This article presents the collaborative work of Alibaba, Alluxio, and Nanjing University in tackling the problem of Artificial Intelligence and Deep Learning model training in the cloud. We adopted a hybrid solution with a data orchestration layer that connects private data centers to cloud platforms in a containerized environment. Various performance bottlenecks are analyzed with detailed optimizations of each component in the architecture. Our goal was to reduce the cost and complexity of data access for Deep Learning training in a hybrid environment, which resulted in over 40% reduction in training time and cost.

© Copyright Alluxio, Inc. All rights reserved. Alluxio is a trademark of Alluxio, Inc.
The rising popularity of artificial intelligence (AI) and deep learning (DL), where artificial neural networks are trained with increasingly massive amounts of data, continues to drive innovative solutions to improve data processing. Distributed DL model training can take advantage of multiple technologies, such as:

- Cloud computing for elastic and scalable infrastructure
- Docker for isolation and agile iteration via containers and Kubernetes for orchestrating the deployment of containers
- Accelerated computing hardware, such as GPUs

The merger of these technologies as a combined solution is emerging as the industry trend for DL training.

Data access challenges of the conventional solutions

Data is often stored in private data centers rather than in the cloud for various reasons, such as security compliance, data sovereignty, or legacy infrastructure. A conventional solution for DL model training in the cloud typically involves synchronizing data between the private data storage and its cloud counterpart. As the size of data grows, the associated costs for maintaining this synchronization becomes overwhelming:

- **Copying large datasets:** Datasets continually grow so eventually it becomes infeasible to copy the entire dataset into the cloud, even if copying to a distributed high performance storage such as GlusterFS.
- **Transfer costs:** New or updated data needs to be continuously sent to the cloud to be processed. Not only does this incur transfer costs, it is also costly to maintain because it is often a manually scripted process.
- **Cloud storage costs:** Keeping a copy of data in the cloud incurs the storage costs of the duplicated data.

A hybrid solution that connects private data centers to cloud platforms is needed to mitigate these unnecessary costs. Because the solution separates the storage and compute architectures, it introduces performance issues since remote data access will be limited by network bandwidth. In order for DL models to be efficiently trained in a hybrid architecture, the speed bump of inefficient data access must be addressed.
We designed and implemented a model training architecture based on container and data orchestration technologies as shown below:

1. **Kubernetes** is a popular container orchestration platform, which provides the flexibility to use different machine learning frameworks through containers and the agility to scale as needed. Alibaba Cloud Kubernetes (ACK) is a Kubernetes service provided by Alibaba Cloud. It can run Kubernetes workloads on CPUs, GPUs, NPUs (including optical 800 chips), and Shenlong bare metal instances on the Alibaba Cloud platform.

2. **Kubeflow** is an open source Kubernetes-based cloud-native AI platform used to develop, orchestrate, deploy, and run scalable, portable machine learning workloads. Kubeflow supports two TensorFlow frameworks for distributed training, namely the parameter server mode and AllReduce mode. Based on Arena developed by Alibaba Cloud Container Service Team, users can submit these two types of distributed training frameworks.

3. **Alluxio** is an open source data orchestration system for hybrid cloud environments. By adding a layer of data abstraction between the storage system and the compute framework, it provides a unified mounting namespace, hierarchical cache, and multiple data access interfaces. It is able to support efficient data access for large-scale data in various complex environments, including private, public, and hybrid cloud clusters.
The original design of the Alluxio project aimed to solve performance bottlenecks caused by various compute frameworks accessing filesystems storing data on disk, such as HDFS. The project was started in UC Berkeley AMPLab and was made open source in 2013. After many years of continuous development, Alluxio’s virtual distributed filesystem is emerging as the industry’s mainstream solution to solve I/O performance problems on the cloud, bridging the data access chasm for hybrid cloud solutions.

Recently introduced features in Alluxio have further cemented its utility for machine learning frameworks. A POSIX filesystem interface based on FUSE provides an efficient data access method for existing AI training models. Helm charts, jointly developed by the Alluxio and Alibaba Cloud Container Service teams, greatly simplify the deployment of Alluxio in the Kubernetes ecosystem.
In our initial experiments, we ran a DL workload including Alluxio in the stack. As we scaled the compute hardware by introducing more GPU cores, we found that Alluxio did not scale as expected and thoroughly analyzed the observed performance degradations surrounding Alluxio.

Deep learning experiment environment

- We used the ResNet-50 model and the ImageNet dataset. The size of the dataset is 144GB and the data is stored in TFRecord format. Each TFRecord size is about 130MB. Each GPU’s batch_size is set to 256.
- The model training hardware selected is 4 V100 highly equipped with GPU models for a total of 32 GPU cards.
- The data is stored in the Alibaba Cloud object storage service. The model training program reads the data through Alluxio which automatically caches the data in memory. Each machine configured 40GB of memory for a total of 160GB of memory storage. Note that none of the data was preloaded into Alluxio.

Initial performance

In the performance evaluation, we found that upgrading from NVidia P100 to NVidia V100 resulted in a 3x improvement in the training speed of a single card. This computing performance improvement puts additional pressure on data storage access, which also poses new performance challenges for Alluxio’s I/O.

This bottleneck can be quantified by comparing the performance of Alluxio with a synthetic data run of the same training computation. The synthetic run represents the theoretical upper limit of the training performance with no I/O overhead since the data utilized by the training program is self generated. The following figure measures the image processing rate of the two systems against the number of GPUs utilized.
Initially both systems perform similarly but as the number of GPUs increase, Alluxio noticeably lags behind. At 8 GPUs, Alluxio is processing at 30% of the synthetic data rate. While monitoring resource utilization, we observed that we were not hitting the limits of the system’s computing, memory, or network resources. This illustrates that using Alluxio’s default settings is not suitable for this training scenario using 8 GPUs and further tuning is needed.

Analysis

To investigate what factors are affecting performance, we analyzed Alluxio’s technology stack as shown below and identified several major areas of performance issues.

FILESYSTEM RPC OVERHEAD

Alluxio file operations require multiple RPC interactions to fetch the requested data. As a virtual filesystem, its master processes manages filesystem metadata, keeps track of where data blocks are located, and fetches data from underlying filesystems. Even in the best case scenario where an Alluxio client requests for data that is cached in a local Alluxio worker, multiple RPCs are involved, such as checking with the master node to ensure data consistency.

DATA CACHING AND EVICTION STRATEGIES

As Alluxio reads data from the underlying storage system, the worker will cache data in memory. As the allocated memory fills up, it will choose to evict least recently accessed data. Both of these operations are asynchronous and can cause a
noticeable overhead in data access speeds, especially as the amount of data cached on a node nears saturation. This is also exacerbated by the fact that if the requested data is cached by a remote Alluxio worker, the Alluxio worker local to the client may cache its own local copy as it retrieves the data from the remote worker.

**ALLUXIO AND FUSE CONFIGURATION FOR HANDLING NUMEROUS CONCURRENT READ REQUESTS**

Reading files through the FUSE interface is fairly inefficient using its default configuration. FUSE reads are handled by the libfuse non-blocking thread pool that does not necessarily recycle its threads. Since the FUSE kernel module acts as a bridge for the application to connect to the Alluxio filesystem, each time the application reads a inode or dentry of a file or directory, the FUSE kernel module needs to make a request to the Alluxio filesystem.

When the Alluxio client passes data to FUSE, the data ends up being copied multiple times due to the API limitations of third party Java libraries used by the Alluxio and Fuse integration, which will be referred to as AlluxioFUSE hereinafter. Similarly, these third party libraries are only compatible with older versions of FUSE, which do not handle high concurrency scenarios well.

Alluxio currently only supports `direct_io` mode in FUSE as opposed to `kernel_cache` mode which would further improve I/O efficiency with the help of page cache. This is because Alluxio’s integration requires each thread to use its own file input handle in a multi-threaded scenario.

**IMPACT OF RUNNING ALLUXIO IN CONTAINERS ON ITS THREAD POOL**

Running Java 8 in a containerized environment may not fully utilize a machine's CPU resources due to how the size of thread pools is calculated. Before patch 191 for Java 8, thread pools would be initialized with a size of 1, which severely restricts concurrent operations.
Data access in a hybrid environment is a complex problem with multiple facets to optimize on. Given our use case of DL model training, we will focus on improving performance and data consistency of read-only data sets, rather than more flexible scenarios that involve simultaneous reading and writing operations or streaming datasets. Based on the analysis above, we will examine the FUSE, Alluxio, and Java on Kubernetes layers for:

- Resources constraints such as limited thread pools and suboptimal JVM configurations in a containerized environment
- Use of different caching layers such as in FUSE and Alluxio metadata
- Unnecessary overhead such as metadata operations

### Optimization of FUSE

#### UPGRADE LINUX KERNEL VERSION

FUSE implementation is divided into two layers: libfuse running in user mode and FUSE Kernel running in kernel mode. The latest version of the Linux Kernel has made a lot of optimizations for FUSE. After upgrading the Linux Kernel from 3.10 to 4.19, we found a 20% improvement in read performance.

#### OPTIMIZE FUSE PARAMETERS

**Extend FUSE metadata validity time**

Each operation on a file in Linux needs to obtain two pieces of metadata: struct dentry and struct inode. The FUSE kernel module needs to request for this metadata from Alluxio which would stress the Alluxio Master in high concurrency situations. To avoid making these requests to Alluxio, the metadata can be cached with a longer expiration period by setting the options: `–o entry_timeout=T –o attr_timeout=T`

**Avoid CPU overhead caused by frequent thread creation and destruction**

FUSE sets a maximum number of worker threads to process requests; this can be configured with `max_idle_threads` and defaults to 10. When this value is exceeded, threads will be deleted instead of being recycled. This has significant performance implications when the application process makes a large number of concurrent requests. Unfortunately, this configuration setting is only supported in libfuse3, whereas AlluxioFUSE only supports libfuse2. To workaround this problem, we patched libfuse2 to support configuring `max_idle_threads`.

---

/ 8
Optimization of Alluxio

The FUSE process interacts with Alluxio by calling the embedded Alluxio client during runtime. For DL scenarios, we need to customize FUSE specific Alluxio properties to optimize performance.

AVOID FREQUENT CACHE EVICTION

Alluxio’s default caching strategy is to cache incoming data as it is read from the underlying storage system. This is optimal for scenarios where a subset of the stored data is read so that when the same data is requested at a subsequent time, it can be quickly retrieved from cache. This is not the case for DL training because it typically reads the entire dataset. This amount of data easily overwhelms Alluxio’s cache capacity as datasets are generally on the order of terabytes. There are several properties we can tune to how Alluxio caches data to optimize for DL training.

**alluxio.user.ufs.block.read.location.policy**

This property determines how to manage blocks as they are cached. The default value is `alluxio.client.block.policy.LocalFirstPolicy`, which means that Alluxio will cache the data on an Alluxio worker local to the Alluxio client requesting for the data. When the worker cache saturates, the worker will start to evict blocks to make space for newer blocks. We can set this property to `alluxio.client.block.policy.LocalFirstAvoidEvictionPolicy`, which will avoid evicting data to reduce the load on each individual worker.

**alluxio.user.file.passive.cache.enabled**

This property determines if an Alluxio worker local to the Alluxio client should cache its own copy when reading data from another Alluxio worker. This property is enabled by default to maximize data locality. We can set this property to `false` to avoid caching additional copies.

**alluxio.user.file.readtype.default**

This property determines how Alluxio handles data that it reads from the underlying storage system. The default value is `CACHE_PROMOTE` which will cache the newly read data in the highest tier. If the highest tier is full, then Alluxio will move older blocks into a lower tier. This scenario is expected to occur when reading a large training dataset that easily exceeds the capacity of the memory tier. The movement of blocks from one tier to another introduces significant overhead especially due to numerous locking operations required to safely migrate blocks. To avoid this situation, we set the read type to `CACHE`.
REduce RPCs TO THE Alluxio Master

While running DL training, the metadata for each data file used is read before starting the actual training task. It is not necessary to request for this metadata throughout the entire DL training task since the dataset is unchanged. This can be cached by setting alluxio.user.metadata.cache.enabled to true. The cache size and expiration time, determined by alluxio.user.metadata.cache.max.size and alluxio.user.metadata.cache.expiration.time respectively, should be set to keep the cached information relevant throughout the entire workload.

A recurring RPC that occurs between Alluxio masters and workers is a periodic heartbeat. The heartbeat interval can be extended to reduce the frequency of updates by setting the property alluxio.user.worker.list.refresh.interval to be 2 minutes or longer.

The Alluxio master keeps track of when files were last accessed. In the DL training workload, the dataset is constant and we don’t necessarily care about tracking this information. We introduced a new property flag to disable this feature to improve the performance of the Alluxio master.

TAke advantage of Data Locality

Data locality aims to move the compute process to the node where the data is located to avoid network data transmission as much as possible. This is vital in a distributed parallel computing environment. In a containerized environment, this can be achieved with two short-circuit read and write methods: Unix socket method and direct file access method.

Enabling unix sockets allows the Alluxio client and worker containers to achieve locality without requiring them to run with the same resources. Although the two are isolated, the performance is worse than direct file access and it could cause errors like Netty’s OutOfDirectMemoryError.

The direct file access method requires that the hostnames and IP addresses of Alluxio Worker and AlluxioFUSE to be the same, so that the Alluxio client and worker share the same cache directory. We decided to pursue this method.

Optimization for Java & Kubernetes

Configure ActiveProcessorCount

Since Alluxio runs on Java 8, various thread pool size calculations depend on the result of Runtime.getRuntime().availableProcessors(), which computes the number of cores the process can utilize by via cpu_shares()/1024 rounded up to 1 if less than. Because the default value of cpu_shares() is 2 in Kubernetes,
The total training time of the workload takes 65 minutes when using Alluxio on four machines with eight cards each, which is very close to the synthetic data scenario that takes 63 minutes. Compared with training via SSD on the cloud, Alluxio saves 45 minutes in time and 40.9% in costs.

ADJUST GC, JIT THREAD

The default GC of the JVM and the number of JIT compilation threads depends on the value of \(-XX:ActiveProcessorCount\), but it can also be configured through parameters such as \(-XX:ParallelGCThreads\) \(-XX:ConcGCThreads\) \(-XX:CICompilerCount\). These should be set to be a small number to avoid performance degradation from frequent preemptive switching of these processes.

Results

After optimizing Alluxio, the single-machine eight-card training performance of ResNet50 is improved by 236.1%. The performance gains are reflected in the four-machine eight-card scenario with the performance loss of 3.29% when compared against the synthetic data measurement. Compared to saving data to an SSD cloud disk in a four-machine eight-card scenario, Alluxio’s performance is better by 70.1%.

The total training time of the workload takes 65 minutes when using Alluxio on four machines with eight cards each, which is very close to the synthetic data scenario that takes 63 minutes. Compared with training via SSD on the cloud, Alluxio saves 45 minutes in time and 40.9% in costs.
In this article, we present the challenges of using Alluxio in a high-performance distributed deep learning model training and dive into our work in optimizing Alluxio. Various optimizations were made to improve the experience of AlluxioFUSE performance in high concurrency reading workload. These improvements enabled us to achieve a near optimal performance when executing a distributed model training scheme in a four-machine eight-card scenario with ResNet50.

For future work, Alluxio is working on enabling page cache support and FUSE layer stability. Alibaba Cloud Container Service team is collaborating with both the Alluxio Open Source Community and Nanjing University through Dr. Haipeng Dai and Dr. Rong Gu. We believe that through the joint effort of both academia and the open-source community, we can gradually reduce the cost and complexity of data access for Deep Learning training when compute and storage are separated to further help advance AI model training on the cloud.
Special thanks to Dr. Bin Fan, Lu Qiu, Calvin Jia, and Cheng Chang from the Alluxio team for their substantial contribution to the design and optimization of the entire system. Their work significantly improved the metadata cache system and advanced Alluxio’s potential in AI and Deep Learning.

AUTHORS

Yang Che, Sr. Technical Expert at Alibaba Cloud. Yang is heavily involved in the development of Kubernetes and container related products, specifically in building machine learning platforms with cloud-native technology. He is the main author and core maintainer of GPU shared scheduling.

Rong Gu, Associate Researcher at Nanjing University and core maintainer of Alluxio Open Source project. Rong received a Ph.D. degree from Nanjing University in 2016 with a focus on big data processing. Prior to that, Rong worked on R&D of big data systems at Microsoft Asia Research Institute, Intel, and Baidu.