Characterizing Deep-Learning I/O Workloads in TensorFlow

Steven W. D. Chien[1], Stefano Markidis[1], Chaitanya Prasad Sishtla[1],
Luis Santos[2], Pawel Herman[1], Sai Narasimhamurthy[3], Erwin Laure[1]
[1] *KTH Royal Institute of Technology*, Sweden
[2] *Instituto Superior Técnico*, Portugal
[3] *Seagate Systems UK*, UK

Abstract

The performance of Deep-Learning (DL) computing frameworks rely on the performance of data ingestion and checkpointing. In fact, during the training, a considerable high number of relatively small files are first loaded and pre-processed on CPUs and then moved to accelerator for computation. In addition, checkpointing and restart operations are carried out to allow DL computing frameworks to restart quickly from a checkpoint. Because of this, I/O affects the performance of DL applications.

In this work, we characterize the I/O performance and scaling of TensorFlow, an open-source programming framework developed by Google and specifically designed for solving DL problems. To measure TensorFlow I/O performance, we first design a micro-benchmark to measure TensorFlow reads, and then use a TensorFlow mini-application based on AlexNet to measure the performance cost of I/O and checkpointing in TensorFlow. To improve the checkpointing performance, we design and implement a burst buffer.

We find that increasing the number of threads increases TensorFlow bandwidth by a maximum of 2.3× and 7.8× on our benchmark environments. The use of the tensorFlow prefetcher results in a complete overlap of computation on accelerator and input pipeline on CPU eliminating the effective cost of I/O on the overall performance. The use of a burst buffer to checkpoint to a fast small capacity storage and copy asynchronously the checkpoints to a slower large capacity storage resulted in a performance improvement of 2.6× with respect to checkpointing directly to slower storage on our benchmark environment.

Index Terms

Parallel I/O, Input Pipeline, Deep Learning, TensorFlow

I. Introduction

Deep-learning (DL) frameworks are increasingly taking advantage of HPC systems with accelerators to speed-up training of large and complex neural networks. During a training process, connection weights in a network are iteratively adjusted to minimize a loss function which reflects the error of a network. This process involves feeding the network with large amount of training samples which implies computationally intensive matrix multiplications. For this reason, accelerators such as GPUs are commonly used. Similar to many HPC applications, I/O is one of the major performance bottlenecks as training process requires ingestion of large amount of training samples. Training datasets typically contain several Gigabytes and up to several Terabytes of data. For instance, the YouTube-8M Kaggle challenge provides 1.71 TB of frame-level data[1]. However, the size of individual training samples is usually limited. The Large Scale Visual Recognition Challenge 2013 (LSVRC2013) [1] training dataset contains 395,909 images, yet the average image resolution is only 482×415 pixels[2]. With ever faster accelerators for computation [2] [3], it is important to maintain a high data ingestion rate such that devices will not be kept idle due to slow I/O.

TensorFlow is among the most successful open-source frameworks for neural networks development [4]. The TensorFlow GitHub repository[3] shows more than 1,600 active contributors with an average of more than a thousand commits per month, making it one of the fastest growing computing framework. TensorFlow is accelerator-centric: neural networks are expressed as computation graphs where nodes represent operations and edges represent data consumed or produced by operations. These operations can be easily offloaded to different specialized accelerators for DL workload [2] [3] and computation will be completely performed on these devices. CPUs on the host system, on the other hand handle all I/O operations such as reading of training or validation samples from files to memory and checkpointing for restart.

TensorFlow attempts to alleviate the I/O bottleneck by performing independent reading of these training samples in parallel through an input pipeline. This reduces idling time of accelerators due to I/O through better utilization of bandwidth thus increases ingestion rate. For large systems with a large number of computing nodes, failure is becoming increasingly common and failure recovery must be properly handled [5]. DL training is no exception. During a long running training process, snapshots of weight variables can be taken periodically to allow later restarting of training in case of system failure. TensorFlow implements this as a checkpoint feature and saves the state of variables of a computation graph to disk. The file size of these

---

[1] https://www.kaggle.com/c/youtube8m
[2] http://www.image-net.org/challenges/LSVRC/2013/
[3] https://github.com/tensorflow/tensorflow

checkpoints depend on the network in use and each checkpoint can easily reach hundreds of Megabytes size in the case of most modern architectures.

For all these reasons, it is important to characterize the impact of I/O on training performance and optimize accordingly. In particular, we study the performance of the I/O interface provided by TensorFlow: Dataset API. Furthermore, we study how emerging storage technologies such as non-volatile memory can help eliminate bottleneck due to slow I/O. We first develop a benchmark tool to measure raw performance of the data I/O pipeline. We proceed to develop a mini-application for image classification with AlexNet, which illustrates typical computation of a simple neural network with a data ingestion pipeline. Finally, we extend the mini application with checkpointing capability and implement a proof-of-concept burst buffer for temporary staging of checkpoints. With these tools, we evaluate experimentally the I/O performance of TensorFlow.

The contributions of this work are:

- We design and implement a TensorFlow I/O benchmark, mimicking the STREAM benchmark, to evaluate the bandwidth of TensorFlow I/O on HPC systems.
- We design and implement a TensorFlow mini-application to mimic the typical input data pipeline and computation in a DL application. We use an AlexNet network with batches of images as input and estimate the impact of the I/O pipeline.
- We evaluate the impact of multi-threading, prefetching and caching on the performance of the micro-benchmark and mini-application.
- We extend the mini-application with checkpointing capability and implement a proof-of-concept burst buffer where checkpoints are staged in non-volatile memory and evaluate checkpointing performance comparing to difference storage devices.

The paper is organized as follows. Section II provides an overview of the I/O system in TensorFlow. We describe the design and implementation of the benchmark and mini-application in Section III. Results are presented in Section V. Finally, Section VII concludes the paper, summarizes and discusses the results.

II. Overview of TensorFlow I/O

TensorFlow is an open-source programming framework developed by Google specifically designed for solving deep-learning problems. Application developers can implement neural networks on TensorFlow with a variety of APIs of different language bindings, such as Java, Go and Rust. The most complete API is the Python API and is used by the majority of TensorFlow users. For this reason, Python is the language of choice for the tools developed in this paper.

Operations in TensorFlow are powered by the TensorFlow runtime which is written in C++. The runtime provides mechanisms for managing and scheduling data movement across hosts and devices, offloading computation to accelerators, memory allocation and most importantly for this paper: I/O operations. Fig. 1 illustrates the design of TensorFlow file system module. Currently, TensorFlow 1.10 supports four underlying file systems, namely: traditional POSIX file system; S3, a cloud based object storage system by Amazon Web Service; Google Cloud Storage, another cloud based file system by Google; and Hadoop Distributed File System (HDFS). Different I/O adapters are developed such that users can interact with these file systems through the TensorFlow API. These adapters inherit the same interface for easy switching and extension from POSIX I/O to other I/O systems. For example, the POSIX adapter implements writing of file through fwrite() and reading through pread(). From a user's point of view, the switching of a file system can be easily done by substituting the prefix of a file path.

TensorFlow also provides distributed runtime support which enables applications to run on clusters and HPC systems. Communication in distributed TensorFlow is based on gRPC, an open-source RPC (Remote Procedure Call) library developed by Google. However modules are developed to support different transport layers such as InfiniBand RDMA (Remote Direct Memory Access) and MPI.

A. Input Pipeline

TensorFlow neural network training follows a data-driven programming model which is similar to that of a producer-consumer model. As soon as the accelerators complete a training step (consumer), it ingests training samples from the input pipeline for computation (producer). The input pipeline typically performs actual I/O, decoding and pre-processing. One example would be the ingestion of JPEG images where the images are read from files. These image data are decoded into tensors and pre-processing e.g. resizing, is applied to suit the input size of a given neural network.

The TensorFlow Dataset implementation attempts to optimize this process by overlapping the input pipeline with the computation pipeline. A common approach in neural network training is the use of mini-batches. Since a training set typically does not fit into GPU memory, training is instead done in batches per iteration. When the computation pipeline becomes ready for the next iteration, one new batch is ingested from the input pipeline to the computation pipeline. By overlapping the preparation of batches with the computation, new batches can be buffered and become ready for ingestion as soon as the computation pipeline becomes available. More specifically, the input pipeline can proceed to prepare the next batch as soon as the current batch is completed, rather than waiting for the computation pipeline to request for a new one. Furthermore, since the input pipeline operates on CPU and computation is typically done on GPU, this allows efficient utilization of both devices on a machine.
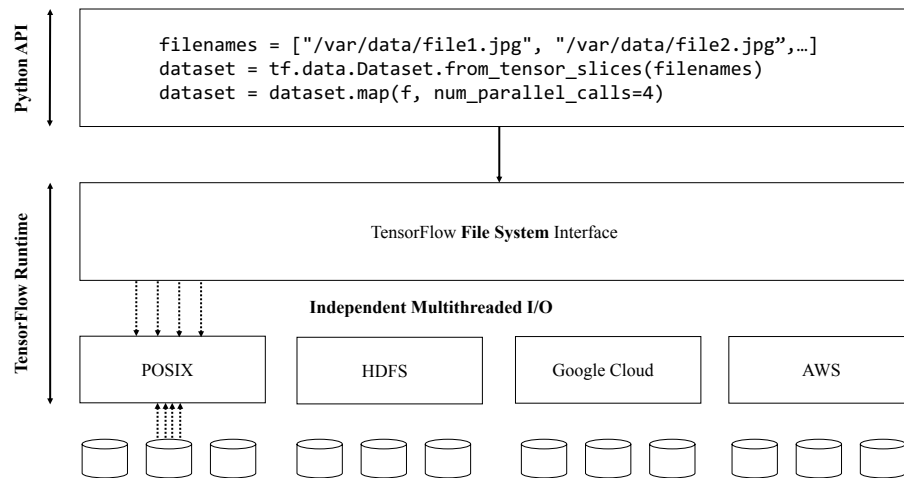
Fig. 1. TensorFlow is able to interact with different types of storage systems, including POSIX file system, Hadoop Distributed File System, Google Cloud Storage and Amazon S3 through file system adapters. Through parallel mapping or parallel interleaving it is possible to read files concurrently.
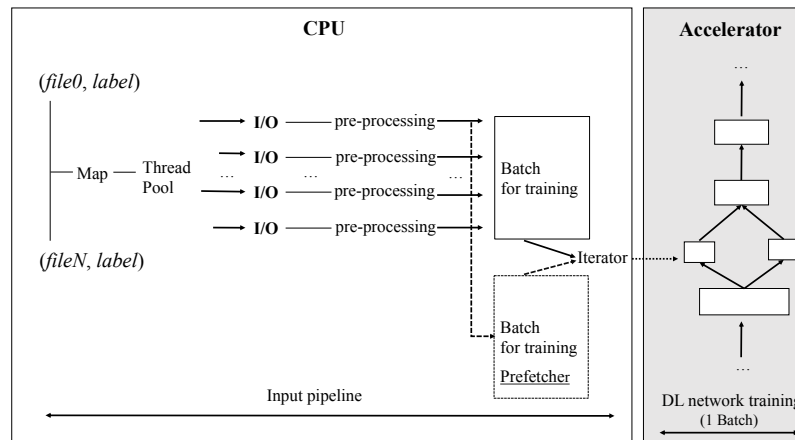

Fig. 2. A simplified TensorFlow input pipeline. In a typical use case, a list of file paths and their labels form a source dataset. The file names are sent to a transformation function through map to perform actual I/O and decoding. Finally these transformed data from mapping are collected into a batch. Optionally a prefetch operation can be used to cache one or more batches in memory such that they can be sent through the iterator immediately when requested.

1) Parallel I/O: The reading and pre-processing is applied to individual files with individual outputs. In other words, this is an embarrassingly parallel process. Therefore, it is possible to parallelize the I/O of many files by mapping a list of file names for transformation in form of tf.dataset.map(). Threads will be spawned by the runtime to execute the I/O function and the number of threads used by the map can be specified with num_parallel_calls argument. The result from mapping will be collected by the down stream operation. The user can optionally apply shuffling of the dataset by tf.dataset.shuffle() and specify the size of the shuffle buffer. The tf.dataset.batch() can be applied by specifying the batch size. This operation accumulates the number of training samples from upstream operation and form a batch.

2) Prefetching: In order to overlap the computation pipeline and input pipeline, tf.dataset.prefetch() can be used to specify how many batches of data to be buffered. This ensures that there will be a specified number of batches ready for consumption. tf.data.Dataset.prefetch() transformation is used to prefetch data from the previous pipeline operation. When applied after tf.dataset.batch(), this allows buffering of one or more batches which are ready for direct consumption. The TensorFlow runtime implements a prefetcher as a background thread and a consumption function. The thread maintains a buffer which stores elements that are prefetched from the upstream operation. The buffer uses a double ended queue implementation from standard library. The thread itself contains an infinite loop which waits for a condition variable. When a Tensor is consumed from the buffer using a consumer function, the thread is notified through the condition variable and wakes up to fetch another element from upstream.

A simplified input and training pipeline is presented in Fig. 2. In the tf.data API, tf.data.Dataset represents a container of

Tensor objects. The input pipeline itself is constructed with a list of operations provided by the tf.data API. For instance, a source dataset can be a vector of file name of samples and their respective labels for a classification problem. The dataset can feed one slice of the vector into the pipeline with Dataset.from_tensor_slices(). These samples can be read and decoded for the next stage of the pipeline by applying a transformation function. This function obtains the slice of information given and returns another slice of transformed data to the downstream operation. One example is to read a file through the TensorFlow file system API, decode data and apply pre-processing. When given the file name of a JPEG image, the transformation function reads it with tf.read_file() and decodes with tf.image.decode_jpeg(), then converts the decoded image into a tensor with tf.image.convert_image_dtype() and finally applies resizing with tf.image.resize_images(). The function returns the transformed image in form of tensor to the next operation, which accumulates to a batch of data. One or more batches are prefetched and buffered by a prefetcher and finally consumed by an iterator.

B. Checkpointing and Restart

TensorFlow uses the tf.train.Saver() to checkpoint training sessions to disk and allows restarting computation from a given checkpoint. If no argument is provided to tf.train.Saver(), all variables in the computation graph are checkpointed. Each tensor is saved under the name that was given when the tensor was created. When a checkpoint is being saved, the checkpoint saver generates three files: a metadata file with suffix .meta which stores the structure of the graph, an index file with suffix .index which describes the tensors of a graph and a data file with suffix .data which stores actual data in variables. When a checkpoint is being restored, the structure of the graph is first restored followed by the restoration of variable contents into the graph. tf.train.Saver() is responsible both for checkpointing and restoring the training state. The Saver can additionally perform cleanup of old checkpoints. By default only the first five checkpoints are retained.

III. METHODOLOGY

In this section, we present the design and development of benchmark and mini-applications to benchmark I/O performance of TensorFlow.

A. TensorFlow I/O Micro-Benchmark

We develop a STREAM-like benchmark application [6] to measure the performance of I/O with TensorFlow dataset API. We achieve this by loading a set of images from storage, and perform simple pre-processing. When the application is initialized, a number of user configurable I/O parameters are evaluated. These include batch size and the number of current I/O threads to use. A file containing a list of paths to images to be consumed is also provided which will be loaded as a Python list. This list becomes the source element.

We first apply shuffle transformation to the list to randomize the order of elements, then immediately apply map transformation, where slices of the image path list will be given to a transformation function. The number of threads to perform concurrent mapping is set according to the parameter provided by the user. This function performs I/O with tf.read() and data from read is immediately passed to tf.image.decode_jpeg() followed by a resize operation tf.image.resize_images(). A tf.contrib.data.ignore_errors() is applied after mapping to avoid exceptions in the mapped function from terminating all execution. This is useful in processing large amount of data where data completeness is not guaranteed. The tensors returned by this transformation through mapping will be collected by a batch transformation where tensors will be accumulated until a batch is formed. The batch size is set according to user configuration. Finally an iterator is attached to the end of the dataset. We directly invoke the iterator of the dataset in a session to draw new batches without passing them through any computation phase. The iterator is continuously invoked until a defined number of batches is consumed. By measuring the time between the first invocation and last invocation it is possible to compute the ingestion rate with the number of images passing through the iterator and the number of seconds used. We also perform another execution with a extremely simplified input pipeline where all steps are omitted in the mapped function except tf.read() to isolate effect due to preprocessing.

B. TensorFlow Mini-Application: AlexNet

We develop a mini-application using the AlexNet DL network architecture to mimic the typical DL I/O workload. We choose AlexNet because it is easy to implement (roughly 200 lines of code) yet it retains basic characteristics of DL networks. AlexNet was developed by Alex Krizhevsky (hence the name), and was winner of the ImageNet challenge in 2012. AlexNet [7] consists of five convolution layers, three max pooling layers and three fully connected layers. The network uses the ReLU (Rectified Linear Unit) activation function and the convolution layers contribute to feature extraction while the fully connected layers implement the classifiers.

During initialization, the user provides a list of paths to images and labels, batch size, total number of batches per epoch, number of I/O threads to use and indicates if prefetch is to be enabled or not. The input pipeline consists of three stages. Similar to the micro-benchmark application, we read the list of image paths and their classes. The labels are converted to one-hot vectors to align the number of outputs of the classifier. We apply shuffle transformation to the list of samples to randomize
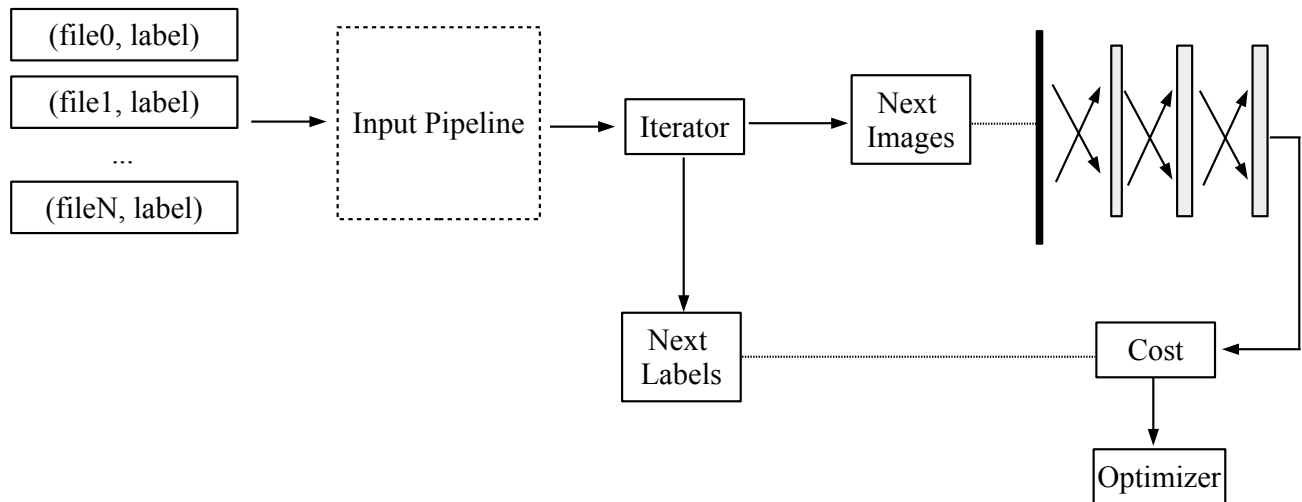
Fig. 3. Workflow of AlexNet mini-application. Image files are ingested to form batches of image data and labels through an input pipeline. The neural network takes a batch of image data and perform classification through AlexNet. The output of the classifier is compared with the ground truth (labels) and a cost value which represents the error of classification is derived. The cost value is passed to a TensorFlow optimizer, which base on a minimization target adjusts variables in the network with DL methods. The mini-application executes one training step and consume one batch of images by calling the optimizer.

order of elements. We also perform map transformation where paths will be sent to a transformation function. Again, the number of threads used to perform concurrent mapping is set according to the parameter provided by the user. This function performs I/O with tf.read() and data from read is immediately passed to tf.image.decode_png() followed by a resize operation tf.image.resize_images(). tf.image.decode_png() supports both decoding of JPEG and PNG images. In our implementation, input images are resized to dimensions 224×224 with three channels. The tensors returned by this transformation through mapping will be collected by a batch transformation where tensors will be accumulated until a batch is formed. The batch size is set according to user configuration. Finally an iterator is attached to the end of the dataset. We derive a cost value from the network to reflect the error from classification and pass it to the Adam Optimizer provided by TensorFlow. In each invocation, a batch is drawn from the dataset through the neural network to compute the cost value and the optimizer minimizes the cost by modifying weights variables of the network accordingly. A simplified work flow of the mini-application is presented in Fig. 3.

C. Checkpoint and Burst Buffer

We extend the AlexNet mini-application to add checkpointing capability with tf.train.Saver(). A checkpoint prefix and the interval between checkpoints are defined by the user. A checkpoint prefix refers to the path and prefix name used for the checkpoint file. For every defined number of iterations, the checkpoint saver will be called to take a snapshot of all variables. In order to ensure that the checkpoints are safely written to disk, we perform disk synchronization of disk with syncfs() immediately after Saver returns.

In order to take a snapshot of all variables, training process must be paused in favor of variable saving process. A simple network such as AlexNet in our mini-application can produce up to several hundreds of megabytes of snapshot files. The process of writing these snapshot files seriously hamper training performance. For this reason, we introduce a simple proof-of-concept burst buffer implementation in our extended mini-application. In particular, we take advantage of non-volatile memory storage device for fast writing of snapshot files.

Non-volatile memory is an emerging storage technology which features access speed while being persistent [8]. For this reason, it has been increasingly adopted as a burst buffer, which acts as an intermediary between slower but larger; and faster but smaller storage devices to absorb bursty I/O which affect application performance. Burst buffers are increasingly adopted in large scale cluster systems [9]. Two of the most famous burst buffer solutions are DataWarp [10] by Cray and IME [11] by DNN. In DataWarp, user requests resources from the system through job scripts. After allocation the system can be interacted through POSIX APIs. Data movement can be requested explicitly when used as a scratch drive or done implicitly when used as a cached instance. IME is designed as an intermediate storage layer between a compute system and an external storage system, connected through switches. The storage system and IME uses NVM-based devices to absorb I/O traffic between the external storage system and compute system.

Checkpointing is a suitable use case for using burst buffer as staging [12]. Since I/O due to checkpointing is bursty, by absorbing the staging with a fast storage it is possible to minimize disruption to execution. In our implementation, when the

checkpoint saver is called, a checkpoint is created and synchronized to a fast non-volatile memory device. At the same time a process is spawned in background to copy the just created files to hard disk for storage. Since the checkpoint was already written to persistent memory, it is possible to continue training without disruption.

IV. EXPERIMENTAL ENVIRONMENT

The first benchmark environment is a workstation with local storage two Hard Disk Drives (HDD) and one Solid-State Drive (SSD) and one Intel Optane card (Opt.). The second benchmark environment is a supercomputer at KTH Royal Institute of Technology, with network-based storage provided by a Lustre parallel file system.

The two systems have the following characteristics:

- **Blackdog** is a workstation with an eight core Xeon E5- 2609v2 processor running at 2.5GHz and an NVIDIA Quadro K4000. The workstation is equipped with a total of 72GB DRAM. The storage consists of two 4TB HDD (non-RAID) and a 250GB SSD (Samsung 850 EVO). The machine also has one Intel Optane SSD 900p with 480GB on PCIe. The file system used is ext4 and OS is Ubuntu Server 16.04 with Kernel 4.4.0-112-generic. TensorFlow version 1.10 was compiled with gcc v7.3.0 and NVIDIA CUDA 9.2.
- **Tegner** is a GPU cluster at KTH PDC. Each node has two Intel E5-2690v3 Haswell processors, 512 GB of RAM and one Nvidia Quadro K420 or K80 GPU and provides 1GB and 24GB of ram respectively. The parallel file system used is Lustre and operating system is CentOS 7.4.1708. The nodes are connected by EDR InfiniBand network. We compiled TensorFlow 1.10 with support of Python 3.6, NVIDIA CUDA, OpenMPI and InfiniBand. The versions used are CUDA 9.1, cuDNN 7.0.5, OpenMPI 3.0 and the compiler used is GCC 6.2.0.

TABLE I
IOR BENCHMARK RESULTS

Platform	Devices	Max Read	Max Write
Blackdog	HDD	163.00 MB/sec	133.14 MB/sec
	SSD	280.55 MB/sec	195.05 MB/sec
	Intel Optane	1603.06 MB/sec	511.78 MB/sec
Tegner	Lustre	1968.618 MB/sec	991.914 MB/sec

Benchmarking I/O is a notoriously difficult task. The reason is that the Operating System often caches files which are previously opened in memory, thus the real cost of I/O cannot be assessed. For this reason, we used a small C program which passes a POSIX advice with posix_advice() for the files used with POSIX_FADV_DONTNEED, which states that the file is no longer needed. Furthermore, on our local workstation where we have roots privileges we use # echo 1 > /proc/sys/vm/drop_caches to instruct the operating system to drop caches. We also monitor I/O activities using iosnoop and dstat tools to ensure I/O activities actually occurred. On the clusters, we only execute the small C program to pass POSIX advice as access to root privilege is not possible. In order to establish a upper bound of I/O performance on our test platform, we summarize IOR benchmarking results in Table I. We obtain this result by reading from and writing to a file with 5GB on the respective devices for six times and obtain the bandwidth. The execution run is for warm up and the result is discarded. The median bandwidth is reported here. Again caches are dropped before the tests.

A. TensorFlow I/O Micro-Benchmark

We execute our micro-benchmark tool with a subset of images from ImageNet totaling 16,384 JPEG images with median image size 112KB. We choose the particular batch size 64 to present the test results. Therefore, we invoke the iterator through a TensorFlow session 256 times to fully consume the images. We perform strong scaling by varying the number of threads used for map transformation, which are responsible for performing individual I/O of images. The number of threads used are one, two, four and eight and we report the bandwidth with number of images processed per second. On our local workstation we repeat the tests with sample images placed on different devices: HDD, SSD and Intel Optane. On Tegner, we perform the tests with sample images placed on the parallel file system Lustre. We execute each test six times, with the first test being warm up and report the median bandwidth value. Error bars are not shown as bandwidth measurements vary less than 1% and 4% with the respect to median bandwidth on the Blackdog and Tegner respectively.

B. TensorFlow Mini-Application: AlexNet

We perform the tests with Caltech 101 data set, a small scale data set with 9,144 images of 101 classes plus one extra *Google background* class. The median image size is approximately 12kB while the average size is around 14kB. We report our results based on batch size 64 where our GPU is well utilized. Since 9,144 is not divisible by 64, in each test we execute 142 iterations which in total process 9,088 images. In order words, we only execute one epoch of training. The reason is
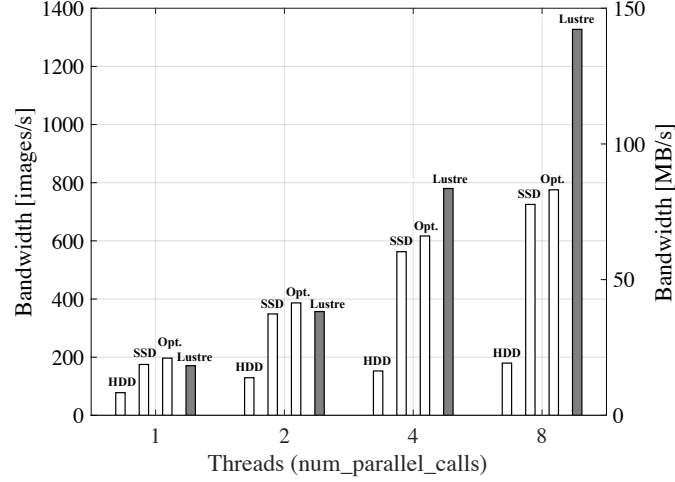
Fig. 4. Median bandwidth in images per second from strong scaling tests with micro-benchmark application using a typical input processing pipeline for images. The higher the rate the better. The bandwidth is translated to MB/s scale on the right. Tests that are performed on Blackdog are colored in white and the test conducted on Tegner is performed in gray.

that after the first epoch all samples will be seen by the Operating System and will potentially be cached in memory, thus avoiding actual I/O operations. We vary the number of threads used for mapping between one, two, four and eight and report the runtime in seconds. We additionally test for the effect with and without prefetching by setting the number of batches to prefetch to one or zero batch. Similar to the tests with micro-benchmark tool, we repeat the tests when sample images are placed on different storage devices: HDD, SSD and Intel Optane on our local workstation. On Tegner, the images are placed on Lustre. We execute each test six times, with the first test being warm up and report the median run time. Error bars are not shown as execution time measurements vary less than 1% and 6% with the respect to median bandwidth on the Blackdog and Tegner respectively.

In order to understand the effect of prefetching, we use *dstat*[4] to perform tracing of system I/O activities. dstat is a system resources monitoring tool which is able to produce statistics on different system activities. Statistics are sampled once per second and can be reported as a comma separated values file (CSV). In particular, we trace for disk activities on devices where samples are placed. We report disk activities from the first and the last invocation of the iterator.

C. Checkpoint and Burst Buffer

We execute the same tests as used with our mini-application, except that we only perform the first 100 iterations instead of full 142 iterations. We perform a checkpoint every 20 iteration and call for file system synchronization to ensure the checkpoint files are safely written to disk. In this particular tests, we use batch size 64 where images are placed on SSD and prefetch is enabled. We test for checkpoint to HDD, SSD, Intel Optane, and Intel Optane as burst buffer. The total runtime and median checkpoint in seconds are reported in addition to the disk activities measured using *dstat*.

V. RESULTS

In this section, we present the results of parametric study described in Section IV. The tests are performed on Blackdog and Tegner.

A. TensorFlow I/O Micro-Benchmark

Threading is an effective way of increasing bandwidth utilization for file ingestion. Fig. 4 shows that by increasing the number of threads from one to two it effectively almost double the bandwidth on most devices. The effect is particularly visible on fast storage devices such as SSD and Optane. On HDD the effect of scaling is also visible but soon flattens when the number of threads used exceeds four. A not very surprising result is that scaling on Tegner with Lustre file system shows the best scalability when the number of threads increases as the files are likely shared on different object store targets, thus allowing better bandwidth utilization. The bandwidth when comparing to I/O performance measured with IOR is unfavorable. One reason is due to preprocessing functions such as decoding used inside the mapped function which uses computation. Fig. 5 shows the performance of an input pipeline without any preprocessing, where the only operation is to read the file with the given path.
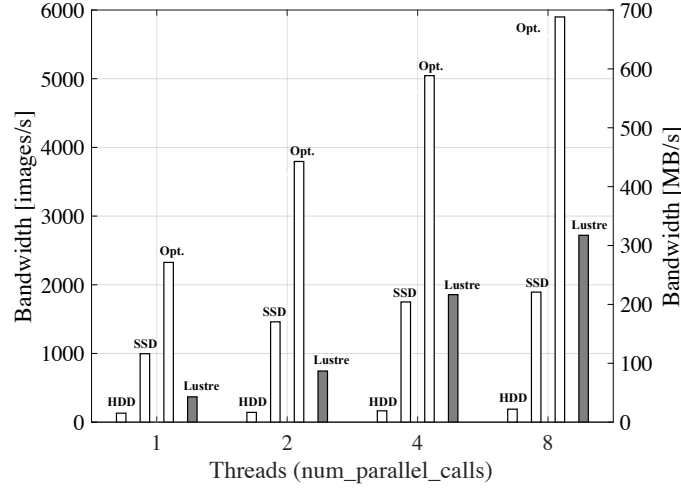
[4]http://dag.wiee.rs/home-made/dstat/

Fig. 5. Median bandwidth in images per second when the input pipeline of the micro-benchmark application contains simply a read file operation without further preprocessing. The bandwidth is translated to MB/s scale on the right. Tests that are performed on Blackdog are colored in white and the test conducted on Tegner is performed in gray.
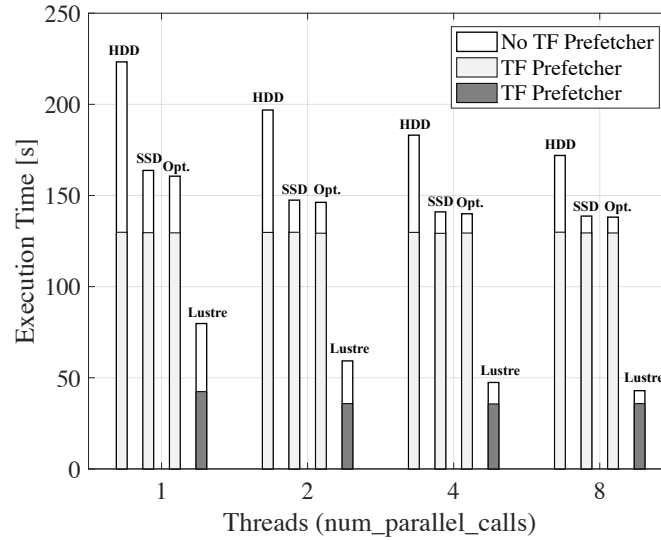

Fig. 6. Median execution time of mini-application in seconds with different number of threads when prefetch is disabled or enabled. Tested storage devices where images are stored are HDD, SSD and Optane. Gray bar represents runtime when one batch is prefetched and the white area represents excess in runtime when prefetch is not used. The case of prefetching of one batch on Tegner is shown in dark gray.

B. TensorFlow Mini-Application: AlexNet

The impact of overlapping input pipeline on CPU with computation pipeline on GPU is extremely effective. We show that runtime of the mini-application with prefetch enabled in light gray in Fig. 6 while the white portion of the bars represents the excess in runtime when prefetch is not enabled. It can be seen that scaling of I/O thread has moderate effect on the total time. This is particular visible on HDD. An interesting observation is that the delay due to I/O can be completely hidden by prefetching. This is indicated by the execution time for different configurations which becomes the same regardless of the number of threads or storage technology used. In other words, the excess in execution time when prefetch is disabled can be conceived as the cost of I/O.

In order to confirm this observation, we trace I/O activity from the execution of the first to the last with *dstat* in Fig. 8. The white area represents the number of megabytes read each second when prefetch is disabled while the gray area represent results when one batch is prefetched. The panel on the top shows the tracing of execution when samples are placed on HDD while the lower panel shows the case when samples are placed on SSD.
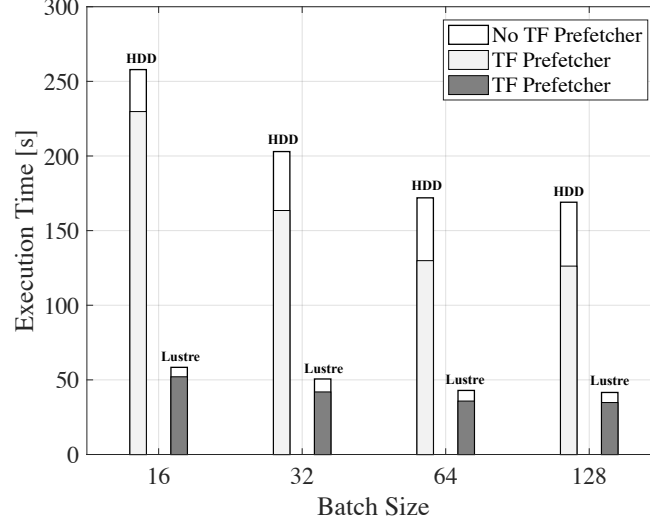
42

chen shuaiben

Fig. 7. The variation of batch size can impact training performance. A larger batch size tends to allow better utilization of GPUs. The figure shows median training performance of mini-application with eight parallel mapping threads.
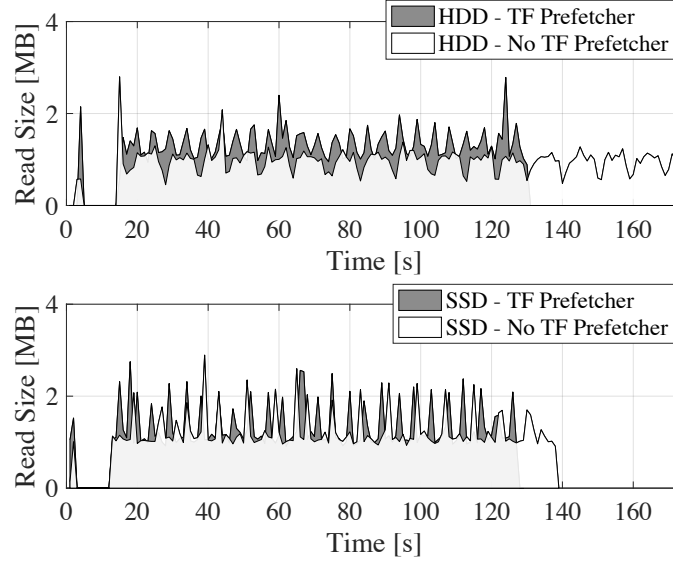

Fig. 8. I/O tracing of mini-application in Megabyte over time using *dstat*. Area in gray represents amount of data read in MB read when one batch is prefetched and area in white represents data read when prefetch is not used. Panel on top presents result when samples are placed on HDD and panel on the bottom shows result when samples are placed on SSD.

For the case when image files are read from HDD without prefetch, a stable interleaving pattern can be observed between batching drawings. As for the case when one batch is prefetched, intervals are considerably closer and the number of megabyte being read is higher in general.

For the case when images are being read from SSD without prefetch, a similar pattern can also be observed. When one batch is prefetched, high peaks can be observed. However, as compared to reading from HDD, the benefit of prefetching is less obvious. One reason is the high I/O rate of SSD and relative low sampling rate of once per second by *dstat*. However, benefits can still be seen by overlapping I/O and computation with prefetching.

In terms of batch size, the effect can be seen in Fig. 7, where we vary the batch size and fix the number of threads to 8. Execution time for both with and without prefetch reduces as the batch size increases. This can be explained by higher utilization rate of GPU for parallelization.
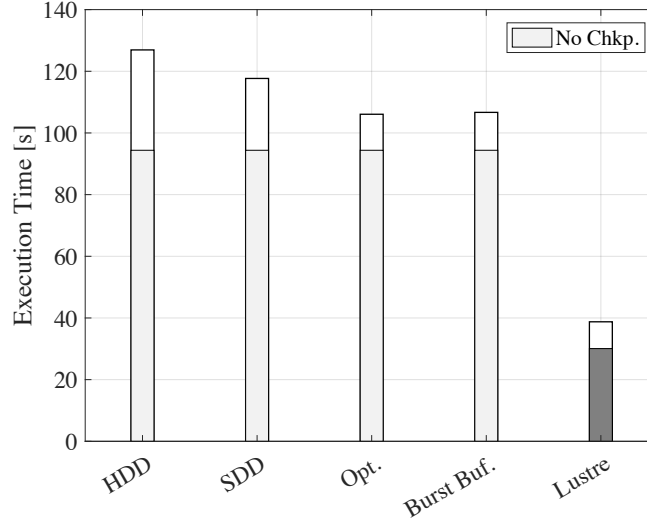
Fig. 9. The median execution time of the extended mini-application in seconds, checkpointing to different storage devices. The gray area shows the same execution with no checkpoints being made. The tests on Tegner are shown in the dark gray bar.

C. Checkpoint and Burst Buffer

We perform checkpoint performance test with our extended mini-application. 100 iterations are executed and one checkpoint is written and synchronized to disk every 20 iterations. Check-pointing of networks with large number of variables can be expensive, and since training must stop in favor for saving it can seriously hamper training speed. Top panel of Fig. 9 shows the total run time of 100 iterations with five checkpoints to different storage media while the gray area represents the run time of 100 iterations without any checkpoint.

Not surprisingly, checkpoint to Intel Optane gave the lowest run time on Blackdog, followed by checkpoint to SSD. Checkpointing to burst buffer also presents very favorable results, showing little difference then only checkpointing to Intel Optane without moving data to slower storage. The bottom panel of Fig. 10 shows a time series of tracing collected by *dstat*, which shows disk activity during execution of the mini-application. It can be seen that checkpointing to HDD uses considerable amount of time, thus hampering training performance. Checkpoints made to Intel Optane are shown in gray. It can be seen that Intel Optane performed extremely well in absorbing data written from the checkpoint process. Once the checkpoint is synchronized, a process is spawned to perform copying of the checkpoint files to HDD. The underlying file system of HDD ext4 performs caching of disk activities and flushes journal information and data to disk when necessary. Thus data written is not flushed to disk immediately, but when caches are flushed by the Operating System. Disk activity from copying to HDD is represented in gray areas, which are delayed comparing to the writes to Intel Optane and the flushing continues after the application ends. Since data is already written to Intel Optane, it is not necessary to enforce immediate synchronization of the checkpoint files when moved to HDD. The tests show considerable improvement when burst buffers are used. Since these devices used for burst buffer are typically smaller in size, by moving these files to HDD for archiving it is possible to cleanup the buffer for other data.

VI. RELATED WORK

Improving training time of DL networks is an active research area [13]. Attempts are made to benchmark several state-of-the-art GPU accelerated deep learning software tools such as Caffe [14], CNTK [15], Tensorflow [4] and Torch where training performance does not scale very well on many-core CPUs. However through the use of GPUs together with a high-speed communications infrastructure they are able to provide performance improvements [16] [17]. Performance of different frameworks often vary with different neural networks [17]. Bottlenecks of various DL frameworks have been studied through performance characterization of different convolution algorithms and their implementations [18]. Layer wise execution times and the batch processing times are also studied to identify limiting factors of each framework. There has been an effort to mitigate the inefficiencies of cuDNN in selecting a convolution algorithm based on the workspace size, so that faster algorithms are used with tight workspace constraints [19]. In terms of computing frameworks, Spark [20] is a successful example of scaling different machine learning networks [21] [22].

Analysis and optimization of I/O is of vital importance for scalable machine learning which employ large datasets. Parallel I/O on HPC systems is an active research area in terms of parallelizing data access [23]. I/O optimizations range from application-side collective I/O to network and disk level request scheduling on the file system [24]. Benchmark tools such as
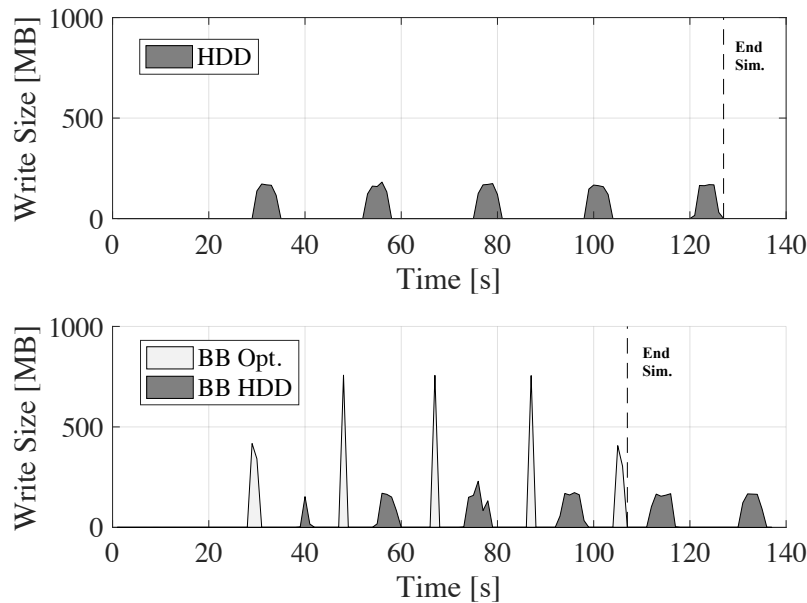
Fig. 10. The top panel shows result from *dstat* for checkpoint written to HDD. The bottom panel shows the case when checkpoint is written to Intel Optane as burst buffer and then moved to HDD. End of execution for the two cases are shown with dotted lines.

IOR are developed for characterizing I/O performance in HPC systems [25]. Assess pattern is also an important aspect for optimization. [26] and [27] looks into data assess pattern of distributed memory systems and proposes a modified framework S-Caffe. S-Caffe is a result of co-designing Caffe framework and MVAPICH2-GDR MPI runtime [28]. Theano-mpi is a Theano based distributed training framework for DNNs based on MPI which is used to drive inter-process communication [29]. DeepIO is an I/O framework for training deep neural networks featuring RDMA-assisted in-situ suffling, input pipelining and entropy-aware opportunistic ordering [30]. The framework is benchmarked against the TensorFlow dataset API, and a portable API for TensorFlow is developed to leverage DeepIO on different storage systems.

VII. DISCUSSION AND CONCLUSION

TensorFlow is one of the most used programming frameworks for implementing DL networks. During the training of the network, a large amount of data, consisting of relatively small files, is read and intermediate results are saved as checkpoints at given time. For these reasons, the performance of the TensorFlow I/O might impact the overall performance of the application and it is important to understand how TensorFlow performs and optimize it. This is the goal of our work.

Using a micro-benchmark, similar in nature to the STREAM benchmark, we measured the read bandwidth of TensorFlow parallel I/O on different devices. We showed that increasing the number of threads, the I/O, the bandwidth in read increases. On Blackdog benchmarking environment, reading from HDD with two, four and eight threads resulted in a 1.65×, 1.95× and 2.3× performance improvement when comparing with the I/O performance with one thread. On Tegner, we observed a 7.8× increase of bandwidth when using eight threads.

One of the major results of this paper is that the TensorFlow prefetcher is a key factor for the input pipeline performance in TensorFlow. The whole input data is divided into smaller datasets, called batches. While a batch is used to train the DL network on the GPU, the next batch is loaded and pre-processed on the CPU when using the TensorFlow prefetcher. In our mini-application, we found a complete overlap of per-batch computation and I/O pipelines leading to a minimal impact of I/O performance on the overall performance of the mini-application. We note that the overlap of computation and I/O depends on how long the per-batch computation lasts. The use of simpler DL networks, deeper input pipeline, faster accelerators and a decreased batch size leads to a decrease of the computation phase and a consequent overlap of computation and I/O. However, in practice we found the computation for one batch with a rather simple DL network, such as AlexNet, spans over 1-2 seconds period in most of the benchmark configurations, e.g. batch sizes and different GPUs, allowing for a complete overlap of computation and I/O.

While the input pipeline has a minimal impact because of overlap of computation and input pipeline, we found that checkpointing of intermediate weights of DL network depends on the I/O device in use and impacts the overall performance. The reason for this is the large amount of data to be written, roughly 600 MB in the case of AlexNet, and the fact that TensorFlow currently does not support overlap of checkpointing and computation. In this, the performance of checkpointing is limited by the latency and bandwidth of the device in use. For this reason, we implemented a burst-buffer solution for

checkpointing to fast storage with small capacity (Intel Optane) and asynchronously copy the checkpoints to slower and high-capacity HDD. This resulted in obtaining the I/O performance of using the fastest I/O device with a performance improvement of 2.6× with respect to checkpointing directly to HDD.

As future work, we plan to focus on two main research lines related to this work. We first intend to investigate the performance of TensorFlow I/O using distributed systems and TensorFlow distributed datasets. In this work, we focused on single-node I/O performance, albeit using parallel I/O and one parallel file system in one test environment. As a second future plan, we intend to investigate the TensorFlow I/O performance using object-store for HPC [31], such as local installations of Ceph [32] and Seagate's Mero [33]. This work requires the implementation of a TensorFlow file systems in the runtime, already supporting other remote object stores, such as AWS and Google Cloud.

Overall, we showed that the current TensorFlow provides high-performance I/O on single node in the data-ingestion phase and that burst-buffer can be an effective technique for fast check-pointing.

ACKNOWLEDGMENT

REFERENCES

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 1–12.

[3] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA Tensor Core Programmability, Performance & Precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pp. 522–531.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning," in *OSDI*, vol. 16, 2016, pp. 265–283.

[5] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014. [Online]. Available: https://doi.org/10.1177/1094342014522573

[6] J. D. McCalpin *et al.*, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19–25, 1995.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[8] I. B. Peng, S. Markidis, E. Laure, G. Kestor, and R. Gioiosa, "Exploring application performance on emerging hybrid-memory supercomputers," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec 2016, pp. 473–480.

[9] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia *et al.*, "Accelerating science with the nersc burst buffer early user program," *CUG16*, 2016.

[10] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and design of Cray DataWarp," *Cray User Group CUG*, 2016.

[11] W. Schenck, S. El Sayed, M. Foszczynski, W. Homberg, and D. Pleiter, "Evaluation and performance modeling of a burst buffer solution," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 12–26, 2017.

[12] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 19.

[13] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *CoRR*, vol. abs/1802.09941, Feb. 2018.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[15] F. Seide and A. Agarwal, "CNTK: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 2135–2135.

[16] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 3. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 1337–1345. [Online]. Available: http://proceedings.mlr.press/v28/coates13.html

[17] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking State-of-the-Art Deep Learning Software Tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, Nov 2016, pp. 99–104.

[18] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of CNN frameworks for GPUs," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 55–64.

[19] Y. Oyama, T. Ben-Nun, T. Hoefler, and S. Matsuoka, "Accelerating Deep Learning Frameworks with Micro-batches." IEEE, Sep. 2018, to appear in IEEE International Conference on Cluster Computing (Cluster'18).

[20] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[21] D. Harnie, M. Saey, A. E. Vapirev, J. K. Wegner, A. Gedich, M. Steijaert, H. Ceulemans, R. Wuyts, and W. D. Meuter, "Scaling machine learning for target prediction in drug discovery using Apache Spark," *Future Generation Computer Systems*, vol. 67, pp. 409 – 417, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X1630111X

[22] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "Sparknet: Training deep networks in spark," *arXiv preprint arXiv:1511.06051*, 2015.

[23] R. Thakur, W. Gropp, and E. Lusk, "A case for using MPI's derived datatypes to improve I/O performance," in *Supercomputing, 1998. SC98. IEEE/ACM Conference on*. IEEE, 1998, pp. 1–1.

[24] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 155–164.

[25] H. Shan and J. Shalf, "Using IOR to Analyze the I/O Performance for HPC Platforms," in *Cray User Group Conference (CUG07)*, 2007.

[26] S. Pumma, M. Si, W. Feng, and P. Balaji, "Towards Scalable Deep Learning via I/O Analysis and Optimization," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec 2017, pp. 223–230.

[27] S. Pumma, M. Si, W.-c. Feng, and P. Balaji, "Parallel I/O Optimizations for Scalable Deep Learning," in *Parallel and Distributed Systems (ICPADS), 2017 IEEE 23rd International Conference on.* IEEE, 2017, pp. 720–729.

[28] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-Caffe: co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters," *Acm Sigplan Notices*, vol. 52, no. 8, pp. 193–205, 2017.

[29] H. Ma, F. Mao, and G. W. Taylor, "Theano-MPI: A theano-based distributed training framework," in *Euro-Par 2016: Parallel Processing Workshops*, F. Desprez, P.-F. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, V. Scarano, M. A. Vega-Rodríguez, A. L. Varbanescu, S. Hunold, S. L. Scott, S. Lankes, and J. Weidendorfer, Eds. Cham: Springer International Publishing, 2017, pp. 800–813.

[30] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems," in *IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2018)*, 2018.

[31] S. W. Chien, S. Markidis, R. Karim, E. Laure, and S. Narasimhamurthy, "Exploring Scientific Application Performance Using Large Scale Object Storage," *arXiv preprint arXiv:1807.02562*, 2018.

[32] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation.* USENIX Association, 2006, pp. 307–320.

[33] S. Narasimhamurthy, N. Danilov, S. Wu, G. Umanesan, S. W. D. Chien, S. Rivas-Gomez, I. B. Peng, E. Laure, S. de Witt, D. Pleiter *et al.*, "The SAGE project: a storage centric approach for exascale computing," in *Proceedings of the 15th ACM International Conference on Computing Frontiers.* ACM, 2018, pp. 287–292.

Characterizing deep-learning I/O workloads in tensorflow

Chien, Steven W.D.; Markidis, Stefano; Sishtla, Chaitanya Prasad; Santos, Luis; Herman, Pawel; Narasimhamurthy, Sai; Laure, Erwin

01	chen ping	Page 1

15/8/2019 2:39

02	chen ping	Page 1

15/8/2019 2:39

03	chen ping	Page 1

4/8/2019 5:33

04	chen ping	Page 1

4/8/2019 5:34

05	chen ping	Page 1

4/8/2019 5:34

06	chen ping	Page 1

4/8/2019 5:35

07	chen ping	Page 1

4/8/2019 5:35

08	chen ping	Page 1

4/8/2019 5:35

09	chen ping	Page 1

4/8/2019 5:35

10	chen ping	Page 1
	4/8/2019 5:36	

11	chen ping	Page 1
	4/8/2019 5:39	

12	chen ping	Page 1
	4/8/2019 5:39	

13	chen ping	Page 1
	4/8/2019 5:55	

14	chen ping	Page 1
	4/8/2019 9:29	

15	chen ping	Page 1
	12/8/2019 7:54	

16	chen ping	Page 1
	12/8/2019 7:54	

17	chen ping	Page 1
	12/8/2019 8:01	

18	chen shuaiben	Page 1
	15/8/2019 11:53	

19	chen shuaiben	Page 2
	15/8/2019 11:53	

20	chen shuaiben	Page 2
	15/8/2019 11:54	

21	chen ping	Page 2
	12/8/2019 8:12	

22	chen ping	Page 2
	12/8/2019 8:13	

23	chen ping	Page 2
	12/8/2019 8:13	

24	chen ping	Page 2
	12/8/2019 8:13	

25	chen ping	Page 2
	12/8/2019 8:16	

26	chen ping	Page 2
	12/8/2019 8:17	

27	chen ping	Page 2
	12/8/2019 8:44	

28	chen ping	Page 3
	6/8/2019 1:44	

29	chen ping	Page 3
	12/8/2019 8:47	

30	chen ping	Page 3
	12/8/2019 13:32	

31	chen ping	Page 4
	12/8/2019 9:00	

32	chen shuaiben	Page 4
	15/8/2019 11:53	

33	chen shuaiben	Page 5
	15/8/2019 11:53	