FeatureBox: Feature Engineering on GPUs for Massive-Scale Ads Systems

Weijie Zhao[1], Xuewu Jiao[2], Xinsheng Luo[2], Jingxue Li[2], Belhal Karimi[1], Ping Li[1]
[1]Cognitive Computing Lab, Baidu Research
[2]Baidu Search Ads (Phoenix Nest), Baidu Inc.
10900 NE 8th St. Bellevue, Washington 98004, USA
No. 10 Xibeiwang East Road, Beijing 10193, China
{weijiezhao, jiaoxuewu, luoxinsheng, lijingxue01, belhalkarimi, liping11}@baidu.com

***Abstract*—Deep learning has been widely deployed for online ads systems to predict Click-Through Rate (CTR). Machine learning researchers and practitioners frequently retrain CTR models to test their new extracted features. However, the CTR model training often relies on a large number of raw input data logs. Hence, the feature extraction can take a significant proportion of the training time for an industrial-level CTR model. In this paper, we propose FeatureBox, a novel end-to-end training framework that pipelines the feature extraction and the training on GPU servers to save the intermediate I/O of the feature extraction. We rewrite computation-intensive feature extraction operators as GPU operators and leave the memory-intensive operator on CPUs. We introduce a layer-wise operator scheduling algorithm to schedule these heterogeneous operators. We present a light-weight GPU memory management algorithm that supports dynamic GPU memory allocation with minimal overhead. We experimentally evaluate FeatureBox and compare it with the previous in-production feature extraction framework on two real-world ads applications. The results confirm the effectiveness of our proposed methods.**

***Index Terms*—CTR Prediction; GPU; Large-Scale Machine Learning Framework**

I. Introduction

Deep learning has been widely employed in many real-world applications, e.g., computer vision [1], [2], [3], data mining [4], and recommendation systems [5], [6], [7], [8]. In recent years, sponsored online advertising also adopts deep learning techniques to predict the Click-Through Rate (CTR) and improve recommender systems. Unlike common machine learning applications, the accuracy of the CTR prediction is critical to the revenue. In the context of a multi-billion-dollar online ads industry, even a 0.1% accuracy increase will result in a noticeable revenue gain. We identify two major paths to improve the model accuracy. The first area is to propose different and enhanced model architectures. Every improvement in this direction is considered a fundamental milestone in the deep learning community—and does not happen often in the CTR prediction industry. The other (more practical) is feature engineering, i.e. to propose and extract new features from the raw training data. The benefit of feature engineering is usually neglected in common deep learning applications because of the general belief that deep neural networks inherently extract the features through their hidden layers. However, recall that CTR prediction applications are accuracy-critical, hence, the gain from an improved feature engineering strategy remains attractive for in-production CTR prediction models. Therefore, in order to achieve a better prediction performance, CTR deep learning models in real-world ads applications tend to utilize larger models and more features extracted from the raw data logs.

Testing on the historical and online data is the rule-of-the-thumb way to determine whether a new feature is beneficial. Every new feature with positive accuracy improvement (e.g. 0.1%) is included into the CTR model. Machine learning researchers and practitioners keep this feature engineering trial-and-error on top of the current in-production CTR model. As a result, the in-production CTR model becomes larger and larger with more and more features. To support the trial-and-error research for new features, it requires us to efficiently train massive-scale models with massive-scale raw training data in a timely manner. Previous studies [9] propose hierarchical GPU parameter server that trains the out-of-memory model with GPU servers to accelerate the training with GPUs and SSDs. With a small number of GPU servers, e.g., 4, can obtain the same training efficiency as a CPU-only cluster with hundreds of nodes. The training framework focuses on the training stage and assumes the training data are well-prepared—the training data are accessed from a distributed file system.

However, preparing the training data is not trivial for industrial level CTR prediction models—with $\sim 10^{11}$ features. The feature extraction from raw data logs can take a significant proportion of the training time. In addition to the frequent re-training for new feature engineering trials, online ads systems have to digest a colossal amount of newly incoming data to keep the model up-to-date with the optimal performance. For the rapid training demands, optimizing the feature extraction stage becomes one of the most desirable goals of online ads systems. This latter point is the scope of our contribution.

Training workflow. The upper part of Figure 1 depicts a visual illustration of the feature extraction. Due to the large amount of raw data, the original feature extraction task is constructed as MapReduce [10] jobs that compute feature combinations, extract keywords with language models, etc. Those MapReduce jobs frequently read and write intermediate files with the distributed file system (i.e., HDFS). The intermediate I/O can be as large as 200 TB. Once the features are

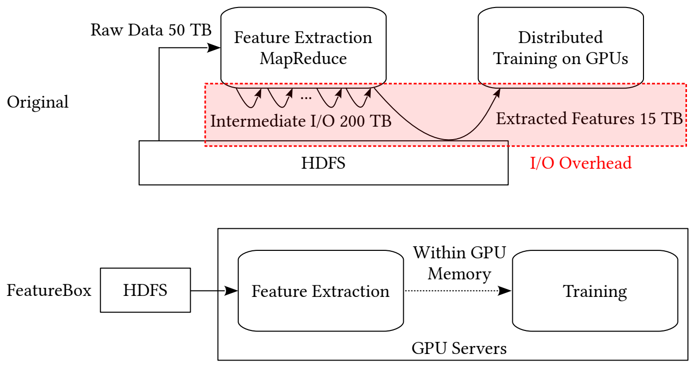


Fig. 1. A visual illustration for the original feature extraction and training workflow (upper); and our proposed FeatureBox (lower).

extracted, we also need to materialize them to the ∼15 TB extracted features to the HDFS so that the following distributed training framework can read them from the distributed file system. This training workflow incurs rapid communication with HDFS that generates heavy I/O overhead.

One straightforward question can be raised: *Can we perform the feature extraction within GPU servers to eliminate the communication overhead?* In the lower part of Figure 1, we depict an example for the proposed training framework that combines the feature extraction and the training computation within GPU servers. The intermediate I/O is eliminated by integrating the feature extraction and the training computation into a pipeline: for each batch of extracted features, we feed the batch to the model training without writing them as intermediate files into HDFS.

Challenges & Approaches. However, moving the feature extraction to GPU servers is non-trivial. Note that the number of GPU nodes is much fewer compared with the CPU-only cluster. We acknowledge two main challenges in embedding the feature extraction phase into GPU servers:

1) *Network I/O bandwidth*. The network I/O bandwidth of GPU servers is by orders of magnitude smaller than the bandwidth of CPU clusters because we have fewer nodes—the total number of network adapters is lower. We materialize frequently-used features as basic features so that we can reuse them without extra I/O and computations. In addition, we use column-store that reads only the required columns in the logs to reduce I/O.
2) *Computing Resources*. With a smaller number of nodes, the CPU computing capability on GPU servers is also orders of magnitudes less powerful than the CPU cluster. We have to move the CPU computations to GPU operations to bridge the computing power gap.
3) *Memory Usage*. The feature extraction process contains many memory-intensive operations, such as dictionary table lookup, sort, reduce, etc. It is desired to have an efficient memory management system to perform these memory-intensive operations on GPU servers with limited memory.

We summarize our **contributions** as follows:

- We propose FeatureBox, a novel end-to-end training framework that pipelines the feature extraction and the training on GPU servers.
- We present a layer-wise operator scheduling algorithm that arranges the operators to CPUs and GPU.
- We introduce a light-weight GPU memory management algorithm that supports dynamic GPU memory allocation with minimal overhead.
- We experimentally evaluate FeatureBox and compare it with the previous in-production feature extraction framework on two real-world ads applications. The results confirm the effectiveness of our proposed methods.

II. Preliminary

In this section, we present a brief introduction of CTR prediction models and the hierarchical GPU parameter server. Both concepts are the foundations of FeatureBox.

A. CTR Prediction Models

About a decade ago, CTR prediction strategies with large-scale logistic regression model on carefully engineered features are proposed in [11], [12]. With the rapid development of deep learning, deep neural networks (DNN) attract a lot of attention in the CTR research community: The DNN model, with wide embedding layers, obtains significant improvements over classical models. The model takes a sparse high-dimensional vector as input and converts those sparse features into dense vectors through sequential embedding layers. The output dense vector is considered a low-dimensional representation of the input and is then fed into the following layers in order to compute the CTR. Most proposed CTR models share the same embedding layer architecture and only focus on the following neural network layers, see for e.g., Deep Crossing [13], Product-based Neural Network (PNN) [14], Wide&Deep Learning [7], YouTube Recommendation CTR model [15], DeepFM [16], xDeepFM [17] and Deep Interest Network (DIN) [18]. They introduce special neural layers for specific applications that capture latent feature interactions.

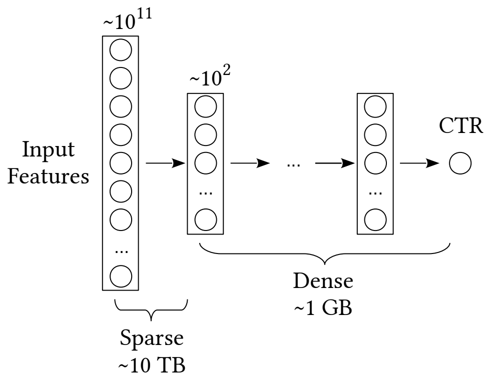


Fig. 2. An example for the CTR prediction network architecture.

We summarize those architectures in Figure 2. The input features are fed to the neural network as a sparse high-dimensional vector. The dimension of the vector can be as large as $\sim 10^{11}$. The input features for CTR models are usually from various resources with categorical values, e.g., query words, ad keywords, and user portrait. The categorical values are commonly represented as a one-hot or multi-hot encoding. Therefore, with categorical values with many sources, the number of dimensions is high ($\sim 10^{11}$) for industry CTR prediction models. Note that feature compression or hashing strategies that reduce the number of dimensions are not applicable to the CTR prediction model because those solutions inevitably trade off the prediction accuracy for better computational time [19]—recall that even a small accuracy loss leads to a noticeable online advertising revenue decrease, which is unacceptable. We embed the high-dimensional features through an embedding layer to obtain a low-dimensional ($\sim 10^2$) representation. The number of parameters in the embedding layer can be as large as 10 TB due to the high input dimension. After the low-dimensional embedding is achieved, we fed this dense vector to the following neural network components to compute the CTR.

B. Hierarchical GPU Parameter Server

Due to the extremely high dimension of the embedding layer, the model contains around $10TB$ parameters which do not fit on most computing servers. Conventionally, the huge model is trained on an MPI cluster. We partition the model parameters across multiple computing nodes (e.g., 150 nodes) in the MPI cluster. Every computing node is assigned a batch of training data streamed directly from the HDFS. For each node, it retrieves the required parameters from other nodes and computes the gradients for its current working mini-batch. The gradients are then updated to the nodes that maintain the corresponding parameters through MPI communications. Recently, hierarchical GPU parameter servers [9] are proposed to train the massive-scale model on a limited number of GPU servers. The key observation of the hierarchical GPU parameter server is that the number of referenced parameters in a mini-batch fits the GPU memory because the input vector is sparse. It maintains three levels of hierarchical parameter servers on GPU, CPU main memory, and SSD. The working parameters are stored in GPUs, the frequently used parameters are kept in CPU main memory, and other parameters are materialized as files on SSDs. The upper-level module acts as a high-speed cache of the lower-level module. With 4 GPU nodes, the hierarchical GPU parameter server is able to be 2X faster than 150 CPU-only nodes in an MPI cluster. Our proposed FeatureBox follows the design of the training framework in the hierarchical GPU parameter server and absorbs the feature engineering workload into GPUs to eliminate excessive intermediate I/O.

III. FeatureBox Overview

In this section, we present an overview of FeatureBox. We aim at allowing the training framework to support pipeline processing with mini-batches so that we can eliminate the excessive intermediate resulting I/O in conventional stage-after-stage methods.

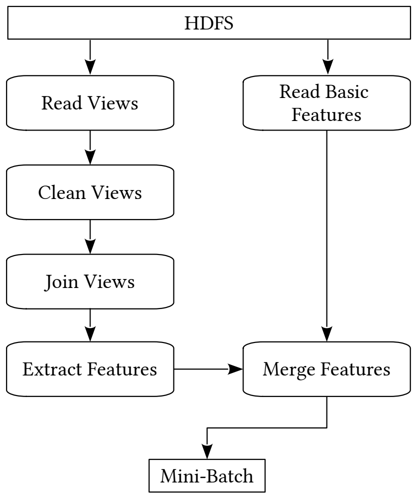


Fig. 3. FeatureBox pipeline.

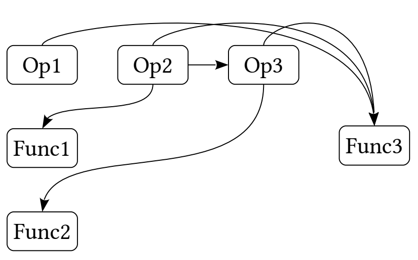
Figure 3 depicts the detailed workflow of the FeatureBox pipeline. The workflow has two major tracks: – extract features from input views and – reading basic features. A view is a collection of raw data logs from one source, e.g., user purchase history. CTR prediction models collect features from multiple sources to obtain the best performance. The views are read from the network file system HDFS. We need to clean the views by filling null values and filtering out unrelated instances. Afterwards, the views are joined with particular keys such as user id, ads id, etc. We extract features from the joined views to obtain the desired features from the input views. Then, these features are merged with the basic features, read in a parallel path. We provide a detailed illustration for these operations as follows:

Read views and basic features. The views and basic features are streamed from the distributed file system. The features are organized in a column-wise manner so that we only need to read the required features.

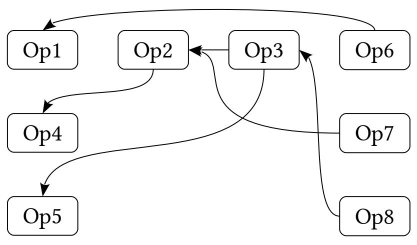
Clean views. Views contain null values and semi-structured data, e.g., JSON format. At the view cleaning stage, we fill the null values and extract required fields from the semi-structured data. Following the cleaning, all columns have non-empty and simple type (as integer, float, or string) fields. Note that the resulting views contain all the logged instances. For an application, it may not need to include all instances, e.g., an application for young people. A custom filter can be applied to filter out unrelated instances of the current application.

Join views. We now have one structured table for each view. Data from different views are concatenated by joining their keys, e.g., user id, ad id, etc. We recall that the join step combines multiple views into a single structured table.

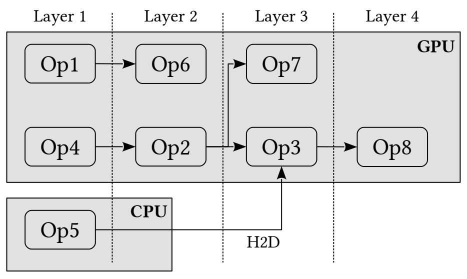
Extract features. Every time CTR model engineers propose a new feature, an operator that computes the new feature extraction on the structured table is created. A collection of those operators are executed in the feature extraction stage. The FeatureBox framework figures out the dependencies of operators and schedules the execution of the operators.



(a) Function call graph



(b) Dependency graph



(c) Heterogeneous Scheduling

Fig. 4. An example for the heterogeneous operator scheduling.

Merge features. The extracted features are further merged with the basic features read from HDFS. The merging is also realized by a join operation on the instance id, which is a unique value generated when an instance is logged. Subsequent to the merging, a mini-batch of training data is generated and is fed to the neural network for the training.

IV. Heterogeneous Operator Scheduling

The stages discussed above are represented as operators in the FeatureBox pipeline. Note that those operators are heterogeneous: Some operators are network I/O intensive, e.g., read views and read basic features; some operators are computation-intensive, e.g., clean views and extract features; and the remaining operators with joining, e.g., join views and merge features, rely on heavy memory consumption for large table joins (which corresponds to a large dictionary lookup). Therefore, we introduce a heterogeneous operator scheduler that manages the operator execution on both CPUs and GPUs.

Scheduling. Figure 4 shows an example for the heterogeneous operator scheduling algorithm. We first present a function call graph for operators in Figure 4(a). Three operators and three major functions are displayed in the example. Op1 calls Func3; Op2 calls Func1 and Func3; and Op3 calls Func2 and Func3, where Func1 and Func2 are pre-processing calls, and Func3 is a post-processing call. We make a fine granularity pipeline so that the initialing overhead of the pipeline is minimized. The fine-granularity is obtained by viewing each function call as a separate operator. Then, we obtain 5 more operators: Op4 is a call for Func1; Op5 is a call for Func2; Op6, Op7, and Op8 are the Func3 calls from Op1, Op2, and Op3, respectively. Their dependency graph is illustrated in Figure 4(b). Now we have a directed acyclic graph (DAG) for the operators. As shown in Figure 4(c), we perform a topological sort on the dependency graph, assign the operators with no dependencies (root operators) to the first layer, and put the remaining operators to the corresponding layer according to their depth from the root operators. With this layer-wise partition, we observe that the operators in the same layer do not have any execution dependency. We issue the operators in the same layer together and perform a synchronization at the end of each layer to ensure the execution dependency. We prefer to execute operators on GPUs unless an operator requires a significant memory footprint that does not fit in the GPU memory. For instance, Op5 (Func2) in Figure 4 is a word embedding table look up operation that requires a considerable amount of memory. We assign this operation to CPU workers and move its results from the CPU main memory to GPUs as a host-to-device (H2D) CUDA call.

TABLE I
The kernel launching overhead with an empty kernel on Nvidia Tesla V100-SXM2-32GB.

#Launches	1	10	100	1,000	10,000
Time (us)	4	35	360	3,619	34,515

Inner-GPU operator launching. After the layer-wise DAG operator scheduling, we have determined the execution device for each operator and the synchronization barriers. However, CUDA kernel launching is has a noticeable overhead. We report the CUDA kernel launch overhead in Table I. The test is performed on an Nvidia Tesla V100-SXM2-32GB GPU for an empty kernel with 5 pointer-type arguments. The CUDA driver version is 10.2. The average launching time for a kernel is around 3.5 us. Since we have fine-granularity operators, we have to rapidly launch CUDA kernels to execute the large number of operators. In order to eliminate the launching overhead, we rewrite the operator kernel as a CUDA device function for each operator in the same layer and create a meta-kernel that sequentially executes the operator device functions in a runtime-compilation manner. The overhead of the meta-kernel generation is disregarded—we only need to create this meta-kernel for each layer once as a pre-processing of the training since we determine the operator execution order before the actual training phase and keep the scheduling fixed. With the generated meta-kernels, we only need to launch one kernel for each layer.

V. GPU Memory Management

Feature extraction operators usually need to cope with strings of varying length, e.g., query keywords and ads titles. The execution of the operator commonly dynamically allocates memory to process the strings. For example, splitting a string with a delimiter needs to allocate an array to store the result of the splitting operation. We propose a light-weight block-level GPU memory pool to accelerate this dynamic allocation.

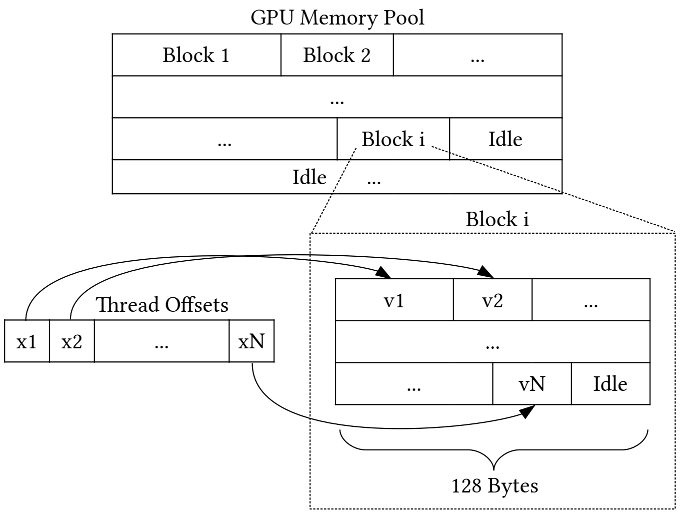


Fig. 5. A visual illustration for the GPU memory pool architecture.

Figure 5 presents a visual illustration for our proposed block-level GPU memory pool. The `Thread Offsets` denotes an array that stores the pointers to the dynamically allocated memory in the GPU memory pool. The memory in the GPU memory pool is pre-allocated in the GPU global memory. For each block, the allocated memory is aligned in 128 bytes for a cache-friendly execution.

Algorithm 1 In-Kernel Dynamic Memory Allocation

Input: allocation memory size for the i^{th} thread, $size_i$; global memory pool head pointer, *idle_memory_head*;
Output: thread offsets, $offsets_i$;

```
1: $prefix_{1..N} \leftarrow parallel\_prefix\_sum(size_{1..N})$
2: $address \leftarrow atomic\_add(idle\_memory\_head, prefix_N)$
3: for each thread $i$ in the current block concurrently do
4:     $offsets_i \leftarrow address + prefix_i - prefix_1$
5: end for
```

Dynamic GPU memory allocation. Algorithm 1 describes the workflow of the in-kernel dynamic memory allocation. We maintain a global variable *idle_memory_head* that stores the pointer of the head address of our pre-allocated GPU memory pool. We assume each GPU thread in a block has computed their required allocation size $size_i$. We first compute an in-block parallel prefix sum on $size_{1..N}$ to obtain the prefix sum $prefix_{1..N}$, where N is the number of threads in a block. The prefix sum is used to compute the total size of the requested memory. In addition, we can easily compute the thread offsets by adding the prefix sum to the head of the allocated memory address. After that, we let one thread in the block, e.g., thread 1, to apply the memory for the entire block—the total size is $prefix_N$. The memory allocation is implemented by an *atomic_add* operation. Line 1 calls the CUDA atomic add that adds $prefix_N$ to *idle_memory_head* and returns the old value of *idle_memory_head* to *address* in an atomic fashion—no data race within this operation. Once the requested memory is allocated for the block, we increment the *idle_memory_head* pointer in the memory pool. We finalize the allocation by letting all threads in the block compute their corresponding offsets by adding the prefix sum to the allocated address. The memory allocation is called inside the meta-kernel that we generated in the operator scheduling. The entire allocation process has very little overhead costs—it does not require any inter-block synchronization or any kernel launches.

Reset GPU memory pool. Our light-weight memory allocation strategy only maintains a pointer on a pre-allocated continuous global memory. However, the single-pointer design does not support memory freeing. We have to maintain an additional collection of freed memory and allocate the requested memory chunks from this collection—the maintenance of this additional data structure leads to significant memory allocation overhead. We observe that our operators are in fine-granularity and are scheduled layer by layer. Therefore, we can assume that the total required memory for dynamic allocations fits the GPU memory. We perform the memory release in a batch fashion: the memory pool is reset after each meta-kernel. The reset can be done in a constant time—we only need to set *idle_memory_head* to the original allocated memory address for the memory pool so that the allocation request in the meta-kernel for the following layer gets the allocation from the beginning of the memory pool.

VI. Experimental Evaluation

In this section, we investigate the effectiveness of our proposed framework FeatureBox through a set of numerical experiments. Specifically, the experiments are targeted to address the following questions:

- How is the end-to-end training time of FeatureBox compared with the previous MapReduce solution?
- How much intermediate I/O is saved by the pipelining architecture?
- What is the performance of FeatureBox in the feature extraction task?

Systems. The MapReduce feature extraction baseline is our previous in-production solution to extract features for the training tasks. It runs in an MPI cluster with CPU-only nodes in a data center. Commonly, a feature extraction job requires 20 to 30 nodes. Each node is equipped with server-grade CPUs ($\sim$100 threads). The training part is executed on GPU nodes. Each GPU node has 8 cutting-edge 32 GB HBM GPUs, $\sim$1 TB main memory, $\sim$20 TB RAID-0 NVMe SSDs, and a 100 Gb RDMA network adaptor. The training framework is the hierarchical GPU parameter server. All nodes are interconnected through a high-speed Ethernet switch.

Models. We use CTR prediction models on two real-world online advertising applications. The neural network backbones of both models follow the design in Figure 2. The major difference between the two models is the number of input features. Both models have $\sim$10 TB parameters. We collect real user click history logs as the training dataset.

TABLE II
END-TO-END TRAINING OF MAPREDUCE FEATURE EXTRACTION WITH HIERARCHICAL GPU PARAMETER SERVER AND FEATUREBOX.

	Application A		Application B	
#Instances	$\sim 1 \times 10^9$		$\sim 2 \times 10^9$	
Log Size	∼15 TB		∼25 TB	
Framework	MapReduce + GPU	FeatureBox	MapReduce + GPU	FeatureBox
#Machines	20 CPU + 1 GPU	1 GPU	30 CPU + 2 GPU	2 GPU
Execution Time	18h	3.5h	27h	2.65h
Speedup	-	5.14X	-	10.19X
Intermediate I/O Saving	-	∼50 TB	-	∼100 TB

A. End-to-End Training

We report Table II specifications about the training data and the end-to-end training comparison between our proposed FeatureBox and the MapReduce feature extraction with hierarchical GPU parameter server training as a baseline. Both training datasets contain billions of instances. The size of the logs is ∼15 TB for application A, and ∼25 TB for application B. The end-to-end training time includes the features extraction from the log time and the model training time. FeatureBox uses 1 GPU server for application A and 2 GPU servers for application B. In addition to the GPU servers, the baseline solution also employs 20/30 CPU-only servers to perform feature extraction. The baseline solution first extracts features using MapReduce, saves the features as training data in HDFS, and streams the generated training data to the GPU servers to train the model. On the other hand, FeatureBox processes the data in a pipeline fashion: features are extracted on GPU servers and then are immediately fed to the training framework on the same GPU server. For application A, FeatureBox only takes 3.5 hours to finish the feature extraction and the training while the baseline solution requires 18 hours—with fewer number of machines, FeatureBox has a *5.14X* speedup compared to the baseline. Meanwhile, Application B presents a bigger volume of log instances. Hence, we use two GPU servers to perform the training. We can observe a larger gap between FeatureBox and the baseline when the data size scales up: FeatureBox outperforms the baseline with a *10.19X* speedup. One of the major reason of the speedup is that FeatureBox eliminates the huge intermediate I/O from the MapReduce framework. We save ∼50-100 TB intermediate I/O while using FeatureBox.

B. Feature Extraction

Although the improvement of FeatureBox in the end-to-end training time mainly benefits from the pipeline architecture, we also investigate the feature extraction performance to confirm that our proposed GPU feature extraction framework is a better alternative to the baseline MapReduce solution.

We report, in Figure 6, the time to extract features from $10,000$ log instances of Application B. MapReduce runs on 30 CPU-only servers and FeatureBox runs on 2 GPU servers. The pre-processing time includes the stages to prepare the data for the feature extraction, such as read, clean, and join views. The pre-processing time of both methods are comparable because the executed operations are mostly memory and network I/O. Regarding the time to extract features, FeatureBox is more than 3 times faster than MapReduce. FeatureBox only takes around half of the time to extract the features than the baseline.

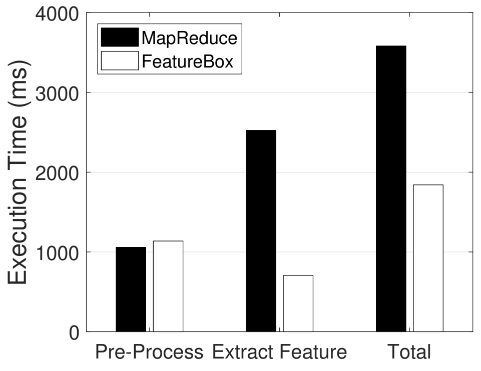


Fig. 6. Feature extraction time of MapReduce and FeatureBox.

C. Discussion

Based on these results, we can answer the questions that drive the experiments: The end-to-end training time of FeatureBox is 5-10 times faster than the baseline. Due to the pipeline design, FeatureBox saves us 50-100 TB intermediate I/O. For feature extraction only tasks, FeatureBox on 2 GPU servers is 2X faster than MapReduce on 30 CPU-only servers.

VII. CONCLUSIONS

In this paper, we introduce FeatureBox, a novel end-to-end training framework that pipelines the feature extraction and the training on GPU servers to save the intermediate I/O of the feature extraction. We rewrite computation-intensive feature extraction operators as GPU operators and leave the memory-intensive operator on CPUs. We introduce a layer-wise operator scheduling algorithm to schedule these heterogeneous operators. We present a light-weight GPU memory management algorithm that supports dynamic GPU memory allocation with minimal overhead. We experimentally evaluate FeatureBox and compare it with the previous in-production MapReduce feature extraction framework on two real-world ads applications. The results show that FeatureBox is 5-10X faster than the baseline.

References

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems (NeurIPS)*, Montreal, Canada, 2014, pp. 2672–2680.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, 2016, pp. 770–778.

[3] A. Voulodimos, N. Doulamis, A. D. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," *Comput. Intell. Neurosci.*, vol. 2018, pp. 7 068 349:1–7 068 349:13, 2018.

[4] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. L. García, I. Heredia, P. Malík, and L. Hluchỳ, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.

[5] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*. ACM, 2016, pp. 191–198.

[6] J. Wei, J. He, K. Chen, Y. Zhou, and Z. Tang, "Collaborative filtering and deep learning based recommendation system for cold start items," *Expert Systems with Applications*, vol. 69, pp. 29–39, 2017.

[7] H. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah, "Wide & deep learning for recommender systems," in *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, (DLRS@RecSys)*, Boston, MA, 2016, pp. 7–10.

[8] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–38, 2019.

[9] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical GPU parameter server for massive scale deep learning ads systems," in *Proceedings of Machine Learning and Systems (MLSys)*, Austin, TX, 2020.

[10] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] B. Edelman, M. Ostrovsky, and M. Schwarz, "Internet advertising and the generalized second-price auction: Selling billions of dollars worth of keywords," *American economic review*, vol. 97, no. 1, pp. 242–259, 2007.

[12] 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 *Proceedings of the 27th International Conference on Machine Learning (ICML)*, Haifa, Israel, 2010, pp. 13–20.

[13] Y. Shan, T. R. Hoens, J. Jiao, H. Wang, D. Yu, and J. C. Mao, "Deep crossing: Web-scale modeling without manually crafted combinatorial features," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, San Francisco, CA, 2016, pp. 255–262.

[14] Y. Qu, H. Cai, K. Ren, W. Zhang, Y. Yu, Y. Wen, and J. Wang, "Product-based neural networks for user response prediction," in *IEEE 16th International Conference on Data Mining (ICDM)*, Barcelona, Spain, 2016, pp. 1149–1154.

[15] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys)*, Boston, MA, 2016, pp. 191–198.

[16] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: A factorization-machine based neural network for ctr prediction," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, Melbourne, Australia, 2017, pp. 1725–1731.

[17] J. Lian, X. Zhou, F. Zhang, Z. Chen, X. Xie, and G. Sun, "xdeepfm: Combining explicit and implicit feature interactions for recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, London, UK, 2018, pp. 1754–1763.

[18] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai, "Deep interest network for click-through rate prediction," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, London, UK, 2018, pp. 1059–1068.

[19] W. Deng, J. Pan, T. Zhou, D. Kong, A. Flores, and G. Lin, "Deeplight: Deep lightweight feature interactions for accelerating ctr predictions in ad serving," in *Proceedings of the 14th ACM international conference on Web search and data mining*, 2021, pp. 922–930.