ratio of number of GPUs per Isilon node will vary from the above data points based on several factors:

- DL algorithms are diverse and don't necessarily have the same infrastructure demands as the benchmarks listed above.
- Characteristics and size of the data sets will be different from the data sets described in this document and used to generate the above data points.
- Accounting for node/drive/network failures, admin jobs or other workloads accessing Isilon simultaneously.

An understanding of the I/O throughput demanded per GPU for the specific workload and the total storage capacity requirements can help provide better guidance on Isilon node count and configuration. It is recommended to reach out to the Dell EMC account and SME teams to provide this guidance for the specific DL workload, throughput and storage requirements.

Conclusions

This document discussed key features of Isilon that make it a powerful persistent storage for DL solutions. We presented a high-performance architecture for DL by combining NVIDIA DGX-1 servers with Tesla V100 GPUs and Dell EMC Isilon F800 All-Flash NAS storage. This new reference architecture extends the commitment that Dell EMC and NVIDIA have to making AI simple and accessible to every organization with our unmatched set of joint offerings. Together we offer our customers informed choice and flexibility in how they deploy high-performance DL at scale.

To validate this architecture, we ran several image classification benchmarks and reported system performance based on the rate of images processed and throughput profile of IO to disk. We also monitored and reported the CPU, GPU utilization and memory statistics that demonstrated that the server, GPU and memory resources were fully utilized while IO was not fully saturated. Throughout the benchmarks we validated that the Isilon All-Flash F800 storage was able to keep pace and linearly scale performance with NVIDIA's DGX-1 servers.

It is important to point out that DL algorithms have a diverse set of requirements with various compute, memory, IO, and disk capacity profiles. That said, the architecture and the performance data points presented in this whitepaper can be utilized as the starting point for building DL solutions tailored to varied set of resource requirements. More importantly, all the components of this architecture are linearly scalable and can be independently expanded to provide DL solutions that can manage 10s of PBs of data.

While the solution presented here provides several performance data points and speaks to the effectiveness of Isilon in handling large scale DL workloads, there are several other operational benefits of persisting data for DL on Isilon:

- The ability to run AI in-place on data using multi-protocol access
- Enterprise grade features out-of-box
- Seamlessly tier to more cost effective nodes and/or to scale up to 58 PB per cluster

In summary, Isilon-based DL solutions deliver the capacity, performance, and high concurrency to eliminate the IO storage bottlenecks for AI. This provides a rock-solid foundation for large scale, enterprise-grade DL solutions with a future proof scale-out architecture that meets your AI needs of today and scales for the future.

Appendix – System configuration

ISILON

Configuration

The Isilon cluster configuration is simple yet it provides excellent performance.

There are three subnets, each with one pool. One subnet is for 40gige-1 on all nodes, another for 40gige-2, and the final one for mgmt-1. In total, sixteen (16) 40 GbE ports are used in the Isilon cluster.

# isi network interfaces list					
LNN	Name	Status	Owners	IP Addresses	
1	40gige-1	Up	groupnet0.subnet66.pool0	10.55.66.211	
1	40gige-2	Up	groupnet0.subnet5.pool0	10.55.5.211	
1	mgmt-1	Up	groupnet0.subnet0.pool0	10.55.67.211	
2	40gige-1	Up	groupnet0.subnet66.pool0	10.55.66.212	
2	40gige-2	Up	groupnet0.subnet5.pool0	10.55.5.212	
2	mgmt-1	Up	groupnet0.subnet0.pool0	10.55.67.212	
3	40gige-1	Up	groupnet0.subnet66.pool0	10.55.66.213	
3	40gige-2	Up	groupnet0.subnet5.pool0	10.55.5.213	
3	mgmt-1	Up	groupnet0.subnet0.pool0	10.55.67.213	
4	40gige-1	Up	groupnet0.subnet66.pool0	10.55.66.214	
4	40gige-2	Up	groupnet0.subnet5.pool0	10.55.5.214	
4	mgmt-1	Up	groupnet0.subnet0.pool0	10.55.67.214	
5	40gige-1	Up	groupnet0.subnet66.pool0	10.55.66.215	
5	40gige-2	Up	groupnet0.subnet5.pool0	10.55.5.215	
5	mgmt-1	Up	groupnet0.subnet0.pool0	10.55.67.215	
6	40gige-1	Up	groupnet0.subnet66.pool0	10.55.66.216	
6	40gige-2	Up	groupnet0.subnet5.pool0	10.55.5.216	
6	mgmt-1	Up	groupnet0.subnet0.pool0	10.55.67.216	
7	40gige-1	Up	groupnet0.subnet66.pool0	10.55.66.217	
7	40gige-2	Up	groupnet0.subnet5.pool0	10.55.5.217	
7	mgmt-1	Up	groupnet0.subnet0.pool0	10.55.67.217	
8	40gige-1	Up	groupnet0.subnet66.pool0	10.55.66.218	
8	40gige-2	Up	groupnet0.subnet5.pool0	10.55.5.218	
8	mgmt-1	Up	groupnet0.subnet0.pool0	10.55.67.218	

Total: 24

To have high availability in the event of an Isilon node failure, using the dynamic IP allocation method is recommended. This will ensure that all IP addresses in the pool are always available. Refer to the [OneFS Best Practices](#) document for details.

Configuring Automatic Storage Tiering

This section explains one method of configuring storage tiering for a typical DL workload. The various parameters should be changed according to the expected workload.





1. Build an Isilon cluster with two or more node types. For example, F800 all-flash nodes for a hot tier and H500 hybrid nodes for a cold tier.
2. Obtain and install the Isilon SmartPools license.
3. Enable Isilon access time tracking with a precision of 1 day. In the web administration interface, click File System → File System Settings. Alternatively, enter the following in the Isilon CLI:





```
isilon-1# sysctl efs.bam.ptime_enabled=1  
isilon-1# sysctl efs.bam.ptime_grace_period=86400000
```
4. Create a tier named *hot-tier* that includes the F800 node pool. Create a tier named *cold-tier* that includes the H500 node pool.

Storage Pools

Summary
File Pool Policies
SmartPools
CloudPools
SmartPools Settings
CloudPools Settings

Tiers & Node Pools
+ Create a Tier

Name	State	Nodes	Requested Protection	SSD/L3	HDD % Used	SSD % Used	Actions
 cold-tier	Good	5-8	--	L3 Cache	0.1%	--	View / Edit / More
 h500_60tb_3.2tb-ssd_128gb	Good	5-8	+2d:1n	L3 Cache	0.1%	--	View / Edit / More
 hot-tier	Good	1-4	--	Has SSDs	0.0%	0.0%	View / Edit / More
 f800_48tb-ssd_256gb	Good	1-4	+2d:1n	Has SSDs	0.0%	0.0%	View / Edit / More

 = Tier
 = Node Pool
 = Manual Node Pool
 = Unprovisioned Node

- Edit the default file pool policy to change the storage target to *hot-tier*. This will ensure that all files are placed on the F800 nodes unless another file pool policy applies that overrides the storage target.

View Default Policy Details
Help

* = Required field

Policy Information

Policy Name
Default Policy

Description
This policy applies to all files not selected by higher-priority policies.

Select Files to Manage

File Matching Criteria
Matches all files not already matched by any other policies

Apply SmartPools Actions to Selected Files

Storage Settings

Move To Storage Pool or Tier
Storage Target: hot-tier (tier)
Use SSDs for data and metadata (Requires the most SSD space)

Move Snapshots to Storage Pool or Tier
Snapshot Storage Target: hot-tier (tier)
Use SSDs for data and metadata (Requires the most SSD space)

Requested Protection
Using requested protection of the node pool or tier (Suggested)

I/O Optimization Settings

Write Performance
SmartCache is enabled

Data Access Pattern
Optimized for concurrent access

6. Create a new file pool policy to move all files that have not been accessed for 30 days to the H500 (cold) tier.

View File Pool Policy Details

* = Required field

[Help](#) ?

Description

CloudPools State
No access

CloudPools State Details
Policy has no CloudPools actions

*** Policy Name**
Idle data to cold tier

Description
No value

Select Files to Manage

*** File Matching Criteria**
IF
Accessed is older than 1 month ago

Apply SmartPools Actions to Selected Files

Storage Settings

Move To Storage Pool or Tier
Storage Target: cold-tier (tier)
Use SSDs for metadata read acceleration (Recommended)

Move Snapshots to Storage Pool or Tier
Snapshot Storage Target: cold-tier (tier)
Use SSDs for metadata read acceleration (Recommended)

Requested Protection

[I/O Optimization Settings](#)

Testing Automatic Storage Tiering

7. Start the Isilon SmartPools job and wait for it to complete. This will apply the above file pool policy and move all files to the F800 nodes (assuming that all files have been accessed within the last 30 days).

```
isilon-1# isi job jobs start SmartPools
isilon-1# isi status
```
8. Use the `isi get` command to confirm that a file is stored in the hot tier. Note that first number in each element of the inode list (1, 2, and 3 below) is the Isilon node number that contains blocks of the file. For example:

```
isilon-1# isi get -D /ifs/data/imagenet-scratch/tfrecords/train-00000-of-01024
* IFS inode: [ 1,0,1606656:512, 2,2,1372672:512, 3,0,1338880:512 ]
* Disk pools:      policy hot-tier(5) -> data target f800_48tb-ssd_256gb:2(2),
metadata target f800_48tb-ssd_256gb:2(2)
```
9. Use the `ls` command to view the access time.

```
isilon-1# ls -lu /ifs/data/imagenet-scratch/tfrecords/train-00000-of-01024
-rwx----- 1 1000 1000 762460160 Jan 16 19:32 train-00000-of-01024
```
10. Instead of waiting for 30 days, you may manually update the access time of the files using the `touch` command.

```
isilon-1# touch -a -d '2018-01-01T00:00:00' /ifs/data/imagenet-scratch/tfrecords/*
isilon-1# ls -lu /ifs/data/imagenet-scratch/tfrecords/train-00000-of-01024
-rwx----- 1 1000 1000 762460160 Jan 1 2018 train-00000-of-01024
```
11. Start the Isilon SmartPools job and wait for it to complete. This will apply the above file pool policy and move the files to the H500 nodes.

```
isilon-1# isi job jobs start SmartPools
isilon-1# isi status
```
12. Use the `isi get` command to confirm that the file is stored in the cold tier.

```
isilon-1# isi get -D /ifs/data/imagenet-scratch/tfrecords/train-00000-of-01024
* IFS inode: [ 4,0,1061376:512, 5,1,1145344:512, 6,3,914944:512 ]
* Disk pools:      policy cold-tier(9) -> data target h500_60tb_3.2tb-
ssd_128gb:15(15), metadata target h500_60tb_3.2tb-ssd_128gb:15(15)
```

13. Run the training benchmark as described in the previous section. Be sure to run at least 4 epochs so that all TFRecords are accessed. Monitor the Isilon cluster to ensure that disk activity occurs only on the H500 nodes. Note that since the files will be read, this will update the access time to the current time. When the SmartPools job runs next, these files will be moved back to the F800 tier. By default, the SmartPool job runs daily at 22:00.
14. Start the Isilon SmartPools job and wait for it to complete. This will apply the above file pool policy and move the files to the F800 nodes.

```
isilon-1# isi job jobs start SmartPools
isilon-1# isi status
```
15. Use the *isi get* command to confirm that the file is stored in the hot tier.

```
isilon-1# isi get -D /ifs/data/imagenet-scratch/tfrecords/train-00000-of-01024
* IFS inode: [ 1,0,1606656:512, 2,2,1372672:512, 3,0,1338880:512 ]
* Disk pools:          policy hot-tier(5) -> data target f800_48tb-ssd_256gb:2(2),
metadata target f800_48tb-ssd_256gb:2(2)
```

NVIDIA DGX-1

All four NICs on the DGX-1 server should be connected to allow direct communication between the GPUs on different DGX-1 servers, avoiding the PCIe bus and the CPU. This provides the best performance.

Follow the DGX-1 server administrator's manual to configure the NICs to use Ethernet instead of InfiniBand. Use the commands below to confirm that they are configured correctly.

```
root@DGX1-1:~# ibdev2netdev
mlx5_0 port 1 ==> enp5s0 (Up)
mlx5_1 port 1 ==> enp12s0 (Up)
mlx5_2 port 1 ==> enp132s0 (Up)
mlx5_3 port 1 ==> enp139s0 (Up)

root@DGX1-1:~ # ibv_devinfo
hca_id:mlx5_3
transport:          InfiniBand (0)
fw_ver:             12.18.1000
node_guid:           506b:4b03:00f5:cef4
sys_image_guid:      506b:4b03:00f5:cef4
vendor_id:           0x02c9
vendor_part_id:      4115
hw_ver:              0x0
board_id:             MT_2180110032
phys_port_cnt:       1
Device ports:
  port: 1
    state:            PORT_ACTIVE (4)
    max_mtu:           4096 (5)
    active_mtu:        4096 (5)
    sm_lid:            0
    port_lid:          0
    port_lmc:          0x00
    link_layer:        Ethernet
```

Below is an example of the file `/etc/network/interfaces`.

```
auto lo
iface lo inet loopback

auto enp5s0
iface enp5s0 inet static
address 10.55.5.231
netmask 255.255.255.0
mtu 9000

auto enp12s0
iface enp12s0 inet static
```

```

address 10.55.12.232
netmask 255.255.255.0
mtu 9000

auto enp132s0
iface enp132s0 inet static
address 10.55.66.231
netmask 255.255.255.0
gateway 10.55.66.1
dns-nameservers 8.8.8.8
mtu 9000

auto enp139s0
iface enp139s0 inet static
address 10.55.139.234
netmask 255.255.255.0
mtu 9000

```

Below is an example of the file /etc/hosts.

```

127.0.0.1    localhost
127.0.1.1    DGX1-1
10.55.66.231 DGX1-1
10.55.66.235 DGX1-2
10.55.66.239 DGX1-3
10.55.66.243 DGX1-4
10.55.66.247 DGX1-5
10.55.66.251 DGX1-6
10.55.66.155 DGX1-7
10.55.66.159 DGX1-8
10.55.66.163 DGX1-9

```

ARISTA DATA SWITCHES

The 100G switch configuration is also very simple. Although we have defined four layer 3 subnets, we do not need to separate the traffic into different VLANs for a network as small as this one.

All ports to the DGX-1 servers plus the inter-switch ports should be running at 100G (100GBASE-SR4). All ports to the Isilon nodes should be running at 40G (40GBASE-SR4).

Below are the most important configuration commands.

```

transceiver qsfp default-mode 4x10G
spanning-tree mode mstp
interface Port-Channel10
interface Ethernet31/1
channel-group 10 mode active
interface Ethernet32/1
channel-group 10 mode active
ip route 0.0.0.0/0 10.55.67.1
ip routing

```

ISILON VOLUME MOUNTING

Isilon is used for two types of file storage. First, it is used for scripts, binaries, and logs. This requires low bandwidth and must support NFS locks for consistency and proper visibility of changes. This generally uses the default mount options and is mounted with:

```
mount -t nfs 10.55.66.211:/ifs /mnt/isilon
```

Next, there is the data that will be read or written at high speed. To ensure an even balance of traffic across both 40 Gbps interfaces on each Isilon node and across two of the four interfaces on each DGX-1 server, we'll want to carefully create several mounts explicitly to the IP addresses of several Isilon nodes.

As an example, the commands below can be run on each DGX-1 server to mount to each of the 16 Isilon interfaces.


```

mount -t nfs 10.55.66.211:/ifs \
  rsize=524288,wsiz=524288,nolock /mnt/isilon1
mount -t nfs 10.55.66.212:/ifs \
  rsize=524288,wsiz=524288,nolock /mnt/isilon2
...
mount -t nfs 10.55.5.211:/ifs \
  rsize=524288,wsiz=524288,nolock /mnt/isilon9
...
mount -t nfs 10.55.5.218:/ifs \
  rsize=524288,wsiz=524288,nolock /mnt/isilon16

```

Note that since training is purely a read workload, it is safe to add the `nolock` parameter to the mounts that contain the input data. In some cases, this can improve performance.

If you are using the recommend dynamic IP allocation policy in the Isilon IP address pool, all IP addresses will remain accessible, even in the event of an Isilon node failure.

In such a mounting scheme, you must ensure that the application evenly uses the data in the 16 mount points. This can be easily accomplished with a script like `round_robin_mpi.py` which uses the rank of the MPI process to select a mount point.

As a simple but less effective alternative, you may mount `/mnt/isilon1` to a different Isilon interface on each DGX-1 server, but this may result in a sub-optimal balance of traffic. Also, it has been observed that a single NFS mount can read about 2500 MB/sec which is only half of the 40 Gbps front-end Ethernet links on the Isilon nodes. Therefore, it is best to have at least two mounts per Isilon interface.

Note that different mounts do not share the Linux buffer cache. If your dataset is small enough to fit in the DGX-1 server RAM, consider using fewer mounts to allow your entire dataset to be cached.

Appendix – Benchmark setup

This section uses several Bash and Python scripts that are available at

<https://github.com/claudiofahey/tensorflow-benchmark-util>.

CREATING THE IMAGENET TFRECORD DATASETS

To run the TensorFlow Benchmarks suite, the standard 148 GB ImageNet TFRecord dataset was created based on the documentation at <https://github.com/tensorflow/models/tree/master/research/inception#getting-started>.

To create the 22.2 TB dataset, consisting of 150 copies of the above, the script `expand_records.sh` was used. To create the 22.5 TB dataset, consisting of larger resized images, the script `resize_tfrecords.sh` was used to resize the images and then the resulting dataset was copied 13 times using `expand_records2.sh`.

OBTAIN THE TENSORFLOW BENCHMARKS

The TensorFlow Benchmark suite can be obtained from the following Git repository.

```

cd /mnt/isilon/data
git clone https://github.com/claudiofahey/benchmarks tensorflow-benchmarks
cd tensorflow-benchmarks
git checkout 6cdfc0c

```

Note that the commit above differs from the official repository in only one significant way. It removes an unnecessary file name wildcard globbing step which has a huge performance impact when the number of TFRecord files exceeds 10,000. See <https://github.com/claudiofahey/benchmarks/commit/6cdfc0c>.

To allow the TensorFlow Benchmarks to be modified quickly, the scripts are not included (burned in) to the Docker image.

START TENSORFLOW CONTAINERS

In a basic bare-metal deployment of TensorFlow and MPI, all software must be installed on each node. MPI then uses SSH to connect to each node to start the TensorFlow application processes.

In the world of Docker containers, this becomes a bit more complex but significantly easier to manage dependencies. On each DGX-1 server, a single Docker container is launched which has an SSH daemon that listens on the custom port 2222. This Docker container also has TensorFlow, OpenMPI, and NVIDIA libraries and tools. We can then `docker exec`

the `mpirun` command on one of these containers and MPI will connect to the Docker containers on all other DGX-1 servers via SSH on port 2222.

First, a custom Docker image is created using the following Dockerfile.

```
FROM nvcr.io/nvidia/tensorflow:18.09-py3

# Install SSH.
RUN apt-get update && apt-get install -y --no-install-recommends \
  openssh-client \
  openssh-server \
  && \
  rm -rf /var/lib/apt/lists/*
# Configure SSHD for MPI.
RUN mkdir -p /var/run/sshd && \
  mkdir -p /root/.ssh && \
  echo "StrictHostKeyChecking no" >> /etc/ssh/ssh_config && \
  echo "UserKnownHostsFile /dev/null" >> /etc/ssh/ssh_config && \
  sed -i 's/^Port 22/Port 2222/' /etc/ssh/sshd_config && \
  echo "HOST *" >> /root/.ssh/config && \
  echo "PORT 2222" >> /root/.ssh/config && \
  mkdir -p /root/.ssh && \
  ssh-keygen -t rsa -b 4096 -f /root/.ssh/id_rsa -N "" && \
  cp /root/.ssh/id_rsa.pub /root/.ssh/authorized_keys && \
  chmod 700 /root/.ssh && \
  chmod 600 /root/.ssh/*

EXPOSE 2222
```

As you can see, this Dockerfile is based on the [NVIDIA GPU Cloud \(NGC\) TensorFlow image](#). You must create an account on [NGC](#) (there is no charge for this account) and configure Docker to login using this account using the **`docker login nvcr.io`** command.

Once logged in, run the following command to build the Docker image. Replace user with your NGC ID, Docker ID, or **`host:port`** if you are using an on-premise container registry.

```
docker build -t user/tensorflow:18.09-py3-custom.
```

Note that during the build process, a new RSA key pair is randomly generated and stored in the image. This key pair allows containers running this image to SSH into each other. Although this is convenient for a lab environment, a production environment should never store private keys in an image.

Next, you must push this image to a Docker container registry so that it can be pulled from all other DGX-1 servers. You can use an NGC private repository, Docker Hub, or your own private on-premise container registry. To run an unsecure on-premise registry, refer to <https://docs.docker.com/registry/> and <https://docs.docker.com/registry/insecure/>.

Once logged in to your container registry, run the following command to upload the container.

```
docker push user/tensorflow:18.09-py3-custom.
```

You are now ready to start the containers on all DGX-1 servers. Repeat this command for each DGX-1 server, replacing host with the server name.

```
ssh host \
nvidia-docker \
run \
--rm \
--detach \
--privileged \
-v /mnt:/mnt \
--network=host \
--shm-size=1g \
--ulimit memlock=-1 \
--ulimit stack=67108864 \
--name tf \
```

```
user/tensorflow:18.09-py3-custom \
bash -c \
"/usr/sbin/sshd ; sleep infinity"
```

The above command uses **nvidia-docker** which is like the docker command except that it allows the container to directly access the GPUs on the host. The final line starts the SSH daemon and waits forever. At this point, the container can be accessed by MPI via the SSH daemon listening on port 2222.

Choose any one of the DGX-1 servers as the primary node and enter the container by running the following command. This will give you a bash prompt within the container.

```
docker exec -it tf bash
```

Confirm that this container can connect to all other containers via password-less SSH on port 2222.

```
ssh dgx1-1 hostname
ssh dgx1-2 hostname
```

Next, test that MPI can launch processes across all DGX-1 servers.

```
mpirun --allow-run-as-root -np 2 -H dgx1-1 -H dgx1-2 hostname
```

To stop the containers and all processes within them, run the following command on each DGX-1 server

```
docker stop tf
```

Note that the script **start_containers.sh** automates some of these steps.

Appendix – Monitoring Isilon performance

INSIGHTIQ

To monitor and analyze the performance and file system of Isilon storage, the tool InsightIQ can be used. InsightIQ allows a user to monitor and analyze Isilon storage cluster activity using standard reports in the InsightIQ web-based application. The user can customize these reports to provide information about storage cluster hardware, software, and protocol operations. InsightIQ transforms data into visual information that highlights performance outliers, and helps users diagnose bottlenecks and optimize workflows. For more details about InsightIQ, refer to the [Isilon InsightIQ User Guide](#).

ISILON STATISTICS CLI

For a quick way to investigate the performance of an Isilon cluster when InsightIQ is not available, there is a wealth of statistics that are available through the Isilon CLI, which can be accessed using SSH to any Isilon node.

This first command shows the highest level of statistics. Note that units are in bytes/sec so in the example below Node 1 is sending (NetOut) 2.9 GB/sec to clients.

```
isilon-1# isi statistics system --nodes all --format top
Node   CPU    SMB FTP HTTP   NFS HDFS  Total   NetIn NetOut DiskIn DiskOut
All 72.7%  0.0 0.0  0.0 10.0G 0.0 10.0G 304.4M 10.4G 315.4M 10.8G
1 79.2%  0.0 0.0  0.0 2.9G 0.0 2.9G 295.0M 2.9G 70.5M 2.3G
2 80.6%  0.0 0.0  0.0 2.7G 0.0 2.7G 3.2M 2.7G 95.5M 2.8G
3 71.9%  0.0 0.0  0.0 2.4G 0.0 2.4G 3.4M 2.6G 75.5M 2.8G
4 59.2%  0.0 0.0  0.0 2.0G 0.0 2.0G 2.9M 2.1G 73.9M 2.9G
```

The following command shows more details related to NFS. All statistics are aggregated over all nodes.

```
isilon-1 # isi statistics pstat --format top
NFS3 Operations Per Second
access          2.33/s  commit          0.00/s  create          0.75/s
fsinfo          0.00/s  getattr         2.09/s  link            0.00/s
lookup          0.99/s  mkdir           0.00/s  mknod           0.00/s
noop            0.00/s  null            0.00/s  pathconf        0.00/s
read            18865.24/s  readdir         0.00/s  readdirplus     0.00/s
readlink        0.00/s  remove          0.00/s  rename          0.75/s
rmdir           0.00/s  setattr         0.00/s  statfs          0.00/s
symlink         0.00/s  write           0.75/s
Total           18872.91/s
```

CPU Utilization	
user	1.4%
system	72.5%
idle	26.1%

OneFS Stats	
In	73.81 kB/s
Out	8.96 GB/s
Total	8.96 GB/s

Network Input	
MB/s	12.64
Pkt/s	150368.20
Errors/s	0.00

Network Output	
MB/s	9868.11
Pkt/s	6787182.27
Errors/s	0.00

Disk I/O	
Disk	272334.03 iops
Read	11.73 GB/s
Write	99.63 MB/s

Appendix – Isilon performance testing with iPerf and FIO

iPerf is a tool for active measurements of the maximum achievable bandwidth on IP networks. It can be used to validate the throughput of the IP network path from an Isilon node to a compute node NIC. It can easily be scripted to run concurrently to allow all nodes to send or receive traffic.

FIO is a disk benchmark tool for Linux. It can be used to easily and quickly produce a storage workload that is nearly identical to the TensorFlow benchmark used in this document.

This section shows how to use iPerf and FIO.

To begin, on your head node (or first compute node), create a hosts file containing one host name (or IP address) per client. For example:

```
server1.example.com
server2.example.com
```

IPERF

Install iPerf on all compute nodes. Note that iPerf is already installed on all Isilon nodes. All versions should match.

```
cat hosts | xargs -i -P 0 ssh root@{} yum -y install \
iperf-2.0.4-1.el7.rf.x86_64.rpm
cat hosts | xargs -i -P 0 ssh root@{} pkill -9 iperf
cat hosts | xargs -i ssh root@{} "iperf --server --daemon \
> /dev/null 2>&1 &"
client_opts="--t 300 --len 65536 --parallel 2"
ssh root@isilon-1.example.com iperf -c server1.example.com ${client_opts} &
ssh root@isilon-2.example.com iperf -c server2.example.com ${client_opts} &
wait
```

To view the aggregate bandwidth, run the following on any Isilon node.

```
isi statistics system --format top --nodes all
```

FIO

The procedure below shows how to use FIO to benchmark NFS I/O from multiple clients concurrently. Mount to Isilon. Each client will mount to a different Isilon IP address.

```
cat hosts | xargs -i -P 0 ssh root@{} mkdir -p /mnt/isilon
cat hosts | xargs -i -P 0 ssh root@{} umount /mnt/isilon
ssh root@server1.example.com mount -t nfs 10.1.1.1:/ifs \
-o rsize=524288,wsiz=524288 /mnt/isilon
ssh root@server2.example.com mount -t nfs 10.1.1.2:/ifs \
-o rsize=524288,wsiz=524288 /mnt/isilon
```

Install FIO servers.

```
cat hosts | xargs -i -P 0 ssh root@{} yum -y install epel-release
cat hosts | xargs -i -P 0 ssh root@{} yum -y install fio
cat hosts | xargs -i ssh root@{} fio -version
```

Start FIO servers.

```
cat hosts | xargs -i ssh root@{} pkill fio
cat hosts | xargs -i ssh root@{} fio --server --daemonize=/tmp/fio.pid
```

Create a FIO job file shown below, named `fiol.job`. Set `numjobs` to the number of GPUs per host. This job file performs I/O that is like the TensorFlow CNN benchmark. It creates 30 files per GPU and then reads them sequentially concurrently.

```
[global]
name=job1
directory=/mnt/isilon/tmp/fio
time_based=1
runtime=600
ramp_time=60
ioengine=libaio
numjobs=8
create_serialize=0
iodepth=32
kb_base=1000
[job1]
rw=read
nrfiles=30
size=64GiB
bs=1024KiB
direct=1
sync=0
rate_iops=83
```

Run the FIO job.

```
mkdir -p /mnt/isilon/tmp/fio
fio --client=hosts fiol.job
```

Appendix – Collecting system metrics with ELK

While performing benchmarks on a distributed system such as this, it is valuable to collect and display all relevant metrics such as CPU utilization, GPU utilization, memory usage, network I/O, and more. This can be performed with a variety of tools. For this document, data was collected using Elasticsearch, Kibana, Metricbeat, and [NVIDIA GPU Beat](#). This suite of tools from [Elastic](#) is often referred to as the ELK stack, however, Logstash was not used.

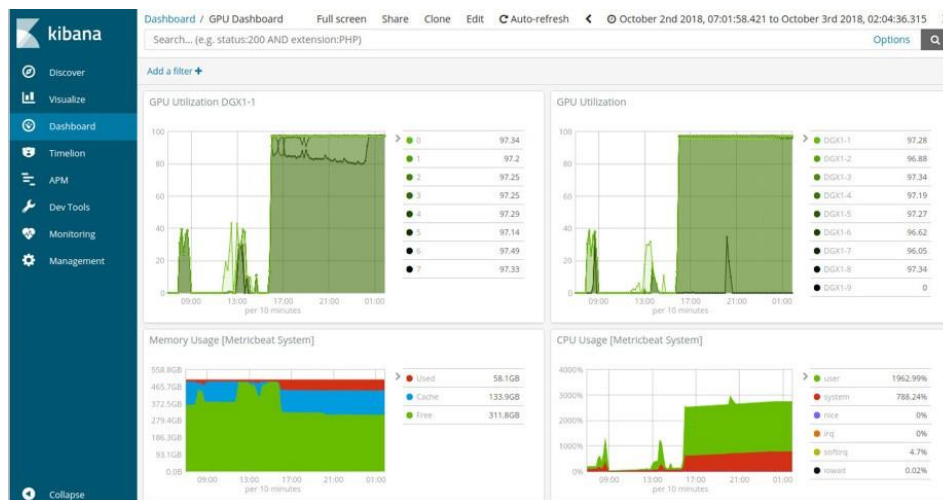


Figure 18: A custom Kibana dashboard showing GPU, CPU, and memory utilization. Data is collected by Metricbeat and NVIDIA GPU Beat.

A single-node deployment of Elasticsearch and Kibana can be brought up in a few minutes using the Docker Compose file `elk/docker-compose.yml` in the `tensorflow-benchmark-util` repository.

```
cd tensorflow-benchmark-util/elk
docker-compose up -d
```

Next, Metricbeat must be started on all DGX-1 servers. Edit the file `metricbeat/hosts` to contain the FQDN for all DGX-1 servers. Edit the file `metricbeat/etc/metricbeat/metricbeat.yml` to contain the IP address or FQDN for the server running Elasticsearch. Then run the `script start_metricbeats.sh`. This will SSH into each host and start the Metricbeat container.

For NVIDIA GPU Beat, repeat the above steps but in the `nvidiagpubeat` directory. Note that the version of NVIDIA GPU Beat dated September 25, 2018 has a resource leak that causes resource exhaustion after about 24 hours. Be sure to restart the containers every few hours to avoid this.

Open your browser to port 5601 of the server running Kibana. Import the customized Kibana dashboards and visualizations in `elk/kibana.json`. Finally, open the dashboard named **GPU Dashboard**.

Appendix – Tips

1. When MPI TensorFlow applications are terminated by the user with Control-C or by application errors, some processes may not terminate completely. This may result in subsequent executions getting a GPU out-of-memory (OOM) error. This condition can be confirmed by running `nvidia-smi` and checking the used memory. To fix this, simply stop and restart the TensorFlow containers.
2. Avoid significant amounts of network traffic on the inter-switch links. Since there are no VLANs set up, this can be done accidentally if NICs are not assigned to the proper IP subnets. Check the interface packet counters on the switches before and after a training run to confirm this

References

Name	Link
Arista 7060x Series	https://www.arista.com/en/products/7060x-series
Elasticsearch, Kibana	https://www.elastic.co/
Horovod	https://github.com/uber/horovod
ImageNet	http://www.image-net.org/challenges/LSVRC/2012/
Isilon InsightIQ	https://www.emc.com/collateral/TechnicalDocument/docu65870.pdf
Isilon Storage Tiering	https://www.dell EMC.com/resources/en-us/asset/white-papers/products/storage/h8321-wp-smartpools-storage-tiering.pdf
Isilon OneFS Best Practices	https://www.emc.com/collateral/white-papers/h16857-wp-onefs-best-practices.pdf
Isilon OneFS Technical Overview	https://www.emc.com/collateral/hardware/white-papers/h10719-isilon-onefs-technical-overview-wp.pdf
MPI All Reduce	http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/
NVIDIA DGX-1 server	https://www.nvidia.com/dgx1
NVIDIA GPU Cloud (NGC) TensorFlow image	https://ngc.nvidia.com/registry/nvidia-tensorflow
NVIDIA mixed precision training	https://docs.NVIDIA.com/deeplearning/sdk/mixed-precision-training/index.html
RAPIDS	https://rapids.ai/
TensorFlow	https://github.com/tensorflow/tensorflow
TensorFlow Benchmarks	https://www.tensorflow.org/performance/benchmarks
TensorFlow Benchmark Utilities	https://github.com/claudiofahey/tensorflow-benchmark-util
TensorFlow Inception	https://github.com/tensorflow/models/tree/master/research/inception