Object Storage for Deep Learning Frameworks

Or Ozeri
IBM Research - Haifa
oro@il.ibm.com

Effi Ofer
IBM Research - Haifa
effio@il.ibm.com

Ronen Kat
IBM Research - Haifa
ronenkat@il.ibm.com

ABSTRACT

The advent of big datasets and high speed GPUs is fueling the growth in machine and deep learning techniques. In this paper we explore storing the training data in object storage and demonstrate how this can be done effectively while providing sufficient throughput to high performance GPUs.

CCS CONCEPTS

• **Information systems** → **Cloud based storage**; • **Computing methodologies** → **Machine learning**;

KEYWORDS

Machine Learning, Deep Learning, Object Storage

ACM Reference Format:
Or Ozeri, Effi Ofer, and Ronen Kat. 2018. Object Storage for Deep Learning Frameworks. In *Second Workshop on Distributed Infrastructures for Deep Learning (DIDL '18), December 10–11, 2018, Rennes, France.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3286490.3286562

1 INTRODUCTION

In recent years, machine learning and deep learning techniques have become prevalent in diverse fields including computer vision, speech recognition, natural language processing, social network analysis, bioinformatics, medicine, and others; Producing results comparable to and in some cases surpassing human experts. Enabling these advancements are the availability of big data sets, such as the 14,197,122 images in ImageNet [3] or the 640,000 mammography images from over 86,000 subjects in the Digital Mammography DREAM Challenge [1]. Enabling the processing of all of these data in modern machine learning models are high throughput processing capabilities from high-end GPUs and accelerators which can process these large datasets using parallel and efficient execution of trillions of floating-point operations per second.

For these GPUs to reach their full potential, data delivery must keep up with the rate at which they can consume it. Table 1 shows the storage throughput requirements of various machine learning workloads when running with varying number of Nvidia Volta V100 GPUs. We can see that the workloads requirements are peaking at around 570 MBytes/sec. While CPUs are currently advancing at a rate of about 1.1x performance improvement per year, GPUs are advancing at a rate of about 1.5x per year [7], or 10x performance improvement every 5 to 6 years. So a throughput of 570 MBytes/sec may suffice for the current generation of GPUs, but future generations will require higher throughputs, i.e., 5700 MBytes/sec in 2023 for a single training workload run.

GPUs	Resnet152	Resnet50	VGG11	Alexnet	Speech LSTM
1	28.8	62.9	77.0	246.1	17.8
2	58.3	136.9	122.9	423.9	29.4
4	107.4	224.4	174.4	570.1	64.3
8	180.6	370.6	208.9	526.4	107.0

Table 1: Storage bandwidth in MBytes/sec of popular deep learning workloads running with varied numbers of GPUs

High-end storage system can satisfies the requirements stated in Table 1 today and possibly will continue doing so in the near future. However with the volume, variety, and velocity of training data in an AI driven world, the cost of storing data on such system is becoming increasingly higher. In this paper we explore how capacity oriented storage system like object storage, which can scale up at a lower cost of ownership, can satisfy the requirements of deep learning training workloads.

2 STORAGE FOR DEEP LEARNING

In order to feed GPUs with data at the required high throughput, deep learning systems, such as TensorFlow, Caffe, and PyTorch, traditionally use a POSIX file system interface to access training data. Furthermore, the data usually resides locally on the same machine as the GPUs. However, this model does not scale with the number of users, the volume of data, and the variety of workloads that are running on deep learning systems today.

While moving the data to a high-end storage system can be the answer in some use-cases, we explored the use of object storage services which are available either on-prem or in the cloud and offer scalable and inexpensive storage for structured and unstructured data. Such systems include IBM Cloud Object Storage, Amazon S3, OpenStack Swift, and others.

Deep learning systems were originally designed to use a POSIX file system interface and are optimized for it. Taking advantage of object storage requires more than just adding support for its REST based interface and unique APIs, as has been done in TensorFlow [4]. Object storage requires different semantics than a file system [10]. Object storage is not designed for high performance single threaded data transfers. Furthermore, standard practices for feeding deep learning models data from disaggregated and remote object storage can result in poor training performance.

3 IMPLEMENTATION

To enable deep learning frameworks to access object storage, we have added a FUSE [2] based file system, s3fs [9], to our deep learning stack. s3fs is an open source code that translates POSIX

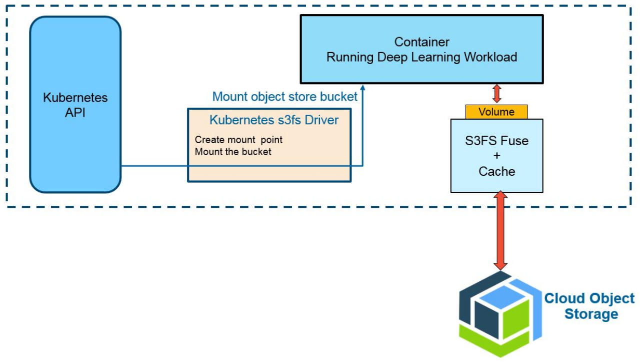

Figure 1: Accessing object storage from deep learning frameworks using s3fs

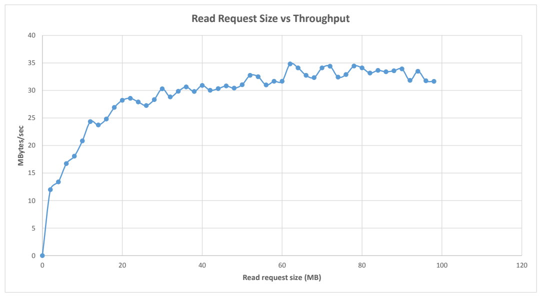

Figure 2: Effects of request size on the throughput

API requests into REST API requests to the object storage; Providing a file system implementation which is actaully backed up by an object store and making object storage access transparent to the deep learning framework. s3fs leverages a local cache to hold objects that are accessed. In our work we augmented the s3fs cache and logic to provide high throughput data transfer for the GPUs. The improvements and new logic that was created in this work were contributed to the s3fs open source repository and are now part of the s3fs standard distribution.

The s3fs file system can yield good performance results when reading or writing files sequentially. However as s3fs is backed up by an object storage it does not fit workloads with frequent updates, such as transactional workloads. For example, a small update operation to an existing file will result in writing the whole file again as objects are immutable and an update translates to a copy-like operation. In addition, data locking is not supported in s3fs, so it is less suitable for workloads which require concurrency control.

Moreover, as s3fs caches the data it reads but lacks a synchronization mechanism between the remote objects and locally cached files, s3fs is not suitable for accessing data that is not immutable. Deep learning workloads are a good fit for s3fs since the training portion of the workloads makes use of immutable data. The training data resides in object storage with s3fs enabling access to the data through a POSIX interface. The results of the training are then written back into the object storage. Figure 1 shows how our s3fs deployment fits into the context of a deep learning framework running on Kubernetes. To reduce the number of data hops, s3fs runs locally on each learner node and have a local in-memory cache in order to provide high throughput.

4 PERFORMANCE CONSIDERATIONS

Object storage services, such as IBM Cloud Object Storage, are able to support a large number of concurrent requests from multiple users, while their single threaded sequential performance is capped well below the total available bandwidth. This behavior is driven by the inherent distributed nature of object storage which distribute the data chunks across multiple nodes and storage devices. While a single request is capped, when serving multiple requests, an object storage can easily saturate the network between the storage and the client issuing the requests. In order to deliver high throughput we leverage the aggregated throughput of multiple requests by converting a read request issued by the deep learning framework into multiple concurrent requests.

Our deployment of s3fs converts each client read requests into multiple concurrent read requests. We have configured s3fs to issue multiple concurrent read requests[1], which provides an overall read throughput from the object store that is comparable to local disk access. The maximum number of concurrent read requests can be set in s3fs using the parallel_count parameter.

In our deployment we have chosen to define our data chunk size to be 52 MB. The chunk can be smaller when the object is smaller than 52 MB or when it is the last chunk in an object which does not divide evenly by 52 MB. The chunk size is a parameter that affects latency and throughput. A smaller chunk size reduces latency but can result in lower throughput if the chunk size is too small. In our performance evaluation, as shown in figure 2, we observed that we reached a single reader throughput of 35 MBytes/sec for the object storage environment we used for our evaluation. Furthermore, for workloads that read just a small portion of the data from an object, keeping a smaller chunk size is advantageous. A smaller chunk size also makes it possible to increase the number of concurrent read requests. We experimented with different chunk sizes and found that the optimal chunk size for our environment is around 50MB, different environments will need to be configured for different chunk sizes. Our object storage best practices suggest using range read requests that are multiples of 4 MB and thus we set the chunk size to 52 MB. The s3fs chunk size can be configured using the multipart_ size parameter.

Converting POSIX read requests into multiple object storage chunk read request speeds up data delivery and also enables a type of rudimentary read ahead prefetching since rather than read just the range specified by the read request, we read entire objects segmented into chunks. In the case of deep learning workloads where generally the entire dataset is used, this proves highly beneficial. Moreover, while latency is generally linear to the request size, as shown in figure 3, small objects suffer disproportionately from

[1]In our specific configuration we set the number of concurrent requests to twenty.

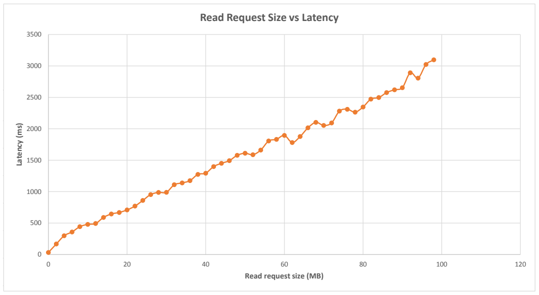

Figure 3: Effects of request size on latency

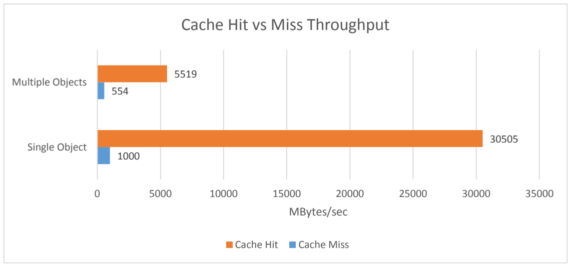

Figure 4: Impact of the cache on throughput

the overhead of the time-to-first-byte from the object storage service. But when data is packaged into larger objects, the read-ahead feature of the cache avoids this overhead all together.

To avoid reading the same data multiple times, we have deployed an in-memory cache using the Linux kernel page cache [8]. Chunks read from the object storage are stored inside this cache and are served back to the deep learning framework asynchronously. Since deep learning frameworks often run their training in multiple epochs this results in substantial speed improvement as long as the data fits within the cache. Choosing the appropriate cache size depends therefore on the expected workloads, and is a balancing act between the cost of providing the cache and the improved throughput to the GPUs.

To facilitate low latency and high throughput, the cache is kept in-memory and is collocated with the deep learning compute nodes. To keep it simple, it is not shared between multiple nodes or users. A future expansion of a cache would be to enable a shared cache. We cache both objects as well as meta data. Typical file system usage makes frequent access to metadata via a stat() system call which maps to an HTTP HEAD request to the object storage system. By caching the objects meta data we are able to reduces the number of redundant HTTP HEAD requests and reduce the amount of time it takes to list a directory or retrieve file attributes. We have configured s3fs to cache the meta data of 100,000 objects, with each cached entry taking up to 512 bytes of memory. By bounding the number we ensure that the meta data does not consume more than approximately 50 MB of the cache memory. It is possible to update the limit using the max_stat_cache_size parameter in s3fs.

5 DEPLOYMENT EXPERIENCES

Our FUSE based architecture has been implemented in a deep learning service offering on the IBM Cloud and our s3fs enhancements have been contributed to the s3fs project repository. We evaluated the performance of our solution using the TensorFlow deep learning framework with the training data residing in an IBM Cloud Object Storage instance on the IBM Cloud.

We measured the throughput of reading a single file using a single client thread for a non-cached object. The base throughput we get when using the native implementation of reading from object storage is around 35 MB/sec. Using our solution, the throughput improves close to linearly with the size of the read object up to objects of about 1 GB in size. Reading of small files is relatively slow compared to reading from a local disk due to time-to-first-byte from the object storage service. But with larger objects, such as TensorFlow recommended TRecords format [5], s3fs is able to make use of additional concurrent connections and thus throughput improves.

When reading a single object we achieve a maximum throughput of 544 MBytes/sec for cache miss requests. However, when reading multiple objects concurrently, we are able to get an aggregate throughput of 1000 MBytes/sec. The difference between reading a single object compared to multiple object is due to the way data is copied into the client thread. In our testing, each object is about 200 MB and the bare metal machines we use are equipped with a 10 Gbit/sec NIC. For comparison, the same set of objects can be downloaded, using concurrent read threads, with Python HTTP library (httplib) at 1158 MBytes/sec. While s3fs achieves very good performance, compared to a native HTTP read throughput, its performance is impacted by overheads from the FUSE file system and client side copy.

Incorporating a memory cache in each compute node, has proven to boost throughput dramatically. For a cache hit, the throughput increases from 544 MBytes/sec to 5519 MBytes/sec for a single object, and from 1000 MBytes/sec to 30505 MBytes/sec for multiple objects, as shown in Figure 4. In our usage scenario data is usually read into the cache as part of the first epoch of a machine learning job. Since we utilize read ahead prefetching, it is not uncommon to observe cache hits even on the first epoch. Subsequent epochs are always able to take advantage of the data already in the cache, unless the dataset size is larger than the cache total size.

6 CONCLUSIONS

The challenge in using object storage for deep learning frameworks lies in providing sufficient data throughput to the GPUs. Using our s3fs based architecture with concurrent reads, read ahead prefetching, and an in-memory cache we are able to surpass this number as well as make allowance for faster future generation of GPUs. Our solution is running in production in IBM's Deep Learning Service on the IBM Cloud, since its official launch on March 2018.

REFERENCES

[1] 2018. The Digital Mammography DREAM Challenge. https://www.synapse.org/#!Synapse:syn4224222/wiki/401743
[2] 2018. FUSE. https://github.com/libfuse/libfuse
[3] 2018. ImageNet. http://www.image-net.org/

[4] 2018. TensorFlow on S3. https://www.tensorflow.org/deploy/s3
[5] 2018. TensorFlow Performance Guide. https://www.tensorflow.org/performance/performance_guide
[6] David Kung. 2018. *Storage bandwidth of popular deep learning workloads*. Technical Report. IBM.
[7] Nvidia 2018. Supercharged computing. https://www.nvidia.com/en-us/about-nvidia/ai-computing/.
[8] David Povet and Marco Cesati. 2005. *Understanding the Linux Kernel* (3rd ed.). O'Reilly Media.
[9] Randy Rizun. 2018. s3fs-fuse. https://github.com/s3fs-fuse/s3fs-fuse
[10] G Vernik, M Factor, E Kolodner, P Michiardi, E Ofer, and F Pace. 2018. Stocator: Providing High Performance and Fault Tolerance for Apache Spark over Object Storage. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) (2018)*. IEEE, Washington, DC, 462–471. http://doi.ieeecomputersociety.org/10.1109/CCGRID.2018.00073