Stochastic Communication Avoidance for Recommendation Systems

Lutfi Eren Erdogan*[1], Vijay Anand Raghava Kanakagiri*[2], Kurt Keutzer[1], Zhen Dong[1]
lerdogan@berkeley.edu, vijay7anand@tamu.edu, keutzer@berkeley.edu, zhendong@berkeley.edu

Abstract*— One of the major bottlenecks for efficient deployment of neural network based recommendation systems is the memory footprint of their embedding tables. Although many neural network based recommendation systems could benefit from the faster on-chip memory access and increased computational power of hardware accelerators, the large embedding tables in these models often cannot fit on the constrained memory of accelerators. Despite the pervasiveness of these models, prior methods in memory optimization and parallelism fail to address the memory and communication costs of large embedding tables on accelerators. As a result, the majority of models are trained on CPUs, while current implementations of accelerators are hindered by issues such as bottlenecks in inter-device communication and main memory lookups. In this paper, we propose a theoretical framework that analyses the communication costs of arbitrary distributed systems which use lookup tables. We use this framework to propose algorithms that maximize throughput subject to memory, computation, and communication constraints. Furthermore, we demonstrate that our method achieves strong theoretical performance across dataset distributions and memory constraints, applicable to a wide range of use cases from mobile federated learning to warehouse scale computation. We implement our framework and algorithms in PyTorch and achieve up to $6\times$ increases in training throughput on GPU systems over baselines, on the Criteo Terabytes.

I. Introduction

A significant portion of machine learning research has advanced due to better memory and computational speeds of accelerators, alongside faster interconnects and more efficient parallelization in large systems. However, accelerators often have limited memory compared to CPUs, rendering many memory-intensive algorithms infeasible for deployment. One approach to mitigate this issue is to increase memory, but this can't keep up with the rapid growth of machine learning models. An alternative is to develop new parallelization strategies that balance memory use and communication, as explored in strategies like those in [4]. Another optimization strategy involves quantization, where quantization aims to minimize the memory footprint and computational requirements of embedding tables without significantly impacting accuracy. Embedding tables in DLRMs, which map sparse categorical features to dense vectors [3], are often very large and are thus prime targets for quantization. By quantizing the embeddings to lower bit-width representations like 4-bit [5] or performing tensor train decomposition [9], memory usage is significantly reduced, making it more feasible to train and deploy large DLRMs and improving computational efficiency during training and inference phases. These methods mainly emphasize on reducing the embedding size see [8]. Another solution is to manipulate how the model is trained to save memory. Some examples using this solution include reversible networks, which change the structure of the model. Others, such as Checkmate [7], change the execution pattern of the model, adding additional operations to backpropagation to decrease the number of intermediate values that need to be stored in memory.

Recent algorithms have significantly increased the scale of NLP and CV models by reducing the memory demands per GPU, allowing the use of accelerators for extremely large models. However, these methods struggle with models that have large embedding tables, which are not easily managed by pipeline parallelism and remain large in parameter count. Data parallelism also falls short as it is better suited for compute-heavy tasks rather than memory-intensive embedding operations. Furthermore, techniques like recomputation or checkpointing do not suit embeddings well, as their high memory cost does not justify the modest savings from managing intermediate activations.

This forces the use of model parallelism, but oftentimes the number of accelerators required to fit the large embedding tables are too great to make model-parallelism a financially viable solution.

These embeddings can often be terabytes large, but as observed in practice, they are not accessed uniformly at random. In real datasets, the access pattern of these embeddings varies, generally with a small portion of embeddings being accessed far more frequently than others [2]. Existing methods [10] have explored the usage of distributed communication to decrease the communication cost, but the theoretical bounds on communication efficiency has not been analyzed in previous work. In addition, there has been no existing work exploring how various methods of distributing embeddings across GPUs and CPUs impact the system performance.

To summarize, our contributions are as follows:

1) We develop a simple framework to calculate the expected communication cost under a training regime. And we expand this framework to address considerations such as determining the optimal levels of communication and caching, as well as methods for adjusting them accordingly.
2) We use the above framework to obtain communication strategies that can minimize expected communication

*Equal contribution, [1]University of California Berkeley, [2]Texas A&M University cost without caching. We also demonstrate that these methods also decrease main memory I/O proportionally to the decrease in communication.

3) We demonstrate how assumptions about ML training motivate different strategies for caching through empirical analysis with our framework.
4) We extensively test our algorithms on a variety of datasets and models. Furthermore, we also test various theoretical distributions, and observe that that our algorithms can generalize well, with performance gains on a wide range of potential synthetic datasets.

II. Methodology

As shown in Fig. 1, the existing training paradigm for the above model is as follows

1) A batch of training examples is retrieved
2) For every training example, the embeddings are retrieved from embedding tables in main memory
3) The batch of embeddings is sent to the GPU

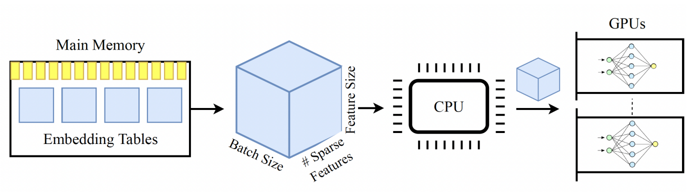

Fig. 1. Existing training paradigm without coalescing.

In the following chapters, we will analyze the communication cost of the embedding layers of the recommendation systems and propose algorithms that maximize throughput subject to memory, computation, and communication constraints.

A. Coalescing on device

To start, we wish to estimate how much overlap exists per batch of the dataset. To do so, we want to find the expected number of unique elements in the list and the expected total communication given a batch size b. If we transmit the batch of b training examples, the communication cost per feature is $b * d$, where d is number of lookups per sample. We call this "communicating the batch", as we send over the entire batch.

As depicted in Fig. 2 we can alternatively coalesce the embeddings into a list of unique embeddings in the batch as well as there indices. For embedding e, let $P(e)$ be the probability that e refers to a categorical feature in a training example chosen uniformly at random from the dataset. The likelihood that e is transmitted within our batch is computed by taking complement of the probability that no embedding for that feature in our batch is e:

$$1 - (1 - P(e))^b \tag{1}$$

Using (1), if E is the set of possible embeddings for this feature, we can then find our expected embedding communication cost as:

$$\sum_{e \in E} 1 - (1 - P(e))^b \tag{2}$$

In addition to transmitting the unique embeddings as described in (2), we also have to transmit the indices of each embedding. This means that we have a constant b cost. As a result, our net cost is

$$b + \sum_{e \in E} 1 - (1 - P(e))^b \tag{3}$$

We call this method "coalescing" because coalesce, or combine distinct, embeddings for each feature.

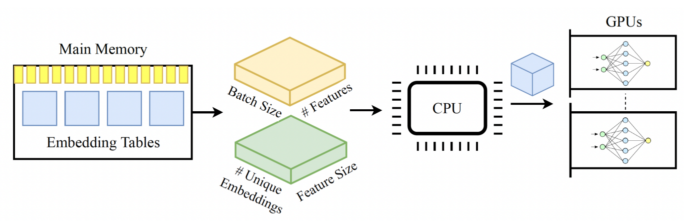

Fig. 2. Training paradigm with coalescing.

B. Caching on device

We can also use the GPU as a cache for our embeddings as described in Fig. 3, we examine several ways of determining what embeddings to store on GPU.

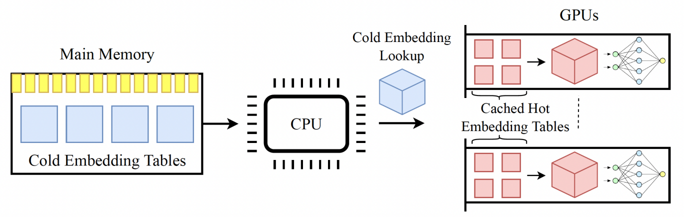

Fig. 3. Training Framework with Caching Mechanism.

The first method: minimize communication bandwidth per epoch. For this method, we assume that the batch size is sufficiently large on GPU to saturate communication during training, and as a result the time it takes to execute computations using larger batch sizes is proportional to the batch size itself. As a result, the expected time spent on computation per epoch is constant, and the primary source of change in the latency how much time is spent on communication per epoch.

We shall let the number of samples in the dataset be Q, batch size b, and the lookups per sample be d. Our expected communication cost is equal to the expected number of batches times the expected communication per batch. Without caching or overlap, this is equal to:

$$\frac{Q}{b} \times b \times d = Q \times d \tag{4}$$

If we exploit the overlap between lookups, the communication cost changes to

$$Q + \frac{Q}{b}\left(\sum_{e \in E} 1 - (1 - P(e))^b\right) \times d \tag{5}$$

The first term in (5) represents the cost of sending indices, while the second term is the cost of sending the embeddings. Generally speaking, the communication cost without caching will decrease as batch size increases, but the memory requirements will also increase. The tradeoff between these two values depends on the distribution of embeddings.

If we utilize the remaining memory to cache embeddings, we can remove the communication cost of embeddings that are cached on device. Let the set of embeddings cached on device be C

$$Q + \frac{Q}{b}(\sum_{e \in E/C} 1 - (1 - P(e))^b) \times d \tag{6}$$

However, due to the fact that we have limited memory there is a direct trade off between the batch size and the number of cached embeddings.

As a result, if we model the memory usage of the cached embeddings, we can theoretically calculate the largest potential batch size on device. While in practice compilers do not allocate memory with this theoretical efficiency, it helps illustrate the mathematics of the tradeoffs. Define the total number of parameters that can fit on device to be M, the parameters used for running the model per sample to be a, i.e. space taken up by both model's weights, biases and also space required for intermediate activations of a single sample. Then, given that $|C|$ embeddings are cached, and each take up d parameters on device, as before, the maximum batch size possible is

$$b = \frac{M - |C|d}{a} \tag{7}$$

This introduces a tradeoff between the amount of communication saved by overlap and the amount of communication saved by caching, because the higher our batch size is the more overlap will occur but the less memory will be available for caching.

In order to understand the efficacy of this tradeoff, we need to examine the relative change in communication from caching one additional embedding, e'.

Let C be the current set of cached embeddings and b be the maximum batch size with C cached.

Let $C' = C \cup e'$, b' be be the maximum batch size with C' cached,

If we store e', our expected decrease in communication cost is $1 - (1 - P(e))^b$, which is the likelihood e' is in a batch.

However, using (7), our expected batch size decreases by $b - b' = \frac{d}{a}$, or the size of one embedding divided by the size of activation. This both decreases communication and decreases potential overlap, using (6) and (7), we can mathematically say that

$$\Delta communication/epoch = \text{commn}_1 - \text{commn}_2 \tag{8}$$

where,

$$\text{commn}_1 = \frac{\sum_{e \in E/C'} (1 - (1 - P(e))^{b'}) \times Q}{b'} \tag{9}$$

$$\text{commn}_2 = \frac{\sum_{e \in E/C} (1 - (1 - P(e))^{b}) \times Q}{b}. \tag{10}$$

For the communication to decrease, we need (8) to be negative. We can separate out the term for e' to get that

$$(1 - (1 - P(e'))^b) \geq t1 \tag{11}$$

where

$$t1 = \sum_{e \in E/C'} \frac{(b(1 - (1 - P(e))^{b'}) - b'(1 - (1 - P(e))^b))}{b'} \tag{12}$$

When $M >> a > d$, or when available memory is significantly larger than the memory needed for running model on a one sample, and the memory needed for running model on a single sample is larger than the memory needed for an embedding, caching generally improves performance in use cases where the dataset is randomized, such as offline training. Because this equation minimizes communication bandwidth, it only serves as an accurate model of communication when communication is dominated by bandwidth.

In practice, this means the caching generally decreases the cost of communication relative to relying on coalescing for large batch size training.

In addition to using this to understand the tradeoff between the batch and the number of cached embeddings, it is possible to find the theoretically optimal number of embeddings to minimize communication bandwidth in $O(log(|E|))$ steps using binary search. However, the embedding memory and activation memory need to be measured empirically, as existing machine learning platforms often allocate more memory than necessary to improve performance.

Furthermore, (13) is also a good proxy for the expected number of accesses to main memory for embeddings. This is because cache performance for these embeddings are poor, and as a result most embeddings communicated over the channel are looked up in main memory.

$$(1 - P(e'))^{(b-b')} \geq \frac{d}{M - |C'|d} \sum_{e \in E/C'} \tag{13}$$

If the batch size is too small to saturate computation after communication, the size of the cached set can be determined through measuring the runtime of the model on various splits of the data. This is much more time intensive, but can potentially lead to more accurate results.

When analyzed with the different dataset distributions, our method displays strong theoretical performance. We use three distributions, Zipf distribution ($P(x) \sim 1/x = e^{-log(x)}$), exponential distribution ($P(x) \sim 1/x = e^x$), and half normal distribution ($P(x) \sim 1/x = e^{-x^2}$), to scale to $5\times$ number of embeddings and $5\times$ batch size. Half normal distribution is of particular importance since Criteo Terabyte Dataset is most similar to that distribution. With our method, the total communication cost increases by $< 1.5\times$ for the exponential and the normal distributions and by $< 2\times$ for Zipf distribution while with prior methods, the total communication cost increases by $5\times$. That is a $3\times$ increase in theoretical performance.

III. Experiment Settings

We evaluate our method using DLRM [1] on the Criteo Terabyte Dataset. Since our method and our theoratical framework is applicable to any arbitrary distributed system that use lookup tables, we expect the same performance gains if we have used TBSM [6] on the Alibaba UBA dataset.

Both models, for their respective dataset, are usually trained on CPUs. This is due to the limited memory availability for GPU training.

We compare our method against baseline Deep Learning Recommendation Model (DLRM) implementations that use only CPUs or a combination of CPUs and GPUs without caching. The CPU-only setup processes the entire computation graph, including the DNN's large tensor operations, on the CPU—tasks for which the CPU is not optimized. The CPU-GPU setup assigns the memory-intensive embedding layer to the CPU and the compute-intensive DNN layers to the GPU, resulting in CPU-GPU communication overhead during the backward pass and when handling intermediate results, which extends the training time.

The effectiveness of caching hot embeddings in mini-batches hinges on using only the cached (hot) embeddings and avoiding the cold embeddings stored on the CPU, thus eliminating data shuffling between the CPU and GPUs during the embedding layer. While it's unrealistic to expect all mini-batches to require only hot embeddings, it is feasible to ensure that most do. This is achieved by classifying training samples into "hot" (those that only need hot embeddings) and "normal" (those that require both hot and cold embeddings). We can then create mini-batches exclusively composed of hot samples, and others of normal samples. Given that hot samples generally predominate, this method maximizes the use of cached embeddings. This classification involves constructing a ranked skew table for all embeddings, selecting the top few embeddings for caching, and then categorizing training samples based on their dependency on these cached embeddings.

For all of the experiments, we benchmark the iteration time and the total training time for one epoch.

IV. Experiments

We first evaluate our results on the Criteo Terabyte dataset. For this method, we trained a DLRM model with the default parameters on two NVIDIA TITAN RTX with 24GB of memory. For our first experiment, we wanted to devise a simple experiment to verify the benefit of overlapping and caching on the iteration time and the total training time for one epoch. We set the batch size to 2048 for all runs, and we cache 256MB of the hot embeddings with our method.

From the Table I, we can see that models with caching and coalescing achieve large performance improvements relative to baselines. We even argue that without caching, the CPU-GPU implementation is worse than the CPU only implementation. This means that the state-of-the-art implementations cannot utilize the hardware accelerators, leading to scalability problems; while, caching enables us to integrate fast accelerators into deep recommendation systems training by avoiding CPU-GPU communication.

TABLE I
Comparing total time in seconds and one epoch for the two baselines and our method with coalescing and caching.

Method	**One Iteration (s)**				One
	Fwd	Bckwd	Optzn	***Total***	Epoch(m)
CPU-only	12.05	15.1	41.94	69.09	38.03
CPU-GPU baseline	20.37	19.14	54.17	93.68	46.12
SCARS (ours)	50.88	5.36	0.41	**56.65**	**27.12**

Note: Fwd, Bckwd, Optzn means Forwad, Backward, Optimization

For our second set of experiments, we wanted to see if caching more embeddings meant better performance. For that, we set the batch size at 2048 and changed the GPU memory allocated for caching the hot embeddings. From Table II, in each case when compared with CPU-GPU baseline, we see major savings in backward pass and optimization stages of one iteration, and the savings across different GPU memory settings are very similar. This is to be expected since our first experiment showed that caching hot embeddings significantly improves these stages thanks to decreased DRAM memory writes. However, forward time increases drastically and almost directly proportionally as the memory cached hot embeddings increase. As a result, the forward time dominates the total iteration time.

TABLE II
The absolute time in seconds for one iteration of SCARS with differing memory allocated for cached hot embeddings.

Method	**One Iteration (s)**			
	Fwd	Bckwd	Optzn	Total Time
CPU-GPU baseline	20.37	19.14	54.17	**93.68**
128MB	26.65	4.13	0.39	**31.23**
256MB	50.88	5.36	0.41	**56.65**
512MB	108.46	5.79	0.42	**114.75**
1024MB	191.75	5.03	0.44	**197.22**

Note: Fwd, Bckwd, Optzn means Forwad, Backward, Optimization

Such a pattern is caused by the increased size of the embedding lookup layer: As we cache more embeddings, the size of this layer increases. Since retrieving the embedding indices from a large lookup layer is slower, the forward time of the network, which is bounded by this lookup operation, increases. Hence, we conclude that we cannot just cache as many embeddings as we want and that we need a more intelligent way to cache hot embeddings.

Driven by the question of the optimal size of the cached hot embeddings, we analyzed how much of the cached embeddings in the 512MB setting from the previous experiment were frequently accessed by a batch of 1024 random training samples. We looked at the four 128MB portions of the 512MB, with the first portion containing the "hottest" cached embeddings and the last portion containing the least hot embeddings. As can be seen from the Figure 4, almost all of the samples in the batch accessed the first and the second portions while the third and the last portions were rarely or even never accessed. Thus, the last two 128MB portions are not really ”hot” embeddings, and so caching them is unnecessary. All in all, increasing the memory for cached embeddings excessively causes us to cache cold embeddings and unnecessarily slow down the forward pass of the embedding layer. Thus, the best practice is to profile our data beforehand and see how much memory is really needed for cached embeddings. For our experiments, we decided to cache 256MB of hot embeddings.

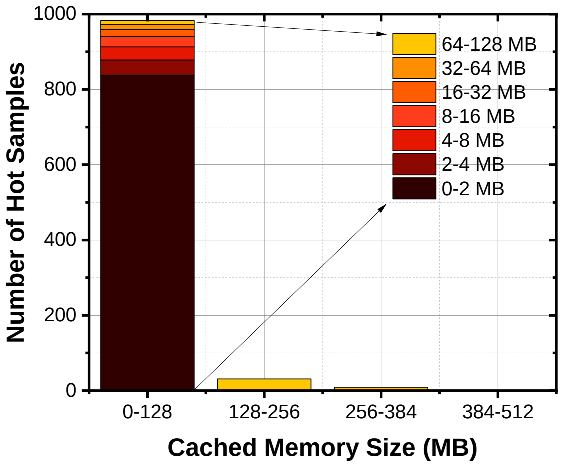

Fig. 4. The number of samples using cached embeddings in a mini-batch of size 1024 from the Criteo Terabyte Dataset varies with the memory size. The 0-128MB range, containing the ”hottest” embeddings, is used by almost all samples in a mini-batch, whereas the 128-256MB and 384-512MB ranges, representing progressively ”colder” embeddings, see significantly less usage. The least hot 128MB range is seldom or never used. This data suggests that caching need not be indefinite, as caching colder embeddings eventually uses up lots of memory space without proportional usage benefits.

Our next set of experiments in which we varying the batch size showed that caching hot embeddings allows us to have larger batch sizes, as can be seen from the Table III. For a ”normal” iteration, that is an iteration that accesses both the hot and the cold embeddings, larger batch size means dramatically slower backward pass and optimization stages since a larger batch size means more DRAM writes which can be inferred from Table IV. Thus, state-of-the-art implementations use a batch size no larger than 2048. For a ”hot” iteration, that is an iteration that accesses only the cached embeddings, we found that increased batch size doesn’t affect the iteration time. The reason for such performance gain is that caching embeddings have major savings in backward and optimization stages already due to decreased DRAM writes and that since we now can limit the embeddings layer to an intelligent size (we choose 256MB for Criteo Terabyte Dataset), the time for forward pass doesn’t change much with increased batch size.

We also note the total time for one epoch of Criteo Terabyte with varying batch sizes in Table V and see that caching hot embeddings allows us to use large batch sizes, which greatly improves the training time. However, after some very

TABLE III
THE ABSOLUTE TIME IN SECONDS FOR ONE ITERATION OF SCARS WITH DIFFERING BATCH SIZES

Method	**One Iteration (s)**			
	Forward	Backward	Optimization	Total Time
2048	50.88	5.36	0.41	**56.65**
4096	54.23	8.14	0.62	**62.99**
8192	51.60	10.43	0.88	**62.91**
16384	54.70	19.14	1.56	**75.4**

TABLE IV
THE ABSOLUTE TIME IN SECONDS FOR ONE ITERATION OF THE BASELINE CPU-GPU IMPLEMENTATION WITH DIFFERING BATCH SIZES

Method	**One Iteration (s)**			
	Fwd	Bckwd	Optzn	Total Time
2048	20.37	19.14	54.17	**93.68**
4096	26.71	25.04	95.81	**147.56**
8192	40.06	74.81	149.52	**264.39**
16384	72.21	139.84	253.36	**465.41**

Note: Fwd, Bckwd, Optzn means Forwad, Backward, Optimization

large batch size, increasing the batch size indefinitely doesn’t correspond to equally greater performance gains.

TABLE V
THE ABSOLUTE TIME IN MINUTES FOR ONE EPOCH OF BASELINE CPU-GPU IMPLEMENTATION AND SCARS WITH 256MB OF CACHED HOT EMBEDDINGS

Method	Batch Size				
	2048	4096	8192	16384	32768
CPU-GPU baseline	46.12	38.33	35.54	30.10	-
256MB Cached SCARS	27.12	15.10	7.54	4.52	4.31

From Table VI, we can see that although first few increases in the batch size reflect as remarkable gains (a speed up of 3.6× from 2048 to 8192), the next few increases don’t show an equally great gain (a speed up of 1.7× from 8192 to 16384). Hence, it is not essential to increase the batch size indefinitely further than some point.

TABLE VI
SPEEDUP RATIO AT P TO Q I.E. TOTAL TIME IN ONE ITERATION FOR BATCHSIZE P TO TOTAL TIE IN ONE ITERATION FOR BATCHSIZE Q

Batch Size	Speedup (s) for one iteration of SCARS		
	4096(p)	8192(p)	16384(p)
2048(q)	1.8	3.6	6.01
4096(q)	-	2	3.342
8192(q)	-	-	1.7

Speedup(p/q) = (Absolute Time for one iteration at Batch Size p) /(Absolute Time for one iteration at Batch Size at q

Next, we wanted to analyse at how increasing the batch size of SCARS affects convergence and training accuracy. We keep the memory used for cached embeddings constant and this is considerably less than the device memory; therefore, we can technically indefinitely increase the batch size. Since our method mainly focuses on improving performance in terms of time and efficiency, we let models with different batch sizes train for an absolute time of 10 hours rather than using some certain epoch count since, as shown before, the time for an epoch for different batch sizes is not the same. These time frames, as a result, would mean more epochs for large batch sizes while it only corresponds to a few epochs for smaller ones. Our experiments showed that after you increase the batch size too much, although the speed and efficiency gains are remarkable, the model cannot converge as fast. Since the size of mini-batches essentially determines the frequency of updates, very large mini-batches correspond to fewer updates and therefore to slower convergence; therefore, it is best to choose large batch size reasonably since a slow convergence is suboptimal. This finding, when paired with our previous finding that increasing the batch size after some point doesn't correspond to equally great performance gains in terms of speed, leads us to such insights: Since increasing the batch size "too much" ceases to offer striking speed ups in training time and also slows down convergence, it is suboptimal to increase the batch size indefinitely. Besides, we found that too large batch size training is more prone to overfitting.

Since after 10 hours, the models with reasonable batch sizes (2048-8192) are all close to convergence (they are very far into their training), their accuracy are all high and close to each other as well; hence, we can't really observe the speed at which the model initially gets close to a minima, which usually happens in the first few iterations of the training. This initial stage is the defining factor for measuring the speed of convergence since the most accuracy gains happen here. Thus, we need to see which model has a faster initial stage than the others, and for that matter, we need a shorter time frame. We chose a time frame of 1 hour to repeat the experiment and observed if SCARS with large batch sizes converge faster. This time, however, the difference between the small and large batch sizes were greater and more stark. As can be seen from Table VII, we found that reasonably large batch size schemes converge faster than the model with a batch size of 2048; indeed, SCARS with a batch size of 8192 achieves 0.214% increase in accuracy compared to 2048; similarly, 4096 achieves 0.06% increase in accuracy. The model with 2048 batch size can only close this accuracy gap after ~ 1.6 more hours of training. This further shows that SCARS is able to train faster with large batch sizes and is time-friendly. This could prove to be very beneficial for commercial scenarios where the training time is limited and for research purposes where you would want to get quick results (or any kind of situation in which waiting for one day is not optimal).

TABLE VII
TEST ACCURACY AFTER 1 AND 10 HOURS OF TRAINING

	Speedup (s) for one iteration of SCARS					
	2048	4096	8192	16384	32768	65536
1 hour	80.510	80.573	**80.724**	80.457	80.066	79.587
10 hours	81.171	81.168	**81.176**	81.094	81.021	80.816

Consequently, when the hot embeddings are intelligently cached and therefore when we can use a reasonably large batch size, our method greatly reduces the training time with a faster convergence by achieving up to $6\times$ speed up from Table VI compared to the CPU-GPU baseline implementation of DLRM.

V. Conclusion

In this work, we develop a handy framework that can model the communication cost of various distributed systems. Based on that, we are able to minimize the expected cost by applying wise communication strategies for large recommendation system training. Specifically, our method utilizes the overlapping embeddings to transmit only the unique embeddings with their indices for each feature. We call this "coalescing." The other method utilizes the GPU as a cache for hot embeddings. We investigate several ways to determine which embeddings to store on GPU and come up both with a theoretical answer (the tradeoff between batch size and the size of the cached embeddings) as well as theory-inspired experimental solutions. As a result, our method generalize well to different datasets and systems with different memory constraints. Furthermore, our PyTorch implementation shows up to $6\times$ improvement on the training and convergence time of massive GPU systems, for both the Criteo Terabyte and Alibaba User Behavior datasets.

References

[1] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sun-daraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. arXiv preprint arXiv:1906.00091, 2019.

[2] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In Proceedings of the 29th Symposium on Operating Systems Principles, pages 348–363, 2023.

[3] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye,Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In Proceedings of the 1st workshop on deep learning for recommender systems, pages 7–10, 2016.

[4] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 431–445, 2021.

[5] Hui Guan, Andrey Malevich, Jiyan Yang, Jongsoo Park, and Hector Yuen. Post-training 4-bit quantization on embedding tables. arXiv preprint arXiv:1911.02079, 2019.

[6] Tigran Ishkhanov, Maxim Naumov, Xianjie Chen, Yan Zhu, Yuan Zhong, Alisson Gusatti Azzolini, Chonglin Sun, Frank Jiang, Andrey Malevich, and Liang Xiong. Time-based sequence model for personalization and recommendation systems. arXiv preprint arXiv:2008.11922, 2020.

[7] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph E Gonzalez, Ion Stoica, and Kurt Keutzer. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In Proceedings of Machine Learning and Systems, volume 2, pages 497–511, 2020.

[8] Shiwei Li, Huifeng Guo, Xing Tang, Ruiming Tang, Lu Hou, Ruixuan Li, and Rui Zhang. Embedding compression in recommender systems: A survey. ACM Computing Surveys, 56(5):121, 2024.

[9] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. Tt-rec: Tensor train compression for deep learning recommendation models. Proceedings of Machine Learning and Systems, 3:448–462, 2021.

[10] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. Proceedings of Machine Learning and Systems, 2:412–428, 2020.