Data-Aware Storage Tiering for Deep Learning

Cong Xu†, Suparna Bhattacharya‡, Martin Foltin†, Suren Byna‡, Paolo Faraboschi†
†Hewlett Packard Labs, {cong.xu,suparna.bhattacharya,martin.foltin,paolo.faraboschi}@hpe.com
‡Lawrence Berkeley National Lab, sbyna@lbl.gov

***Abstract*—DNN models trained with very large datasets can perform rich deep learning tasks with high accuracy. However, feeding huge volumes of training data exerts significant pressure on IO subsystems as the entire data is re-loaded in random order on every iteration to enable convergence, with very little scope for reuse. To address this challenge, we co-optimize data tiering and iteration in DNN training for any given dataset and model with bandwidth and convergence conscious mini-epoch training (MET). This approach can substantially reduce the IO bandwidth required to provide sustained read throughput matching the processing speed of accelerators. Further, we introduce two different feedback mechanisms to adjust the repeating factor over each mini-epoch during the training. We have evaluated three different applications with MET. Most of them work out-of-box with modest MET parameters. The adaptive repeating factor design was able to gain back most of the accuracy drop due lo large MET parameters.**

I. Introduction

Advancements in computational capabilities for training deep neural networks (DNN) favor larger models trained on increasingly bigger datasets to achieve results with significant improvements in accuracy than was possible before [1]. Many HPC applications using AI could have arbitrary large datasets because they are generated through simulation [2]. However, with this approach, training models for the real world could involve iterating through huge volumes of data (TBs to PBs), too expensive to fit in the performance tier of the underlying storage system. The massive IO bandwidth requirements for feeding training data to keep powerful accelerators maximally utilized is becoming a scalability bottleneck for AI training. For example, training different CNNs on 8 NVidia A100 GPUs in a single compute node requires up to 130 GB/s of IO bandwidth to keep the GPUs busy as shown in Figure 1.

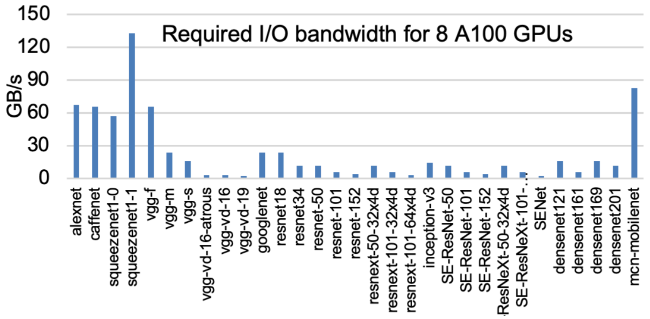

Fig. 1. Required IO bandwidth to feed 8 A100 GPUs for CNNs

Traditional storage data caching and tiering solutions are ineffective in addressing this bottleneck even though the data gets processed repeatedly within a training run and across multiple training runs for hyperparameter tuning. DNN training algorithms, e.g., mini-batch stochastic descent (SGD) iterates through the entire dataset in a different random order for every epoch to achieve theoretical convergence estimates [2], issuing IO to reload data from the capacity tier to the performance tier, causing performance to be limited by the bandwidth of the capacity tier. A recent study on data stalls observed when training several DNNs shows that when no more than 35% of the dataset can be cached in memory, 10-70% of epoch time may be spent blocking on IO (fetch stalls) [3].

Several researchers have explored strategies that are less wasteful of IO [3], [4], [5], [6], [7]. However the effectiveness of such optimizations depends not just on the system performance characteristics but also on the characteristics of the dataset and influence on model convergence. We propose co-optimizing the data tiering and iteration scheme for DNN training with a systematic approach that is not just bandwidth aware but also model convergence conscious and data sample influence aware. There are two main challenges of this radical change. The first one is to provide sustained read throughput for the local accelerators that matches the required IO bandwidth. The second one is how to introduce the technology with minimal disruption to current ML software pipelines and processes.

The contributions of this work are listed as follows,

- We propose a simple but effective storage tiering technique called mini-epoch training (MET) to reduce the IO bandwidth required to fetch data from the capacity tier of the storage hierarchy.
- We analyze, in detail, the impact of MET parameters including number of mini-epochs and repeating factor (number of times each mini-epoch is repeated in an epoch) on the convergence of training for different deep learning models from image classification, video understanding to climate segmentation.
- We introduce two different feedback mechanisms to adjust the repeating factor during training so that some accuracy drop due to large MET parameters can be gained back.

II. Preliminary

A. IO bottleneck analysis of training deep learning models

We built an IO roofline model to analyze the impact of peak FLOPs and IO bandwidth on various CNNs with different IO arithmetic intensity. The IO arithmetic intensity is defined as the ratio of FLOPs per sample to the size of each input sample. The lower the metric is, the more IO intensive the model is. With a hypothetical DL training system that has an aggregated peak performance of 126 PF/s, we plot the performance of several CNNs in the IO roofline analysis. As Figure 2 shows, even though some really deep models with high IO arithmetic intensity are categorized as "happy models" as they are most compute bound, there is a significant portion of the CNNs that are IO IO bound with several TB/s read bandwidth.

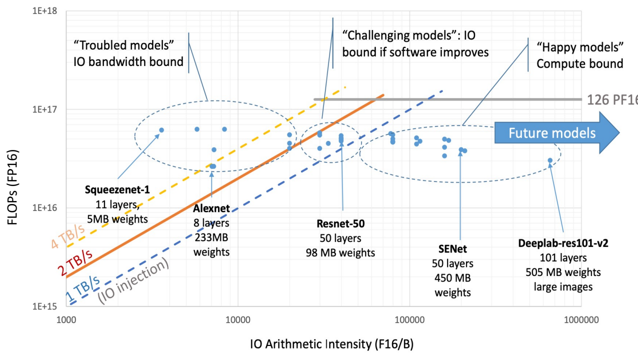

Fig. 2. IO roofline analysis for CNNs in a system with peak 126 PF/s

Figure 3 illustrates a systemic view of two-tier storage hierarchy in HPC environment. The training data sits at the storage capacity tier which is typically a parallel file system (PFS), i.e., Lustre or GPFS. For example, the total space of the capacity tier C_2 can be as large as $250PB$ in the Summit supercomputer [8]. However, the aggregated IO bandwidth (B_2) from the capacity tier is limited at $2.5TB/s$. A performance tier is usually available to "cache" hot data in the storage tier in two different forms: (a) dedicated IO nodes with NVMe drives such as Datawrap burst buffer nodes; (b) local NVMe drives in each compute node. Summit uses node-local NVMe providing an aggregated $26.1TB/s$ of read bandwidth (B_1).

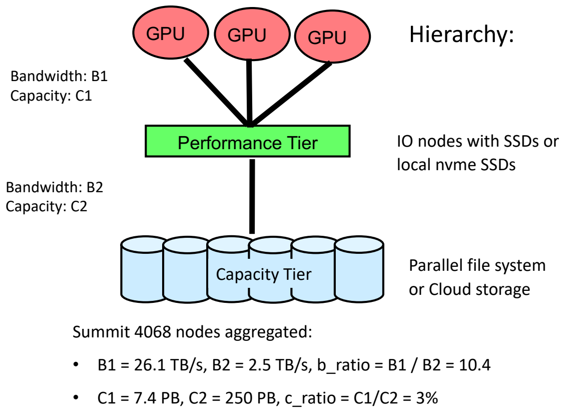
GPU
GPU
GPU
Hierarchy:
Bandwidth: B1
Capacity: C1
Performance Tier
IO nodes with SSDs or local nvme SSDs
Bandwidth: B2
Capacity: C2
Capacity Tier
Parallel file system or Cloud storage
Summit 4068 nodes aggregated:
B1 = 26.1 TB/s, B2 = 2.5 TB/s, b_ratio = B1 / B2 = 10.4
C1 = 7.4 PB, C2 = 250 PB, c_ratio = C1/C2 = 3%

Fig. 3. A two-tier storage hierarchy in HPC

A recent study [9] shows that the scaling efficiency of some HPC AI applications, such as CosmoFlow, dropped to less than 58% at 1024 nodes due to IO bottleneck. By placing the data on the performance tier, which is the burst buffer file system on Cori at NERSC, scaling efficiency can be improved to greater than 85% on 1024 nodes.

Another example is the 2018 ACM Gordon Bell Prize winner - DeepCAM - exascale deep learning for climate analytics [10]. Training the DeepLabV3+ model on the entire Summit supercomputer would require 65,000 samples per second, which translates to 3.8 TB/s data read bandwidth. Figure 4 shows that such bandwidth requirement cannot be simply met by typical global PFS or burst buffer system. Therefore, caching data in the local NVMe drives in the compute nodes would help mitigate the IO bottleneck.

Dataset Size	Required BW (27K GPUs)	GPFS/LUSTRE	BurstBuffer	NVMe or DRAM
20 TB (~63K samples)	3.8 TB/s	~400 GB/s	~2 TB/s	~26 TB/s

Fig. 4. IO bandwidth mismatch between DeepCAM data read and peak bandwidth of different storage components, reprinted from [11]

B. Related Work

Several IO optimizations for DL training have been proposed by researchers that take advantage of data samples cached from a previous epoch when constructing randomly shuffled mini-batches for the next epoch. These commonly include a shared distributed cache across nodes (where samples loaded on another node can be accessed over the network), coordinated IO pipelining and cooperative miss handling. Certain techniques such as entropy aware IO pipelining in DeepIO [5] and substitutable cache hits in Quiver [7] alter the order of samples to better utilize data already present in the cache, trading off some randomness (entropy) compared to a full dataset re-shuffling between epochs. Entropy-aware pipelining computes the randomness level based on the number of samples, but does not have a way to account for how the effect of this tradeoff could vary depending on input data characteristics and the model. On the other hand, the MinIO cache in CoorDL [3] avoids replacement of cached items between epochs, without affecting the random ordering, so that at least some fraction of data for an epoch is always accessible from the cache. The rest still incur IO stalls. For example, a very recent study [6] built a machine learning IO middleware to provides a scalable solution to the IO bottleneck by predicting where a sample will be read from given the seed generating the read access pattern during training.

Most of these approaches require non-trival modifications of the entire input pipeline architecture and are intrusive to the application code. Our approach is simple yet effective: mini-epoch training only requires changing a couple of lines of code and training feedback module is just adding a training callback. At the same time, it is more robust and can also achieve higher savings as it automatically adapts to characteristics of the dataset and model.

Many samping techniques have been explored to the reduce the amount of data read required during the training. For example, Dong et al. [12] used a bandit based sampling strategy with a multi-armed bandit algorithm to accelerate the convergence of coordinate descent by learning which coordinates will yield more aggressive descent. Borsos et al. [13] proposed a novel importance sampling technique for variance reduction in an online learning formulation which finds a sequence of importance sampling distributions competitive with the best fixed distribution in hindsight. Namkoong et al. [14] employ a bandit optimization procedure which learns probabilities for sampling coordinates in non-smooth optimization problems to achieve tighter convergence guarantees than their non-adaptive counterparts. Most of these techniques are orthogonal to our proposed mini-epoch training method and thus can be combined to further reduce IO traffic during training.

III. Design of Mini-Epoch Training

In this Section, we first describe the main design of mini-epoch training (MET) in a two-tier storage hierarchy. We then describe different possible implementations of MET in Tensorflow and other ML frameworks. Finally, we introduce two different feedback mechanisms to adapt repeating factor at run time.

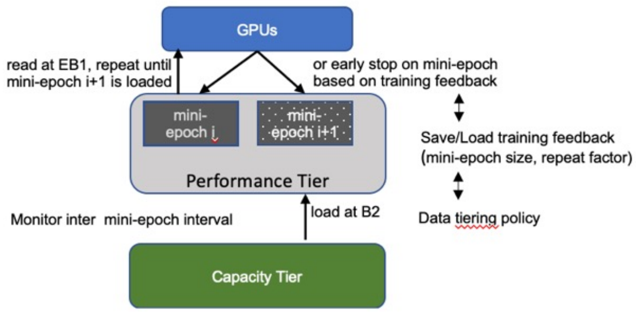
GPUs
read at EB1, repeat until mini-epoch i+1 is loaded
or early stop on mini-epoch based on training feedback
mini-epoch i
mini-epoch i+1
Performance Tier
Save/Load training feedback (mini-epoch size, repeat factor)
Monitor inter mini-epoch interval
load at B2
Data tiering policy
Capacity Tier

Fig. 5. Diagram of mini-epoch training (MET) design

A. Bandwidth-aware iteration

When the entire training dataset is larger than the available space for a given user (EC_1) in the performance tier (note that EC_1 can be significantly smaller than C_1 given than many users share data on the performance tier), we introduce mini-epochs by splitting the entire epoch into nm mini-epochs such that each mini-epoch is smaller than $0.5 * EC_1$ for non-overlapping mini-epochs. As shown in Figure 5, while GPUs are iterating over mini-epoch i, we prefetch the next mini-epoch $i+1$ from the capacity tier to performance tier at B_2. Assuming GPUs are consuming data at EB_1, MET iterates over mini-epoch i by a repeating factor (rf) number of times before the next mini-epoch is completely loaded. The size of mini-epochs depends on the space available in the performance tier, while the repeating factor reflects desired IO reduction achieved. A higher repeating factor reduces the IO bandwidth demand, freeing bandwidth for other nodes and applications to share the same storage. If repeating factor is higher than EB_1/B_2, the GPU stall times due to I/O is fully eliminated. However, repeating a mini-epoch trades some randomness (compared to shuffling a full epoch), which could affect model convergence in certain situations.

B. Implementation in ML framework

The required code change to the ML applications is minimum. We implemented MET by simply adding a stage in the training pipeline that first splits the dataset with Tensorflow Dataset `split` API and then repeats the mini-epoch with Tensorflow Dataset `repeat` API. For example, with $nm = 8$ and $rf = 16$, the added stage looks like,

```
epoch = tfds.load(<dataset_name>, split=[f'train[{k}%:{k+10}%]' for k in range(0, 100, 12.5)])
mini-epochs = [mini-epoch.repeat(16) for mini-epoch in epoch]
```

There are several alternative ways to implement the mini-epoch training, i.e,. with Tensorflow Dataset `window` API,

```
epoch = tfds.load(<dataset_name>)
mini-epochs = [epoch.window(<mini_epoch_size>)]
```

where mini-epoch size is the number of training samples in one mini-epoch. If the size of mini-epoch is not uniform, Dataset `take` and `skip` APIs are used for the implementation as follows,

```
epoch = tfds.load(<dataset_name>)
mini_epoch_sizes = [me1_size, me2_size, ...] # store the sizes for each mini-epoch
mini-epochs = []
for size in mini_epoch_size:
        mini-epoch.append(epoch.take(size))
        epoch = epoch.skip(size)
```

All the above implementations require a few lines of code change when constructing the input pipeline stages in the ML applications. And they can be very similar in other ML frameworks such as Pytorch and CNTK.

Currently we implement per-task data prefetching technique as part of the input pipeline for each ML application. For example, the mini-epoch $i+1$ is manually loaded from capacity tier to performance tier at the start time of mini-epoch i and deleted from performance tier at the completion time of mini-epoch $i+1$.

C. Adaptive repeating factor with feedback mechanism

The baseline design of MET use a fixed rf during training, when rf increases we observed that the convergence of the model could be affected depending on the dataset and the model. Therefore we next introduce two mechanisms to adapt repeating factor (ARF) based on feedback from model monitoring during training.

```
loss = logs.get("loss")
train_err = 1 - logs.get("accuracy")
val_err = 1 - logs.get("val_accuracy")
metrics = np.array([loss, train_err, val_err])
current_score = np.dot(self.score_weights, metrics)

if np.less(current_score, self.score):
        self.score = current_score
        self.wait = 0 # Record the weights if current results is better (less).
else:
        self.wait += 1
        if self.wait >= self.wait_th:
                # stop iterating over this mini-epoch and
                # move forward to next mini-epoch when it's loaded
```

Fig. 6. Pseudo code for score based adaptive repeating factor (ARF)

1) Score based adaptive repeating factor: We use a score to measure the convergence of the training,

- Monitor the training loss/accuracy, validation accuracy and other metrics at the end of each mini-epoch. A score is calculated based on a combination of the monitored metrics.
- If the score does not improve on repeating over mini-epoch i repeats for a number of times, early stop on this mini-epoch and wait until mini-epoch i+1 is fully loaded then move forward.

- Remember the optimal strategy for given dataset and model, so this gets automatically reflected in subsequent training runs.

The feedback module is implemented as a Tensorflow training callback as shown in Figure 6. The combination of different metrics is being explored and an optimal solution will be selected. For example, a simple linear combination of training loss, training error, and validation error can be used to construct the score.

2) Bollinger band-based adaptive repeating factor: Instead of waiting for a predetermined numbers of times before fast-forwarding to the next mini-epoch, we try to understand the range of fluctuations for the score during training. Thus we borrow the concept of Bollinger Bands from the financial market, which consist of an N-period moving average MA of the score, an upper band at K times an N-period standard deviation above the moving average $MA + K * std$, and a lower band at K times an N-period standard deviation below the moving average $MA - K * std$. If the score goes below the lower band, we decide the convergence of the model is fluctuating out of the normal range and the training triggers fast forwarding to next mini-epoch.

IV. Experiments

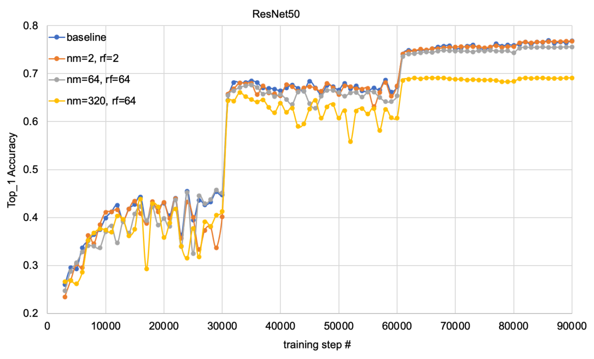

Fig. 7. Top 1 Accuracy for MET with Resnet50 on ImageNet

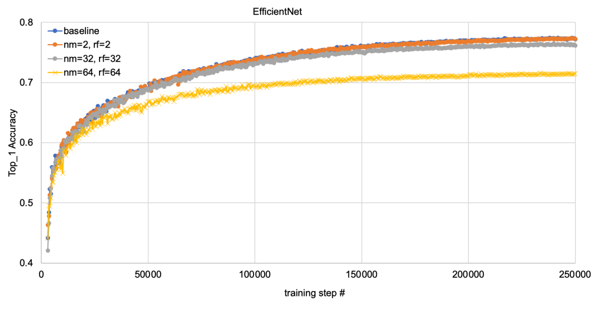

Fig. 8. Top 1 Accuracy for with EfficientNet on ImageNet

We evaluated MET on one image classification task with two different CNN models, one video understanding task, and one HPC application. Most experiments are performed with TensorFlow except the HPC use case which is written in PyTorch. We applied scored-based ARF on the video understanding task, and both score and bollinger band-based ARFs on the HPC use case.

A. Image Classification

We used ImageNet dataset which consists of 1.28 million training images and 50,000 validation images. We adapt the baseline implementation of ResNet-50 and EfficientNet-B0 from the official Tensorflow models github repository. Both models are trained with momentum stochastic gradient descent (SGD). For ResNet-50, we used a global batch size of 1024 and piece-wise constant learning rate scheduler. For EfficientNet-B0, we used a global batch size of 256 and learning rate schedule with exponential decay.

As shown in Figure 7, MET with small nm and rf can match the convergence curve of baseline ResNet-50 training. MET with modest parameters, i.e., (nm, rf) = (64, 64), results in about 1% accuracy drop. MET with nm greater than 300 incurs more than 7% accuracy loss. Similarly, for EfficientNet-B0, the accuracy drop is within 1% for nm and rf up to 32. But with (nm, rf) = (64, 64), we already saw 6% accuracy drop for EfficientNet-B0, which means the upper bound of (nm, rf) without feedback mechanism depends on the model architecture or the training parameters, i.e., batch size.

B. Video Understanding

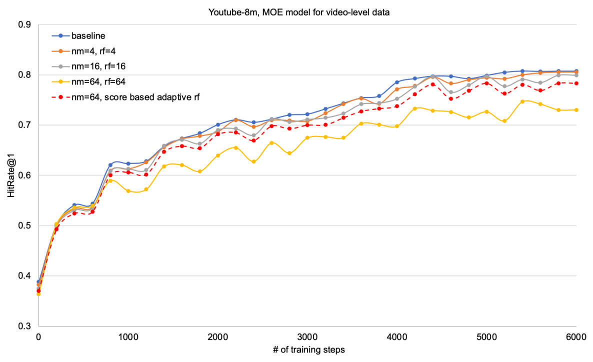

Fig. 9. HitRate@1 for MET with a softmax over a mixture of logistic models

We use the YouTube-8M dataset which consists of 3862 classes and a total of 6.1 million videos. We trained a softmax over a mixture of logistic models, which is referred as MOE model, over the video-level features including feature vectors extracted from the video and audio, to predict the labels associated with each video. The primary evaluation metric we used with this dataset is "Hit@1". We first compute scores for each possible label for each video. The test video is successfully classified if the highest-scoring label is one of its ground truth labels.

As shown in Figure 9, MET with fixed rf works well until rf hits 64 with an accuracy drop of 7%. Therefore we applied score-based adaptive rf for $nm = 64$ and achieved a final HitRate@1 at 0.783 compared to that at 0.803 of baseline.

C. Deep Learning Climate Segmentation

We implemented MET in the PyTorch version for the climate segmentation benchmark as part of the ML. The dataset for this benchmark comes from CAM5 simulations and is hosted at NERSC. The samples are stored in HDF5

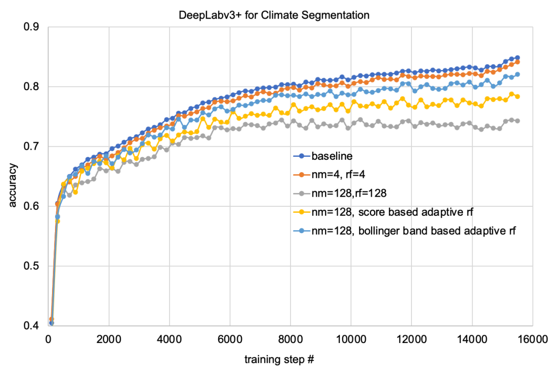

Fig. 10. Accuracy for MET of DeepLabv3+ model on Climate Segmentation

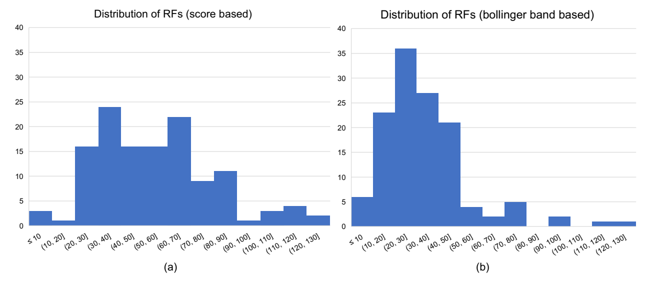

Fig. 11. Distributions of repeating factors over 128 mini-epochs for adaptive repeating factor based on: (a) score and (b) bollinger band

files with input images of shape (768, 1152, 16) and pixel-level labels of shape (768, 1152). The labels have three target classes (background, atmospheric river, tropical cyclone). We used the DeepLabv3+ model in the paper [10] with a global batch size of 64. We used the LAMB optimizer [15] with piece-wise learning rate scheduler.

When scaling (nm, rf) to (128, 128), MET with DeepLabv3+ incurs more than 10% accuracy drop compared to baseline. We applied both the score based and bollinger band-based adaptive repeating factors for $nm = 128$. Score based ARF was able to gain back about 4% accuracy while Bollinger band-based ARF could maintain a accuracy drop of smaller than 3% from the baseline.

Figure 11 shows the distributions of repeating factors for both score based and Bollinger band-based ARFs for $nm = 128$. The score-based ARF has a wider distribution with an average repeating factor of 55, while the bollinger band-based ARF has a tighter distribution with an average repeating factor of 35. We noted that there are a few mini-epochs that have a relatively large repeating factor close to 128.

Assuming we run the DeepCAM on the entire Summit system with $B_2 = 400GB/s$ and $B_1 = 26TB/s$ as illustrated in Figure 3, the baseline is IO bottlenecked as explained in Figure 4. We estimate the average training throughput and data read bandwidth (EB_2) with MET under using the following equation,

$$EB_2 = \frac{\sum_{i=1}^{nm} t_i * MAX(3.8TB/s/rf_i, B_2)}{\sum_{i=1}^{nm} t_i} \quad (1)$$

where t_i and rf_i are the training time and repeating factor of mini-epoch i.

Figure 12a shows that with fixed $rf >= 10$ the IO bottleneck can be mitigated and training throughput peaked at 65,000 samples/s. Both ARFs have less than 5% of mini-epochs with $rf < 10$ when there is IO stall time, and their overall training throughput degradation is negligible. With (nm, rf) = (128,128), the average EB_2 can be reduced from 400GB/s to 30.4GB/s, while the EB_2 for scored based and bollinger band based ARFs are 90.7 GB/s and 147 GB/s respectively.

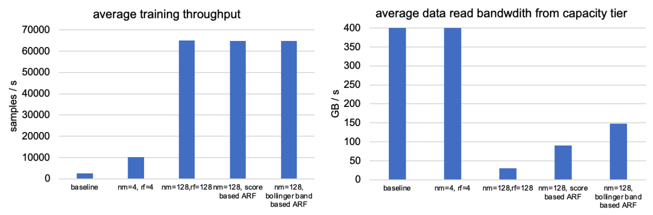

Fig. 12. Analytical modeling results for average (a) training throughput, and (b) data read bandwidth (EB_2) with MET on the Summit system

V. Discussion

A. Co-optimize data tiering policies

Prefetching of mini-epochs is performed by the underlying storage tiering system at the bandwidth corresponding to the optimal strategy from the feedback module. Usually the convergence is more sensitive to repeating factor at early phases during the training. As a result, the system will try to increase repeating factor as the training is making progress and the convergence is not impacted. More data reuse means the prefetching can be done at a data rate lower than B_1. If the impact of repeating factor on convergence speed fluctuates across mini-epochs, more states will be tracked to enable adaptive repeating factor to different mini-epochs, and the prefetching rate also changes dynamically. If too much bias is added with large repeating factor of mini-epochs, there are two directions for co-optimization: (1) compose each mini-epoch randomly in each pass to reduce the bias; or (2) increase batch size to reduce the bias at every iteration.

B. IO bound vs. Data processing bound

Data read bandwidth is not necessarily the only performance bottleneck in the input pipeline of all training workflows. Instead, we observed input pipeline stalls due to the slowness of data preprocessing. For example, in some of the image-based models, there are image augmentation operations including padding, scaling, rotations, resizing, distortion, flipping, brightness adjustment, contrast adjustment, and noising which consume a lot of compute resources and may not keep up with the pace of the model training on GPUs. Under these scenarios, MET could also help accelerate the input pipeline performance by storing the preproccessed version mini-epoch data in the performance tier. However, the downside is the reduced level of randomness of the input data fed to the model because for any given sample, its preprocessed version will remain the same for rf times. In comparison, there are some random effects if preprocessing is performed online in some operations, i.e., random rotation, resizing, and flipping.

VI. Conclusion

We have evaluated three different applications with MET. Many of them work out-of-the-box with modest MET parameters. With larger MET parameters, i.e. both nm and rf greater than 64, there could be about 5% to 11% accuracy drop with fixed rf design compared to the baseline, while the adaptive repeating factor was able close most of the accuracy gap.

Acknowledgment

The authors would like to thank Sergey Serebryakov for helpful discussions. This research is supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] S. W. D. Chien, S. Markidis, C. P. Sishtla, L. Santos, P. Herman, S. Narasimhamurthy, and E. Laure, "Characterizing deep-learning i/o workloads in tensorflow," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2018, pp. 54–63.

[2] Q. Meng, W. Chen, Y. Wang, Z. Ma, and T. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling," *Neurocomputing*, vol. 337, pp. 46–57, 2019. [Online]. Available: https://doi.org/10.1016/j.neucom.2019.01.037

[3] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in dnn training," *Proc. VLDB Endow.*, vol. 14, no. 5, p. 771–784, Jan. 2021. [Online]. Available: https://doi.org/10.14778/3446095.3446100

[4] C.-C. Yang and G. Cong, "Accelerating data loading in deep neural network training," 12 2019, pp. 235–245.

[5] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "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, pp. 145–156.

[6] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," 2021.

[7] A. V. Kumar and M. Sivathanu, "Quiver: An informed storage cache for deep learning," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, ser. FAST'20. USA: USENIX Association, 2020, p. 283–296.

[8] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley, and V. V. Larrea, "End-to-end i/o portfolio for the summit supercomputing ecosystem," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356157

[9] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook, K. Maschhoff, J. Sewall, N. Kumar, S. Ho, M. F. Ringenburg, Prabhat, and V. Lee, "Cosmoflow: Using deep learning to learn the universe at scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018. [Online]. Available: https://doi.org/10.1109/SC.2018.00068

[10] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, "Exascale deep learning for climate analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[11] Q. Koziol, "I/o for deep learning at scale," in *35th Symposium on Mass Storage Systems and Technologies (MSST*, 2019.

[12] J. Dong, J. Zhang, and Y. Shi, "Bandit sampling for faster activity and data detection in massive random access," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 8319–8323.

[13] Z. Borsos, A. Krause, and K. Y. Levy, "Online variance reduction for stochastic optimization," 2018.

[14] H. Namkoong, A. Sinha, S. Yadlowsky, and J. C. Duchi, "Adaptive sampling probabilities for non-smooth optimization," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 2574–2583. [Online]. Available: https://proceedings.mlr.press/v70/namkoong17a.html

[15] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large batch optimization for deep learning: Training bert in 76 minutes," 2020.