ALLUXIO

When AI Meets Alluxio at Bilibili

Building an Efficient AI Platform for Data Preprocessing and Model Training

Lei Li, *AI Platform Lead*
Zifan Ni, *Senior Software Engineer*

What's Inside

Lei Li, AI Platform Lead, and Zifan Ni, Senior Software Engineer from Bilibili, share how they applied Alluxaio to their AI platform to increase training efficiency, as well as best practices including technical architecture and specific tuning tips.

CASE STUDY

1 / Overview

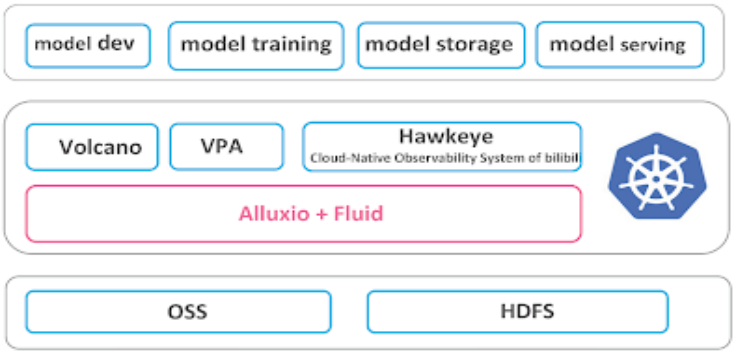
About Bilibili

Bilibili (NASDAQ: BILI) is a leading video community with a mission to enrich the everyday life of the young generations in China. With our initial website launch in June 2009 and official branding as ‘‘Bilibili’’ in January 2010, we have evolved from a content community inspired by anime, comics and games (ACG) into a full-spectrum video community covering a wide array of interests from lifestyle, games, entertainment, anime and tech & knowledge to many. In the last financial report, our MAU reached 230 million.

Bilibili’s Coeus AI Platform

Coeus is our self-developed cloud-native AI platform of Bilibili. Currently, Coeus supports a wide range of use cases, including Ads, CV, NLP, Speech, e-commerce, etc.

From a functional perspective, Coeus supports model development, model training, model storage, and model serving.



The above diagram depicts the architecture and components. Coeus is implemented on Kubernetes and integrates with many cloud-native components, including Volcano, VPA, Hawkeye (a self-developed cloud-native observability system), Alluxio and Fluid.

Coeus uses Alluxio to bridge the underlying storage systems (OSS and HDFS) and the AI applications (video and image training jobs based on Pytorch and Tensorflow).

2 / Our Challenges and Alluxio's Solution

Alluxio is an open source project with a large and vibrant community. Alluxio provides a data layer between storage and compute engines for large-scale analytics and machine learning. This data layer provides virtualization across data sources to serve data to applications, whether in the cloud or on-premises, bare metal or containerized.

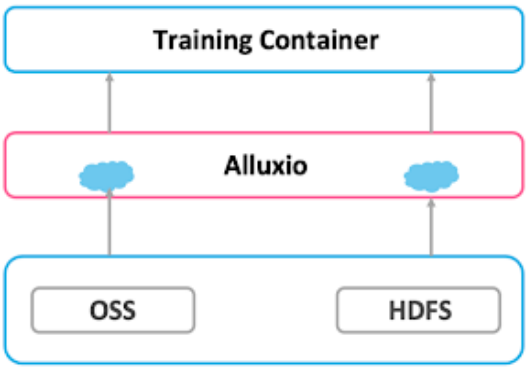
We use Alluxio as an intermediate layer between computation and storage of our AI platform. We came across four major challenges before adopting Alluxio and Alluxio has helped us overcome them.

Challenge 1: Container Crashes

Before Alluxio

The training data is downloaded through the container and read locally during the training phase. However, if the container crashes during the download process, we need to restart the container to download the data again, which is a huge waste of time.

Alluxio's Solution: Caching



Alluxio can hold huge data in a distributed way. Alluxio workers serving as distributed caching helps handle all the data management logic to make the model training read data as if it's local.

Because the data is cached by Alluxio, even if the container crashes, no data will be lost. Upon restarting the container, users can access the original data without having to download it again.

Challenge 2: Users Had To Change The Code Of Applications To Access OSS And HDFS

Before Alluxio

The second challenge is on the application side. Users (algorithm engineers) have to (re)write the code of training jobs for OSS and HDFS to access data in the pipeline.

Alluxio's Solution: FUSE API + Unified Namespace

After adopting Alluxio, it simplifies data reading by providing FUSE API and unified namespace.

FUSE is compatible with the POSIX interface so that algorithm engineers don't have to rewrite any code to access data through Alluxio. By doing so, users are not burdened with the details of reading data.

Unified namespace allows data access as simple as doing configurations. With both OSS and HDFS mounted to Alluxio, the single unified namespace logically decouples the applications from storage, so that the AI applications simply communicate with Alluxio while Alluxio handles the communication with the different underlying storage systems on applications' behalf.

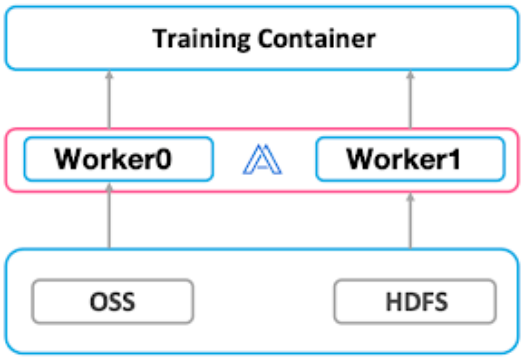
Challenge 3: Data is Too Large to Fit into One Single Machine

Before Alluxio

We have a huge amount of training data, which is too large to fit on a single machine. Also, we need to implement data reading and data retransmission components to ensure that the data is read correctly during the model training phase.

Alluxio's Solution: Distributed Caching on Alluxio Workers

Alluxio itself is a distributed file system. Having multiple workers caching data solves the capacity limitation of a single machine.

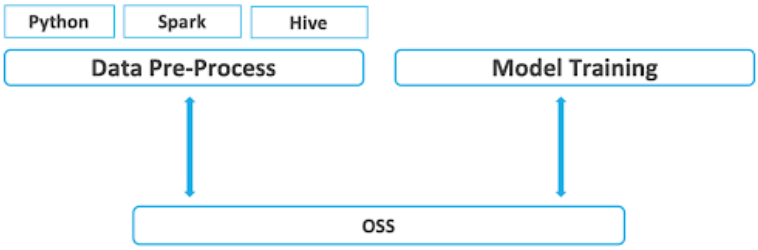


Challenge 4: Repeatedly Pulling Data From Remote Storage Is Slow

Before Alluxio

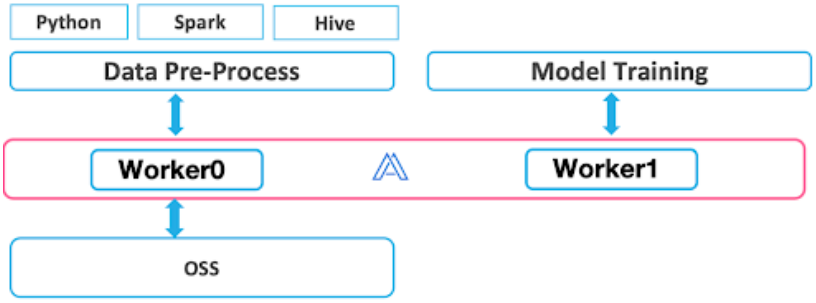
Our algorithm engineers pre-process data by using Jupyter service on the AI platform. For pre-processing, they use Python to pre-process image files and then write the results back to OSS; or, they use Spark or Hive for pre-processing and write the results back to HDFS. During model training, they had to pull the pre-processed data from the remote storage again.

Without a caching layer, they had to write the results of pre-processed data to the under storage and read the data from the storage for model training, which was slow, and also have burdened the under storage.



Alluxio's Solution: Shared Cache as an Intermediate Data Layer

Our AI platform uses Alluxio as a data sharing layer across pre-processing and model training, as well as the buffer between applications and storage. We load data into the Alluxio cluster in Kubernetes during pre-processing. Then, Alluxio serves as an intermediate caching layer on top of HDFS and OSS. This reduces the pressure on remote accessing storage, and speeds up the entire model training process.



3 / Best Practices Using Alluxio

We have done a lot of work on the deployment and optimization of Alluxio to achieve the best outcome possible.

Deployment

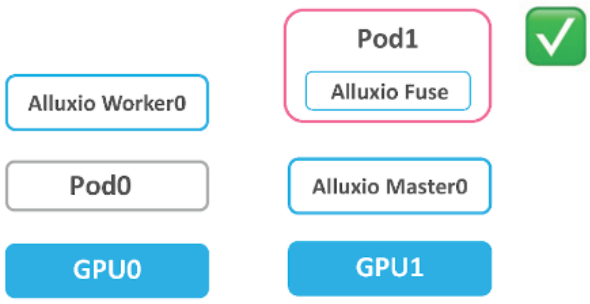
We use serverless mode to start Alluxio FUSE to ensure the maximum utilization of resources and the stability of service during the training cycles. We choose serverless because of the following typical scenario.

Let's say, there are two machines, GPU 0 and GPU 1, and we have launched the Alluxio Cluster on these two machines. The Alluxio FUSE is on the same machine as the Alluxio Worker. On top of that, Pod 0 is occupying the GPU 0 to perform some training tasks.

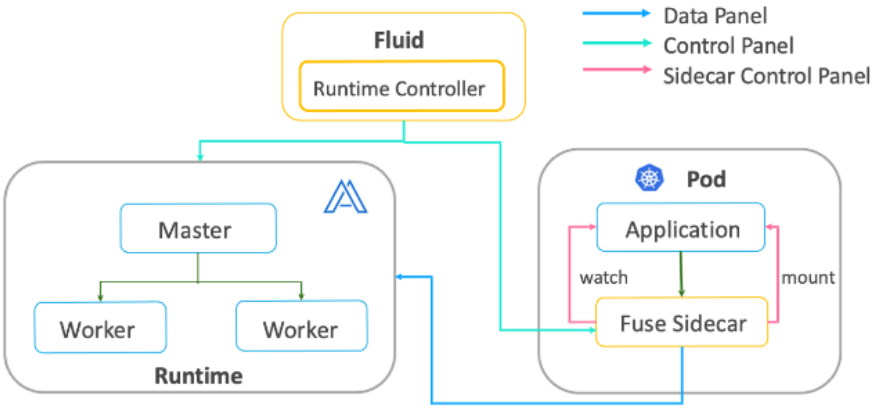
Now, a newly-created pod (pod 1) needs to use the cache on GPU 0 and GPU resources. Since pod 0 has already taken over the GPU resources and cache on GPU 0, pod 1 must wait for pod 0 to release before it can read data from the cache properly. It is unacceptable because pod 0 may take up resources for a long time without releasing them. It is impossible to schedule pod 1 to the node on the right-hand-side as there's no Alluxio FUSE service and no Alluxio cache. This is the limitation of the serverful model, as the service is tied to machines.

Another option is to deploy Alluxio FUSE on all nodes. However, since each FUSE is a separate pod, starting these pods will cause a waste of resources.

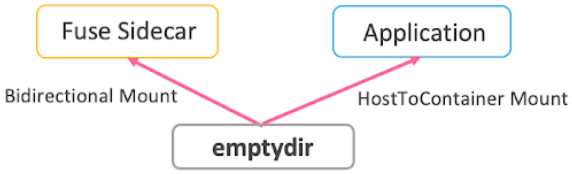
Therefore, we use serverless FUSE. Serverless FUSE will start FUSE as part of the training task pod. This ensures that the pod is started on the same node with FUSE, regardless of which node it is scheduled to by Kubernetes.



After all the nodes are started and deployed, when the user submits a task, both the task and the FUSE process are injected into the training container as sidecars. The FUSE Sidecar mounts the FUSE Path and Host Path in both directions, while the task sidecar accesses cached data in Alluxio via this path.



At the same time, two processes start along with the FUSE sidecar. The first process monitors the lifecycle of the training container. If the training container terminates, this process will stop the FUSE sidecar. The second process monitors the health status of the FUSE sidecar. It ensures the availability of the FUSE process.



Tuning

While deploying Alluxio in production, we found that we need to optimize the configuration of Alluxio to achieve the best functionality and performance.

Problem	Solution
1.Master stop world gc 30s / 30m	• Java: MaxGCPauseMillis 👇, ParallelGCThreads👆 • Alluxio Master: Memory request👆
2. Fuse frequently throw I/O exception	• alluxio.user.rpc.retry.max.duration👆 • alluxio.user.rpc.retry.base.sleep👆
3. Poor performance in reading bunches of small files	• Upgrade Alluxio >= 2.6.2

Large Amount of Metadata Introduces GC to the Alluxio Master

When we launch an Alluxio cluster to handle over 10 million files in a default configuration, we find that the Alluxio Master service is not very stable, and the read speed is not high enough.

By monitoring and analyzing, we found that it is due to the constant restart of the Alluxio Master, which is caused by the constant garbage collection (GC) of the JVM. GC will cause the master to do the stop-the-world GC 5 to 10 times in an hour. Each time it will take 30~60 seconds, which will cause the service to be interrupted constantly.

To solve this problem, the first thing is to reduce MaxGCPauseMillis, because it will make the JVM do GC more aggressively and improve the ParallelGCThreads. In the meanwhile, we can give Alluxio master higher memory requests to ensure that there is enough memory to store large amounts of metadata.

FUSE Request Timeout

In some heavy-load tasks, stop-the-world GC cannot be avoided, even after tuning the parameters mentioned above. As a result, some FUSE requests will time out, making the training tasks fail.

To solve this problem, we found that it is helpful to increase alluxio.user.rpc.retry.max.duration and alluxio.user.rpc.retry.base.sleep.

Performance Regression in Reading Multiple Small Files

We also found that Alluxio's performance is insufficient when reading multiple small files. This problem was solved by upgrading Alluxio to 2.6.2 and above.

4 / Results

Alluxio has demonstrated the performance improvement to Coesus in our production environment with two cases.

Case 1: Audio Language Recognition Model

Framework	Pytorch
Network	TDNN neural network model
Total Epoches	20
Number of Files	~2.55Millions
File Size	~ 300Ki/file; ~800Gi total
GPU	4 * Nivida V100/16G GPU
Alluxio	2 * 500 Gi worker

Case 1 is an audio language recognition model. We used 4 Nvidia V100 GPUs, the total file number reached 2.55 million, and the total file size reached 800 G. To handle this scale of training data, we deployed two Alluxio Workers with 500 G storage space.

Speed Comparison			
	OSS S3Fuse	Local SSD	OSS Alluxio Cache
Completion Time	242.56h	63.48h	64.17h
Speed-up Ratio	1	3.82	3.78

In loading training data, two methods were compared. The first option is to download to the local SSD. The second option is to use S3 FUSE. We can see that S3 FUSE took over 242 hours to complete the training of 20 epochs. The results indicate that multiple read tasks seriously reduce S3 FUSE's performance. The third option is with Alluxio where the time was reduced from 242.56h to 64.17h. This shows that reading from Alluxio in multiple tasks does not significantly affect the performance since the overall training time is almost the same as reading from local SSD.

Case 2: Video Portrait Matting

Framework	Pytorch
Network	TDNN neural network model
Total Epoches	50
Number of Files	~20Millions
File Size	~100K/file; ~2Ti in total
GPU	4 * Nividia V100/32G
Alluxio	4 * 600 Gi worker

Case 2 is a video portrait matting model. We used 4 Nvidia V100 GPUs. The scale of training data is 50 epochs, with the total file number reaching 20 million and the total file size reaching 2 TB.

<u>W/O Alluxio</u>

- not enough disks to support large-scale training in local storage
- S3Fuse: bad performance and availability with large metadata

<u>With Alluxio</u>

- each epoch completed stably in ~18 hours
- quality of trained model significantly improved
- intersection of union(IoU) of portrait matting improved by about 2%

To handle this scale of training data, we deployed 4 Alluxio Workers with 600G storage space. Such a large scale of training was almost impossible in the past because there was simply no machine that could meet the requirements of both disk and GPU. However, with Alluxio, each epoch could be completed in about 18 hours, and the IoU (intersection of union) of portrait matting improved by about 2%.

5 / Summary

Alluxio has powered our AI platform in both data pre-processing and model training stages. Using Alluxio as a data layer between compute and storage, our AI platform has improved the training efficiency and the data access is simplified from the user side.

About the Authors

Lei Li works as an AI platform lead at Bilibili. Previously, he worked in Microsoft Azure. He is interested in k8s, docker, and other cloud-native tech. Lei likes traveling and meeting new friends in his spare time.

Zifan Ni is a Senior Software Engineer working on the AI platform at bilibili. He graduated from Fudan University and worked on Azure storage at Microsoft previously.