Introduction

The development of extreme-scale deep learning models necessitates the use of datasets that far exceed the storage capacity of a single machine. These workloads often require raw data to be stored in remote cloud platforms and preprocessed before they can be ingested by a hardware accelerator (e.g., GPU, TPU). Consequently, the time spent loading and transforming data from remote buckets can become a significant bottleneck, leading to substantial hardware accelerator downtime and reduced training efficiency.

Our project addresses this challenge by implementing a proxy-assisted caching and prefetching service designed to minimize GPU stalls. We employ a sharded dataset approach, where data is partitioned into fixed-size chunks to facilitate parallel access and management. Our architecture introduces a proxy layer between the remote storage (e.g., Google Cloud Storage) and the training client. A key feature of our design is the ability for the trainer to pre-determine and communicate shard access orders to the proxy at the beginning of each epoch. This allows the proxy to overlap data transfer and preprocessing with GPU computation by prefetching the next required shards into a local cache.

Inspired by the Extract, Transform, Load (ETL) pipelines of high-performance storage systems like NVIDIA's AIStore [1], we evaluate our architecture's ability to hide remote storage latency and improve end-to-end training throughput using the CIFAR-100 dataset [2]. By decoupling data ingestion from the core training loop, we aim to provide a scalable solution for high-performance machine learning workflows that operate on datasets distributed across multiple regions.

Design and Implementation

Our system is designed as a distributed microservice architecture optimized for high-throughput data ingestion in deep learning workflows. The system consists of two primary components: the **Frontend Service** and the **Proxy Service**. All services communicate using gRPC to ensure efficient, strongly-typed data transfer. We also instrumented a mock **Origin Service** to emulate remote cloud storage, allowing us to simulate various latency profiles and evaluate the effectiveness of our prefetching strategy under realistic conditions.

System Architecture

The architecture follows a tiered approach to decouple remote storage from the training compute. Data flows from the Origin (emulated cloud storage) through a layer of Proxies (caching and transformation) to the Frontend (coordinator), which serves the ML Client.

Microservices

- **Frontend Service**: The Frontend serves as the primary entry point for the ML trainer. It is responsible for routing incoming requests to the appropriate proxies and, most importantly, managing the system's prefetching lifecycle. It maintains the shard schedule and ensures that prefetch signals are dispatched to keep the cache filled ahead of the trainer's requests.
- **Proxy Service**: The Proxy acts as the distributed caching and preprocessing layer. It fetches shards from the Origin on demand or via prefetch signals from the Frontend and stores them in an in-memory map. This service is designed to be horizontally scalable, allowing for parallel transformation of raw data and distributed cache management.
- **Origin Service**: Implemented in Go, this service emulates a cloud storage bucket (e.g., Google Cloud Storage). It serves the CIFAR-100 dataset partitioned into fixed-size binary shards. To evaluate performance under realistic conditions, the service includes a latency simulation layer that can model different storage tiers, such as local SSDs or high-latency cloud objects.

Prefetching Logic

A core contribution of our implementation is the prefetching mechanism that overlaps data transfer with GPU computation. The system utilizes the fact that ML training access patterns (shard orders) are often known at the start of an epoch.
At the start of training, the ML client sends a `PreFetchRequest` containing the full sequence of shards for the session to the Frontend. The Frontend populates a internal `cache_queue` and immediately triggers the first k prefetches (where k is the window size) to the Proxy layer. Every time the client requests a shard via `GetObject`, the Frontend initiates a background gRPC call to prefetch the next shard in the queue. This ensures that while the GPU is processing the current shard, the Proxies are already pulling the subsequent shards from the Origin.

PyTorch Integration

We implemented a custom `CIFAR100` dataset class in Python, inheriting from the standard torchvision `VisionDataset` base class. To avoid local I/O bottlenecks, we replaced standard file system calls with a gRPC client that retrieves binary data from our Frontend. This allows our system to be used as a drop-in replacement for standard datasets, maintaining compatibility with PyTorch's `DataLoader` while transparently benefiting from our distributed prefetching infrastructure.

Deployment

The system is containerized using Docker and orchestrated via Kubernetes. We provide deployment configurations for each service, enabling easy scaling and resource management. The use of Kubernetes `Services` and `ClusterIP` allows for seamless internal discovery between the Frontend, proxies, and origin nodes.

Evaluation

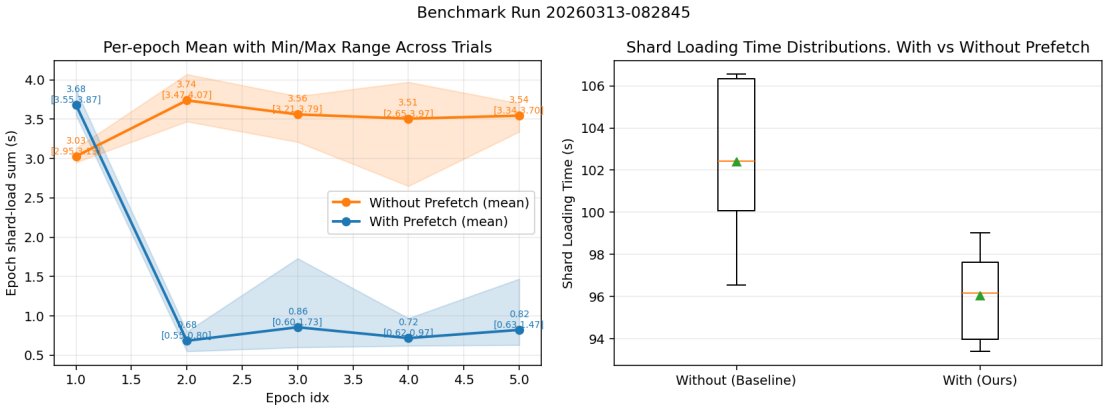

Figure 1: Prefetching benchmark without data transformation. Left: epoch-level shard loading time (mean with min–max range) across trials. Right: distribution of total shard loading time with and without prefetching.

- In the first epoch, both the baseline and our prefetching system exhibit high latency which is approximately ranging from 3.0s to 3.7s of delays. This occurs because the proxy service's in-memory cache is initially empty, forcing a synchronous fetch from the high-latency origin service before the prefetching can even finish asynchronously.
- From the second epoch and onward, the benchmark with prefetching show significant improvement averagely when comparing to trials without prefetching. The baseline average time is stay-

ing around the range from 3.51s to 3.74s while the average time is 0.68s to 0.86s with prefetching, maintaining a significantly lower delay through caching and asynchronously fetching information.

- The median loading time is approximately 102s for the baseline without the prefetching. After using the prefetching proxy, the median loading time drop from 102s to 96s which shows how the prefetching proxy is decreasing the cloud latency, hiding it through asynchronous fetching.
- The larger variant between each trials in the shard loading time without prefetching compare to the smaller variance in the shard loading time in our design with prefetching shows how prefetching also makes the loading time more consistant because the cloud transferring isn't as stable and as reliable in providing data one by one at exact same time. Our prefetching acting as a buffer when network jitter and disrupt the fetching from origin to prevent any downtime for the model training.

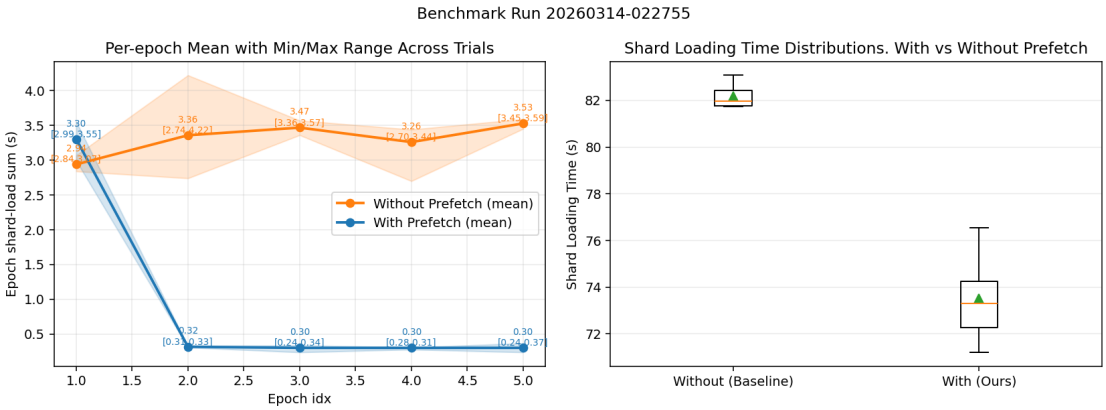

Figure 2: Prefetching benchmark with grayscale data transformation.

- After the transformation is added, the steady-state latency dropped to around 0.3s for with prefetching while the previous is around 0.6s to 0.8s. This happened since after the grayscale is applied transforming the prefetched data to smaller, effectively reduce the latency even further while the data without prefetch isn't transformed, which will be passed with the exact same data size. This also effectively reduce the variation of the per trial difference for the test with prefetching as the dataset are a lot smaller.
- The shard loading time is decreased significantly from median of 102 to 82 for without prefetching and 96 to 74 for with prefetching due to the reduction of data size from colored image to grayscale.

Team Contributions

- **Jasper Huang:** Implemented model trainer test script. Design the prefetching structure. Ran and analyzed preliminary results. Analyzed Lab 4 post-preliminary results and wrote the Lab 4 Evaluation section.
- **Kyle Trinh:** Provide the original idea. Design and implemented the origin-proxy-frontend structure. Implement the service prefetching logic. Implement and test the transformation.
- **Rebecca Chen:** Wrote the proposal and designed the framework from looking into AIStore's documentation. Designed and implemented python trainer's dataloader and dataset classes. Implemented all of the origin (local CIFAR-100 and GCS Storage variants). Wrap trainer client to Kubernetes. Set up the non-grayscale data prefetching experiment. Set up plotting script for Lab 4 report.

References

[1] NVIDIA AIStore Documentation. https://aistore.nvidia.com/docs/overview
[2] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. 2009. https://www.cs.toronto.edu/~kriz/cifar.html