2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00251

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

(Extended Abstract)

Xupeng Miao[†§] Lingxiao Ma[†] Zhi Yang[†] Yingxia Shao[‡] Bin Cui[†$] Lele Yu[§] Jiawei Jiang[♯]

[†] Department of Computer Science & Key Lab of High Confidence Software Technologies (MOE), Peking University

[$]Institute of Computational Social Science, Peking University (Qingdao)

[‡]School of Computer Science, BUPT, [♯]ETH Zurich [§]Tencent Inc.

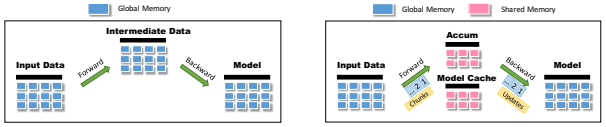
[†]{xupeng.miao, xysmlx, yangzhi, bin.cui}@pku.edu.cn

[‡]shaoyx@bupt.edu.cn [§]leleyu@tencent.com [♯]jiawei.jiang@inf.ethz.ch

***Abstract*—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. Comprehensive experiments show that cuWide can be up to more than 20× faster than the state-of-the-art GPU solutions and multi-core CPU solutions.**

I. Introduction

Wide model [1] has been widely used in many practical big data applications, which can be expressed as a linear combination of sample features followed by an activation function. Given such increasing demands for high dimensional workloads, more efficient model training on GPU with thousands of computing units and high memory bandwidth becomes promising and attractive.



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

Fig. 1. The two strategies for training.

However, the GPU training for the wide models is far from optimal due to the sparsity and irregularity in wide models. Existing popular GPU-based systems, e.g., TensorFlow (TF), adopt stage-based strategy during training, as illustrated in Figure 1(a). It suffers from large amount of random memory accesses and redundant read/write of intermediate values, which makes the wide model training on GPU even slower than the ones using CPU.

In this paper [2], we propose a novel flow-based training schema in Figure 1(b) and provide a vertex-centric programming model Aggregate-Loss-Apply (*ALA*) to express the execution of various wide models over a bi-graph. Furthermore, we exploit the fine-grained data access pattern of wide models and utilize the limited shared memory (up to 96 KB) to cache the important model parameters and cross-stage accumulations, which alleviates the global memory access and achieves a high-performance training.

II. Problem Definition

Wide model. GLMs (e.g., LR, LSVM, LSR and FTRL) and FM are most representative and popular wide models for industrial applications because they are simple, scalable and interpretable. Formally defined as: $\hat{y}(\boldsymbol{x}) = \boldsymbol{w}\boldsymbol{x}$, 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: $L(\boldsymbol{w}) = \sum_{i=1}^{n} l(y_i, \boldsymbol{w}\boldsymbol{x_i})$. The goal of wide model is to minimize the loss function and find $\boldsymbol{w} = argmin_{\boldsymbol{w}} L(\boldsymbol{w})$.

Stage-based strategy problem. The stage-based strategy calculates prediction errors in the forward stage and uses gradients to update the model in the backward stage. The computation is often expressed by operations among tensors on GPU global memory. Such implementation exhibits several attractive features in programming and flexibility, but it has potential efficiency limitation due to inefficient memory accesses. 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 (up to 96 KB per streaming multiprocessor), while the number of features can easily exceed millions (more than 1 MB memory).

III. Methods

Spatial Locality. In many large-scale machine learning problems [3], it is very common that the feature frequency almost follows a power law distribution. More concretely, most features only show up in few samples, and the top features are nonzero for almost all samples. 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 `Accum`s, which can be aggregated from input features and model parameters. Taking the LR model as an example:

$$\hat{y} = \frac{1}{1+e^{-\boldsymbol{w}\boldsymbol{x}}} = \texttt{eval}(\boldsymbol{w}\boldsymbol{x}) \quad (1)$$

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00251

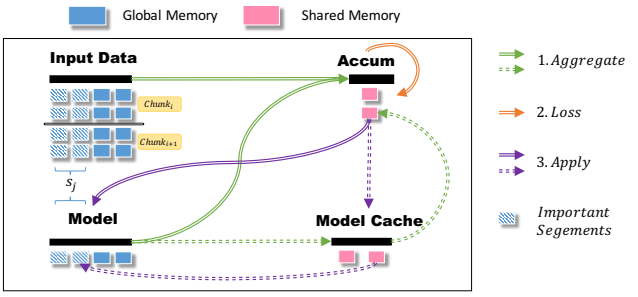


Fig. 2. Illustration of cuWide execution. Note that the solid lines refer to the Accum caching and the dotted lines refer to the model caching.

$$\boldsymbol{g}_w = \frac{-y\boldsymbol{x}}{e^{y\boldsymbol{w}\boldsymbol{x}}+1} = \texttt{grad}(\boldsymbol{w}\boldsymbol{x}, \boldsymbol{x}, y) \tag{2}$$

where $\boldsymbol{w}$ are the model parameters, $\boldsymbol{x}$, y are the inputs and $\boldsymbol{w}\boldsymbol{x}$ are the Accums. Such temporal locality of Accum data during both stages implies that we can reduce the global memory access for writting/reading intermediate data by caching aggregated feature in shared memory.

Flow-based strategy. As illustrated in Figure 1(b), we perform a two-dimensional partition over the mini-batch, into chunks over the sample dimension and segments over the feature (i.e., model) dimension, respectively. In Figure 2, 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, 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.

Programming Abstraction. The general GPU machine learning frameworks completely loses the locality access pattern in wide model with the primitive of tensor abstractions. cuWide provides a vertex-centric programming model Aggregate-Loss-Apply (*ALA*), allowing to express the execution of various wide models over a bi-graph. In Figure 3, for each iteration of mini-batch SGD, sample vertices represent samples, parameter vertices represent different features and edges represent the relations between samples and features. Figure 4 shows the LR implementation using *ALA* programming model.

System Implementation. To effectively utilize the GPU, we further propose several GPU-oriented optimizations, including feature-oriented data layout to enhance the data locality, shared memory conflicts resolution and multi-stream scheduling to overlap data transferring and kernel computing. The source code has been made public at [4].

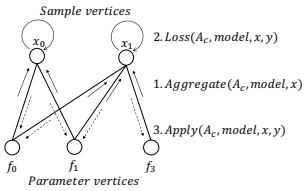


Fig. 3. The *ALA* processing model

```
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.

IV. Experiments

End-to-end comparison. Figure 5 shows the results of comparing cuWide with stage-based solutions, including TF and cuWide-stage (our C++ implementation). We evaluate the LR performance on criteo-s, kddb and url datasets. cuWide outperforms the others by a significant margin (up to 36.1× for TF on kddb) due to the flow-based strategy, which harnesses the spatial-temporal locality of wide models to leverage the GPU memory hierarchy.

Breakdown comparison. We also provide breakdown comparison to illustrate each caching mechanism's contribution. It reveals that the model caching and Accum caching both improves the performance. For kddb, with the Accum caching, cuWide achieves 4.9× speedup than TF and 1.9× than stage-based training baseline. When further applying model caching, cuWide further bring 7.4× speedup. More detailed experimental results are in [2].

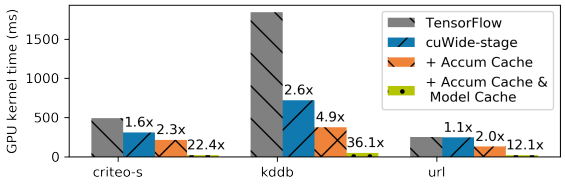


Fig. 5. Performance comparison with TensorFlow and cuwide-stage using Nvida GTX 1080ti for LR with batch size = 1024

Acknowledgement

This work is 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, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir *et al.*, "Wide & deep learning for recommender systems," in *Workshop on DLRS*, 2016.
[2] X. Miao, L. Ma, Z. Yang, Y. Shao, B. Cui, L. Yu, and J. Jiang, "Cuwide: Towards efficient flow-based training for sparse wide models on gpus," *TKDE*, pp. 1–1, 2020.
[3] 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," *PVLDB*, pp. 716–727, 2012.
[4] "cuwide," https://github.com/DMALab/cuWide.