Quiver: An Informed Storage Cache for Deep Learning

Abhishek Vijaya Kumar and Muthian Sivathanu, *Microsoft Research India*

https://www.usenix.org/conference/fast20/presentation/kumar

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-12-0

Open access to the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
is sponsored by

the time per mini-batch is known in advance, one can predict how sensitive each job is to I/O performance, which can in turn allow the cache placement and eviction to give higher priority to jobs that benefit the most from caching.

4 Design of *Quiver*

In this section, we present the design of *Quiver*, a distributed cache that improves I/O efficiency for DLT jobs running in a cluster of GPUs. *Quiver* is a *co-designed* cache that is tightly coupled with the DLT framework (*e.g.*, PyTorch or Tensorflow). By modifying the DLT framework, the cache client integrates deeply into the I/O access of the DLT job, and shares richer information with the cache server.

4.1 System Architecture

Before getting into the details of *Quiver*, we describe the broader context in which *Quiver* fits. *Quiver* is designed for a shared GPU cluster that an organization creates on GPU VMs allocated in the cloud. Each GPU VM has a certain amount of local SSD storage. A DLT job runs within its own container, thus isolating jobs of multiple users from each other. A higher level scheduler and container manager such as Kubernetes [7] manages submission of jobs and scheduling of DLT job containers on a specific VM. *Quiver* is agnostic to the exact scheduling mechanism used, and makes no assumptions about the scheduler.

The input training data for jobs of a particular user is stored in the cloud storage account of the corresponding user, which ensures privacy and access control; while general datasets such as ImageNet do not require this, in general, users sometimes augment standard datasets with their own private training samples which may be sensitive, or train on entirely private data (*e.g.*, surveillance videos, enterprise data). The DLT job running in a VM would perform random reads on this remote storage (*e.g.*, an Azure blob [21] or Amazon S3 [2]).

A DLT job may move across VMs, because of VM redeployments, because of job migration [37], or because it runs on cheaper preemptible VMs [1, 3, 22]. Hence, the local SSD data at each VM is *soft state*. Thus, even if the whole training dataset fits in one VM's local SSD, the simple solution of copying data once from the remote store to local SSD does not work. With *Quiver*, a job can move around across VMs and still transparently benefit from a shared distributed cache.

4.2 Security model

Quiver is a cache that is shared across multiple jobs and multiple users, so the security semantics are important. *Quiver* guarantees that a user can see only data content that she has access to otherwise (*i.e.*, no training data is leaked across multiple users). This requirement of *data isolation* conflicts with the need to share/reuse the cache for effective performance.

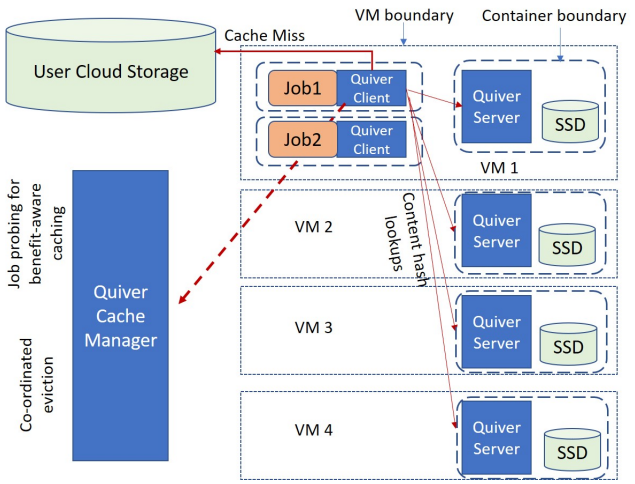


Figure 1: **Architecture of Quiver** Cache servers run on all VMs, but within their own container. *Quiver* clients run in the address space of the DLT job, and include changes to the DLT framework. User's input training data is fetched from cloud storage on a cache miss. Each job's dataset is sharded across multiple cache servers, and looked up using content hashes.

For example, if two different users have their own copies of the ImageNet dataset, those would be two different sets of files in two different cloud storage accounts and thus would logically need to be cached separately, thus preventing reuse across users. *Quiver* uses *content-addressed capabilities* to achieve cache reuse while preserving isolation.

4.3 Content-addressed Cache

The cache in *Quiver* is addressed not by file names and offsets, but by content hashes, similar to content-addressed file systems [9, 23]. The granularity of a cache entry in *Quiver* is decided by the DLT job, and it could be either an individual data item (*e.g.*, image training data where each item is hundreds of KB) or a group of data items (*e.g.*, in text training data). For simplicity, we assume in this paper that the granularity is a single data item. For each data item in the dataset, a content hash (*e.g.*, SHA1) of that item is calculated, and the resulting hash acts as the index for cache inserts and lookups. The benefit of content addressing is that the same data items across multiple copies (say copies of ImageNet data in different storage accounts of different users), will map to the same hash, allowing reuse across users.

To ensure isolation, *Quiver* uses the notion of digest files for the input training data. For each dataset that a user owns, the user computes the content hashes of each data item, and stores just the hashes in a digest file. The digest file contains entries of the form `<content_hash: file_location>`, where the `file_location` indicates the path and offset where that particular data item resides within the cloud storage account of

this particular user. Thus, across multiple users sharing the same data set, while the hash component would be the same, each user will have a different entry in the `file_location` component, as they would point to that particular user's backing store in the cloud. Because the DLT job is calculating these hash digests only from data that the user already has access to, the very presence of a hash value serves as a capability for that user to access that content. As a result, when a cache server gets a lookup for a certain hash value, it can safety return the data associated with that key. The user cannot manufacture or guess legal hashes without having the content, because of the sparsity of the hash function and its collision-resistance properties.

As the digest file is small (few MBs), it is stored locally within the container of the DLT job. The DLT job first looks up the cache with the hash capabilities. If the content is not in the cache, it fetches it from remote storage (using the `file_location` corresponding to the hash entry), and then adds that content into the cache keyed by the hash value.

4.4 *Quiver* Server

The cache server in *Quiver* is a distributed, peer-to-peer service that runs on all GPU VMs in the cluster. The cache server runs as a separate "privileged" user (*e.g.*, organization admin) in its own container, so other users running DLT jobs do not have access to that container. DLT jobs interact with the cache server through RPC interfaces for `lookup` and `insert`. Internally, the cache server is a key-value store maintained on local SSD. The key space is partitioned across multiple cache-server instances via consistent hashing; each cache server instance handles its partition of the key space.

4.5 Cache Manager

Because *Quiver* is a distributed cache, it needs to co-ordinate eviction and placement decisions so that all cache servers roughly agree on the which parts of which data sets to cache. The cache manager in *Quiver* interacts with both the *Quiver* clients and *Quiver* servers to co-ordinate these decisions. The cache manager is also responsible for measuring the likely benefit that each job would get from caching, by probing DLT jobs. It does this by instructing cache servers to temporarily return cache misses for all data read by the DLT job for a few mini-batches. It then compares this execution time with the time during normal operation with caching, and uses this to prioritize cache placement (§ 5).

4.6 *Quiver* Client

A significant part of the intelligence in *Quiver* exists at the cache client. The cache client runs as part of the DLT job within the user's container, in the same address space as the DLT framework such as PyTorch, and interposes at the interface used by the DLT script to access training data. For example, in PyTorch, the `DataSet` abstraction is used to iterate over training data, and it has a simple `Next` interface to get the next set of input data items for the next mini-batch. Internally, the `DataSet` abstraction maintains a randomly permuted list of indices that determines the order in which the data items are fetched. *Quiver* augments this component to also manage the digest-file of hashes, and when a set of indices are to be fetched from the store, it first does the lookup in the cache using the hash values in the digest.

In addition, the *Quiver* client also exports job-specific information to the cache servers, such as the time taken per mini-batch on the GPU. This allows the cache servers in *Quiver* to probe and perform a controlled measurement of performance of the DLT job with and without caching, and use that to prioritize cache placement.

4.7 Substitutable hits

Quiver incorporates the notion of substitutable I/O into the data fetch component of the DLT framework. Today, if a mini-batch requires 512 data items, the dataset loader provides 512 indices to be fetched from the store; if only data pertaining to a subset of the indices was cached, some items may be missing, resulting in remote I/O in the critical path. In *Quiver*, the loader looks up lot more (*e.g.*, 10x) indices from the cache and fills the mini-batch opportunistically with whichever 512 it is able to fetch from the cache, so that the DLT job can make progress without blocking on the cache misses. It then marks the indices that missed in cache as "pending". The data loader continues with the remaining indices for subsequent mini-batches. Once it reaches the end of that list, it makes additional passes over the index list, this time focusing only on the indices previously marked pending.

To see why this would work, assume that only 10% of the training dataset is in cache (for simplicity, a contiguous 10% in the original data set order *i.e.*, without any random permutation). Now, because the lookups from the DLT job are a randomly permuted order of indices, each sequence of k indices is *expected* to get cache hits for $k/10$ indices; hence, if it looks up a sequence of length $10*k$, it can fill its mini-batch of size k. During its second pass over the pending entries, a different, non-overlapping 10% of the dataset may be in the cache, which means it would get hits for $1/9th$ of the lookups. Note that this property also holds across multiple jobs each with their own random permutations. For the same 10% that is cached, regardless of the permutation each job has, each job is expected to get hits for $1/10th$ of its lookups. Thus, multiple jobs can proceed at full-cache-hit speeds although each of them is accessing a completely different random permutation. Such a workload would normally cause thrashing on a small cache that contains only 10% of the data items. With substitutable cache hits, we prevent thrashing and provide cache-hit

performance. Of course, this assumes an intelligent cache eviction policy, which we describe in § 5.

Impact on Accuracy: A natural question that arises with substitutable hits is whether it impacts training accuracy. As we show in § 7 across multiple models, substitutable hits do not affect accuracy of the job, as the randomness within a reasonable fraction of the training data (*e.g.*, 10%) is sufficient.

4.8 Failure recovery

The substitutability property also helps mask failures of cache servers, such as due to VMs going away. In a traditional cache, failure of a cache server would cause a spike in miss traffic to fetch the lost cache items from the store. *Quiver* can handle it gracefully by simply returning substitute data items, while fetching the contents of the failed cache server in the background. The DLT jobs do not incur the miss handling cost in the critical path; they just continue with whatever data is available in the live cache servers; a subsequent pass over the list of indices will use the re-populated data.

4.9 Locality of cache servers

While the simple version of *Quiver* (focus of this paper) has a unified cache spread across all VMs, the *Quiver* design also permits a locality-aware cache layout. For example, datasets used by VMs within a rack in the data center (or a top-level switch) could be cached only within other VMs under the same switch, so that most fetches avoid the over-subscribed cross-rack switches. In such a setting, each rack would have its own logical *Quiver* instance with its own cache manager. *Quiver* can thus also help save cost for the cloud provider by reducing cross-rack network traffic.

5 Cache Management

In this section, we describe various aspects of cache management in *Quiver*.

5.1 Co-ordinated eviction

As described in § 4.7, when only a part of the dataset (say 10%) is cached, *Quiver* does multiple passes over the list of permuted indices of the dataset within a single epoch. To get good hit-rate during the second pass, a *different* part of the dataset must be cached during that second pass. In a scenario where multiple DLT jobs (*e.g.*, a multi-job doing hyper-parameter exploration) are accessing the same dataset, this is tricky because different jobs may exhaust their first pass over the list of permuted indices at different times.

Quiver handles this by allocating cache space for two *chunks* of the data set, and using a technique similar to double-buffering [35]. First, the digest file representing the complete dataset, is partitioned into a fixed number of *chunks*, such that each chunk is, say, 10% of the dataset. The chunking of the dataset has to be done intelligently, to ensure randomness of the input data within each chunk. Some datasets such as LibriSpeech [24] order data items by the sequence length; chunking them in logical byte order would result in the first chunk comprising entirely of short sequences, thus affecting randomness. Recurrent neural networks (RNNs) [4, 36] require all inputs within a mini-batch to be of the same sequence length; if a mini-batch comprises of inputs with different sequence lengths (*e.g.*, randomly chosen inputs), they pad all inputs to match the length of the longest input within the mini-batch. Thus, for compute efficiency, it makes sense for all inputs within the mini-batch to be roughly of the same length.[1]. To allow for such efficient bucketing of inputs within a mini-batch, we define the chunk to be a *striped partition*; let us refer to each contiguous 10% of the input dataset as a *partition*. Each partition is chunked into 10 *stripe units*; a logical chunk is simply the complete stripe formed by stitching the corresponding stripe unit within each partition. As much as possible, a mini-batch is formed purely from inputs in a single stripe unit, for homogeneity of sequence lengths, while also ensuring uniform distribution of inputs.

Dataset chunking allows co-ordinated access of the cache across multiple jobs. While the jobs operate on the first chunk, the second chunk is brought into the cache, so that it is ready when (some of) the jobs switch to the next pass, possibly in a staggered manner. An important question is when to evict the first chunk from the cache. If evicted too soon, a subset of jobs that are still in their first pass and accessing that chunk will see misses, whereas if it remains in the cache for too long, the next (third) chunk cannot be preloaded. *Quiver* uses a two-step process to handle eviction. A chunk is *marked for eviction* when another chunk of the dataset is fully loaded into cache; all new jobs will now get hits only from the latest chunk. However, existing jobs that are still running their pass over the first chunk, will continue to get hits on the first chunk. When all existing jobs have exhausted their pass over the first chunk (and notify the cache server), the first chunk is *actually evicted*. At this point, the preload for the third chunk of the data set can start.

In the above example, note that if a job proceeds at a much slower rate compared to other jobs accessing the same dataset, it could continue to access the first chunk for a long time, preventing the load of the third chunk into the cache. Different jobs in a multi-job are typically designed to proceed at a similar pace, so this is not a common occurrence within a multi-job, but could happen across very different models on the same dataset. Interestingly, a job that is much slower than other jobs on the same dataset means that it spends more time per mini-batch on the GPU, which means it is less sensitive to I/O performance(§ 5.3); a cache miss would not affect that

[1] Dynamic graph computation in modern frameworks such as PyTorch [25] ensures that a mini-batch with short sequence length uses correspondingly lesser computation

job by much. Hence, *Quiver* does a forced-eviction of a chunk after a threshold time has expired from the completion of the first job on that chunk.

Algorithm 1 Substitutable hits & Co-operative miss handling

```
1:  global gChunkIndex = -1
2:  ▷ Returns: List of indices of data items to be fetched for
    current mini-batch
3:  function GETBATCH(SIZE)
4:      ▷ Try to randomly sample 10 x size unused elements
5:      pendingIndices = getPendingIndices(size * 10)
6:      cacheHits = cacheClient.lookup(pendingIndices)
7:      if len(cacheHits) >= size then
8:          return pickAndMarkUsed(cacheHits, size)
9:      end if
10:     ▷ Not enough cache hits, perform co-operative
11:     ▷ cache miss handling
12:     result = List()
13:     result.addAll(
            pickAndMarkUsed(cacheHits, len(cacheHits)))
14:     if gChunkIndex < 0 then
15:         ▷ cacheManager returns 0 if no chunk is cached
16:         gChunkIndex =
              cacheManager.getCurrentChunk(datasetId)
17:     end if
18:     chunksChecked=0
19:     while chunksChecked < totalChunks do
20:         ▷ Tell cache servers that I am using this chunk
21:         ▷ (if not done already)
22:         informServers(jobId, datasetId, gChunkIndex)
23:         unusedIndices = getRandomUnusedIndices (
                gChunkIndex, size - len(result))
24:         if len(unusedIndices) == 0 then
25:             informServersDoneUsingChunk(
                    jobId, datasetId, gChunkIndex )
26:         end if
27:         result.append(unusedIndices)
28:         if len(result) == size then
29:             return result
30:         end if
31:         gChunkIndex =
                (gChunkIndex + 1) % totalChunks
32:         ++chunksChecked
33:     end while
34: end function
```

5.2 Co-operative cache miss handling

A common workload that places significant demand on the storage bandwidth, is a multi-job [37] where a DLT user runs tens or hundreds of jobs for the same model on the same dataset, but with different hyper-parameter configurations. Without *Quiver*, each of these jobs will read the same data from the remote store, causing the remote store to become a bottleneck, resulting in poor I/O throughput per job. *Quiver* uses co-operative miss handling, where it *shards* the cache fetches across multiple jobs, to avoid multiple fetches of the same data items by multiple jobs. This sharding is done implicitly by simply randomizing the order of fetch of missing files, thus avoiding direct co-ordination among the (independent) jobs. Thus, each job first checks the cache if a set of (say 2048) data items exist, then reads a random subset of those items, and adds the read items into the cache. After the additions, it performs another cache lookup, but this time it would get hits for not only the data items it added, but also the other (mostly non-overlapping) data items that were added simultaneously by other jobs that performed a similar random fetch. Thus, even in the case of a cold cache, or if the entire dataset cannot fit in cache, *Quiver* provides benefits by conserving remote store bandwidth, reading most data items only once across multiple jobs within a single epoch.

A high-level algorithm for substitutable cache hits and co-operative miss handling is presented in Algorithm 1.

5.3 Benefit-aware Cache placement

When total cache space is constrained, *Quiver* utilizes job heterogeneity to preferentially allocate cache space to the jobs that benefit the most from the cache. A DLT job performs both compute (on the GPU) and I/O. Intuitively, if the compute time is higher than the I/O time to read from the remote store, the I/O time can be overlapped, and the job performance would be the same whether it reads from the cache or from the remote store. However, this is a complex phenomenon to model, because it depends on the degree of parallelism of the job (*i.e.*, number of GPUs it runs on), how large the model is, whether the model is written in a way to pipeline computation and I/O, etc.

Interestingly, the tight integration with the DLT framework allows *Quiver* to intelligently probe and measure the job's performance with and without caching. When a new job requests for adding entries into the cache, the cache manager picks the job for probing. Probing operates in two steps. In the first step, the cache manager instructs all cacheservers to reject all cache lookups for that job, thus forcing the job to fetch from the remote store. At the end of this probing phase, *e.g.*, 100 mini-batches, the cache manager gets the total elapsed time from the cache client (which runs as part of the DLT job). The cache manager then monitors the job's performance periodically with the default caching policy. If the times with the default caching policy and without caching don't differ by much, it concludes that the job is not bottlenecked on remote I/O bandwidth, and decides to turn off caching for that job. A dataset touched only by such jobs would thus never enter the cache, freeing up space for other datasets that benefit job performance. *Quiver* runs the probing phase not only at job start time, but periodically , as effective I/O throughput may

have reduced because of increased load on the remote store (*e.g.*, newer jobs reading from the same store), thus making the job more sensitive to I/O performance, or vice versa.

Let t_h^i be the average per-mini-batch time for job i under cache hit, and t_m^i be the corresponding time under cache miss. The benefit from caching for job i is thus $b^i = t_m^i/t_h^i$. Let n^i be the number of GPUs taken by job i. The GPU resources saved for job i by caching its dataset is thus $g^i = b^i * n^i$.

For each data set D^k, there could be multiple jobs in the cluster accessing the same data set. Because the cache is shared by all such jobs, if N jobs access D^k, the total GPU resources saved by caching the dataset is $G_{D^k} = \sum_{i=0}^{N} g^i$. Interestingly, the cache manager has to decide only among three options for each data set: (a) fully cache (space cost is the full size of the dataset (b) enable co-operative miss by caching a fixed size chunk (*e.g.*, 15G), or 10% dataset whichever is smaller (cost is 2 chunks for double buffering), or (c) no caching (zero cost). Note that intermediate sizes for caching are useless, as the benefits are the same as with caching two chunks, given the substitutable cache-hits in *Quiver*.

Given a total cluster-wide cache space budget of S, the cache manager uses a greedy algorithm to preferentially assign cache space to datasets or dataset chunks with the highest ratio of benefit-to-cost.

5.4 Cache sharing scenarios

Quiver transparently benefits a variety of DLT scenarios:

Single job accessing a dataset: If the entire dataset can fit in cache, *Quiver* caches the data items accessed in the first epoch of the DLT job. As the DLT job runs several epochs over the same data, subsequent epochs get full cache-hits from *Quiver*, and run faster. If the dataset does not fit in cache, the DLT job does not benefit from *Quiver* as it reads from remote store in the steady state.

A multi-job accessing a single dataset: A multi-job is a set of jobs run by the same user on the same dataset, but with different configurations of hyperparameters [37]. Today, each jobs reads the same content in different random orders from remote storage. With *Quiver*, if the data fits in cache, all jobs share the cache and get full cache-hits. Interestingly, even if only 10% of the data fits in cache, *Quiver* still gives better performance, because it shards the reads across jobs with co-operative miss handling (§ 5.2).

Different jobs accessing the same dataset: Another scenario that *Quiver* benefits is opportunistic sharing of popular datasets *across* jobs even from multiple users. By doing so, *Quiver* extracts more value out of the same SSD space especially for popular datasets such as ImageNet.

6 Implementation

The *Quiver* client is implemented in PyTorch 1.1.0 (about 900 LOC). Pytorch's data model consists of three abstractions: Dataset, Sampler, and DataLoader. Dataset returns a data item corresponding to a given *index*. Sampler creates random permutations of indices in the range of dataset length. Dataloader fetches one mini-batch worth of indices from the sampler, and adds these to the index queue. The worker threads of DataLoader consume these indices, and fetch data items from Dataset. To use *Quiver*, instead of `torch.utils.Dataset`, the model must use `QuiverDataset` (same interface as existing `Dataset`), that handles the digest file containing hashes. Similarly, the model must extend from `QuiverDataLoader` (same interface as standard `DataLoader`), that probes and monitors the job's mini-batch progress in the `__next__` routine; it also ignores the default Sampler passed into the DataLoader API, instead using its custom Sampler that handles substitutable hits, by creating a list of hashes from indices sampled from chunks of the dataset.

The cache client uses RPC endpoints to look up the cache using hashes, fetch files from cache, and finally, to write to cache and communicate mini batch times to the cache manager. Data fetch from Azure blob on the cache miss path happens over a regular TCP socket connection. QuiverDataset uses either the cache client or the blob client depending on whether it is looking up the cache, or filling a cache miss.

The *Quiver* server is a network server written in C++ in about 1200 lines of code. In addition to batched interfaces for lookup/insert on cache, the server also exposes interfaces to get the current active chunk and notify "ref_chunk" and "unref_chunk"; the cache client uses these to assist with coordinated eviction at the server. The server also exposes an interface to set the *caching mode*, used by the cache manager, *e.g.*, to disable caching for a job during probe phase.

The cache manager is a simple python server with an RPC endpoint used by the client to report mini-batch times, and it informs the cache servers which datasets to cache in which mode, based on its benefit-aware allocation decisions.

Config	Top-1 Acc. (%)	Top-5 Acc. (%)
Baseline sampling	75.87	92.82
Quiver sampling	75.89	92.76

Table 1: **ResNet50 on ImageNet: Final Accuracy after 90 epochs (higher is better)** Average of two runs.

Config	Word error rate (WER) (%)
Baseline sampling	22.29
Quiver sampling	22.32

Table 2: **Accuracy of DeepSpeech2 on LibriSpeech: Final WER (*lower* is better)** Average of two runs (30 epochs).

7 Evaluation

In this section, we evaluate *Quiver* along several dimensions. We answer the following questions in the evaluation:

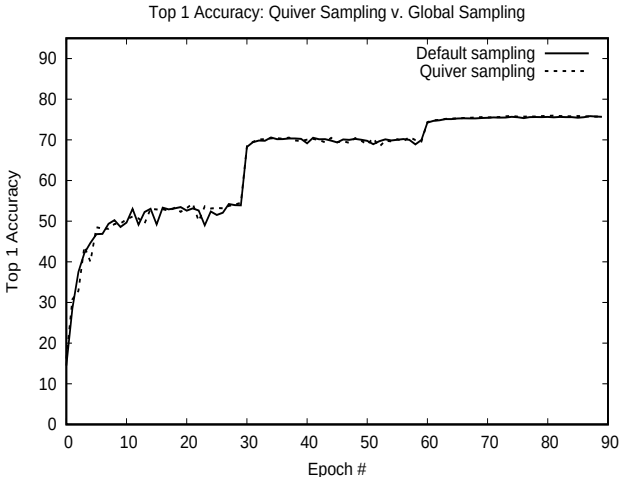


Figure 2: Top-1 Accuracy of ResNet50 ImageNet model under globally random sampling and chunked sampling.

- Do substitutable cache hits impact learning accuracy?
- How much does *Quiver* speed up different DLT jobs?
- How effective is co-ordinated eviction in *Quiver*?
- How effective is benefit-aware caching in *Quiver*?

7.1 Experimental setup

For our evaluation, we use a cluster of 48 GPUs across 12 VMs on Azure. 6 VMs contain 4 NVidia P100 GPUs each while the other 6 contain 4 NVidia P40 GPUs each. All VMs contain 3 TB of local SSD. Different experiments use a subset of these VMs or the full set. The input datasets reside in Azure storage blobs [21] within the same region. We use a diverse set of deep learning workloads: ResNet50 [16] on the 154 GB ImageNet dataset [14], Inception v3 [32] on the 531 GB OpenImages dataset [10], and DeepSpeech2 [4] on the 90 GB LibriSpeech dataset [24]. For substitutable caching, we use a fixed chunk-size of 15GB.

7.2 Accuracy with substitutability

We first show that substitutable caching in *Quiver* (*i.e.*, restricting the shuffle to a fraction of the dataset rather than the entire dataset) has no impact on accuracy. As can be seen from Figure 2, the top-1 accuracy curves closely match. Table 1 shows the final top-1 and top-5 accuracies in both configurations; *Quiver* sampling achieves the same accuracy as globally random sampling. Table 2 shows results for the DeepSpeech2 model on LibriSpeech dataset. Again, the chunked sampling of *Quiver* converges to a similar word-error-rate compared to globally random sampling.

Workload	Time for 7000 mini-batches (s)		
	Baseline	*Quiver*	
	Cache Miss	Cache Hit	Co-op. Miss
ResNet50	2505	646 (3.88x)	1064 (2.35x)
Inception	2874	1274 (2.26x)	1817 (1.58x)
DeepSpeech	1614	1234 (1.31x)	1265 (1.28x)

Table 3: **Speedups from *Quiver* across three workloads**

7.3 Improvement in job throughput

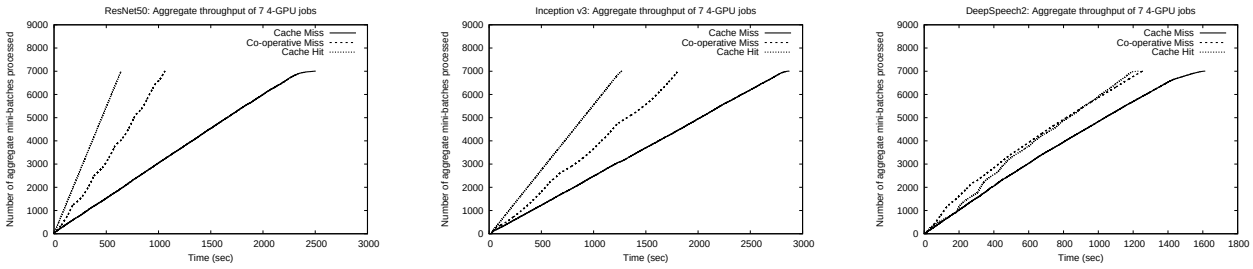
We now evaluate the performance gains from *Quiver*, on three different workloads: ResNet50, Inception, and DeepSpeech2. In each workload, we run a multi-job on 28 GPUs. Recall that a multi-job runs multiple hyper-parameter configurations of the same model/job. For each multi-job, we run 7 jobs (of different configurations), where each job runs on 4 GPUs in a single VM. We show the aggregate throughput (mini-batches/second) of the multi-jobs under three configurations:

1. The baseline configuration, where all jobs read from the remote storage blob. This configuration is referred to as "Cache miss" in the graphs;
2. When all data fetches result in cache hits in *Quiver*. This is the best case performance with *Quiver*, and is shown as "Cache hit" in the graphs;
3. When *Quiver* starts with a cold cache, and the DLT jobs perform co-operative cache miss handling to avoid redundant I/O on the remote store. This also represents the performance when only a 10% or 20% slice of the dataset is cached(§ 4.7).

Figure 3 shows the results for the three workloads. As can be seen, the slope of the "cache hit" curve is consistently much less compared to the "cache miss" curve. In other words, the same number of mini-batches are processed much faster with *Quiver*, resulting in better efficiency. The "co-operative miss" curve is in between the cache hit and cache miss configurations. Thus, even when starting with a cold cache, the ability of *Quiver* to avoid redundant I/O to the remote store from all 7 VMs allows it to extract much higher useful bandwidth out of the remote storage, resulting in better efficiency. Interestingly, in Figure 3(c), the difference between co-operative miss and cache hit is minor, indicating that the workload can run equally fast with just a small slice of the cache (§ 5.3). The overall speedups achieved by *Quiver* for the three workloads is shown in Table 3.

7.4 Interaction with I/O pipelining

DLT frameworks including PyTorch pipeline I/O with computation, to hide I/O latency. In particular, the data loader maintains a queue of fetched mini-batch inputs, and the computation engine picks from the in-memory queue. Both baseline and *Quiver* benefit from pipelining, so the benefits from *Quiver* shown in the previous subsection are *in addition to*



(a) ResNet50/ImageNet (b) Inception/OpenImages (c) DeepSpeech2/LibriSpeech

Figure 3: Multi-job progress timeline with *Quiver* for multi-jobs of 7 jobs each in three models: ResNet50, Inception v3, and DeepSpeech2. Each job runs on 4 GPUs within a single VM.

pipelining. We now analyze the time breakup of multiple pipeline stages within a mini-batch, to understand how exactly the faster I/O due to cache hits improves job performance. For this, we zoom-in on 20 mini-batches of a single ResNet50 job on 4 GPUs within a VM.

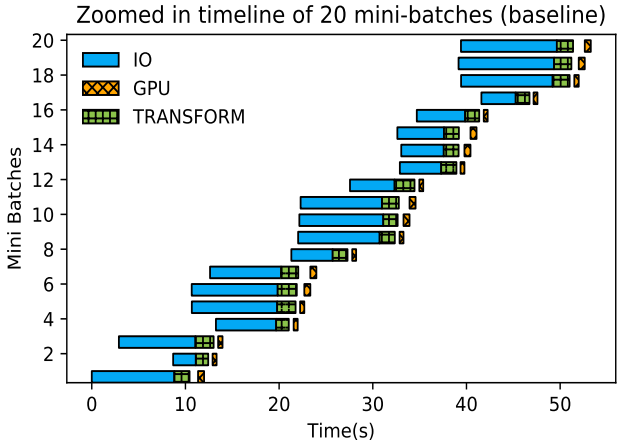


Figure 4: *Detailed timeline of 20 consecutive mini-batches of ResNet50 (different stages), under remote I/O*

Figure 4 is a Gantt chart [34] showing the micro-timeline of a ResNet50 job execution (20 consecutive mini-batches, each processing 512 images) when data is read from remote I/O. The X-axis plots time, while the Y-axis plots the mini-batch index from 1 to 20, starting from a random mini-batch during training. The three boxes in each of the bars pertain to the three main stages of mini-batch processing: *I/O* corresponds to reading the input (from remote storage or Quiver), *Transform* corresponds to performing transformation on the inputs, such as image augmentation (CPU-intensive), and *GPU* is the actual computation on GPU. Ideally, the GPU being the most expensive resource, must not be idle. However, with remote I/O, the GPU is idle most of the time (as seen from the gap between GPU phases for mini-batch i and mini-batch $i+1$), as I/O time constrains job progress. Figure 5 shows the micro-timeline under cache-hit in *Quiver*. As can be seen, the GPU is almost fully utilized in this setting, as the I/O finishes much faster. Although data transformation takes a long time per-mini-batch in both baseline and *Quiver*, because it is parallelized (due to pipelining) across multiple mini-batches on multiple CPU cores, it does not affect GPU utilization in *Quiver*. Thus, while both cases benefit from the I/O pipelining in PyTorch, *Quiver* is able to hide I/O latency much better.

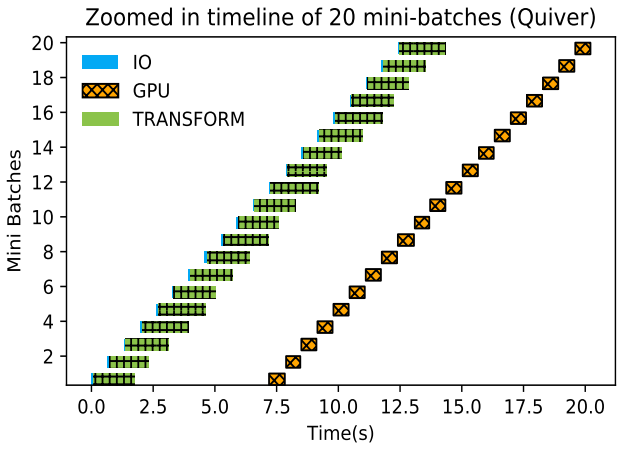
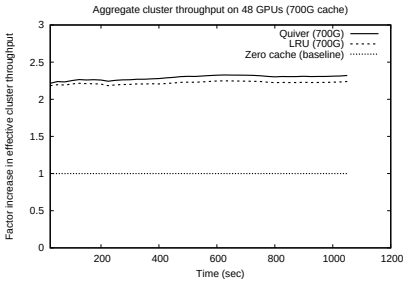


Figure 5: *Detailed timeline of 20 consecutive mini-batches of ResNet50 (different stages), under Quiver hits*

7.5 Cache-constrained scenario

In this experiment, we run 4 ResNet50 jobs (each on 4 GPUs within a single VM) accessing the same ImageNet dataset. After about 15 minutes, we start 3 more ResNet50 jobs in 3 other VMs. We constrain the cache space to be capable of only fitting 20% of the input data. This causes *Quiver* to chop the training data into 10 chunks, and perform double buffering with two chunks at a time (§ 4.7). Figure 7 shows



(a) 100 GB cache (b) 200 GB cache (c) 700 GB cache

Figure 6: Cluster GPU Throughput under multiple simultaneous jobs on 48 GPUs with Quiver, basic LRU, and without caching (baseline). The workload consists of 4 jobs each of ResNet50, Inception, and DeepSpeech2. Each of the 12 jobs runs on 4 GPUs, using a total of 48 GPUs. Average cluster throughput normalized to the non-cached scenario is shown.

Figure 7: *Coordinated eviction with multiple jobs sharing a small slice of the cache.*

the aggregate throughput across these jobs. Every vertical line in the graph indicates the duration for which a specific chunk resides in the *Quiver* cache (the top x-axis plots the chunk number).

There are several aspects that can be seen from this graph. First, if one slices the graph by drawing a line parallel to the x-axis for any time t, it indicates the number of chunks that were cached by *Quiver* at that time, just by counting the number of vertical lines that intersect. It can be seen that at any given time, only 2 chunks are actually resident in the cache, demonstrating co-ordinated eviction. Second, in the aggregate throughput, one can see an increase in the progress rate around roughly 15 mins into the experiment (*i.e.*, when the number of jobs increased from 4 to 7), as more jobs now participate in the co-operative miss handling, improving per-job throughput. Finally, one can notice that when the three jobs start (around t=900 sec), the first two chunks of the cache have already been evicted. Despite that, the jobs are able to make good progress, as they perform substitutable caching, but starting wth the third chunk first (while the first 4 jobs started with the first chunk). This dynamic replaceability is ensured by the cache management policy which directs new jobs to the currently active chunks in order to evict older chunks that other jobs have already exhausted.

7.6 Benefit-aware caching

In this experiment, we demonstrate the efficacy of benefit-aware caching in *Quiver*, and compare it with a simple LRU-based cache replacement policy. For this, we run a workload with a mix of three different DLT models on 48 GPUs. We run four jobs each of ResNet50, Inception, and DeepSpeech, where each job takes a single VM with 4 GPUs. As we previously saw in Figure 3, the three jobs benefit differently from caching. The jobs use three datasets: ImageNet, OpenImages, and LibriSpeech respectively.

Figure 6 shows the steady state timeline (for about 1000 seconds after cache warmup) of normalized cluster throughput. To quantify relative cluster throughput, we calculate the relative improvement in job progress rate (mini-batches processed) for all the 12 jobs compared to the baseline (no cache) configuration. We show cluster throughput under different cache sizes: no caching, 100 GB cache, 200 GB cache, and 700 GB cache. Note that the complete size of the three datasets is about 780 GB. As the 700 GB configuration is close to the complete dataset size, the performance of LRU comes close to *Quiver*. However, thrashing on the remaining 80 GB results in only a 2.2x higher throughput for LRU compared to 2.32x for *Quiver*.

More interesting is the performance of *Quiver* under more constrained caching scenarios, *i.e.*, when the cache size is much lower than the combined sizes of the datasets. In these configs (100 GB and 200 GB), *Quiver* is able to intelligently allocate cache space based on its dynamic mini-batch-aware probing (§ 5.3), besides using co-operative miss handling and substitutable hits to improve throughput. For 100G, it uses co-

operative misses for all three datasets, using a fixed chunksize of 15GB (a total of about 90GB for double buffering of three datasets). At 200 GB cache, *Quiver* automatically chooses to completely cache the ImageNet dataset (as ResNet50 benefits the most from caching), while performing co-operative misses on the other two. At 700G cache, it caches both the ImageNet and OpenImages dataset. *Quiver* is able to preferentially allocate cache space to the jobs benefiting the most, thus maximizing cluster throughput. In both these configurations, LRU performs quite poorly compared to *Quiver*, as it suffers from thrashing because of the random access pattern of the DLT jobs. Overall, even with a tiny cache (100G), *Quiver* still yields sizeable benefits of around 1.6x; the improvement in overall cluster throughput ranges between 1.6x to 2.3x depending on cache size.

8 Related Work

Improving I/O performance for DLT jobs has received some recent attention. DeepIO [39] explored pipelining of I/O fetches with computation by using an in-memory cache, and using an *entropy-aware* sampling technique. DeepIO looks at an individual DLT job in isolation; the benefits from caching for a single job are minimal unless the entire data fits in cache, because workers of a single DLT job read each data item exactly once per epoch. In contrast, *Quiver* achieves cache reuse across *multiple* jobs securely. As a result, even when only a small part of data fits in cache, it improves performance by using substitutable hits and co-operative miss handling to co-ordinate I/O access across multiple jobs. *Quiver* is also benefit-aware in its placement and thus uses the cache frugally, prioritizing jobs that benefit the most. As the authors of DeepIO note, the (modest) benefits from DeepIO in the partial caching scenario are a result of reduced copy overheads and thread scheduling cost by using RDMA shuffling; in contrast, *Quiver* actually reduces the time spent waiting on I/O by employing co-operative miss handling.

Distributed caching in the cluster context has been explored more broadly in the analytics community. For instance, Pac-Man [5] explored co-ordinated caching of data across different workers of an analytics job to extract most benefit for query performance. Similarly, intelligent scheduling of big data jobs to maximize cross-job sharing of cached data was explored in Quartet [8] The co-ordinated eviction policy in *Quiver* has some parallels to these, but the ability to handle partial data caching without thrashing is unique to *Quiver*, as it's possible only because of the substitutability property of DLT jobs. EC-cache [27] is at a distributed cluster cache that uses erasure coding for dynamic load balancing during reads. Because of the regularity of the DLT workload, the simple static partitioning in *Quiver* seems sufficient. There has been other work on caching of various forms in the big data world [15, 19, 38].

Quiver is also related to recent work on systems for deep learning, that use predictability of DLT jobs to improve efficiency. Gandiva [37] uses predictability across mini-batches to introspect on job performance, and uses it to migrate jobs across GPUs or to pack jobs tightly within the same GPU. Astra [31] exploits mini-batch predictability to perform dynamic compilation and optimization of a DLT job by online profiling of multiple choices of optimizations. *Quiver* draws on a similar insight, but uses the predictability for intelligent cache prioritization and prefetching.

Making caching and prefetching decisions informed by application-provided hints has also been studied [26]. General-purpose hints that are application-agnostic are both challenging and limiting; by building a vertically integrated, domain-specific cache exclusively for DLT jobs, the interface in *Quiver* is both simple and powerful. Co-operative caching has also been studied [18, 30]; unlike past work, *Quiver* manages a partial working set with substitutable caching and co-ordinated evictions.

The content hash-based addressing in *Quiver* is based on the notion of using a hash as a *capability*; a similar approach has been explored in content-indexed file systems [9, 23]. *Quiver* applies this idea to the context of a shared cache that simultaneously provides both data isolation and cache reuse.

9 Conclusion

Deep learning has become an important systems workload over the last few years: the number of hardware startups on accelerators for deep learning is testament to its popularity. Systems for deep learning have mostly focused on improving compute efficiency and network efficiency, and the storage layer has been largely handled by ad hoc solutions such as manual staging of data to local SSD, that have significant limitations. With *Quiver*, we provide an automated caching mechanism that helps bridge the storage performance gap in the face of ever-increasing compute capacity for deep learning. *Quiver* achieves cache efficiency by tightly integrating with the deep learning workflow and the framework, and exploits characteristics such as I/O substitutability to ensure an efficient cache even when only a subset of data can fit in cache.

Acknowledgments

We thank our shepherd Robert Ross and the anonymous reviewers for their valuable comments and suggestions. We thank Ashish Raniwala, Subir Sidhu, Venky Veeraraghavan, Tanton Gibbs, and Chandu Thekkath from Microsoft Azure AI Platform team for their useful discussions, as well as providing access to GPU clusters.

References

[1] AMAZON. Amazon ec2 spot instances. run fault-tolerant workloads for up to 90% off. In *https://aws.amazon.com/ec2/spot/*.

[2] AMAZON. Amazon s3: Object storage built to store and retrieve any amount of data from anywhere. In *https://aws.amazon.com/s3/*.

[3] AMAZON. Train deep learning models on gpus using amazon ec2 spot instances. In *https://aws.amazon.com/blogs/machine-learning/train-deep-learning-models-on-gpus-using-amazon-ec2-spot-instances/*.

[4] AMODEI, D., ANUBHAI, R., BATTENBERG, E., CASE, C., CASPER, J., CATANZARO, B., CHEN, J., CHRZANOWSKI, M., COATES, A., DIAMOS, G., ELSEN, E., ENGEL, J. H., FAN, L., FOUGNER, C., HAN, T., HANNUN, A. Y., JUN, B., LEGRESLEY, P., LIN, L., NARANG, S., NG, A. Y., OZAIR, S., PRENGER, R., RAIMAN, J., SATHEESH, S., SEETAPUN, D., SENGUPTA, S., WANG, Y., WANG, Z., WANG, C., XIAO, B., YOGATAMA, D., ZHAN, J., AND ZHU, Z. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR abs/1512.02595* (2015).

[5] ANANTHANARAYANAN, G., GHODSI, A., WARFIELD, A., BORTHAKUR, D., KANDULA, S., SHENKER, S., AND STOICA, I. Pacman: Coordinated memory caching for parallel jobs. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (2012), pp. 267–280.

[6] BOYD, T., CAO, Y., DAS, S., JOERG, T., AND LEBAR, J. Pushing the limits of gpu performance with xla. *https://medium.com/tensorflow/pushing-the-limits-of-gpu-performance-with-xla-53559db8e473*.

[7] BREWER, E. A. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing* (2015), ACM, pp. 167–167.

[8] DESLAURIERS, F., MCCORMICK, P., AMVROSIADIS, G., GOEL, A., AND BROWN, A. D. Quartet: Harmonizing task scheduling and caching for cluster computing. In *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (2016).

[9] FU, K., KAASHOEK, M. F., AND MAZIERES, D. Fast and secure distributed read-only file system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4* (2000), USENIX Association, p. 13.

[10] GOOGLE. Open images dataset. In *https://github.com/cvdfoundation/open-images-dataset* (2018).

[11] GOOGLE. Overview of the open images challenge 2018. In *https://storage.googleapis.com/openimages/web/challenge.html* (2018).

[12] GOOGLE. Youtube-8m dataset. In *https://research.google.com/youtube8m/* (2018).

[13] GRAPHCORE, AND TØRUDBAKKEN, O. Introducing the graphcore rackscale ipu pod. In *https://www.graphcore.ai/posts/introducing-the-graphcore-rackscale-ipu-pod* (2018).

[14] GROUP, D. Dawnbench: Imagenet training on resnet50. *https://dawn.cs.stanford.edu/benchmark/*.

[15] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *OSDI* (2010), vol. 10, pp. 1–8.

[16] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[17] JOUPPI, N. P., YOUNG, C., PATIL, N., PATTERSON, D., AGRAWAL, G., BAJWA, R., BATES, S., BHATIA, S., BODEN, N., BORCHERS, A., ET AL. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), IEEE, pp. 1–12.

[18] KIM, H., JO, H., AND LEE, J. Xhive: Efficient cooperative caching for virtual machines. *IEEE Transactions on Computers 60*, 1 (2010), 106–119.

[19] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–15.

[20] LINLEY, M. Google announces a new generation for its tpu machine-learning hardware. *https://techcrunch.com/2018/05/08/google-announces-a-new-generation-for-its-tpu-machine-learning-hardware/*.

[21] MICROSOFT. Blob storage: Massively scalable object storage for unstructured data. In *https://azure.microsoft.com/en-in/services/storage/blobs/*.

[22] MICROSOFT. Use low-priority azure vms with batch. In *https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms*.

[23] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review* (2001), vol. 35, ACM, pp. 174–187.

[24] OPENSLR. Librispeech asr corpus. In *http://www.openslr.org/12*.

[25] PASZKE, A., GROSS, S., CHINTALA, S., AND CHANAN, G. Pytorch. In *https://pytorch.org* (2017).

[26] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 79–95.

[27] RASHMI, K., CHOWDHURY, M., KOSAIAN, J., STOICA, I., AND RAMCHANDRAN, K. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 401–417.

[28] RASLEY, J., HE, Y., YAN, F., RUWASE, O., AND FONSECA, R. Hyperdrive: Exploring hyperparameters with pop scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (2017), ACM, pp. 1–13.

[29] ROBBINS, H., AND MONRO, S. [a]a stochastic approximation method, [o] annals math. *Statistics 22* (1951), 400–407.

[30] SARKAR, P., AND HARTMAN, J. Efficient cooperative caching using hints. In *OSDI* (1996), pp. 35–46.

[31] SIVATHANU, M., CHUGH, T., SINGAPURAM, S. S., AND ZHOU, L. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, ACM, pp. 909–923.

[32] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. *CoRR abs/1512.00567* (2015).

[33] VOLTA, I. The worlds most advanced data center gpu. *URL https://devblogs. nvidia. com/parallelforall/inside-volta*.

[34] WIKIPEDIA. Gantt chart. *https://en.wikipedia.org/wiki/Gantt_chart*.

[35] WIKIPEDIA. Wikipedia: Multiple buffering. In *https://en.wikipedia.org/wiki/Multiple_buffering*.

[36] WU, Y., SCHUSTER, M., CHEN, Z., LE, Q. V., NOROUZI, M., MACHEREY, W., KRIKUN, M., CAO, Y., GAO, Q., MACHEREY, K., ET AL. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[37] XIAO, W., BHARDWAJ, R., RAMJEE, R., SIVATHANU, M., KWATRA, N., HAN, Z., PATEL, P., PENG, X., ZHAO, H., ZHANG, Q., ET AL. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 595–610.

[38] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 2–2.

[39] ZHU, Y., CHOWDHURY, F., FU, H., MOODY, A., MOHROR, K., SATO, K., AND YU, W. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2018), IEEE, pp. 145–156.