Efficient I/O for Neural Network Training with Compressed Data

Zhao Zhang*, Lei Huang*, J. Gregory Pauloski‡, Ian T. Foster¶
*Texas Advanced Computing Center
Email: zzhang, huang@tacc.utexas.edu
‡University of Texas at Austin
Email: jgpauloski@utexas.edu
¶University of Chicago & Argonne National Laboratory
Email: foster@uchicago.edu

***Abstract*—FanStore is a shared object store that enables efficient and scalable neural network training on supercomputers. By providing a global cache layer on node-local burst buffers using a compressed representation, it significantly enhances the processing capability of deep learning (DL) applications on existing hardware. In addition, FanStore allows POSIX-compliant file access to the compressed data in user space. We investigate the tradeoff between runtime overhead and data compression ratio using real-world datasets and applications, and propose a compressor selection algorithm to maximize storage capacity given performance constraints. We consider both asynchronous (i.e., with prefetching) and synchronous I/O strategies, and propose mechanisms for selecting compressors for both approaches. Using FanStore, the same storage hardware can host 2–13× more data for example applications without significant runtime overhead. Empirically, our experiments show that FanStore scales to 512 compute nodes with near linear performance scalability.**

I. Introduction

Deep learning (DL) methods are increasingly popular in both academia and industry. Researchers and practitioners explore DL methods for classification, extrapolation, interpolation, inverse problems, and many other tasks. The training of neural network models inevitably requires the use of HPC-like systems [1] with their powerful memory and communication architectures. Recent work [2–7] shows the power of supercomputers in reducing image classification (ResNet-50 [8]) training time from hours to 132 seconds, without loss of validation accuracy.

DL training involves repeated steps, in each of which elements of the large training set T are processed in batches of size B, $B \ll |T|$, with the processing of each batch involving each of N processors handling B/N elements, exchanging gradient information with all other processors, and then updating its weights based on gradients.

Traditional shared file systems cannot meet the resulting I/O throughput needs at scale [9–11]. In one recent study, this led to a weak scaling efficiency of only ∼25% with ResNet-50 on 64 GPUs [9], while researchers at Microsoft report a low GPU utilization of 52%, partially due to the ignorance of locality in a cloud setting [12]. On a supercomputer, the repeated, highly concurrent, and frequent file accesses can result in file system slowdown and unresponsiveness, thus negatively impacting other users on the same machine.

The availability of burst buffers on many modern supercomputers suggests a potential solution to this problem. Serving training data from burst buffers can drastically reduce I/O overheads, and thus enable efficient training at scale [9, 13]. However, the use of burst buffers introduces new challenges. If burst buffers are located on every compute node (a so-called "node-local" architecture), it can be difficult to determine the optimal node count for training, given the twin constraints of: data fitting local-node burst buffers and training making efficient use of all processors. We illustrate this challenge and possible consequences in Figure 1. Performance depends on a balance between batch size B and processor count N. If B is too large (i.e., $B > B_{\text{max}}$, for some optimizer-dependent B_{max}), training may converge to a suboptimal target [14]. If B/N is too small, then data assigned to each processor cannot make full use of the parallel architecture. We denote b as the minimum per-node batch size required for 100% utilization of a processor; we need $B_{\text{max}}/N \geq b$ (i.e., $N \leq B_{\text{max}}/b$) for efficient execution. The use of node-local burst buffers introduces a third constraint: we need N to be large enough such that $N \times M \geq |T|$ (i.e., $N \geq |T|/M$), where M is the size of each node's burst buffer.

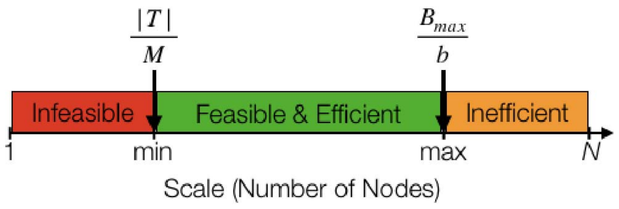


Fig. 1: Node count landscape and associated consequences

For example, the original ResNet-50 model trains on the ∼140 GB ImageNet dataset with a batch size of 256 [14]. On a cluster with four GPUs and 60 GB local storage per node, it requires three nodes (with a total of 12 GPUs) to host the data, but the batch size of 256 can achieve greater than 90% utilization on no more than two GPUs. Thus the overall efficiency is < 2/12 = 17%. We find that many DL applications are in this situation, namely that the need for substantial total aggregate burst-buffer memory to hold training data pushes them too far to the right in Figure 1. In such cases, using fewer compute nodes to host the data can significantly enhance hardware utilization.

In this paper, we investigate the use of lossless compression techniques to reduce the overall data size and thus push the minimum efficient scale to the left along the axis in Figure 1. We evaluate over 180 compressor and option combinations and study the performance impacts for DL training using both benchmarks and real-world datasets and applications. Our experiments show that, with proper compressors, it is possible for DL applications to preserve the baseline performance (with data stored locally and no compression) even with decompression overhead. Such improvement is made possible by the I/O mechanisms in DL frameworks: 1) With asynchronous I/O (also known as prefetch), the decompression overhead can be masked by the computation and communication of each iteration; 2) With synchronous I/O, the decompression overhead may be compensated by the time saved with less I/O quantity (details are in §VI-A). To ease the compressor selection, we design a numerical algorithm to make compressor decisions given the application and I/O performance.

Based on this study, we extend FanStore [15], a distributed object store, with compression techniques in data storage and file access over interconnect. FanStore integrates several metadata and data placement strategies for scalability. It hosts the compressed dataset across nodes and provides a global namespace. Training programs can access the compressed representation using the POSIX-compliant interface completely in user space, so FanStore can be used without intrusive code changes or root privilege. With the selected compressors, the example applications preserve the baseline performance with raw data and can host 2-13x more data on the same storage hardware. Our experiments also show that with FanStore, DL applications scale to 512 processors with over 90% weak scaling efficiency.

This work makes three contributions:

- Efficient and scalable file access to a compressed representation of many datasets, achieved by combining new data structures, scalable metadata placement, and consistency models for different data usage types.
- A compressor selection algorithm that evaluates the benefit and overhead of candidate compressors and selects the compressor with the highest possible storage capacity given the performance constraint.
- The open source implementation of FanStore (https://github.com/tacc/fanstore), tested on the Intel x86_64 and IBM POWER9 architectures that cover over 96.6% of the machines in the TOP500 list as of June 2019.

The rest of the paper is as follows. §II introduces distributed DL training and compression algorithms. §III reviews related work in I/O optimization for distributed DL training. §IV discusses the design of FanStore and §V presents implementation details. We review our I/O implementation and describe our compressor selection algorithm in §VI. We present experiments in §VII and conclude and envision future work in §VIII.

II. Background

We introduce the technical background of distributed DL training, its I/O pattern, and compression techniques.

A. Distributed DL Training

In the popular data parallel approach to distributed training, the model is replicated and training batches are distributed across nodes at each iteration [16]. Researchers have used this approach to scale ImageNet training on thousands of CPUs, GPUs, and TPUs [2–7]. Compared to the single node case, the data parallel approach runs in a similar way in the forward computation, while it divides the backward propagation to three steps. The first step is to compute the gradients on each node; the second step is to run an allreduce function in MPI so that each node has the identical averaged gradients for each weight; the last step is to apply updates to weights. In practice, the allreduce step uses a buffer, and an allreduce is invoked once the buffer is full. Weight updates are streamlined with allreduce operations. An "Epoch" in training is when all items in the training dataset have been visited once statistically. The relation between the number of iterations, epoch count, data size, and batch size is as following: $num\_iter = \frac{num\_epoch * data\_size}{batch\_size}$

B. Distributed DL I/O

Unlike traditional compute-intensive HPC applications that demonstrate bursty I/O behavior [17–20], distributed DL training has a long-lasting, repeated, high volume, and highly concurrent I/O pattern.

1) Metadata access: A training program first enumerates the supplied training data items and calculates the number of iterations per epoch, given the specified batch size. This step cause a workload burst in the metadata server. For example, the ImageNet dataset has 1.3 million files in 2002 directories. Reading all metadata for these files involves 2002 *readdir()* calls and 1.3 million *stat()* calls from one I/O process.

This situation is exacerbated at scale. For example, when running a Keras, TensorFlow, and Horovod stack on four nodes, each with six GPUs, Horovod requires launching one TensorFlow process per GPU, resulting in 4×6=24 processes in total. Keras then launches four I/O threads per process, by default, to read training data. There will then be a total of 4×24=96 independent I/O threads running, each generating the workload described in the previous paragraph. We have observed file system slowdown and unresponsiveness due to this metadata access pattern on our machines.

2) Read: At the start of each iteration, each process reads $batch\_size$ files from the shared file system. Thus, in the above example, the four nodes read a total of $96 \times batch\_size$ files concurrently. The iterative training process makes this read pattern persist until the end of training.

3) Write: DL programs write several output files including checkpoint files, sample output, and log files. It is a common practice to number the checkpoint files with epoch count, as previous models may have a better generalization than later ones. Unless resuming from a checkpoint, these files are not read by the training program again. Sample output files are usually written by Generative Adversarial Networks (GAN), when the generator produces artifacts that are similar to real ones. These sample outputs are examined by researchers with expertise to verify the effectiveness of the GAN. DL programs also write log files, often containing information about each iteration and summaries of each epoch to keep track of training progress. Once written, all these three types of output files are rarely read again by the training program.

C. Data Compression

Compression techniques have been studied in file systems [21] and I/O forwarding [22]. Lossy data compression [23–26] with controlled data distortion was evaluated for its efficacy in scientific computing scenarios. Another family of compression algorithms is lossless compression. These algorithms employ one or several of encoding and dictionary-based algorithms to preserve the original data values. They can be generic, such as Huffman coding [27] and Lempel-Ziv compression [28, 29], or be format specific, such as JPEG2000 [30] and LZW for TIFF [31].

We focus here on lossless compressors, since the impact of lossy compression on training performance, while shown to have promise in some cases [32, 33], is unclear in general.

III. Related Work

I/O optimization has been extensively studied in the context of different classes of target application, including HPC, workflow, and AI.

Model performance (e.g., test accuracy and test loss) is critical for DL training. A global dataset view, in which every node sees the same data set with identical directory structure and files, is a key to preserving model performance [15]. Current DL I/O optimization methods all enforce this constraint.

A common way to reduce the I/O workload is to encapsulate the large dataset into one or several files in a customized format, then let training programs read the dataset through a customized interface. Examples include TFRecord in TensorFlow, IORecord in MXNet, and LMDB in Caffe. The encapsulated dataset can be placed in shared file system or local storage, if the dataset fits. Previous work [34] applies optimization such as inter-process contention reduction, sequential seek elimination, and randomization reduction in I/O with LMDB at large scale. In general, this method can reduce metadata workload on the server, but read/write traffic remains the same, and the heavy I/O traffic persists through the training process.

A technical workaround is to partition the dataset into chunks, and let each node only see its own chunks. After every few epochs, the chunks are permuted across the nodes. Using this method, every compute node sees a partial dataset with a larger variance at a given time, and the global view is maintained eventually when all compute nodes see all data. However, the impact of the time-divided variance on the training convergence is unclear. In addition, permuting the dataset introduces additional overhead.

Another approach to reduce I/O workload for DL is to host a shared file system across the local storage space of the compute nodes [13, 35]. This method extends the storage capacity from single node storage to cumulative storage space and preserves the POSIX-compliant interface to the dataset, but the metadata problem is left unaddressed in most of today's file systems and burst buffer solutions.

In contrast, FanStore localizes metadata server operations and leverages the data compression technique to explore further opportunity to enlarge on-node storage capacity using the same hardware without losing training performance. It also maintains the global data view using different strategies for input and output data. In addition, FanStore enables POSIX-compliant file access with the function interception method which achieves near raw hardware performance.

IV. Design

This section discusses the design of FanStore including the interface, compressed representation, access on the compressed data, and caching policy.

A. Interface

FanStore supports data access through the POSIX-compliant interface. A file or directory can be accessed using its path, e.g., */path/to/dir/file*. FanStore leverages the function interception technique to achieve high performance I/O. It intercepts and implements the functions in GNU C Library.

```
int open(const char *filename, int flags[, mode_t
    mode])
int close(int fd)
ssize_t read(int fd, void *buffer, size_t size)
ssize_t write(int fd, const void *buffer, size_t
    size)
off_t lseek(int fd, off_t offset, int whence)
DIR * opendir (const char *dirname)
struct dirent * readdir(DIR *dirstream)
int closedir (DIR *dirstream)
int stat(const char *filename, struct stat *buf)
```

Listing 1: FanStore Interface

Listing 1 shows the interface conceptually, while the actual implementation intercepts the 64 bit version of these functions. *open()* and *close()* open and release a file, respectively. *read(), lseek(), write()* provide the low level input, positioning, and output functions. *opendir(), readdir(), closedir()* handle operations on directories. *stat()* retrieves metadata (file attributes) from file system.

With these interface implemented, FanStore achieves a minimal POSIX-compliant interface. FanStore now implements a multi-read single-write I/O model, where each file in FanStore can be read multiple times and concurrently, while each file can be written only once by one process. An output file cannot be updated once a *close()* function is applied to the file descriptor. This restricted writing model suffices for distributed DL training, as concurrent writes to the same output file are rare, and those output files are never read by the training program again (as discussed in §II-B).

TABLE I: The compressed data representation

num files 4 bytes	**file path** 256 bytes	**compressor** 2 bytes	**stat** 144 bytes	**size** 8 bytes	**data** variable
	file path 256 bytes	**compressor** 2 bytes	**stat** 144 bytes	**size** 8 bytes	**data** variable
	file path 256 bytes	**compressor** 2 bytes	**stat** 144 bytes	**size** 8 bytes	**data** variable
	...				

B. Compressed Data Representation

FanStore uses a compressed data representation to enhance storage capacity. Table I shows the data layout of the representation. A data preparation tool (details in §V-B) partitions the dataset into several partitions, selects compression algorithms, and concatenates the compressed input files. Each partition starts with the number of files, followed by, for each file, the file path, compressor used (an integer identifier), metadata, compressed data size, and compressed data. The compressed dataset is stored in the shared file system and needs to be prepared only once. Upon use, programs need to load the compressed data representation to local storage.

C. Accessing Compressed Representation

1) Loading: Upon training, a parallel program loads the partitioned compressed dataset from the shared file system to local storage, with each node holding one or several partitions. The program uses knowledge of the partition size and available local storage space to make dynamic decisions on how many partitions to load on each node. The program then scans each partition to extract the metadata and compressed file data. It inserts the locality information into the extra fields in the file metadata. The metadata are stored in RAM using a hash table. The compressed file data are stored as byte arrays in a hash table keyed by the file path, if users specify RAM as the back end; if local disks (e.g., SSD) are the back end, the compressed data files are stored in the local file system. After all partitions are loaded, the parallel program uses *allgather()* to exchange the metadata scattered on compute nodes so as to construct a global metadata view. Thus all subsequent metadata traffic on the training dataset is local.

2) Accessing: Distributed DL training involves three types of data access: 1) directory metadata, 2) file metadata, and 3) file data. Directory info (*readdir()*) is returned immediately with the metadata stored in RAM. File metadata (*stat()*) is handled in a similar way. Thus the high volume and highly concurrent directory and file metadata access are handled completely in RAM on each compute node. No traffic is placed on the metadata servers of the shared file system.

To access file data, the training program issues a request through GNU C library functions (*open()*). Figure 2 shows the internal logic for *open()* request. FanStore intercepts this function call and its parameters, then looks for the file in the backend. If the compressed file data is local, it will decompress the data and stores the plain file data in a cache region. If the compressed file data is remote, it will send an MPI message to retrieve the compressed data, then decompress it and store the plain file in the cache region. Either way, the decompression is done at runtime. To respond to subsequent *read()* calls, FanStore returns the requested data from the cache region, as shown in Figure 3.

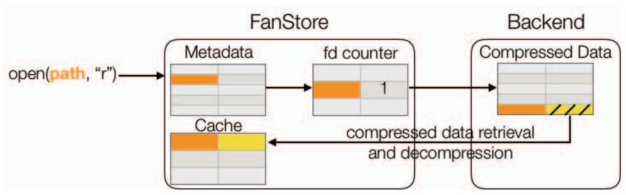


Fig. 2: FanStore handling an *open* request locally. Orange indicates file path, yellow with stripes is compressed data, solid yellow is uncompressed data. Backend is on the same node as FanStore daemon.

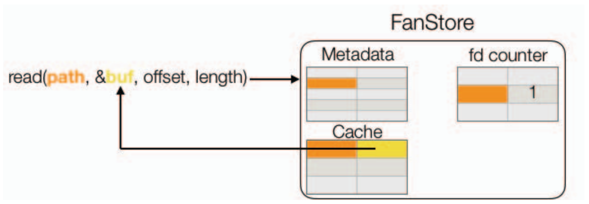


Fig. 3: FanStore handling a *read* request. Orange indicates file path; solid yellow is uncompressed data.

In this way, all file access is processed within the compute nodes over the interconnect, and no I/O traffic is placed to the shared file system.

3) Caching: FanStore uses a shared memory pool to cache the decompressed file data. The design principle is to use a minimum amount of RAM for caching, as the DL training program itself can be memory intensive. One characteristic of the file access patterns in DL training is that every file has an identical probability to be accessed at every iteration. So FanStore implements a variant of the FIFO (first in first out) caching policy, where we enforce the FIFO rule except for the file that is being accessed by more than one I/O thread. As shown in Figure 4, we use a thread-safe hash table to keep track of the opened files and the number of I/O requests on them at runtime. Once a file is opened and decompressed, the counter of this file increases by one; and the counter decreases by one when it is closed. The cache entry is released if the counter of a file is zero.

V. Implementation

We now discuss the implementation details of FanStore on its components, parallel runtime and communication, parallel data loading, and fault tolerance.

A. Overview

Overall, training programs see FanStore as a shared file system with a mount point specified by users. The dataset is accessible using the same relative path when the dataset is prepared. For example, directory *dir/cate1/file1* is accessible as

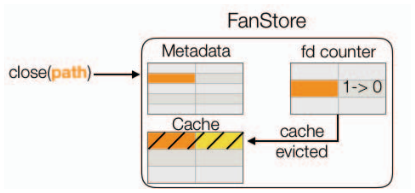


Fig. 4: FanStore handling a *close* request. Orange indicates file path; entries with stripes are evicted.

/fs/dir/cate1/file1. So users only need to change the directory path in the training program to use FanStore.

FanStore has three components: the data preparation tool, the function interceptor, and the FanStore daemon. Functionally, the data preparation tool is used to package the dataset in the shared file system using the compressed data representation. The function interceptor intercepts the I/O function calls in GNU C library and uses FanStore daemon for according functions. The FanStore daemon manages the metadata, local data access, remote data retrieval, data decompression, and cache for decompressed data. It also handles the requests from the function interceptor.

B. Data Preparation

The data preparation tool is a standalone multi-threaded program that takes input parameters of data path, partition count, and compression algorithm. In addition to data scattering, users can also specify a directory to be broadcast to all nodes, for use for the validation dataset from which every node reads all files. The data preparation tool produces datasets in the form of several packaged partitions.

Upon preparation, the tool produces a list of files in the specified path, then divides the list into several chunks. Each thread processes file list chunks in a round-robin manner, traversing all files in the list, compressing and concatenating each file sequentially using the compressed data representation, as discussed in §IV-B. A dataset can be prepared once and used for subsequent training repeatedly, unless it is updated.

C. Function Interception

We combine two techniques for function interception. The first technique is to preload a dynamic library [36] with identical function names. The second is trampoline [37], which rewrites the first several instructions of a function to use a customized implementation in user space, then jumps back to the remainder or the end of the original function.

For some I/O functions in GNU C library, they can be called internally. In such cases, these functions are directly accessed without looking up the dynamical libraries, so the dynamic library preload technique does not work. Instead, we intercept functions such as *open(), close()), read()*, and *stat()* using the trampoline technique. For those functions that are not called internally, such as *seek()* and *write()*, we intercept them using the dynamic library preload technique.

The dynamic library preload interceptor is implemented in a shared library called wrapper.so. To use it, users need to set *export LD_PRELOAD=/path/to/wrapper.so*. It rewrites the corresponding I/O functions in GNU C library when executed.

D. Parallel Runtime and Communication

FanStore is designed to be a scalable shared object store with a target of thousands of compute nodes. Thus, instead of a master slave model, FanStore processes see each other as a peers. Users launch FanStore using an MPI task launcher, such as *mpiexec.hydra*. In practice, FanStore should be launched with one process on each node. Each FanStore process uses its rank as the identifier.

When launched, each process uses its rank to determine which partitions to load. It evaluates first, using the method described in §IV-C1, whether there is enough storage space for the assigned partitions; if so, it also evaluates whether it has space for extra partitions. The more data served from local storage, the less communication passes through the interconnect, improving I/O performance.

FanStore needs communication at four places: 1) Metadata broadcast; 2) storing additional partitions; 3) remote file retrieval; 4) write metadata insertion. In general, FanStore uses MPI for communication. Specifically, the metadata broadcast is implemented with the MPI *Allgather()* collective function. Once decided to load additional partitions, FanStore does not read them off the shared file system, but rather copies it from neighbors in a virtual ring topology. In this way, the additional copy forms a data transfer from one process to its neighbor. Assuming each partition is about the same size, this data transfer does not introduce contention from a topological point of view. The remote file retrieval is implemented with MPI *send()* and *recv()*. The last communication scenario happens when a file is closed. Once the file is closed, the write cache entry is dumped to back end and the metadata is forwarded via an MPI message to the node with the corresponding rank.

E. Fault Tolerance

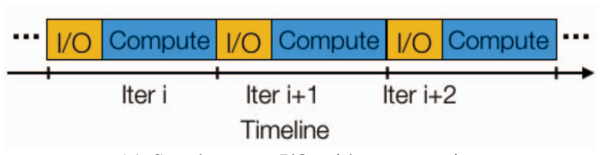
Model performance such as test accuracy or loss is sensitive to the batch size. If one node fails during training, the batch size changes, which may lead the model to a stale state. As a common practice, DL programs usually write checkpoints named with epoch numbers, as discussed in §II-B3. Thus FanStore does not address the fault tolerance issue explicitly, users can resume training from the last checkpoint in the shared file system.

VI. Compression

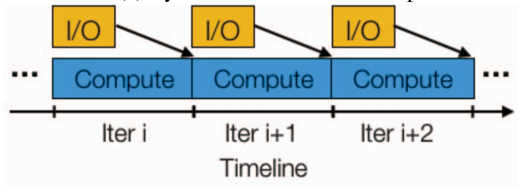
The wide variety of compressors, datasets, and application I/O sensitivities makes it challenging to quantify the benefits and overheads of different compressors for a specific application and target dataset(s). We describe here a numerical algorithm for selecting compressors with the highest compression ratio under the performance constraint.

A. I/O Implementation

I/O in DL training may be implemented either synchronously or asynchronously. In the synchronous approach,



(a) Synchronous I/O with computation



(b) Asynchronous I/O with computation

Fig. 5: Common I/O methods in DL frameworks

I/O and computation are sequential in each iteration, as shown in Figure 5(a); in the asynchronous approach, I/O is performed in parallel with the previous computation, as shown in Figure 5(b), with the training batch read in the I/O phase of *Iter i* fed to the computation of *Iter i+1*. Asynchronous I/O makes better utilization of the compute hardware; Keras, PyTorch, TensorFlow, and Caffe all provide such functionality. Note that "Compute" in Figure 5 includes the forward computation, the allreduce on parameters, and the backward computation.

Fetching compressed data involves two steps: read and decompression. Compressed data lowers the time cost of read, as each iteration reads less data, but introduces decompression costs, which vary depending on the compressor. Decompression performance is thus the key to enhancing storage capacity without losing application performance. With the constraint of no application performance loss, synchronous I/O requires that decompression time be less than the time saved reading compressed data. Asynchronous I/O imposes the weaker condition that decompression time be shorter than the difference between the per-iteration time and compressed data read time.

The overall data fetching cost is a non-linear combination of read performance and decompression performance. The non-linearity is due to the bounding factor of read performance, which may be either throughput, measured in files/sec, if training files are small, or bandwidth, measured in MB/s, if training files are sufficiently large. In practice, we pick the larger of the two possible bounding factors.

B. Compressor Selection

We use Equations 1 and 2 to select compressors. The actual values of C_{batch} and S'_{batch} are known application parameters. S_{batch} and $Tpt_{decom}(c)$ can be estimated with samples using a set of candidate compressors. Tpt_{read} and Bdw_{read} can be determined by an I/O performance benchmark. The resulting C is a set of candidate compressors that meet the performance requirements. Then we select the compressor with the highest compression ratio. Empirically, we evaluate 180 compressor configurations on six real-world datasets and then validate this compressor selection algorithm with three use cases. The details is presented in §VII-D and §VII-E, respectively.

TABLE II: Statistics of the test datasets

Dataset	Format	# files	# dirs	Total size	Avg size
EM	tif	0.6 M	6	500 GB	1.6 MB
Tokamak	npz	0.58 M	1	1.7 TB	1.2 KB
Lung image	nii	1.4 K	2	2.2 GB	1.3 MB
Astronomy image	FITS	17.7K	1	1 TB	6 MB
ImageNet	jpg	1.3 M	2,002	140 GB	100 KB
Language	txt	8	1	32 MB	4 MB

$$\begin{aligned} C_{sync} = \{c | &\frac{C_{batch}}{Tpt_{decom}(c)} + T_{read}(C_{batch}, S_{batch}) \\ &< T_{read}(C_{batch}, S'_{batch})\} \end{aligned} \tag{1}$$

$$C_{async} = \{c | \frac{C_{batch}}{Tpt_{decom}(c)} + T_{read}(C_{batch}, S_{batch}) < T_{iter}\} \tag{2}$$

$$T_{read}(C_{batch}, S_{batch}) = \max(\frac{C_{batch}}{Tpt_{read}}, \frac{S_{batch}}{Bdw_{read}}) \tag{3}$$

where:

C_{batch}	= (files) batch size per iteration
S_{batch}	= (MB) I/O quantity w/ compression per iteration
S'_{batch}	= (MB) I/O quantity w/o compression per iteration
$Tpt_{decom}(c)$	= (files/s) decompress. throughput of compressor c
$T_{read}(Size_{batch})$	= (s) time of reading $Size_{batch}$ files from disk
Tpt_{read}	= (files/s) throughput of file read
Bdw_{read}	= (MB/s) bandwidth of file read
T_{iter}	= (s) time of an iteration

VII. Experiments

We now present the test platforms, datasets, and applications that we use to evaluate the effectiveness of the FanStore design and the compressor selection algorithm.

A. Platforms

We use three clusters. **GTX** has 16 nodes, each with four Nvidia GTX 1080 Ti GPUs and ~60 GB local SSD. **V100** has four nodes, each with four V100 GPUs, a POWER9 CPU, and ~256 GB local RAM disk. Both clusters use a Mellanox FDR Infiniband interconnect with up to 56 Gbps bandwidth and sub-micro second latency. **CPU** is a 512 node cluster, where each node has two Intel Xeon Platinum 8160 processors and ~144 GB SSD, connected by a 100 Gb/sec Intel Omni-Path (OPA) network with a fat tree topology.

Our chosen DL frameworks use CUDA 9.0 and CUDNN 7.0 on **GTX** and CUDA 10.0 and CUDNN 7.4.2 on **V100**. We use TensorFlow 1.12.0, TensorLayer [38] 1.9.1, Keras [39] 2.2.2, and Horovod [40] 0.15.2.

B. Datasets and Applications

We use six real-world datasets to evaluate 180 compressor configurations provided by lzbench [41]. Table II summarizes the statistics of these datasets.

The applications in the experiment are SRGAN [42], FRNN [43], and ResNet-50. SRGAN in this case is used

TABLE III: POSIX-compliant solution read performance (files/sec) comparison

Solution	128 KB	512 KB	2 MB	8 MB
FanStore	28 248	9689	2513	560
SSD-fuse	6687	2416	738	197
SSD	39 480	9752	2786	678
Lustre	1515	149	385	139

to process 3D scanning electron microscope (EM) imaging data of neural tissue samples. It trains a generative adversarial network (GAN) to increase the resolution of undersampled images acquired on a point scanning imaging system (e.g. a scanning electron or scanning confocal microscope). This application trains for 2000 epochs.

FRNN is used to predict disruptions in tokamak reactors, with the goal of improving both their performance and operating/repair costs. Practical disruption prediction in tokamaks has recently improved by utilizing deep learning algorithms with real-time machine diagnostics to predict via long short term memory (LSTM) the onset of major disruptions.

ResNet-50 is a convolutional neural network for image classification. The test case is training ResNet-50 with the ImageNet-1k dataset, which has 1000 categories and 1.3 million images in total.

C. Compression-free Performance

The goal of this experiment is to understand FanStore's performance without compression and how it compares to other technical solutions. Figure 6 shows the read throughput comparison between FanStore and TFRecord. Across three datasets of ImageNet, EM, and RS, FanStore reads 5–10× faster than TFRecord on both the Intel Xeon Platinum 8160 and POWER9 processors.

We then compare FanStore with other POSIX-compliant solutions of 1) FUSE over SSD, 2) SSD, and 3) Shared File System (Lustre) using a simple benchmark with a variable file size of 128 KB, 512 KB, 2 MB, and 8 MB. Table III summarizes the read performance. FanStore achieves 71–99% of raw SSD performance and is 2.9–4.4× faster than FUSE over SSD and 4.0–64.7× faster than the Lustre deployment. The high read throughput attributes to the low overhead of function interception technique described in §V-C, as it bypasses the system calls in kernel space. It is this high speed read that enables the possibility of leveraging compression techniques to enhance storage capacity without losing application performance.

D. Compressor Evaluation

In this experiment, we study the efficacy of a set of 180 compressor configurations on the six real-world datasets shown in Table II. We sample a few files from each dataset, then use lzbench to examine the compression ratio and decompression cost.

Figure 7 shows EM (tif) and RS (npz) results in the compression-ratio and decompression-time tradeoff space on the Intel Xeon Platinum 8160 processors (SKX) and IBM

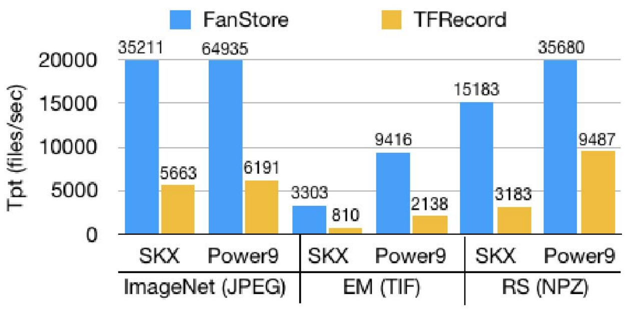


Fig. 6: FanStore vs. TFRecord read performance

TABLE IV: `lzsse8` and `lz4hc` compression ratios on the six datasets

Dataset	EM	Tok.	Lung	Astro	ImageNet	Lang.
`lzsse8`	2.3	2.6	5.7	2.6	1.0	2.8
`lz4hc`	2.0	3.0	6.5	2.2	1.0	2.6
`lzma`	4.0	3.6	10.8	3.4	1.0	4.0
`xz`	4.0	3.4	10.8	3.4	1.0	4.0

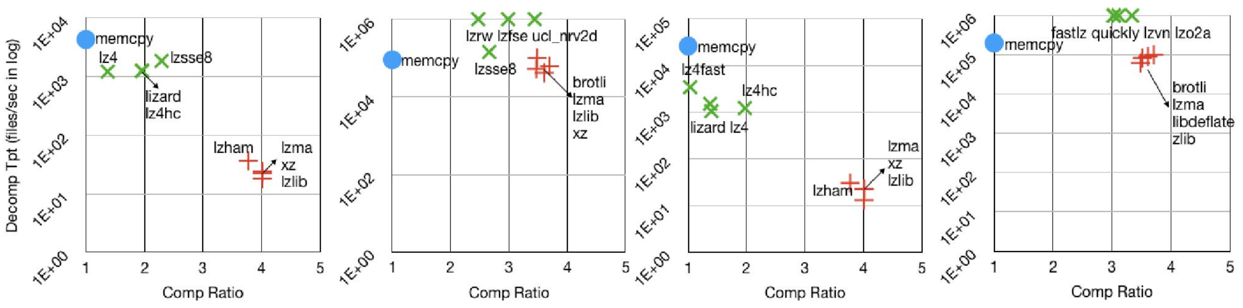
POWER9 processors (POWER9). For the ease of visual perception, we only show the compressors either with highest compression ratio or lowest decompression time. E.g., in Figure 7a, `lzsse8` has the lowest decompression cost (540 μs) and a compression ratio of 2.3. From the results, we see that most of the compressors with low decompression cost have a compression ratio between one and three, while their decompression cost is roughly within an order of magnitude to the *memcpy* baseline. On the other hand, most compressors with highest compression ratio (three to four) have a two to three orders of magnitude higher decompression cost. The reactor status dataset (NPZ) is an exception, as each original file is $\sim$1.2 KB. Table IV summarizes the compression ratio of `lzsse8`, `lz4hc`, `lzma`, and `xz` on the six datasets. Generally, `lzsse8` and `lz4hc` decompress fast and `lzma` and `xz` have the highest compression ratios.

The constraint of compressor selection is to preserve the compression free performance, thus we prefer compressors with low decompression cost. `lzsse8` and `lz4hc` are such compressors with a non-trivial compression ratio on Intel and POWER9 processors, respectively. So we choose these two as the default compressors on the two architectures.

E. Compressor Selection

In this experiment, we use SRGAN and FRNN to examine how decompression cost translates to real application performance and evaluate the effectiveness of the compressor selection algorithm presented in §VI-B. All experiments are run on four nodes of **GTX** (16 1080 Ti GPUs), **V100** (16 V100 GPUs), and **CPU**.

The numerical algorithm presented in §VI-B requires inputs from three sources: 1) T_{iter}, C_{batch}, and S'_{batch} are application parameters; 2) Tpt_{read} and Bdw_{read} are FanStore performance, and they vary on machines; 3) $Tpt_{decom}(c)$ and compression ratio are from compressors and datasets.



(a) EM (tif) on SKX (b) RS (npz) on SKX (c) EM (tif) on POWER9 (d) RS (npz) on POWER9

Fig. 7: Sample results of 180 compressor configurations on TIF and NPZ datasets, for Intel Xeon Platinum 8160 processors (SKX) and IBM POWER9 processors (POWER9). Green crosses indicate the highest decompression throughputs and red pluses the highest compression ratios.

TABLE V: Inputs to the compressor selection algorithm

App	Cluster	IO	T_{iter}	C_{batch}	S_{batch}
SRGAN	GTX	sync	9689 ms	256	410 MB
SRGAN	V100	sync	2416 ms	256	410 MB
FRNN	CPU	async	655 ms	512	615 KB

TABLE VI: FanStore performance for different file sizes, on four nodes

Cluster	file_size	Tpt_{read}(file/s)	Bdw_{read}(MB/s)
GTX	512 KB	9469	4969
	2 MB	3158	6663
V100	512 KB	8654	4540
	2 MB	5026	10546
CPU	1 KB	29103	30

We first profile SRGAN and FRNN with a small dataset hosted in RAM disk to minimize I/O impact and then use the time cost per iteration as T_{iter}. C_{batch} and S'_{batch} can be derived from application parameters. Table V summarizes the required inputs from applications. Then we profile FanStore's performance with various file sizes across scales on all three clusters Table VI shows only a part of the results, which will be used in subsequent analysis. Inputs from compressors are in §VII-D.

Now we use the inputs from application profile, FanStore benchmark, and compressors to examine the effectiveness of the selection algorithm in three cases.

1) SRGAN on GTX: Since SRGAN uses synchronous I/O, we use Equation 1 to select the compressors. In our hardware setting, 4 GTX nodes have an aggregated local storage space of 240 GB. While the complete dataset is 500 GB. A file of 1.6 MB is expected to be compressed to 762 KB, which is close to 512 KB. Thus we use the row of 512 KB and 2 MB on **GTX** in Table VI. So we are expecting a compression ratio of $500/240 \approx 2.1$. Using Equation 3,

$$T_{read}(C_{batch}, S'_{batch}) = \max\left(\frac{256}{3158}, \frac{410}{6663}\right) = 81\,063\ \mu s$$

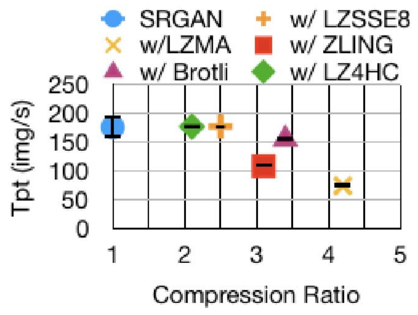
$$T_{read}(C_{batch}, S_{batch}) = \max\left(\frac{256}{9469}, \frac{410/2.1}{4969}\right) = 27\,035\ \mu s$$

With data compression, now we have 81 603–27 035=54 568 μs to decompress 256 files in four-way parallelism. Thus each file has 54 568/256/4=852 μs for decompression.

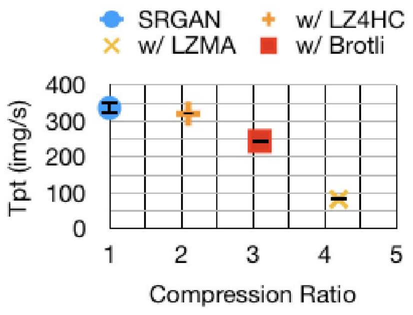
We see in Table VII(a) that `lzsse8` and `lz4hc` meet both the compression ratio and decompression cost constraints. For comparison purposes, we also measure performance with `brotli`, `zling`, and `lzma`, with results shown in Figure 8(a). We see that when using `lzsse8` and `lz4hc`, we achieve identical performance to the baseline, while using compressors with higher decompression costs result in 1.1–2.3× slowdown. From another point of view, if users are bounded by the scale (e.g., if the maximum batch size corresponds to at most four GTX nodes), they may consider trading 10% of performance for the 3.4× more storage capacity offered by `brotli`. Otherwise, hosting the dataset on shared file systems leads to lower performance, as shown in Figure 9(b).

2) FRNN on CPU: FRNN uses asynchronous I/O, so we use Equation 2 to estimate the acceptable decompression cost. In similar calculation as the previous example, the acceptable decompression cost is 4952 μs. This decompression cost can be met by all compressors in the candidate suite. To verify the results, we select compressors with the fastest decompression or highest compression ratio, as shown in Table VII(b). The results in Figure 8(b) show that all three compressors have identical performance to the baseline without compression. Note that the compression ratio of the reactor status dataset is larger than the value estimated from individual files (6.5 for the complete dataset and 2.6 for individual files). This is because each small file takes a fixed block size depending on the specific file system configuration; concatenating them into large chunks can make better use of the storage space.

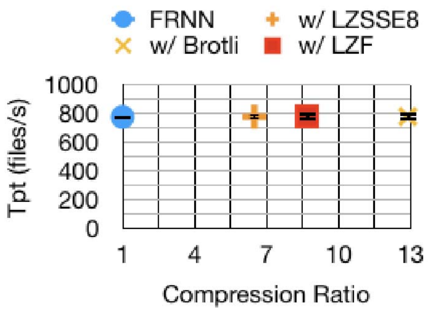
3) SRGAN on V100: SRGAN runs 4× faster, and thus reads files more often, on **V100** than on **GTX**. To sustain the baseline performance with compressed data, FanStore needs a compressor with faster decompression. Using Equation 1 and the data in Table V and VI, we see that it requires a decompression cost that is no larger than 125 μs. `lz4fast` meets the decompression condition, but its compression ratio is close to one. `lz4hc` is the fastest candidate with a



(a) SRGAN on 16 GTX GPUs

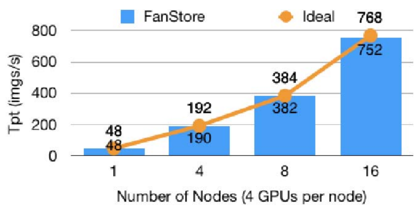


(b) SRGAN on 16 V100 GPUs

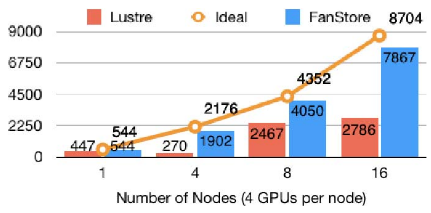


(c) FRNN on 4 SKX CPUs

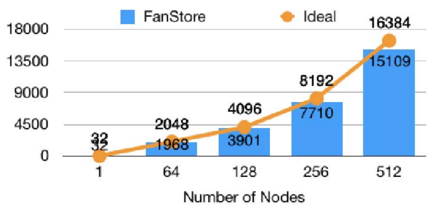
Fig. 8: Application performance with various compressors and processors



(a) SRGAN on GTX



(b) ResNet-50 on GTX



(c) ResNet-50 on CPU

Fig. 9: SRGAN and ResNet performance at scale. Ideal performance is a linear scaling of performance on a single node with data stored in RAM disk.

category	compressor	decom_cost	com_ratio
selected	`lzsse8`	619 ms	2.5
	`lz4hc`	858 ms	2.1
comparison	`brotli`	4741 ms	3.4
	`zling`	17123 ms	3.1
	`lzma`	41261 ms	4.2

(a) SRGAN on GTX

compressor	decom_cost	com_ratio
`lzf`	0.41 μs	8.7
`lzsse8`	0.43 μs	6.5
`brotli`	5.23 ms	13.0

(b) FRNN on CPU

compressor	decom_cost	com_ratio
`lz4hc`	942 us	2.1
`brotli`	5650 us	3.1
`lzma`	43382 ms	4.2

(c) SRGAN on V100

TABLE VII: Selected compressors for three example–platform combinations

compression ratio of two. Thus we select `lz4hc`, `lzma`, and `brotli` to examine the actual impact on the application. The properties of these compressors are in Table VII(c). Figure 8(c) shows the results. `lz4hc` achieves 95.3% of baseline performance, doubling storage capacity for just 4.7% performance loss. `brotli` and `lzma` achieve only 24.6% and 72.8% of baseline, respectively. Another approach, not evaluated here, would be to implement asynchronous I/O, which improves overall performance and enables compression without performance loss.

Discussion. We show the efficacy of the proposed compression selection algorithm empirically. It is a coarse-grained estimation as we simplify the problem. Specifically, the FanStore benchmark only uses one process per node, while in reality, each training process may launch multiple threads. The benchmark is implemented in C, which has a lower overhead compared to the Python runtime, especially for small file access. Even with this simplified approach, the compressor selection algorithm almost always gives appropriate results which meet the compression ratio requirement with the lowest possible decompression cost.

F. Scalability

To showcase the scalability of FanStore, we use SRGAN and ResNet-50 on **GTX** and **CPU**. Figure 9(a) shows the scalability of SRGAN with FanStore using `lzsse8`. The weak scaling efficiency is 97.9% using 64 1080 Ti GPUs compared to the case of one node with four GPUs. Though the ImageNet dataset cannot be compressed further, it is promising to use ResNet-50 to examine the scalability of FanStore. Figures 9(b) and 9(c) show the scaling performance of ResNet-50 on **GTX** and **CPU**. The weak scaling efficiency is 90.4% using 64 1080 Ti GPUs compared to the baseline on a single node. To further show the scalability of FanStore, we run it across scales on **CPU**. The weak scaling efficiency is 92.2% on 512 Intel Xeon Platinum 8160 processors. We also tried the same case using the Lustre file system on 512 nodes, but found that it ran for one hour without starting training. This was because the metadata server of Lustre was processing the metadata workload and did not return.

VIII. Conclusion

We have presented FanStore, a distributed compressed object store that supports massively concurrent file access via a

POSIX-compliant interface. FanStore operates in a new point in the design space of deep learning training performance and storage capacity that allows it to host 2–13× more data on the local storage devices without sacrificing training performance for real-world applications. FanStore scales linearly to hundreds of compute nodes, with the potential to serve thousands of GPUs. Evaluation on three test cases shows that its compressor selection algorithm makes correct decisions based on application and I/O performance. Overall, FanStore enhances the ability of existing supercomputers and clusters to accommodate more DL applications, more efficiently.

In future work we aim to investigate additional applications and compression methods, including lossy compressors such as SZ [24] and ZFP [44] as examined in the CODAR project [45].

Acknowledgments

This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, Contract DE-AC02-06CH11357, the Exascale Computing Project, Project Number 17-SC-20-SC, and by NSF ACI-1540931 and OAC-1818253.

References

[1] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," 2018. [Online]. Available: http://arxiv.org/abs/1802.09941

[2] Y. You *et al.*, "ImageNet training in minutes," in *47th International Conference on Parallel Processing*. ACM, 2018, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225069

[3] V. Codreanu *et al.*, "Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train," *arXiv preprint arXiv:1711.04291*, 2017.

[4] T. Akiba *et al.*, "Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes," *arXiv preprint arXiv:1711.04325*, 2017.

[5] C. Ying *et al.*, "Image classification at supercomputer scale," *arXiv preprint arXiv:1811.06992*, 2018.

[6] X. Jia *et al.*, "Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[7] H. Mikami *et al.*, "ImageNet/ResNet-50 training in 224 seconds," *arXiv preprint arXiv:1811.05233*, 2018.

[8] K. He *et al.*, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[9] Z. Zhang *et al.*, "FanStore: Enabling efficient and scalable I/O for distributed deep learning," *arXiv preprint arXiv:1809.10799*, 2018.

[10] N. Hemsoth, "HPC file systems fail for deep learning at scale," 2018, http://bit.ly/2MgDGSj.

[11] T. Hoefler, "Twelve ways to fool the masses when reporting performance of deep learning workloads," 2018, http://bit.ly/2platwi.

[12] M. Jeon *et al.*, "Multi-tenant GPU clusters for deep learning workloads: Analysis and implications," MSR-TR-2018, Tech. Rep., 2018.

[13] Y. Zhu *et al.*, "Efficient user-level storage disaggregation for deep learning," in *IEEE Cluster*, 2019.

[14] P. Goyal *et al.*, "Accurate, large minibatch SGD: Training ImageNet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[15] Z. Zhang *et al.*, "Aggregating local storage for scalable deep learning i/o," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, Nov 2019, pp. 69–75.

[16] J. Dean *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1223–1231.

[17] A. Kougkas *et al.*, "Leveraging burst buffer coordination to prevent I/O interference," in *12th International Conference on e-Science*. IEEE, 2016, pp. 371–380.

[18] P. Carns *et al.*, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage*, vol. 7, no. 3, p. 8, 2011.

[19] Y. Kim *et al.*, "Workload characterization of a leadership class storage cluster," in *5th Petascale Data Storage Workshop*. IEEE, 2010, pp. 1–5.

[20] N. Mi *et al.*, "Efficient management of idleness in storage systems," *ACM Transactions on Storage (TOS)*, vol. 5, no. 2, p. 4, 2009.

[21] M. Burrows *et al.*, "On-line data compression in a log-structured file system," in *ASPLOS*, 1992, pp. 2–9.

[22] B. Welton *et al.*, "Improving I/O forwarding throughput with data compression," in *International Conference on Cluster Computing*. IEEE, 2011, pp. 438–445.

[23] D. Tao *et al.*, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *International Parallel and Distributed Processing Symposium*. IEEE, 2017, pp. 1129–1139.

[24] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 730–739.

[25] T. Lu *et al.*, "Understanding and modeling lossy compression schemes on HPC scientific data," in *International Parallel and Distributed Processing Symposium*. IEEE, 2018, pp. 348–357.

[26] A. H. Baker *et al.*, "Toward a multi-method approach: lossy data compression for climate simulation data," in *International Conference on High Performance Computing*. Springer, 2017, pp. 30–42.

[27] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[28] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[29] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[30] M. Rabbani, "JPEG2000: Image compression fundamentals, standards and practice," *Journal of Electronic Imaging*, vol. 11, no. 2, p. 286, 2002.

[31] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, Jun. 1984. [Online]. Available: https://doi.org/10.1109/MC.1984.1659158

[32] F. G. Zanjani *et al.*, "Impact of JPEG 2000 compression on deep convolutional neural networks for metastatic cancer detection in histopathological images," *Journal of Medical Imaging*, vol. 6, no. 2, p. 027501, 2019.

[33] S. Dodge and L. Karam, "Understanding how image quality affects deep neural networks," in *8th International Conference on Quality of Multimedia Experience*. IEEE, 2016, pp. 1–6.

[34] S. Pumma *et al.*, "Scalable deep learning via I/O analysis and optimization," *ACM Transactions on Parallel Computing*, vol. 1, no. 1, 2019.

[35] N. Liu *et al.*, "On the role of burst buffers in leadership-class storage systems," in *28th Symposium on Mass Storage Systems and Technologies*. IEEE, 2012, pp. 1–11.

[36] M. Kerrisk and P. Zijlstra, "Linux programmer's manual," *The Linux man-pages project, version*, vol. 3, 2014.

[37] G. Hunt and D. Brubacher, "Detours: Binary interception of Win 3 2 functions," in *3rd Usenix windows NT symposium*, 1999.

[38] H. Dong *et al.*, "TensorLayer: A versatile library for efficient deep learning development," in *Multimedia Conference*. ACM, 2017, pp. 1201–1204.

[39] F. Chollet *et al.*, "Keras," 2015, https://keras.io/.

[40] A. Sergeev and M. D. Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[41] P. Skibinski and et al., "lzbench," https://github.com/inikep/lzbench, 2018.

[42] C. Ledig *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network." in *CVPR*, vol. 2, no. 3, 2017, p. 4.

[43] A. Svyatkovskiy and J. Kates-Harbeck, "PPPL deep learning disruption prediction package," https://github.com/PPPLDeepLearning/plasma-python.

[44] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[45] I. Foster *et al.*, "Computing just what you need: Online data analysis and reduction at extreme scales," in *European Conference on Parallel Processing*. Springer, 2017, pp. 3–19.