CuWide: Towards Efficient Flow-Based Training for Sparse Wide Models on GPUs

Xupeng Miao, Lingxiao Ma, Zhi Yang, Yingxia Shao, Bin Cui, *Senior Member, IEEE*, Lele Yu, and Jiawei Jiang

Abstract—Wide models such as generalized linear models and factorization-based models have been extensively used in various predictive applications, e.g., recommendation, CTR prediction, and image recognition. Due to the memory bounded property of the models, the performance improvement on CPU is reaching the limitation. GPU is known to have many computation units and high memory bandwidth, and becomes a promising platform for training machine learning models. However, the GPU training for the wide models is far from optimal due to the sparsity and irregularity in wide models. The existing GPU-based wide models are even slower than the ones using CPU. The classical training schema of the wide models does not optimized for the GPU architecture, which suffers from large amount of random memory accesses and redundant read/write of intermediate values. In this paper, we propose an efficient GPU-training framework for the large-scale wide models, named cuWide. To fully benefit from the memory hierarchy of GPU, cuWide applies a new flow-based schema for training, which leverages the spatial and temporal locality of wide models to drastically reduce the amount of communication with GPU global memory. To do so, we adopt a bigraph computation model to efficiently realize the flow-based schema and exploit three flexible interfaces for programming. Further, we use the 2D partition of mini-batch (in sample and feature dimensions) with proposed graph abstraction to optimize GPU memory access for sparse data, and apply several spatial-temporal caching mechanisms (importance-based model caching and cross-stage accumulation caching mechanisms) to achieve a high performance kernel. To efficiently implement cuWide, we also propose several GPU-oriented optimizations, including feature-oriented data layout to enhance the data locality, replication mechanism to reduce update conflicts in shared memory, and multi-stream scheduling to overlap data transferring and kernel computing. We show that cuWide can be up to more than 20× faster than the state-of-the-art GPU solutions and multi-core CPU solutions.

Index Terms—Machine learning, wide model, linear model, GPU acceleration, parallel computation, shared memory architecture

1 Introduction

Wide model was first proposed in [1] and has been widely used in many practical big data applications. To describe such model more precisely, we describe its typical setup that the input data is a *sparse* matrix in $\mathbb{R}^{N\times d}$ and the goal is to find a dense vector $w \in \mathbb{R}^d$ that minimizes some (convex) loss function. Here, N is the number of samples and d is the feature dimension. For example, generalized linear model (GLM) class [2], [3], [4] including logistic regression (LR), linear SVM (LSVM), least square regression (LSR) and follow the regularized leader (FTRL) [5] are typical wide models, which can be expressed as a linear combination of sample features followed by an activation function. Since wide models [1], [6], [7] can learn an objective from a wide set of features, they have been applied among many industrial applications like recommender systems [1], click-through-rate prediction [8], [9], and image recognition [10]. Given that increases in model update latencies leads to losses in revenue or accuracy, these applications demand fast model training.

Due to such importance, a significant of computation libraries or systems has been conducted to efficiently implement wide models for multi-core CPU. For example, DIMM-Witted [11] is a state-of-the-art wide model implementation on multi-core CPU systems. However, the performance of existing wide models are still not satisfying. In general, the wide models are memory bounded, and the model training is limited by the memory bandwidth. Emerging hardware (e.g., GPU, TP, ASIC) with more resources are becoming more and more attractive for big data applications. In this paper, we focus on GPU, which provides much higher memory bandwidth than today's CPU architectures. Compared to DRAM, a single GPU's device memory provides $2-4\times$ more bandwidth for random access and $10-20\times$ more bandwidth for sequential access. In addition, GPU shared memory provides $20-50\times$ more bandwidth for sequential access and $200-600\times$ more bandwidth for random access. However, efficient utilization of GPU is challenging because of the (1) low computation-to-communication ratio, and (2) irregular memory access patterns of wide models. As a result, the parallel speedup of these applications is severely

- *Xupeng Miao, Lingxiao Ma, and Zhi Yang are with the Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University, Beijing 100871, China. E-mail: {xupeng.miao, xysmlx, yangzhi}@pku.edu.cn.*
- *Bin Cui is with the Key Lab of High Confidence Software Technologies (MOE), School of EECS, Institute of Computational Social Science, Peking University, Beijing 100871, China. E-mail: bin.cui@pku.edu.cn.*
- *Yingxia Shao is with the School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China. E-mail: shaoyx@bupt.edu.cn, leleyu@tencent.com.*
- *Lele Yu is with Tencent Inc., Shenzhen 518054, China. E-mail: jiawei.jiang@inf.ethz.ch.*
- *Jiawei Jiang is with ETH Zurich, 8092 Zurich, Switzerland.*

Manuscript received 1 Apr. 2020; revised 2 Nov. 2020; accepted 3 Nov. 2020. Date of publication 16 Nov. 2020; date of current version 5 Aug. 2022. (Corresponding author: Zhi Yang.) Recommended for acceptance by B. He. Digital Object Identifier no. 10.1109/TKDE.2020.3038109

limited by the random nature of their memory access patterns, a fundamental property of wide models.

Recent general GPU accelerated machine learning frameworks (e.g., TensorFlow [12], PyTorch [13]) allow to implement wide models on GPU by expressing the computation as operations on tensors. We observe that although such an implementation exhibits several attractive features in programming and flexibility, it has potential efficiency limitation due to inefficient memory accesses, both quantitative as well as qualitative. First, we note that the access frequency distribution of feature (and thus parameter) is often approximate to the power-law distribution [14], [15], such skewed access pattern providing optimization opportunities of leveraging GPU memory hierarchy. However, learning frameworks completely loses such access pattern with the primitive of tensor abstractions. We also observe that the tensor operations have to write many intermediate data (such as the partial sums, predictions, loss and partial gradients) during the forward stage and read them back again in the backward stage, which increases the number of reads and writes on the global memory. To accelerate the model training, the shared memory is a better option for temporally storing the intermediate data. However with the tensor abstraction, the obstacle of doing so is that the capacity of shared memory is small, while the size of gradients can be very large since it is proportional to the number of features. For example, NVIDIA Pascal GPU only has 96 KB per streaming sultiprocessor (SM). But the number of features can easily exceed millions, which incurs more than 1 MB memory.

Our premise is that by changing the focus of computation from a single tensor to a cacheable data partition (e.g., chunks), we can effectively exploit fine-grained data access pattern of wide models to leverage GPU memory hierarchy, through carefully partitioning and scheduling data (and model) onto the GPU shard memory. Based on these insights, we propose an efficient GPU training framework for the wide models, named cuWide. CuWide applies mini-batch SGD algorithm to train the models, and each mini-batch is processed by the GPU. To fully exploit the advantage of the memory hierarchy of GPU, we design a new training schema, called flow-based schema, for the wide models. In this schema, a mini-batch on GPU is further divided into small chunks, and each chuck is computed by the feature aggregation, which computes the sum of weighted feature for each sample in the forward phase and pushes the computation of gradients into the backward phase. The feature aggregation guarantees the size of intermediate data between the two phases is small enough to be stored in shared memory. To harness the characteristics of training data, cuWide mainly uses two optimization techniques to achieve a high-performance training: 1) Importance caching aims at decreasing the number of random global memory access when updating model parameters; 2) Cross-stage accumulation caching aims at removing a large amount of global memory accesses of intermediate data during forward-backward stages.

The main contributions are summarized as follows:

New Flow-Based Training Schema on Bigraph. To realize the flow-based schema on the top of GPU, we introduce a bigraph computation model, where a mini-batch is represented by a bigraph. In the bigraph, samples are represented by sample nodes, features are represented by feature nodes, and the edges indicate the relationship between features and samples. Furthermore, the flow-based schema on bigraph provides a vertex-centric programming model Aggregate-Loss-Apply (*ALA*) to express the execution of various wide models over a bi-graph. From this graph view, cuWide could derive a cache-efficient partition and scheduling scheme that removes the large amount of random memory access from the global memory to the on-chip memory, resulting in optimized memory performance. Specifically, cuWide generates chunks using the 2D partition of mini-batch in sample and parameter dimensions. The size of chunk is selected in such a way that it can be accommodated in GPU shared memory. Each processing unit (e.g., SMs inside a GPU) streams the chunk of given samples on shared memory.

Fine-Grained Spatial-Temporal Caching Mechanisms. To achieve efficient GPU training of cuWide, we carefully investigate the temporal locality of memory access in wide models, and propose two optimization techniques. 1) According to the empirical study, the sparse training data has a skew feature distribution, thus incurring nonuniform global memory access when updating model parameters. The important features with high frequency have more chance to be concurrently updated, which makes parallel computation degenerate into serial computation. By making the trade-off between global memory and shared memory, we design importance cache to improve the efficiency of model update. The importance cache selects a subset of important features on basis of a cost model. 2) Through refactoring the training into `Aggregate`, `Loss` and `Apply` stages, we observe that gradients and predictions of wide models can be formulated as functions on certain intermediate scalars, denoted by `Accums`, which can be aggregated from input features and model parameters. Such temporal locality of `Accum` data across stages implies that we can reduce the global memory access for writting/reading intermediate data by caching aggregated feature in shared memory. In the light of this, we present a cross-stage accumulation caching mechanism that could perform the computation across stages in the shared memory and thus removing a large amount of global memory accesses of intermediated data .

Efficient System Implementation. We carefully implement the prototype of cuWide based on aforementioned technique contributions. To efficiently implement the system, we also propose several GPU-oriented optimizations, including feature-oriented data layout to enhance the data locality, and multi-stream scheduling to overlap data transferring and kernel computing. We conducted comprehensive experiments on four real-world data sets. The experimental results demonstrate that cuWide can be at least 22× faster than the state-of-the-art GPU solution Tensorflow, 18-50× faster than the state-of-the-art multi-core CPU solution DIMMWitted.

2 Preliminaries

In this section, we first briefly introduce two wide models, i.e., GLM and FM. Then we review the training algorithm, mini-batch SGD. Finally, we describe the characteristics of GPU architecture.

TABLE 1
Wide Model Examples: LR, LSVM and LSR

Model	Loss Function	Gradient
LR	$log(1+e^{-y\boldsymbol{wx}})$	$\frac{-y\boldsymbol{x}}{1+e^{y\boldsymbol{wx}}}$
LSVM	$max(0, 1-y\boldsymbol{wx})$	$y\boldsymbol{x}$, if $1-y\boldsymbol{wx} > 0$
LSR	$(y-\boldsymbol{wx})^2$	$2(\boldsymbol{wx}-y)\boldsymbol{x}$

2.1 Wide Model

Wide model has been successfully applied in many recommender systems (e.g., Google [1]). For large-scale online recommendation and ranking systems in an industrial setting, GLMs (e.g., LR, LSVM, LSR and FTRL [5]) are widely used because they are simple, scalable and interpretable. The concept of *wide model* is a substitute for linear model consisting of the multiplication operation of a sparse tensor and a dense vector. Formally defined as: $\hat{y}(\boldsymbol{x}) = \boldsymbol{wx}$, where $\boldsymbol{w} \in \mathbb{R}^m$ is the model parameters, $\boldsymbol{x}$ is the m-dimensional feature vector and $\hat{y}$ is the prediction. The loss function of a given wide model can be formulated as: $L(\boldsymbol{w}) = \sum_{i=1}^{n} l(y_i, \boldsymbol{wx_i})$. The goal of wide model is to minimize the loss function and find $\boldsymbol{w} = argmin_{\boldsymbol{w}} L(\boldsymbol{w})$. Table 1 lists the loss functions of some classical wide model examples.

Based on the computation pattern definition, we further generalize the wide model with factorized models (FM) [16]. FM generalizes to previously unseen feature interactions by learning a low-dimensional latent embedding vector for each feature, with less burden of feature engineering. The equation for a FM can be defined as

$$\hat{y}(\boldsymbol{x}) = \boldsymbol{wx} + \sum_{p=1}^{m} \sum_{q=p+1}^{m} \langle \boldsymbol{v}_p, \boldsymbol{v}_q \rangle x_p x_q, \tag{1}$$

where m is the feature dimension, $\boldsymbol{w} \in \mathbb{R}^m$ are the feature weights, and $\boldsymbol{v}_p$, $\boldsymbol{v}_q \in \mathbb{R}^k$ are hidden vectors describing the variables x_p and x_q with k factors. After the reformulation in [16], it can also be formulated as the weighted summation when the latent dimension k is low (see details in Section 3.2). In consequence, we choose GLMs and FM (with a low latent dimension k) as two representative instances of wide models.

2.2 Mini-Batch Stochastic Gradient Descent

Mini-batch stochastic gradient descent (SGD) is a popular optimization method in machine learning. It makes a balance between convergence speed and accuracy degradation compared to the standard SGD and can be implemented on parallel architectures easily. Algorithm 1 illustrates the classical procedure of mini-batch SGD. Each iteration of the mini-batch SGD (Lines 5-11) consists of the forward stage (Lines 6-8) and the backward stage (Lines 9-11). In the forward stage, the algorithm computes loss and gradient; Then it updates the model with the gradient in the backward stage.

2.3 GPU Architecture

GPU, with SIMT architecture, is often used to accelerate data-intensive machine learning algorithms, especially for deep learning [17], [18]. However, in some cases, GPU programs can be slower than highly-optimized multi-core CPU solutions. Thus, GPU optimization designed for specific models is significant for system performance. We use NVIDIA GPU and its CUDA programming interface [19] in this paper. A GPU function in CUDA programs is called a kernel. Each GPU contains several streaming processors (SMs) and each SM contains individual CUDA cores, warp schedulers and registers.

Algorithm 1. Mini-Batch SGD Algorithm

Input: Data Set: S, Batch Size: n, Epochs: E
1: Initialize model parameters $\boldsymbol{w^{(0)}}$;
2: Shuffle the data set;
3: **for** $i = 0$ to E epochs **do**
4: **for** $t = 0$ to $\frac{|S|}{n}$ iterations **do**
5: $D \leftarrow$ Mini-batch with n elements from S;
6: //Forward stage;
7: $f \leftarrow L_D(\boldsymbol{w^{(t)}})$ //Compute loss;
8: $\boldsymbol{g} \leftarrow \Delta L_D(\boldsymbol{w^{(t)}})$ //Compute gradient;
9: //Backward stage;
10: $\Delta \boldsymbol{w} \leftarrow -\eta \boldsymbol{g}$ //Update rule;
11: $\boldsymbol{w^{(t+1)}} \leftarrow \boldsymbol{w^{(t)}} + \Delta \boldsymbol{w}$ //Model update;
12: **end for**
13: **end for**

GPU supports hundreds of thousands of threads, sharing the same instruction stream. These threads are organized into thread blocks and at the hardware level each thread block only exists in one SM. Every consecutive 32 threads in a block are organized into thread warps that execute the same instruction at a time. Global memory is the largest memory in a GPU (e.g., 11 GB for GTX 1080ti) and addressable for all threads. Compared with global memory, shared memory has much shorter latency and higher throughput. As the special programmable on-chip memory, the shared memory is only addressable for threads in a thread block [20]. In addition, the shared memory is quite small, only up to 96 KB per SM for NVIDIA Pascal GPU.

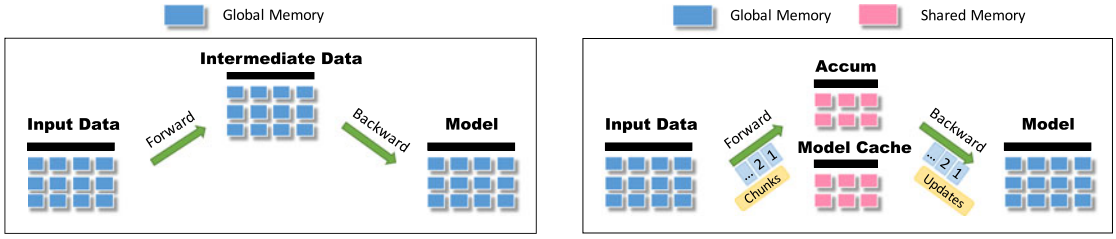
3 Motivation of Flow-Based Training Strategy

In this section, we introduce the flow-based training strategy, which can benefit from the memory hierarchy of GPU. First, we describe the classic training strategy of wide models on GPU, called the stage-based strategy. Then we introduce the data access locality for wide models, followed by the elaboration of the flow-based strategy.

3.1 Stage-Based Strategy

Existing popular ML systems, e.g., TensorFlow, PyTorch and MXNet, adopt stage-based strategy during wide model training on GPU, as illustrated in Fig. 1a. The strategy calculates prediction errors in the forward stage and uses gradients to update the model in the backward stage. Such stage-based model has the following two problems, leading to low memory performance.

Model Access. Although sparse matrix layouts like Compressed Sparse Row (CSR) store all non-zero elements of a row sequentially in memory allowing fast row major traversal of data matrix, the nonzero columns in adjacency



(a) Stage-based strategy (b) Flow-based strategy

Fig. 1. The two strategies for training.

matrix can be scattered anywhere in the matrix and reading their values can result in highly random accesses to the model vector. Similarly, the backward direction process suffers from the similar random write accesses to the model vector.

Intermediate Data Generation. The system has to write intermediate data (such as the partial sums, predictions, loss and partial gradients) during the forward stage and read them back again in the backward stage, which increases the number of reads and writes on the global memory.

As introduced in Section 2.3, shared memory in GPU has much lower latency and higher memory bandwidth than global memory. As wide models are bound by memory access speed, it is highly required but remain challenging to leverage GPU memory hierarchy, given high-dimensional model parameter/itermediate data and limited shared memory. For example, LR on the criteo data set with 1 million features needs 3.8 MB to store the feature weights, but the shared memory only has a limited size (up to 96 KB). Therefore, the stage-based strategy can hardly benefit from the advantages of shared memory.

3.2 Spatial-Temporal Locality of Wide Models

By carefully examining the memory access characteristics of the wide models, we find that these models exhibit spatial-temporal locality in memory access patterns, providing optimization opportunities of leveraging GPU memory hierarchy.

Spatial Locality. For the wide model, the feature frequency indicates the importance of each feature, which is proportional to the memory access frequency of this feature during the training. More concretely, in many large-scale machine learning problems, it is very common that different features are not uniformly updated during the training [14], [21], [22]. Most features only show up in few samples, and the top features are nonzero for almost all samples. During backward stage. Fig. 2 shows the feature distributions of three real data sets. The results imply the feature almost follows a power law distribution, where the feature frequency is computed as

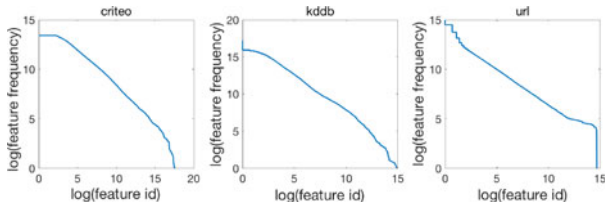


Fig. 2. Feature distributions over three real datasets.

$$f_i = \frac{\#samples\ with\ nonzero\ feature\ i}{\#all\ samples}. \tag{2}$$

This feature skewness implies the spatial locality in accessing model parameters and opportunities for gains by caching importance features in GPU shared memory.

Temporal Locality. Through analysis, we observe that gradients and predictions of wide models can be formulated as functions on certain intermediate scalars, denoted by Accums, which can be aggregated from input features and model parameters. We first take the LR model as an example

$$\hat{y} = \frac{1}{1+e^{-\boldsymbol{wx}}} = eval_{lr}(\boldsymbol{wx}) \tag{3}$$

$$\boldsymbol{g_w} = \frac{-y\boldsymbol{x}}{e^{y\boldsymbol{wx}}+1} = grad_{lr}(\boldsymbol{wx}, \boldsymbol{x}, y), \tag{4}$$

where $\boldsymbol{w}$ are the model parameters, $\boldsymbol{x}$, y are the inputs and $\boldsymbol{wx}$ are the Accums. For the FM model, the feature aggregation is

$$\hat{y} = \boldsymbol{wx} + \frac{1}{2}\sum_{i=1}^{k}\left((\boldsymbol{v}_i\boldsymbol{x})^2 - \sum_{j=1}^{m}\boldsymbol{v}_{ij}^2\boldsymbol{x}_j^2\right) = eval_{fm}\left(\boldsymbol{wx}, \boldsymbol{v}_i\boldsymbol{x}, \sum_{j=1}^{m}\boldsymbol{v}_{ij}^2\boldsymbol{x}_j^2\right), \tag{5}$$

$$\boldsymbol{g}_{\boldsymbol{v}_{ij}} = \boldsymbol{x}_i\boldsymbol{v}_j\boldsymbol{x} - \boldsymbol{v}_{ij}\boldsymbol{x}_i^2 = grad_{fm}\left(\boldsymbol{wx}, \boldsymbol{v}_i\boldsymbol{x}, \sum_{j=1}^{m}\boldsymbol{v}_{ij}^2\boldsymbol{x}_j^2\right), \tag{6}$$

where $\boldsymbol{w}$ and $\boldsymbol{v}$ are model parameters, $\boldsymbol{x}$, y are inputs and $\boldsymbol{wx}$, $\boldsymbol{v}_i\boldsymbol{x}$, $\sum_{j=1}^{m}\boldsymbol{v}_{ij}^2\boldsymbol{x}_j^2$ are Accum data. Such temporal locality of Accum data during the forward-backward stages implies that we can reduce the global memory access for writting/reading intermediate data by caching aggregated feature in shared memory.

3.3 Flow-Based Strategy

Based on the observation of data locality pattern, we design a new training strategy, called flow-based strategy, as illustrated in Fig. 1a. In this strategy, we perform a two-dimensional (2D) partition over the mini-batch, into chunks over the sample dimension and segments over the feature (i.e., model) dimension, respectively. The size of partition is selected in such a way that the range of samples or features contained in it can be accommodated in cache. Then each chunk is processed one-by-one and the Accum, rather than gradient, is temporally stored in the shared memory. In the backward phase, the update operation uses these Accums to calculate gradients,

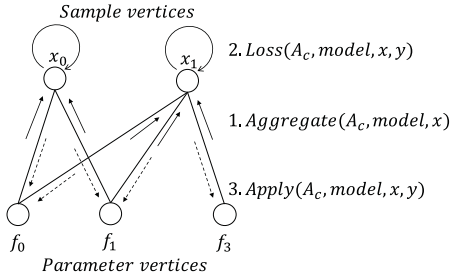


Fig. 3. The *ALA* processing model.

thus enjoying good temporal locality with accesses to the partial sums being served by cache. At the same time, the forward and backward phase read and update important parameters from a model segment cached on shared memory, it enjoys good spatial locality as well. The flow-based strategy processes chunks in a streaming manner, and synchronize the updates to the model when all chunks in a mini-batch have been processed. In this way, the scheme exploits spatial-temporal locality and GPU memory hierarchy to optimize memory accesses of wide model.

4 cuWide Design

Based on the flow-based strategy, we propose cuWide which factors the execution of wide model training into three phases: aggregate the feature, compute the loss and apply the gradient. We formulate the sparse data of wide models by a bigraph representation and design an Aggregate, Loss and Apply processing model on top of the graph. We then design a bigraph-based execution engine with effective caching scheme to optimize the GPU memory access.

4.1 Programming Abstraction

Leveraging the locality access pattern in wide model is nontrivial on GPU with SIMT architecture, which requires carefully partitioning and scheduling data (and model) onto the memory hierarchy in GPU. The general GPU machine learning frameworks completely loses such access pattern with the primitive of tensor abstractions. cuWide adopts a new *graph-based* programming model, which constructs the sample-feature (parameter) interaction during the training as a bi-graph. To do so, cuWide provides a vertex-centric programming model `Aggregate-Loss-Apply`, allowing to express the execution of various wide models over a bigraph. With such programming model, cuWide could enable the flow-based training strategy through exploiting the fine-grained irregularity within the sparse input data matrix, e.g., performing 2D partitioning over the sample and feature dimensions.

We formulate wide models by a bigraph model, where sample vertices represent samples, parameter vertices represent different features and edges represent the relations between samples and features. For each iteration of mini-batch SGD, the number of sample vertices is the batch size, the number of parameter vertices is the feature dimension of data set and the edges between them represent the sparse input. For example, as shown in Fig. 3, the sample vertex $\boldsymbol{x_0}$ has two adjacent parameter vertices (f_0, f_1), which means that $\boldsymbol{x_0}$ has two nonzero features (f_0, f_1) in the dataset. Each parameter vertex stores the model parameters corresponding to the feature, and sample vertices store the `Accum` corresponding to the samples. We construct the bigraph for wide models during the data loading.

```
Aggregate(A_c,x,w):
  A_c ← wx
Loss(A_c,y):
  return log(1 + e^{-yA_c})
Apply(A_c,w,x,y):
  w ← w + α(−yx)/(e^{yA_c} + 1)
```

Fig. 4. LR implementation with *ALA* interfaces.

The standard graph model assumes a homogeneous set of vertices, cannot express bi-graphs that have different types of vertices playing distinct roles (e.g., feature vertices and parameter vertices). So for our bi-graph construction, we present a processing model with three interfaces to describe the computation procedure over the bigraph: Aggregate, Loss and Apply, *ALA* for short, as shown in Fig. 3. CuWide allows a user to customize the computation in the flow-based strategy on bigraph by the *ALA* interfaces.

`Aggregate`: In this stage, sample vertices aggregate adjacent parameter vertices and compute the intermediate `Accum`. Each sample vertex only accesses its own `Accum` during computation.

`Loss`: The Loss interface defines the model and also tracks the training progress. The inputs of the Loss interface are target values and `Accum`, and the output is the current loss of the model.

`Apply`: In this interface, parameter vertices fetch the `Accum` from adjacent sample vertices, finish the gradient computation and update the model.

Figs. 4 and 5 shows the LR and FM implementation using *ALA* programming model. For the FM model in cuWide, we not only calculate **wx** (i.e., the dot product between the dense model parameters **w** and the sparse feature vector **x**), but also $2k$ additional `Accums` including $\mathbf{v}_i\mathbf{x}$ and $\sum_{j=1}^{m} \mathbf{v}_{ij}^2\mathbf{x}_{ij}^2$, where $i \in [1, k]$. The computation of these extra `Accums` are similar to that of **wx** in cuWide. We assume the hidden dimension of FM k is far more less than the feature dimension. Therefore, the main computation operation of FM is still a "multiplication" of a sparse matrix (a mini-batch of **x**) and dense vectors ($\mathbf{v}_i$). If k becomes larger, it turns into a multiplication of a sparse matrix and a dense matrix and requires further optimization. Based on the interface, the user can benefit from our system advantages by easily customizing the loss function to implement other wide models in C++ or Python.

Note that the bi-graph construction and partition is only executed once before the training, which is implemented by

```
Aggregate(A_c,x,w,v):
    A_c1 ← wx, A_c2i ← v_i x, A_c3i ← Σ_{j=1}^{m} v_ij^2 x_j^2
Loss(A_c,y):
    return A_c1 + (1/2) Σ_{i=1}^{k} ((A_c2i)^2 − A_c3i) − y
Apply(A_c,w,v,x,y):
    w ← w + αx, v_ij ← v_ij + α(x_i A_c2j − v_ij x_i^2)
```

Fig. 5. FM implementation with *ALA* interfaces.

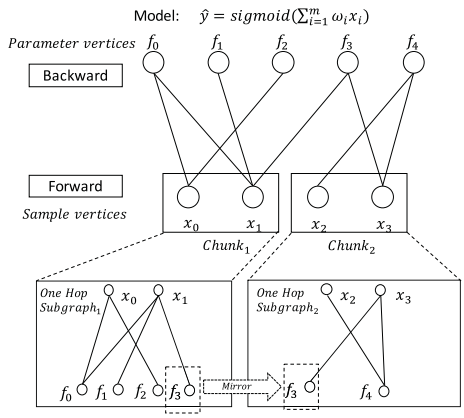


Fig. 6. Bigraph representation by vertex-cut subgraph of LR.

building the chunk index on the input datasets. Its cost is mainly dominated by the time for performing a single pass over the data, which is significantly smaller than that required by training with repeated passing over the data (i.e., epochs). For example, we measured this pre-processing cost for training LR on criteo-s, which is only hundreds of milliseconds and is ignorable compared to training time.

4.2 Bigraph-Based Execution Engine

We present the bigraph-based execution engine as the system realization of the flow-based strategy. The duty of this engine is to execute the customized calculations defined by *ALA* on the bigraph abstraction.

To execute flow-based strategy, the sample nodes are partitioned into chunks so that their `Accum` can be cached in shared memory, as shown in Fig. 1a. In the view of graph, we create a subgraph by extracting the one-hop neighborhood of the sample vertices in the chunk. Fig. 6 shows a vertex-cut example of bigraph representation of LR. $Chunk_1$ contains $(\boldsymbol{x_0}, \boldsymbol{x_1})$, and the neighborhood of the sample vertices $(\boldsymbol{x_0}, \boldsymbol{x_1})$ are (f_0, f_1, f_2, f_3). And $Chunk_2$ contains $(\boldsymbol{x_2}, \boldsymbol{x_3})$, the corresponding subgraph contains sample vertices (f_3, f_4), where f_3 is mirrored from the one in the previous subgraph.

Fig. 7 illustrates the execution process of cuWide engine, where the sparse inputs are organized as mini-batches and transferred into GPU's global memory. The execution engine reads a chunk from the mini-batch, generates a subgraph, and executes the training algorithm with the subgraph. cuWide adopts a fine-granular task-based execution model that treat SMs inside a GPU as standalone workers.

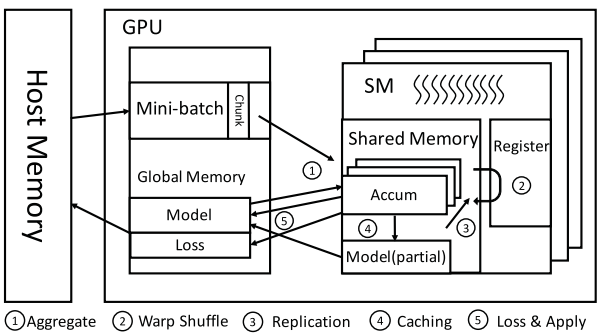


Fig. 7. The execution flow of cuWide engine.

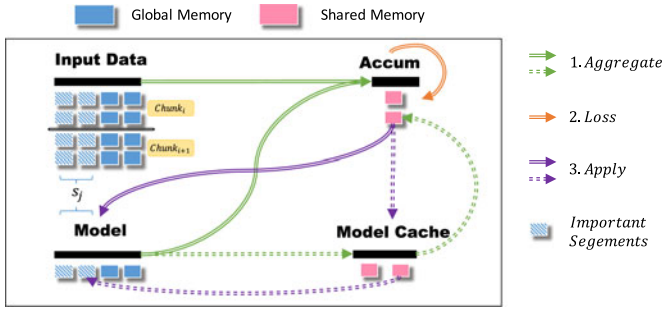


Fig. 8. Illustration of cuWide execution. Note that the solid lines refer to the Accum caching and the dotted lines refer to the model caching.

Each chunk computation is processed by a worker following the ALA processing model. The training procedure is executed on top of a subgraph.

In the `Aggregate` stage, sample vertices are activated. The worker computes `Accum` for sample vertices with inputs and feature weights along edges from parameter vertices, and stores `Accum` into shared memory. In the view of GPU, threads of one GPU block process the subgraph, where each edge represents a thread. The thread executes computation instructions with data from its connected node.

In the `Loss` and `Apply` stage, feature nodes are activated, and they gather `Accum` from adjacent sample nodes and update the model. Similarly, in the view of GPU, each thread of a block uses the `Accum` in the shared memory and inputs in the global memory to calculate gradients and predictions, and updates the model parameters in the global memory. For each subgraph, the `Apply` phase could not start until all sample nodes finish the forward computing. When processing chunks one-by-one, the training procedure generates model updates in sequence as well. When model updates of all the chunks in a mini-batch are collected, the aggregated updates synchronized to the model.

4.3 Locality-Aware Access Optimizations

As previously discussed, both the spatial locality of feature access pattern and temporal locality of feature aggregation data `Accum` providing optimization opportunities of leveraging GPU memory hierarchy. To leverage these opportunities, we propose a cache-efficient partition and scheduling scheme that removes the large amount of (random or extra) memory access from the global memory to the on-chip memory. As illustrated in Fig. 8, cuWide divides the shared memory into two parts, storing important model parameters for spatial locality and aggregation data `Accum` for temporal locality.

Importance-Based Model Caching. As we can see, model parameter (e.g., w for LR) is a source of data appeared in different stages. Large amounts of random access on global memory are involved when updating model parameters, leading to low performance kernel. Therefore, caching the model parameter with shared memory will improve efficiency. However, it's impossible to save the whole high dimensional model parameter into the limited shared memory. The basic idea of caching is to store the frequently accessed model data in high bandwidth memory. For the wide model, the feature frequency indicates the importance of each feature, which is proportional to the memory access frequency of this feature during the training.

Based on the skew feature distribution, we design an importance caching strategy to optimize training performance. Assume that the feature indexes are replaced in descending order of the frequency of features and are partitioned into segments. The size of segment and chunk are selected in such a way that the ranges of feature and sample vertices contained in them can be accommodated in cache, respectively. Then the model parameters, ordered by replaced feature indexes, is equivalent to the parameters ordered by feature importance. The important model parameters, which are ranked high, are cached in shared memory and written back to global memory at the end of each iteration.

To utilize the GPU memory hierarchy, the system processes a chunk at the granularity of segments with importance-based model cache. As illustrated in Fig. 8, in the `Aggregate` stage, the system reads a segment j of the model, and updates the `Accum` data in the shared memory. The system caches the feature node segment (i.e., model segment) in the shared memory if the access locality of this segment is relatively high. We measure the locality in terms of the average degree $E_j[d]$ of feature nodes in this segment. To enhance the locality, the feature node could be sorted by their degree. The whole segment would be loaded onto shared memory and each sample node reads model parameters (at this segment) from the cache if $E_j[d]$ is larger than a threshold th $(th > 1)$. Otherwise, the system directly reads feature nodes from the global memory. Note using cache needs coalesced shared memory access operations read from and write back to global memory, we choose the threshold $th = 2c/r$. Here, c means cost of coalesced memory access and r means cost of random memory access. The `Loss` and `Apply` stages is similar to the `Aggregate` stage, where the system loads the segment in the shared memory if $E_j[d] \geq th$, and perform partial gradient computation.

Cross-Stage Accumulation Caching. The classical training schema of wide models calculates gradients in the forward phase and uses gradients to update the model in the backward phase. Unlike the classical one, we introduce feature aggregation for the wide models, which formulates gradients and predictions of wide models as functions on certain intermediate scalars, denoted by `Accums`. `Accums` can be aggregated from input features and model parameters, So we present the cross-stage accumulation caching which changes the intermediate storage from global memory to shared memory by using feature aggregation. As illustrated in Fig. 8, `Accum` of sample vertices are stored in shared memory. In the `Aggregate` stage, for each segment, the system performs partial feature aggregation for this feature segment, and adds it on the `Accum` in the shared memory, and continues to perform the next segment. After all the segments of a chunk has been processed, the system enters `Loss` and `Apply` stage. Then it reads just a specific segment of the model, and computes the loss and partial gradient for this model segment with the `Accum` data *in the shared memory*. After this segment has been processed, the system adds the partial gradient accumulation back onto global memory and continues to perform the next model segment. As the computation of aggregation, loss and gradients updates are performed in shared memory, the cache of accumulation helps the system to avoid redundant global memory accesses to writing/reading intermediate data across stages.

Note that the processing a segment (i.e., a partial chunk of features) does not restrict the thread parallelism because the multiple partial chunks could be processed in parallel. The processing only limits the data access locations (i.e., shared memory and global memory) for a segment based on the spatial locality, without blocking data access.

5 System Implementation

To effectively utilize the GPU, we further propose two GPU-oriented optimizations for implementing cuWide, including columnar data representation to enhance caching efficiency, and multi-stream execution to overlap the data transfer with computation. The source code of cuWide has been made publicly available at [23]. Below, we will present each optimized implementation component, respectively.

5.1 Data Layout Optimization

In machine learning system, sample-oriented formats, e.g., coordinate list (COO) and compressed sparse row, are used to organize the large-scale sparse data. They store all nonzero elements of a row sequentially in memory allowing fast sample-oriented traversal of feature matrix. However, the neighbors of a node (nonzero columns in adjacency matrix) can be scattered anywhere in the whole graph. The sample-oriented format makes the training performance suffer from a lot of random memory access caused by using feature indexes to access the model parameter. A transaction executed by a warp with 32 threads loads a continuous memory block of size 128 KB. The *coalesced memory access*, which is an efficient memory access pattern, coalesces the loading and storing of the global memory issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. To exploit the advantage of the coalesced memory access, we should align the memory access by using proper data organization.

To optimize data layout, we assume that the feature indexes are replaced in descending order of the frequency of features. We partition sample nodes into chunks, and organize the data of each chunk with feature-oriented (column-oriented) storage format, which sorts entries by feature index first and then by the sample index. The feature-oriented format is similar to the columnar representation that is commonly used in databases [24], [25] for the efficient computation of aggregate queries and other ML systems [26]. Then the model parameters, ordered by replaced feature indexes, is equivalent to the parameters ordered by feature importance. This ensures the locality (and thus the chance of cache) when the system divides them into segments.

Note a mini-batch could contain a large number of chunks. During the training process, cuWide only reshuffles the accessing order of different chunks to randomly form mini-batch at each epoch, with the training instances in each chunk remaining the same. This can avoid data shuffling overhead while ensuring the convergence rate.

5.2 Conflicts Resolution

The naive implementation of the aggregate on the sample vertices of the subgraph incurs many conflicts, because of the internal architecture of shared memory. Physically, shared memory of GPU is divided into equally sized memory

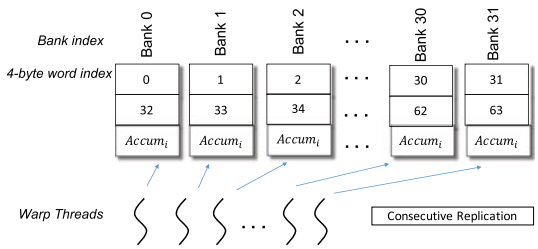
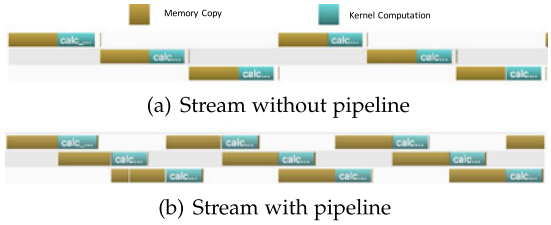


Fig. 9. Replication policy of shared memory.



(a) Stream without pipeline

(b) Stream with pipeline

Fig. 10. Stream Profiling for FM (k=3) on criteo.

modules, called banks. For example, the shared memory in NVIDIA GPU has 32 banks and each bank is 4 bytes wide. Different banks can be accessed simultaneously, but each bank can only accept one access at a time. Therefore, bank conflicts happen if threads in a warp try to load/store data from/to the same bank. Then the access has to be serialized, which is even worse than global memory access. Another potential problem is the atomic conflict. When threads in a warp finish calculation of the forward stage, atomic operations cannot be avoided because of simultaneously memory access of the same entry in Accum vector. Atomic memory access involves lock step and may seriously affect performance.

We adopt a sample vertex replication policy to eliminate or reduce these conflicts. Suppose the Accums form a vector, we replicate every entry and make all replicas consecutive. If all 32 threads in a warp access the same entry of the Accum vector, they can be scheduled to different replicas in different banks. Thus, conflicts are avoided. Fig. 9 shows the situation for 32 consecutive replications of $\texttt{Accum}_i$. Each thread in a warp can only access one bank data to avoid conflicts. However, we can still reduce the number of replications, retain few bank conflicts to save limited shared memory and increase mini-batch size. The number of replicas is not only limited by the size of shared memory, but also achieves a balance between the cost from conflicts and the cost from aggregation. There are several consecutive replicas for entries of Accums. To avoid extra threads synchronization, we aggregate them by exploiting the registers with the warp shuffle instruction "SHFL", which is efficient and supported by hardware.

5.3 Scaling With Stream Pipeline

The computation logic we discussed before is assumed that all training data can be fit into GPU global memory. However, when facing large-scale training data, a streaming copy technique is necessary because of limited CPU-GPU memory bandwidth (only about 16 GB/s (PCIe v3.0x16) or up to 80 GB/s (NVLink)). Fortunately, asynchronous data transfer in GPUs can be achieved by using the asynchronous GPU streams, which could hide memory access latency from GPUs to main memory and improve the utilization of GPU's computing power. Note GPU stream pipeline differs from the GPU warp pipeline in that the former is is managed by the programmers with the CUDA stream APIs hiding the CPU host memory access latency, whereas the later managed by the streaming multiprocessor (SM) scheduler for hiding the GPU global memory access latency.

To support efficient GPU streams, we propose a transmission protocol between main memory and device memory. Assume the training data is divided into several mini-batches in main memory, and there is a replication of model parameters in global memory, which needs to be synchronized after each epoch. The whole workflow has three procedures: 1) Streaming copy batch data from host memory to bigraph buffer and label buffer in GPU global memory. 2) GPU kernel computation. 3) When the kernel is done, synchronize the model parameters with host memory. Then we propose the stream pipeline to overlap the computation and transmission.

We can set more streams for concurrency. However, the number of streams is three in our system, which is determined by dual copy engines of GPU architecture. The hardware has three queues: one computation engine queue and two copy engine queues (one for D2H and one for H2D). Therefore, 3-way stream pipeline can utilize GPU memory bandwidth and computation resources as many as possible. Fig. 10 shows the real profiling results of FM on criteo dataset, we can see the difference of stream pipeline clearly.

5.4 Kernel Optimization

Based on the ALA programming model, we present a brief kernel code sample in Fig. 11. We highlight several standard optimization techniques in the code snippet.

Persistent Threads. Persistent threads is a CUDA programming style which sets the work size just fits the physical SM capacity and each block pulls new work from a queue, rather than launching more thread blocks than the hardware could execute simultaneously. Using persistent threads we can treat SMs inside a GPU as standalone workers. Each chunk computation is processed by a worker following the ALA processing model.

Vectorized Memory Access. GPU's assembly instructions LD.E and ST.E load and store 32 bits from those addresses. We can improve performance of this operation by using the vectorized load and store instructions LD.E.{64,128} and ST.E.{64,128}. These operations can load and store data in 64 or 128 bit widths. Using vectorized loads, we reduce the total

Fig. 11. A brief kernel with highlighted optimization techniques.

TABLE 2
Data Sets Statistics

Dataset	#Samples	#Features	Density	Size
criteo-s	5,134,149	1,000,000	4e-5	3 GB
kddb	19,264,097	29,890,095	1e-6	4.8 GB
url	2,396,130	3,231,961	4e-5	2.1 GB
kdd12	149,639,105	54,686,452	2e-7	21 GB
criteo	45,840,617	1,000,000	4e-5	25 GB

TABLE 3
Configuration of the Evaluated Platform

CPU	2x Intel Xeon CPU E5-2650v4, 256 GB (up to 480 GFLOPS and 76.8 GB/s)
GPU	NVIDIA GTX 1080ti, 28 SM, 11 GB (up to 11.3 TFLOPS and 340 GB/s)
PCIe	3.0x16 (up to 16GB/s)

number of instructions, reduces latency, and improve bandwidth utilization.

On-Chip Cache. NVIDIA GPUs have on-chip L1 cache for each SM and allow programmers to control the cache behavior of each memory instruction. While many GPU applications do not benefit from the cache due to cache contention, some memory instructions may benefit from the cache as the accessed data may be frequently reused in the near future (temporal reuse) or by other threads (spatial reuse). Using the intrinsic instruction _ldg [1] to enable cache-assisted read may help improve system performance

6 Experiment

In this section, we evaluate both the convergence performance and the average epoch time of cuWide by comparing with the state-of-the-art machine learning systems. We select LR and FM from GLMs and factorization models to evaluate cuWide performance, as they are the most representative wide models and have been widely used in many real-world scenarios [1], [8]. We also conduct experiments to demonstrate the effectiveness of caching mechanism. Finally, we evaluate cuWide on the datasets whose size are larger than the GPU memory.

6.1 Experimental Setup

All the experiments are conducted on a server who has 256 GB memory and dual 12-core CPUs with hyper-thread enabled. The experimental results are repeated for ten times and the average results are reported to overcome random errors and warm cache effects. We use NVIDIA GTX 1080ti as the GPU platform, and the detailed configurations of the GPU are shown in Table 3. All the execution time of GPU kernel is collected by `nvprof` [27].

Dataset. We use four high dimensional sparse data sets from real world [28]. Table 2 lists the statistics of these data sets. All of these data sets are collected from CTR prediction problems. We apply *log* loss function for both LR and FM in our experiments. According to the data size, we classify the criteo and kdd12 as large datasets, since their sizes are larger than 20 GB. In addition, we also create a small data set from criteo data set by random sampling, named criteo-s. We use partial dataset (i.e., criteo-s) for kernel evaluation whereas using the complete real datasets (i.e., criteo, kdd12) for incorporating host transfer overhead.

Baselines. In the experiments, we compare these following systems with our cuWide:

TensorFlow [12] is an open source machine learning framework which supports running computations on a variety of types of devices, including CPU and GPU. It translates the optimized computation graph into the intermediate representation according to the predefined operators with advanced optimization, and then the graph is compiled to executable code for the target device. We implemented the mini-batch SGD algorithms with sparse tensor in TensorFlow for LR training using both graph execution (latest 1.15 version) and eager execution (latest 2.1 version). TensorFlow's eager execution is an imperative programming environment that evaluates operations immediately, without building graph.

cuWide-stage. To clearly show the superiority of flow-based strategy cuWide (cuWide-flow), we also provide a C++ implementation (cuWide-stage) with CUDA 10.1 based on the stage-based strategy in Fig. 1a. It calculates gradients in the forward stage and uses gradients to update the model in the backward stage. Gradients are stored in global memory after the forward stage, and are reloaded for the backward stage.

DIMMWitted [11] is one of the state-of-the-art multi-core ML tools on NUMA machines. Ce et.al studied the tradeoff in access methods, model replication, and data replication.

LibFM [16] is a widely used library for factorization machines that features stochastic gradient descent and alternating least squares optimization.

Training Details. In our experiments, LR is optimized by mini-batch SGD with different batch sizes $[512, 1024]$. Much larger batch size evaluation is discussed in Section 6.3. The learning rate of LR is 0.5, which is determined by the grid search among the hyper-parameter space $[0.005, 0.05, \ldots, 500]$ for the best convergence. We split the data set into the train set (90 percent) and the validation set (10 percent). All the systems start training from the same point (0, 0.69) and we just exclude the beginning of the convergence curves and set the $\log$ x-scale for better demonstration [29]. All the systems are convergent with the same early stop [30] criteria (10 epochs) to avoid overfit models, i.e., the training is terminated when the current iteration validation loss is larger than the average loss of the latest 10 epochs. The evaluation time on validation set is not included.

Except TensorFlow, all the systems including cuWide, cuWide-stage, TensorFlow Eager and PyTorch load the train set into the global memory of GPU before the training procedure. TensorFlow, using the graph execution [12], transfers the mini-batch data from devices to the host in each iteration. Although it supports to store the data into constant tensor, which is stored inline in the graph data structure and may be replicated a few times, this method uses more memory and exceeds the limit of 2 GB for serializing individual tensors in TensorFlow.[1]

1. Another choice is to initialize a variable with the data set and split it into mini batches. However, the slice operation leads to a dramatically performance drop.

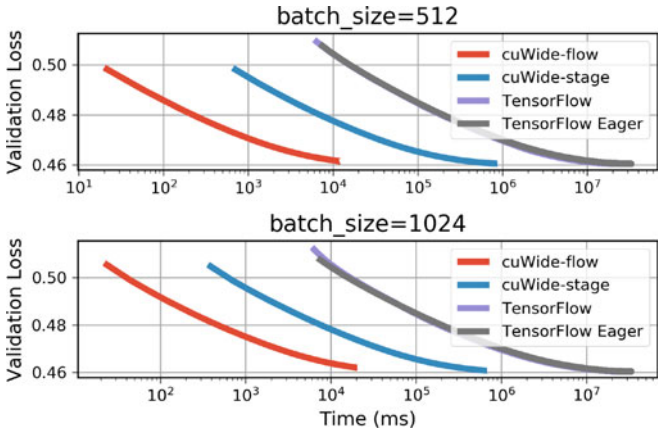


Fig. 12. Convergence performance comparison on criteo-s dataset for LR with different batch size.

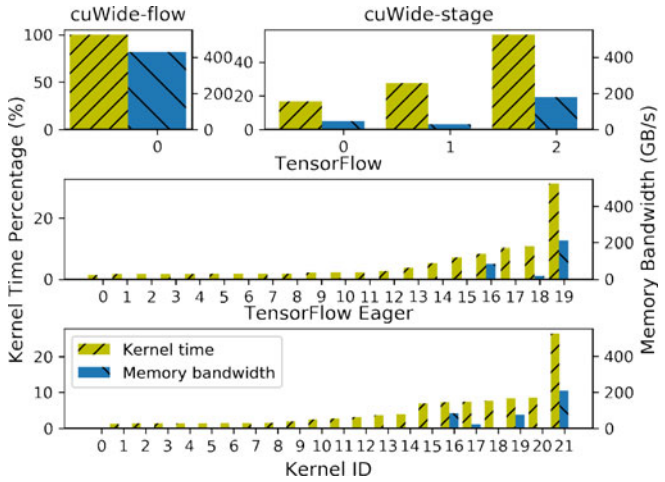


Fig. 13. The percentage of execution time and the utilization of memory bandwidth for GPU kernels for LR on criteo-s with batch size = 1024.

6.2 GPU Performance Comparisons

Fig. 12 shows the end-to-end convergence performance of LR among the state-of-the-art GPU-based approaches on the criteo-s data set with batch size of 1,024. Although cuWide only reshuffles the accessing order of different chunks to randomly form mini-batch at each epoch, it exhibited ignorable impact on final convergent validation loss values (e.g., no more than 0.003 difference with those of other systems). We also see that the convergence speed of cuWide-flow is the fastest for all cases, more than one or two orders of magnitude speedup compared to cuWide-stage and Tensorflow, respectively.

The figure shows that cuWide-stage converges at least 10× faster than TensorFlow. To figure out the reason of the performance gap among these staged-based systems, we show the average epoch time of these systems in Table 4. For TensorFlow, we find that the GPU time only takes up 10 percent of total time. There exists large amount of GPU idle time during the whole procedure. The idle time comes from the extra GPU kernel function calls (i.e., the framework overhead). Specifically, TensorFlow translates the optimized computation graph into the intermediate representation according to predefined basic operators (e.g., Sum, Add, Mul, Neg) and then generates executable code for GPU, thus bringing a large amount of GPU kernel function calls. For TensorFlow Eager, the problem of idle GPU is much more serious. The reason may comes from the static design of TensorFlow which enables more graph operators scheduling optimization than TensorFlow Eager.

For a fair comparison, we evaluate the GPU kernel time for these systems (i.e., excluding the framework overhead). As shown in Table 4, TensorFlow needs hundreds of milliseconds for memory copy between the device and host. While TensorFlow Eager can utilize the GPU computation capacity better because of GPU resident input data. We find that cuWide-flow still at least one order of magnitude faster than other stage-based baselines because of exploiting access locality.

TABLE 4
Average Epoch Time (ms) for LR on criteo-s With Batch Size = 1024

criteo-s	Total time	GPU time	GPU kernel time
cuWide-flow	23	22	22
cuWide-stage	365	311 (×14)	311 (×14)
TensorFlow	7,914	805 (×37)	493 (×22)
TensorFlow Eager	9,107	614 (×28)	597 (×27)

To figure out the speed-up between cuWide and our baselines, we further show the execution time and memory bandwidth of specific GPU kernels of these systems in Fig. 13. Although TensorFlow involves many small model-free GPU kernel calls (such as computation and data management, device query [31], [32]) for more general machine learning applications, the majority of cost is still come from a few large model-related kernels (e.g., multiplication, compute gradient and `ApplyGradient Descent`). So only removing many separate small kernels used in the framework cannot achieve high performance improvement. cuWide-stage further demonstrates this by manually removing the unnecessary operations in TensorFlow, with only three GPU kernels including multiplication, gradient computation, and apply gradients.

With the proposed caching mechanism, cuWide-flow fuses `Aggregate-Loss-Apply` functions into a single GPU kernel. Note that the execution time depends on both the memory bandwidth and the total amount of global memory access. cuWide significantly outperforms other baselines through removing a large amount of (random or extra) global memory accesses with locality-aware optimizations. As shown in the Fig. 13, the importance-based model caching enables cuWide-flow to achieve more than 10× larger memory bandwidth than the baselines that adopt `SparseTensorDenseMatMul` operator for the forward and gradient computations (e.g., kernels 0, 1 in cuWide-stage and kernels 16, 18 in TensorFlow). Also, the cross-stage accumulation caching allows cuWide-flow to avoid generating intermediate gradient data in global memory and to update the model directly, which removes the ApplyGradient Descent operation of baselines (e.g., kernel 2 in cuWide-stage and kernel 19 in TensorFlow).

6.3 Impact of Batch Size

We further analyze the batch size impact on the convergence performance in this subsection. Fig. 12 shows that large batch size can reduce the gap between cuWide and stage-based solutions. Then we conduct an experiment on

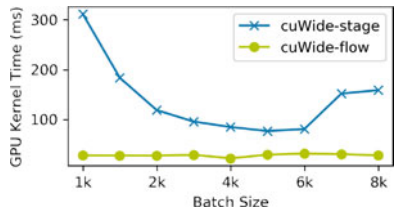


Fig. 14. The GPU kernel time per epoch for LR on criteo-s.

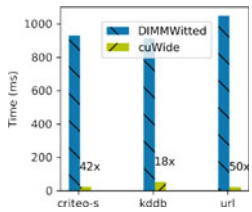


Fig. 15. Average epoch time comparison on LR.

larger batch sizes (up to 16k) and make a comparison for cuWide (cuWide-flow) and cuWide-stage.

As shown in Fig. 14, cuWide always outperforms cuWide-stage. When increasing the batch size, the per-epoch GPU kernel time of cuWide-stage decreases due to the higher utilization ratio of GPU resources in the beginning. It has also been observed in other related works [17], [33] that, as the matrix dimensions are modified, the performance may not strictly monotonic. So the GPU kernel time of cuWide-stage has an increase in the figure. By contrast, per-epoch GPU kernel time of cuWide tends to be stable due to fixed chunk computation under various batch size.

The experimental results also show that large batch size cannot infinitely speed up for stage-based strategy. By contrast, we find that cuWide achieves stable performance as the batch size grows, because it performs at the fine granularity of chunks. The experiment shows that the speed up of cuWide still exists for larger batch sizes due to high shared memory utilization.

In practice, wide models are often memory bounded. Therefore, we prefer to update the model more frequently within the limited data throughput, thus selecting smaller batch sizes. The state-of-the-art CPU-based systems which will be compared in the next subsection apply SGD (batch size = 1) for better speed-up [11]. And as suggested in [34], the batch size is typically chosen between 1 and a few hundreds. Smaller batch sizes lead to faster convergence, but stage-based strategy cannot fully utilize the hardware capacity with these batch sizes.

6.4 Performance Comparison With CPU-Based Approaches

We compare cuWide with the state-of-the-art CPU-based solution DIMMWitted, which uses data replication and model replication for efficiency. The experiments are conducted on criteo-s, kddb and url data sets, as shown in Table 2. We keep the hardware platform and hyperparameter settings the same with cuWide in Section 6.2. DIMMWitted only supports SGD on LR models, rather than mini-batch SGD. Fig. 15 shows the comparison results between the two CPU-based systems and cuWide (batch size=1024). It is clear that cuWide achieves $18\sim 50\times$ speedup of the average epoch time on different data sets.

The convergence performance comparison also demonstrates that highly optimized CPU program can outperform state-of-the-art GPU-based solutions (e.g., TensorFlow). The reason is that wide model training is memory bounded and existing systems only focus on the computation capacity of GPU ignoring the high memory bandwidth of GPU (340 GB/s GPU and 76.8 GB/s CPU, as shown in Table 3). We propose the flow-based strategy to reduce the global memory access bottleneck by involving shared memory of GPU, thus improving the usage of memory bandwidth of GPU. Therefore, cuWide can outperform the state-of-the-art CPU-based solutions in our experiments.

6.5 The Effectiveness of Spatial-Temporal Caching

cuWide outperforms other systems by a significant margin due to the flow-based strategy, which harnesses the spatial-temporal locality of wide models to leverage the memory hierarchy of GPU. In order to demonstrate to what extent each caching mechanism contributes, we analyze the speed up of the optimization in different stages of cuWide by kernel profiling metrics. Fig. 16 shows a breakdown comparison of LR on criteo-s, kddb and url datasets. It is clear to see that the importance-based model caching and cross-stage accumulation caching both improves the performance. Taking kddb as an example, with the accumulation caching, cuWide achieves 4.9× speedup than Tensorflow and 1.9× than stage-based training baseline. When further applying model caching, cuWide further bring 7.4× speedup the baselines.

We find that the different speed-ups may have relevance with feature density or dataset size. Typically, larger dataset size and lower density (i.e., kddb) leads to a more random access and thus significant speed-up over TensorFlow. Moreover, the performance of cuWide is also affected by the feature distribution. As shown in Fig. 3, criteo-s has a more skew distribution, which leads to better locality and thus a larger speed-up over cuWide-stage than the other two datasets.

6.6 The Performance on Large Dataset

To efficiently handle large data set that cannot fit into GPU's global memory, we scale up cuWide with multi-stream technique. We conduct the experiments with FM model using different factor latent dimensions (k=3 and k=10) on three data sets (i.e., criteo-s, criteo, and kdd12) which are larger than the 11 GB GPU global memory. We implement cuWide with stream pipeline enabled and disabled, and compare the performance with LibFM and TensorFlow. Table 5 lists the results. The results show that cuWide could scale to large datasets well, and achieve $1.7\sim 5.2\times$ speedup with stream pipeline than LibFM [16]. Our evaluations on real world data sets have demonstrated that cuWide outperforms CPU-based solutions by applying the stream pipeline even though that the whole dataset cannot fit into the GPU global memory.

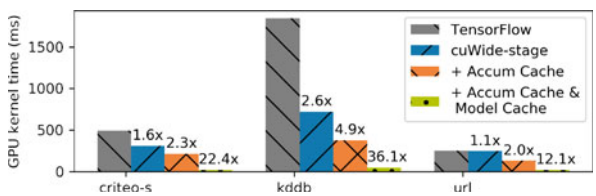


Fig. 16. System speed up from the optimization in cuWide.

TABLE 5
Average Epoch Time (ms) Comparison for FM

Dataset	cuWide w/o pipeline	cuWide w/ pipeline	LibFM (CPU)	TensorFlow (GPU)
		k=3		
criteo-s	2,590	1,450	7,452	10,955
criteo	30,860	17,220	73,130	97,825
. kdd12	39,060	19,550	73,654	2,383,593
		k=10		
criteo-s	4,320	2,820	12,470	12,431
criteo	51,520	31,520	97,574	111,012
kdd12	84,090	58,480	102,081	712,362

TABLE 6
Average Epoch Time (ms) for LR With Batch Size = 1024 on V100

criteo-s	Total time	GPU time	GPU kernel time
cuWide-flow	12	11	11
cuWide-stage	418	330 (×30)	330 (×30)
TensorFlow	2,894	710 (×65)	376 (×34)

Due to the complexity of FM (e.g., multiple accumulation in (5) and (6)), the TensorFlow is still bounded by the large amount of random global memory access of GPU, instead of host-GPU data transfer. So cuWide is able to outperform TensorFlow by significantly removing the random global memory access with the proposed caching mechanism. For models with less arithmetic intensity, such as LR, host-GPU data transfer might be performance bottleneck given the limited PCIe bandwidth. However, cuWide still provides high performance potential for the emerging NVLink GPU systems that provide much higher bandwidth than PCIe.

For more scale training, cuWide is easily to be generalized to such distributed environments by applying data parallel with a model synchronization mechanism (e.g., host memory, parameter server [35], [36], [37] or All-Reduce [38]) due to the scalability of the mini-batch SGD algorithm. We can also embed cuWide into TensorFlow as an special operator and utilize the parameter server in TensorFlow for communication. In such cases, the synchronization communication cost becomes the system bottleneck and many distributed training algorithms [39], [40], [41], [42] has been studied. We leave these as our future works.

6.7 Performance on Advanced GPU

We further evaluate the proposed ideas using more advanced GPUs such as NVIDIA V100 (80 SM, 32 GB, up to 14 TFLOPS and 900 GB/s) that has 10× bigger shared memory and 3× greater effective device memory bandwidth than 1080ti. In such a case, per-kernel execution overhead can become a bigger performance limiter.

Table 6 shows the performance of cuWide-flow is much higher than other baselines on V100. By comparing Tables 6 and 4, we see that speed-ups of cuWide-flow on V100 (see Table 6) becomes significantly larger than that on GTX 1080ti. For example, increasing from 22× on GTX 1080ti to 34× on V100 compared to TensorFlow. This result shows our idea scales with GPU design and resources.

7 Related Work

Machine learning and data mining problems have been gaining popularity because of their successes in many real-world services, e.g., recommendation [1], advertising [43], etc. Wide models like LR [44], LSVM [45], FM etc., learn from a wide set of features. Generalized linear models have been studied by [2] and they train the model over join operations that are easy to implement over existing RDBMSs. [46], [47] further propose the shared computation of aggregates over normalized databases. Inspired by these approaches, we propose the feature aggregation and *ALA* programming interfaces to formulate wide model training. However, there are distinct memory accesses and computation cost for emerging parallel hardware such as GPUs, which lead to new challenges for this topic.

GPU has performed its high efficiency in accelerating general computation tasks, and plenty of GPU-based implementation of machine learning models are proposed to achieve high performance of training models on GPU [48], [49]. There are some open-sourced machine learning systems on GPU, such as H2O.ai [50], cuML of RAPIDS [51]. However, these libraries only support dense wide model training. Snap ML [29] offers a high-performance implementation of SDCA [52] for the distributed GPU cluster environment. The released version is only for the ppc64le architecture.

ThunderSVM [53], [54] provides a high performance GPU-based SVM library. But it concentrates on solving the SVMs with SMO algorithm, which is naturally different from the mini-bath SGD algorithm and not available for other wide models (e.g., LR, LSR, FM). BIDMach [55] adopts a zero memory allocation technique to overcome the allocation barrier during allocate and recycle matrix (or graph) objects in expressions. However, it implements each model with the model-specific design so that optimization cannot be reused among different models. We have made some experiments with BIDMach, however the hardware efficiency of BIDMach is not competitive. Moreover, the loss of BIDMach has to be collected automatically on a transparent validation set sampled from the training procedure, rather than the common test loss or train loss. Due to above reasons, we exclude it from our baselines.

Popular dataflow-based machine learning systems like TensorFlow [12] are mainly designed for neural network computing and lack of optimization for wide models due to the often sparse distribution. PyTorch [13] adopts cuSPARSE to support sparse data computation, which is far from the roofline limit on typical sparse machine learning data with skew distribution [55]. cuWide introduces the bigraph abstraction and represents wide models in a unified framework with system-level optimization. The flow-based strategy is naturally to break a mini-batch into sub-batches and reduce the memory footprint such that the intermediate data of an entire sub-batch fits in on-chip memory within a persistent fused kernel. This insight has been studied in other deep learning models, such as CNN [56] and RNN [57] and even sparse RNN [58]. cuWide targets on wide models which are facing challenges on the spatial-temporal locality and quiet different from deep models.

Several approaches have analyzed thread conflicts on GPUs and achieved a good performance. The work [59] has measured the influence of atomic operations conflicts in both global memory and shared memory of GPU and concluded a linear penalty in conflicts. Garaph [60] further proposed a customized replication method to resolve threads conflicts in processing large-scale graph data. cuWide employs a sample-specific replication policy integrated into the flow-based strategy to optimize the forward stage of wide models.

8 Conclusion

Wide models have been deployed in many real-world applications. One important problem of the models is to design efficient training solutions. In this paper, we proposed a new GPU-training framework for large-scale wide models, called cuWide. To accelerate the training by fully exploiting the memory hierarchy in GPU, cuWide applies a novel flow-based training strategy. We first introduced the ALA programming model for flexibly developing customized wide models while benefiting from our locality-aware optimization techniques in a unified manner. We then proposed caching optimizations for executing flow-based training, exploiting the spatial and temporal locality of data to optimize the memory access of wide model. Through the extensive experiments, it clearly demonstrates that cuWide outperforms other state-of-the-art systems for training large-scale wide models.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (No.2018YFB1004403), the National Natural Science Foundation of China under Grant (No. 61832001, 61972004, 61702015, U1936104, 61702016), the Fundamental Research Funds for the Central Universities 2020RC25, Beijing Academy of Artificial Intelligence (BAAI), PKU-Baidu Fund 2019BD006, and PKU-Tencent Joint research Lab.

References

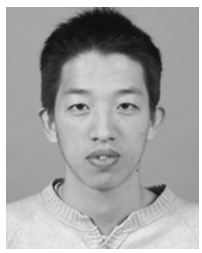
[1] H.-T. Cheng *et al.*, "Wide & deep learning for recommender systems," in *Proc. 1st Workshop Deep Learn. Recommender Syst.*, 2016, pp. 7–10.
[2] A. Kumar, J. Naughton, and J. M. Patel, "Learning generalized linear models over normalized data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1969–1984.
[3] Z. Zhang, J. Jiang, W. Wu, C. Zhang, L. Yu, and B. Cui, "MLlib*: Fast training of GLMs using spark MLlib," in *Proc. IEEE 35th Int. Conf. Data Eng.*, 2019, pp. 1778–1789.
[4] L. Yu, L. Wang, Y. Shao, L. Guo, and B. Cui, "GLM+: An efficient system for generalized linear models," in *Proc. IEEE Int. Conf. Big Data Smart Comput.*, 2018, pp. 293–300.
[5] H. B. McMahan *et al.*, "Ad click prediction: A view from the trenches," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2013, pp. 1222–1230.
[6] W. Liu *et al.*, "Field-aware probabilistic embedding neural network for CTR prediction," in *Proc. 12th ACM Conf. Recommender Syst.*, 2018, pp. 412–416.
[7] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "DeepFM: A factorization-machine based neural network for CTR prediction," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, 2017, pp. 1725–1731.
[8] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich, "Web-scale Bayesian click-through rate prediction for sponsored search advertising in Microsoft's bing search engine," in *Proc. 27th Int. Conf. Mach. Learn.*, 2010, pp. 13–20.
[9] Z. Gharibshah, X. Zhu, A. Hainline, and M. Conway, "Deep learning for user interest and response prediction in online display advertising," *Data Sci. Eng.*, vol. 5, pp. 12–26, 2020.
[10] I. Naseem, R. Togneri, and M. Bennamoun, "Linear regression for face recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 11, pp. 2106–2112, Nov. 2010.
[11] C. Zhang and C. Ré, "DimmWitted: A study of main-memory statistical analytics," *Proc. VLDB Endowment*, vol. 7, pp. 1283–1294, 2014.
[12] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.
[13] A. Paszke *et al.*, "Automatic differentiation in PyTorch," 2017.
[14] E. P. Xing, Q. Ho, P. Xie, and D. Wei, "Strategies and principles of distributed machine learning on big data," *Eng.*, vol. 2, pp. 179–195, 2016.
[15] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritized iterative computations," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, Art. no. 13.
[16] S. Rendle, "Factorization machines with libFM," *ACM Trans. Intell. Syst. Technol.*, vol. 3, 2012, Art. no. 57.
[17] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, pp. 65:1–65:43, 2019.
[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[19] Nvidia cuda programming guide, [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide
[20] M. M. Alam, K. S. Perumalla, and P. Sanders, "Novel parallel algorithms for fast multi-GPU-based generation of massive scale-free networks," *Data Sci. Eng.*, vol. 4, pp. 61–75, 2019.
[21] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, pp. 716–727, 2012.
[22] J. K. Kim *et al.*, "STRADS: A distributed framework for scheduled model parallel machine learning," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 5.
[23] cuwide, [Online]. Available: https://github.com/DMALab/cuWide
[24] M. Zukowski, M. Van de Wiel, and P. A. Boncz, "Vectorwise: A vectorized analytical DBMS," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 1349–1350.
[25] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten, "MonetDB: Two decades of research in column-oriented database," *IEEE Data Eng. Bull.*, 2012.
[26] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 785–794.
[27] Nvidia Crop., "Profiler :: Cuda toolkit documentation," 2019. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html
[28] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, 2011, Art. no. 27.
[29] C. Dünner *et al.*, "Snap ML: A hierarchical framework for machine learning," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 250–260.
[30] G. Raskutti, M. J. Wainwright, and B. Yu, "Early stopping and non-parametric regression: An optimal data-dependent stopping rule," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 335–366, 2014.
[31] Y. Gao *et al.*, "Estimating GPU memory consumption of deep learning models," Microsoft, Redmond, WA, Tech. Rep. MSR-TR-2020–20, May 2020.
[32] P. Yu and M. Chowdhury, "Salus: Fine-grained GPU sharing primitives for deep learning applications," *CoRR*, 2019.
[33] Y. Oyama, A. Nomura, I. Sato, H. Nishimura, Y. Tamatsu, and S. Matsuoka, "Predicting statistics of asynchronous SGD parameters for a large-scale distributed deep learning system on GPU supercomputers," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 66–75.
[34] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*. Berlin, Germany: Springer, 2012, pp. 437–478.
[35] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 583–598.
[36] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 463–478.
[37] Z. Zhang, B. Cui, Y. Shao, L. Yu, J. Jiang, and X. Miao, "PS2: Parameter server on spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2019, pp. 376–388.
[38] S. Kim *et al.*, "Parallax: Sparsity-aware data parallel training of deep neural networks," in *Proc. 14th EuroSys Conf.*, 2019, pp. 43:1–43:15.

[39] S. Ghadimi, G. Lan, and H. Zhang, "Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization," *Math. Program.*, vol. 155, no. 1/2, pp. 267–305, 2016.
[40] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 2737–2745.
[41] J. Jiang, F. Fu, T. Yang, Y. Shao, and B. Cui, "SKCompress: Compressing sparse and nonuniform gradient in distributed machine learning," *VLDB J.*, vol. 29, no. 5, pp. 945–972, 2020.
[42] Z. Zhang, W. Wu, J. Jiang, L. Yu, B. Cui, and C. Zhang, "ColumnSGD: A column-oriented framework for distributed stochastic gradient descent," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1513–1524.
[43] L. Yan, W.-J. Li, G.-R. Xue, and D. Han, "Coupled group lasso for web-scale CTR prediction in display advertising," in *Proc. 31st Int. Conf. Mach. Learn.*, 2014, pp. 802–810.
[44] J. Liang, Y. Song, D. Li, Z. Wang, and C. Dang, "An accelerator for the logistic regression algorithm based on sampling on-demand," *Sci. China Inf. Sci.*, vol. 63, no. 6, 2020, Art. no. 169102.
[45] G. Guo, H. Wang, Y. Yan, L. Zhang, and B. Li, "Large margin deep embedding for aesthetic image classification," *Sci. China Inf. Sci.*, vol. 63, no. 1, 2020, Art. no. 119101.
[46] M. Abo Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich, "In-database learning with sparse tensors," in *Proc. 37th ACM SIGMOD-SIGACT-SIGAI Symp. Princ. Database Syst.*, 2018, pp. 325–340.
[47] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich, "AC/DC: In-database learning thunderstruck," in *Proc. 2nd Workshop Data Manage. End-To-End Mach. Learn.*, 2018, Art. no. 8.
[48] K. Li, J. Chen, W. Chen, and J. Zhu, "SaberLDA: Sparsity-aware learning of topic models on GPUs," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 497–509.
[49] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "CuMF_SGD: Parallelized stochastic gradient descent for matrix factorization on GPUs," in *Proc. 26th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 79–92.
[50] A. Candel, V. Parmar, E. LeDell, and A. Arora, "Deep learning with H2O," H2O. ai Inc, 2016.
[51] cuml: Rapids machine learning library, [Online]. Available: https://rapids.ai
[52] S. Shalev-Shwartz and T. Zhang, "Stochastic dual coordinate ascent methods for regularized loss minimization," *J. Mach. Learn. Res.*, vol. 14, no. Feb, pp. 567–599, 2013.
[53] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, "ThunderSVM: A fast SVM library on GPUs and CPUs," *J. Mach. Learn. Res.*, vol. 19, pp. 21:1–21:5, 2018.
[54] Z. Wen, J. Shi, B. He, J. Chen, and Y. Chen, "Efficient multi-class probabilistic SVMs on GPUs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 9, pp. 1693–1706, Sep. 2019.
[55] J. Canny and H. Zhao, "BIDMach: Large-scale learning with zero memory allocation," in *Proc. BigLearn Workshop NeurIPS*, 2013, Art. no. 117.
[56] S. Lym, A. Behroozi, W. Wen, G. Li, Y. Kwon, and M. Erez, "Mini-batch serialization: CNN training with inter-layer data reuse," in *Proc. Conf. Mach. Learn. Syst.*, 2019.
[57] G. Diamos *et al.*, "Persistent RNNs: Stashing recurrent weights on-chip," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 2024–2033.
[58] F. Zhu, J. Pool, M. Andersch, J. Appleyard, and F. Xie, "Sparse persistent RNNs: Squeezing large recurrent networks on-chip," in *Proc. Int. Conf. Learn. Representations*, 2018.
[59] J. Gomez-Luna, J. M. Gonzalez-Linares, J. I. B. Benitez, and N. G. Mata, "Performance modeling of atomic additions on GPU scratchpad memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 11, pp. 2273–2282, Nov. 2013.
[60] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 195–207.

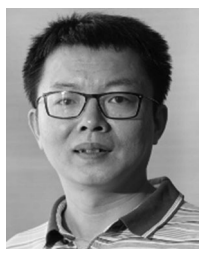
Xupeng Miao received the BSc degree in computer science and technology from Northeastern University, China, in 2017. Now he is currently working toward the third year PhD degree in the School of EECS, Peking University, China. His research interests include gpu acceleration, distributed deep learning system and graph analysis.

Lingxiao Ma is currently working toward the fifth year PhD degree in the School of EECS, Peking University, China. His research works are focused on building efficient parallel systems for large-scale data analytics scenarios, e.g., machine learning, graph processing, through leveraging modern hardware like GPU.

Zhi Yang is an associate professor with the School of EECS, Peking University, China. Topics he has been working on include efficient parallel systems for large-scale data analytics, design and optimization of distributed systems, and social network application.



Yingxia Shao is an associate researcher with the School of Computer Science, Beijing University of Posts and Telecommunications, China. His research interests include large-scale graph analysis, parallel computing framework, and knowledge graph analysis.



Bin Cui (Senior Member, IEEE) is a professor with the School of EECS and director of Institute of Network Computing and Information Systems, Peking University, China. His research interests include database systems, and data mining. He has published more than 100 research papers, and is the winner of Microsoft Young Professorship award (MSRA 2008), and CCF Young Scientist award (2009).

Lele Yu received the PhD degree from Peking University, China, in 2018. He is a researcher in Tencent Inc, China. His interests include distributed ML, big data processing and parallel computing systems, and Topic modeling.

Jiawei Jiang received the PhD degree from Peking University, China, in 2018. He is a postdoc in the Department of Computer Science, ETH Zurich, Switzerland. His interests include distributed ML, communication optimization and GBDT algorithms.