ScalaGBM: Memory Efficient GBDT Training for High-Dimensional Data on GPU

Borui Xu
Shandong Univeristy
Jinan, Shandong, China
boruixu@mail.sdu.edu.cn

Zeyi Wen
Hong Kong University of Science and Technology (Guangzhou), China
Hong Kong University of Science and Technology, Hong Kong SAR
wenzeyi@ust.hk

Yao Chen*
National University of Singapore
Singapore
yaochen@nus.edu.sg

Weiguo Liu*
Shandong Univeristy
Jinan, Shandong, China
weiguo.liu@sdu.edu.cn

Weng-Fai Wong
National University of Singapore
Singapore
wongwf@comp.nus.edu.sg

Bingsheng He
National University of Singapore
Singapore
hebs@comp.nus.edu.sg

Abstract

Gradient Boosted Decision Trees (GBDTs) are classical machine learning algorithms widely employed in recommendation systems, database queries, etc. Due to the extensive memory access involved in histogram-based GBDT training methods, high-bandwidth GPUs have been widely adopted to accelerate the training. However, when handling millions of feature data, it requires significant memory to store the training data and histograms, posing challenges for training on limited GPU memories. In this paper, we develop a GPU-based GBDT framework named ScalaGBM, aiming to accelerate high-dimensional data training with less memory usage. We first employ a CSR-like data format and CSR-based histogram construction to reduce the memory occupation of the training data. Then, we reorganize the training workflow with a double buffer structure to reduce the overall memory consumption for the histogram. Finally, we develop multi-dimensional parallel histogram construction and global optimal split point reduction to speed up the training process. Experimental results demonstrate that ScalaGBM handles real-world datasets with over 100 million instances of 50 million features with a single commercial GPU while existing GBDT frameworks all run into out-of-memory errors. Meanwhile, ScalaGBM achieves a maximum speedup of 39× over state-of-the-art GBDT counterparts without sacrificing the training quality. The code is available at https://github.com/Xtra-Computing/thundergbm.

CCS Concepts

• **Computing methodologies → Parallel algorithms**; **Boosting**;
• **Information systems → Data mining**.

*Corresponding authors

Keywords

High-Dimensional Data, GBDT Training, GPU, Memory Efficiency

ACM Reference Format:
Borui Xu, Zeyi Wen, Yao Chen, Weiguo Liu, Weng-Fai Wong, and Bingsheng He. 2025. ScalaGBM: Memory Efficient GBDT Training for High-Dimensional Data on GPU. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3690624.3709271

1 Introduction

Gradient Boosted Decision Trees (GBDTs) are classical machine learning algorithms that combine the predictions of multiple decision trees, each one correcting the errors of its predecessor. In recent years, GBDTs have gained significant attention due to their superior performance in various tasks such as regression, classification, and query optimizations [7, 12, 27, 30, 32–34]. According to a survey from Kaggle in 2022, 34.7% of respondents used GBDTs in the competitions [4, 18–20, 24].

The increasing demand for robust and efficient GBDT models has led to the development of several optimized frameworks, such as XGBoost [3], LightGBM [17], and CatBoost [26]. These works have been widely adopted in both academic [10, 31, 35] and industrial settings [28, 41] due to their effectiveness. To capitalize on the high memory bandwidth and processing capabilities of graphics processing units (GPUs), these works have also integrated GPU acceleration into the training processes. Specifically, ThunderGBM [38] has been designed to harness the power of GPUs for training GBDTs.

The histogram-based approach is the major method for training GBDTs, prompting significant efforts to accelerate this technique [8, 14, 29]. It relies on constructing histograms for each feature to find the optimal split feature and value for tree nodes, simplifying the process of finding the best splits during tree building. And the histogram-based method typically builds the tree layer by layer, known as a layer-wise manner. Specifically, the instances on each node are used to build gradient accumulation histograms. Then, it computes the gain of bins in histograms and finds the feature and value with the optimal gain as the node splitting criterion.

The data dimension of modern applications has been significantly expanding [6]. For instance, e-commerce datasets contain

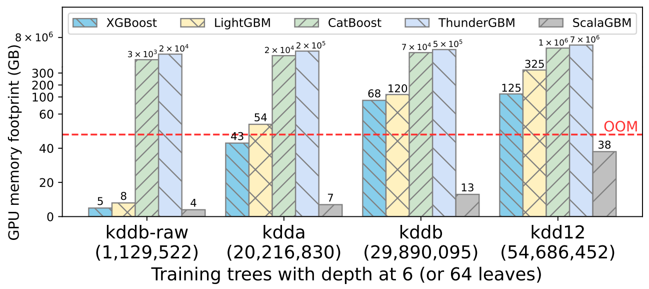


Figure 1: GPU memory consumption on high-dimensional datasets. The feature numbers are in parentheses. We estimate the GPU memory consumption when OOM happens.

the interactions of millions of customers, and disease understanding requires the analysis of millions of genetic variants; the data dimension has even reached tens of millions in datasets from competitions [16, 40]. However, existing GPU-based GBDT frameworks are typically designed for low-dimensional data inputs [3, 17, 26, 38] that can not support data dimensions in millions due to limited GPU memory [37]. Figure 1 depicts the GPU memory usage by various frameworks when training trees with a depth of 6 (or 64 leaves) on datasets with millions of features using a GPU with 48GB memory. Among all these existing frameworks, only XGBoost can train on datasets that do not exceed the feature dimension of *kdda*, which has 20,216,830 features. For datasets with feature dimensions exceeding those of *kdda*, all existing GPU-based frameworks encounter out-of-memory (OOM) failures. We estimate the memory requirements of the frameworks that encounter OOM based on their data management mechanism and plot them in the figure.

We find that significant GPU memory usage in Figure 1 is primarily attributed to the **substantial memory usage of training input data** and **the significant memory usage of the histograms**, both of which are caused by the scaled dimension of input data. Firstly, existing works generally use dense or structured sparse formats to store the input data in GPU memory, which can lead to high memory usage due to the extreme sparsity of high-dimensional data, causing redundancy with many zero elements. Secondly, the scaling up of the dimension of data raises the requirement of storing millions of histograms per tree node, making it impractical to keep all histograms for a layer in GPU memory, especially with deeper trees. Although using more machines or more GPUs can mitigate the issues of memory consumption, the problem of inefficient memory utilization on a single GPU remains unresolved.

To improve memory efficiency in GBDT training with high-dimensional inputs, we propose ScalaGBM, a memory-efficient GPU-based framework. In summary, we make the following contributions in ScalaGBM:

- We develop a CSR-like format and a CSR-based histogram construction method to reduce GPU memory consumption of training input data with high-dimension.
- We restructure the GBDT training flow by reordering and double buffering to further reduce the GPU memory usage of histograms.
- We invent a multi-dimensional parallel histogram construction and a global optimal split point reduction mechanism to further accelerate the training of high-dimensional data.

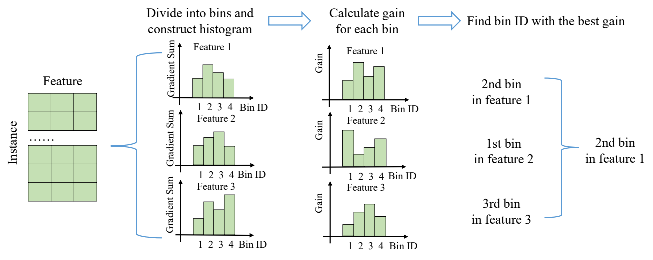


Figure 2: Histogram-based split point finding on one node.

- Experiments show that with 48GB of GPU memory, ScalaGBM can train high-dimensional data that existing GPU-based GBDT systems cannot handle and achieves up to 39x speedup over state-of-the-art GBDT frameworks without compromising quality.

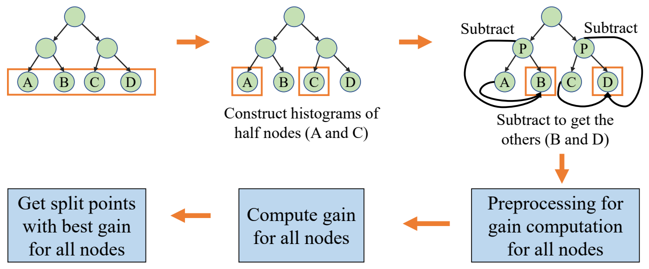
2 Background and Related Work

2.1 Histogram-based GBDT Training

The histogram-based method is one of the most commonly used techniques for training and has significantly improved the performance of GBDTs [3]. It constructs a histogram for each feature at every node of the tree and evaluates the split point of the tree node based on the bins in histograms. In a histogram, the feature values are divided into a fixed number of bins, each covering a specific value range. The partitioning of bins can be determined based on the data distribution or hyperparameter setting [9, 11]. When training GBDTs, the efficiency and effectiveness of the model largely depend on how the split points of leaf nodes in the decision trees are determined.

2.1.1 *Split Point Finding.* In the histogram-based method, a histogram is constructed for each feature. Each bin in the histogram represents an aggregation of gradients and Hessians for all instances whose feature values fall within that bin's range. The bin boundaries serve as candidate split points, and the gain (loss changes) is calculated for each possible split point. The bin boundary with the optimal gain is then selected as the split point. The partitioning of bins is performed once during the initialization phase and remains fixed during training. Histograms are built for all features of the current node whenever a split point is needed. Figure 2 gives an example of the split point finding in the histogram-based method. It first constructs a histogram for each feature using the instances at the current node, and finally, the boundary value of the second bin in the first histogram is used as the split point.

2.1.2 *Subtraction Technique.* The histogram-based training method is often accompanied by the subtraction technique to further improve the efficiency of finding split points. It leverages the fact that the sum of gradients and Hessians of instances is a constant for a given node. When evaluating split points for child nodes that have a common parent, instead of computing the sum of gradients and Hessians for both child nodes, we can calculate the sums for one node (e.g., the left) which has fewer instances and subtract these from the sums of the parent node to obtain the sums for the other node (e.g., the right). This technique can half the computation



A
B
C
D
Construct histograms of half nodes (A and C)
Subtract
Subtract
P
Subtract to get the others (B and D)
Preprocessing for gain computation for all nodes
Compute gain for all nodes
Get split points with best gain for all nodes

Figure 3: Layer-wise node histogram construction and split point evaluation flow with subtraction technique.

cost at least. Figure 3 shows the subtraction technique in the layer-wise method. The solid orange boxes represent nodes for which histograms need to be constructed and stored. We can obtain the histogram for node B by subtracting the histogram for node A from the histogram of their parent node.

2.2 GPU-based GBDT Training

Because of the training efficiency requirement, many efficient GBDT training frameworks are implemented targeting both CPU and GPU platform [1, 3, 5, 8, 13–15, 17, 21–23, 25, 26, 29, 36, 38, 39] with optimizations of different aspects of the training process. Due to the significant performance advantage of GPU-based GBDT training, in this paper, we only focus on GPU-based GBDT training frameworks.

Table 1 summarizes the mainstream GPU-based GBDT training frameworks with their different efficiency optimizations for the training process. XGBoost [3] implemented a histogram-based training approach on GPUs. It performs multi-level parallel computation for constructing histograms and evaluating split points at both the data and feature levels. Additionally, it leverages shared memory to alleviate the impact of irregular memory access patterns. LightGBM [17] and CatBoost [26] support similar histogram-based approaches on GPUs with optimized memory access. Mitchell et al. [23] presented a CUDA version of XGBoost. They proposed two parallel methods to build shallow trees and deep trees respectively, achieving a speedup of 3 to 6 times compared to the CPU. ThunderGBM [36, 38, 39] utilizes run-length encoding compression, two-stage histogram building, and reuse of intermediate results to improve the training efficiency, achieving 10x speedup. SketchBoost [13] is a GPU-based GBDT system focusing on multioutput problems. It uses approximation approaches to reduce the computational complexity without sacrificing accuracy. Shi et al. [29] proposed low-precision gradients to speed up histogram construction and reduce communication costs.

3 Memory Issues for High-Dimensional Data

Existing frameworks store all training data in GPU memory for efficiency. Therefore, when data scales to high dimensions, as shown in Figure 1, the inefficient GPU memory usage for the input data and histograms leads to the failure of training GBDTs on GPUs.

3.1 Extreme Memory Usage of Input Data

In each iteration, input data is used to construct histograms for each node. For low-dimensional data, dense format storage in GPU

Table 1: Comparison of GPU-based GBDT frameworks.

Feature	XGBoost	LightGBM	CatBoost	ThunderGBM	ScalaGBM
Sparse support	✓	×	×	×	✓
Gradient Quantization	×	✓	×	×	×
Exclusive feature bundling	×	✓	✓	×	×
Histogram storage optimization	×	×	×	×	✓
Full GPU implementation	✓	×	×	✓	✓

memory is preferred due to sequential memory access, which speeds up histogram construction. Despite some overhead, the high bandwidth utilization of dense formats offers better overall time efficiency compared with the sparse format. However, for high-dimensional data with even less than 0.1% non-zero elements, dense formats can lead to significant redundant memory use. LightGBM, CatBoost, and ThunderGBM all use the dense format. Although LightGBM and CatBoost reduce the feature dimension through exclusive feature bundling (EFB) and dynamically select the data type to save memory, substantial memory consumption persists. The memory required to store input data in dense format can be calculated using the following formula:

$$data\_size = N \times D \times data\_type \tag{1}$$

where N represents the number of instances, and D denotes the number of features used in the input data. $data\_type$ represents the number of bytes to store each element. The memory requirement is a linear function to the number of instances and number of features. *kddb* in Figure 1 has 19,264,097 instances and 29,890,095 features. With EFB, LightGBM utilizes only 6561 features for training, however, the required memory size is around 117.7GB even if 8-bit datatype is used, which is impractical to store on a single GPU. Different from LightGBM and CatBoost, XGBoost adopts structured sparse formats like ELLPACK [2] to reduce memory consumption for sparse data. However, ELLPACK is sensitive to the maximum number of existing features and consumes substantial memory. Thus, none of the above formats are ideal for high-dimensional data on GPUs.

3.2 Significant Memory Usage of Histograms

Layer-wise GBDT training is the most widely adopted one, such as XGBoost, CatBoost, and ThunderGBM. The large number of features in high-dimensional data makes histogram memory usage significant. Since histograms are needed for each feature, memory consumption on a tree node is proportional to the feature dimension. As shown in Figure 3, in traditional training workflow, histogram construction and split point evaluation are separate processes, with split point evaluation starting only after all histograms for the layer are constructed and stored. The following formula computes the minimum memory cost for histograms in the layer l:

$$hist\_size^l = 2^l \times B \times data\_type \tag{2}$$

where 2^l represents the number of nodes in layer l, B is the total number of bins across all feature histograms at a node, and $data\_type$ is the number of bytes per bin. As tree depth l increases,

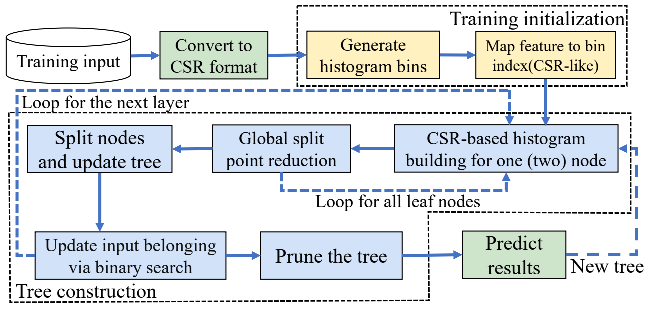


Figure 4: Training overview of ScalaGBM. It only stores the mapping of feature values to bin indices as training input and processes each node independently to avoid entire layer histogram storage.

memory requirements grow exponentially. Additionally, using the subtraction technique to accelerate histogram construction requires additional memory for storing parent node histograms. Thus, the total memory needed for layer l with the subtraction technique is calculated as follows:

$$total\_hist\_size^{l} = 3 \times 2^{l-1} \times B \times data\_type \tag{3}$$

As an instance, XGBoost stores 29,996,908 histogram bins per node on the *kddb* dataset, with each bin using 64 bits for first and second-order derivatives separately. Storing histograms for node splitting in layer 5 requires about 21.5GB. Besides that, due to other GPU memory for input data and split point evaluation, XGBoost fails to train on the *kddb* dataset with a 48GB GPU.

4 ScalaGBM Overview

To address the above issues, we design a GPU-based GBDT framework named ScalaGBM aiming to accelerate training with high-dimensional data with more efficient memory usage. Figure 4 outlines the training process of ScalaGBM.

First, during the initialization phase, the histogram bin boundaries are generated based on the compressed sparse row (CSR) format input data. Then, we map the input data to histogram bin indices and store them in a CSR-like format. During training, we **only keep bin indices in CSR-like format in GPU memory**, reducing the storage needed for input data. In the training phase, we construct histograms for each node based on the CSR-like bin indices and **integrate the histogram construction with the split point evaluation**, avoiding the storage of all histograms for one layer. Additionally, we employ **a double buffer structure** to maintain a single histogram array, optimize memory usage with the **subtraction technique**. After evaluating split points for all nodes in a layer, we update the tree structure and partition the instances. The above process is repeated until the predefined depth is reached.

We implement both the initialization and training process on the GPU. To avoid frequent memory allocations for histogram storage during training. We pre-allocate the maximum required memory during the initialization phase. To optimize training efficiency, we design **multi-dimensional parallel histogram construction** to optimize histogram construction and develop **global optimal split point reduction** to optimize split point evaluation.

In the following sections, we present the detailed memory optimization techniques and system implementations.

5 Memory Consumption Optimization

5.1 CSR-like Data Storage

In the histogram-based GBDT training, the actual data used for training is the histogram bin index data. It is generated from the original input data and stores the index of the histogram bin corresponding to each feature value of each instance. When building the histogram, we can use this index array to determine where the gradient pairs are accumulated. The original feature values are no longer necessary for the training. To reduce memory costs, we use a CSR-like format to store the bin information on the GPU. It consists of two arrays. The first is the bin mapper array which stores the global bin indices corresponding to non-zero feature values, and the second is the row pointer array which stores the starting position of each instance in the bin mapper array. The memory computation can be calculated as follows:

$$data\_size = (nnz + N + 1) \times data\_type \tag{4}$$

where nnz is the number of non-zero elements. It is more memory-friendly for storing high-dimensional sparse data. The traditional bin indices in the bin mapper array are independent in different feature histograms, which means the bin index in different feature histograms can overlap[38]. If the bin mapper array stores such bin indices, an instance may have multiple identical bin indices from different feature histograms. At that point, we need an additional nnz size array to identify which histogram each bin index belongs to during histogram construction. This not only increases the number of memory accesses during histogram construction but also doubles the storage for the input data. Therefore, we use the global index in the bin mapper array, where each feature has its own index range. For example, the index range for the bins of the first feature is from 1 to n, and for the second feature, it is from $n+1$ to $2n$, and so on. Indexes are cumulative, and there is no overlap between indices for different features. It can help save $nnz \times data\_type$ space, which is around 6GB on the *kdd12* dataset.

5.2 CSR-based Histogram Construction

We need to build histograms from the CSR-like data storage. Since the CSR-like format only stores non-zero elements and each instance may have a different number of non-zero elements, we cannot determine the instance ID using array indexing. Therefore, when building histograms for each node, we need to use the row pointer array to obtain the range of each instance in the bin mapper array. Algorithm 1 illustrates the histogram construction.

In Algorithm 1, $rowPointers$ stores the starting position of the bin index of each instance, $featureToBinId$ holds the global histogram bin indices for each instance, $gradientPair$ has first- and second-order gradient values for each instance. We use a double loop to traverse the non-zero elements. In the outer loop, we first obtain the index of the training instance, the gradient values of the instances, and the start and end positions of its bin indices in the bin mapper array. Then, in the inner loop, we iterate through each global bin index of each instance to obtain the accumulated position of the gradient pair. Finally, we use atomic addition to sum the

Algorithm 1: CSR-based Histogram Construction

```
Input: Row pointer array $rowPointers$,
       Feature to bin ID array $featureToBinId$,
       Instance gradient pair array $gradientPair$,
       Total number of instances $instanceCount$
Output: Histogram array $histogram$
1  for $instanceIndex \leftarrow 1$ to $instanceCount$ do
2      $startIndex \leftarrow rowPointers[instanceIndex]$;
3      $endIndex \leftarrow rowPointers[instanceIndex + 1]$;
4      $source \leftarrow gradientPair[instanceIndex]$;
       // scan non-zero elements
5      for $elementIndex \leftarrow startIndex$ to $endIndex$ do
           // The global bin index
6          $binIndex \leftarrow featureToBinId[elementIndex]$;
7          $dest \leftarrow histogram[binIndex]$;
8          if $sourcePair.gradient \neq 0$ then
               // Atomic operation
9              Add($dest.gradient, source.gradient$);
10         if $sourcePair.hessian \neq 0$ then
               // Atomic operation
11             Add($dest.hessian, source.hessian$);
```

gradient pairs in each histogram bin. As we can see, this histogram construction method does not require the original feature values or their corresponding feature IDs that are generally required by other frameworks.

5.3 Reorganization of Training Flow

After completing the histogram construction, we need to consider the efficient histogram storage. There is no need to store all the histogram data for the entire tree layer without the subtraction technique since histograms for different nodes are independent. We reorganize the histogram construction and split point evaluation phases, executing them in a loop. After evaluating a node's split point, its histogram data becomes obsolete, allowing us to reuse its memory to build histograms for the next node. No matter how deep the tree is, only the histograms for one node need to be stored. The required memory is:

$$hist\_size = B \times data\_type \tag{5}$$

The tree depth no longer affects memory consumption.

5.4 Double Buffering Histogram for Subtraction

When using subtraction to accelerate histogram construction, as shown in Figure 5, we can build histograms for two nodes simultaneously and iterate the process until all leaf nodes are processed. However, storing the newly constructed histogram data as parent histograms presents a challenge. The traditional method allocates two arrays, one to store the parent node histograms and the other to store the child node histograms. The size of the child node histogram array is twice the size of the parent node histogram array. To make full use of the parent node array, we pre-allocate the maximum space needed for the parent node array and use a double buffering method to write the child node histograms to the parent node array to reduce memory consumption.

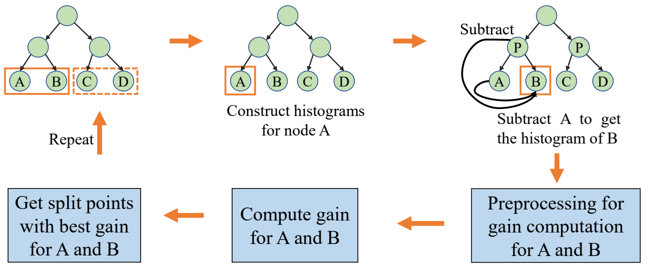

A
B
C
D
Repeat
Construct histograms for node A
Subtract
P
Subtract A to get the histogram of B
Get split points with best gain for A and B
Compute gain for A and B
Preprocessing for gain computation for A and B

Figure 5: Histogram construction and node split point evaluation flow in ScalaGBM. It iterates the flow till all leaves of a layer are processed.

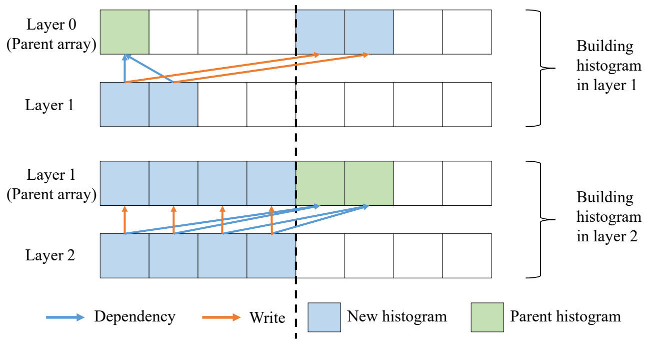


Figure 6: Use a double buffer structure to write histogram data back to the parent histogram array.

Figure 6 illustrates the idea of the double buffer structure to build and store histograms for layer 1 and layer 2 when the tree depth is 5. At this maximum depth, we need to store histograms for up to eight nodes from the third layer as parent histograms. Therefore, we predefine a node histogram array with a length of 8 as the parent array. Each box in the figure represents all histograms for a node. Blue boxes represent the histograms to be built and stored, while green boxes show the histograms currently in the parent node array. Blue arrows indicate dependencies in subtraction-based histogram construction, with child nodes pointing to parent nodes. Orange arrows highlight where new data is written into the parent node array. We divide the parent node array into two halves: the first four are in one buffer, while the remaining four are in another buffer. When building histograms for layer 1, we obtain the parent node histograms from the first half buffer and write the data into the second half. When building histograms for layer 2, we read the parent node data from the second half buffer and write the data into the first half. The system uses the parity of the current tree depth to determine which buffer to write back.

When training a tree with depth L, this double buffering structure can reduce the required histogram memory usage in the subtraction technique from $3 \times 2^{L-2} \times B \times data\_type$ to $2^{L-2} \times B \times data\_type$.

6 Efficient Framework Implementation

Existing frameworks encounter long initialization processes when training GBDTs on high-dimensional data [17, 26]. To avoid it, we implement ScalaGBM that can run fully on the GPU. For the initialization phase, we parallelize the histogram bin partitioning and the generation of CSR-like input data on the GPU and allocate the memory for histogram storage.

6.1 Parallel Histogram Construction

We parallelize the Algorithm 1 on the GPU. One simple parallel approach is to execute the outermost loop in parallel. However, this may lead to load imbalances because the number of non-zero elements in each instance often varies, resulting in some threads in the same warp waiting for a long time. Additionally, as the tree depth increases, the number of instances per node decreases, and only parallelizing the outermost loop may not fully unleash the parallel potential of the GPU. Therefore, to better leverage the parallel power, we use multi-dimensional parallel histogram construction to make parallelization of both the inner and outer loops, taking advantage of the fact that the GPU can perform parallel computation across multiple dimensions. Specifically, we associate the CUDA block structure with instances, using one or more blocks to process each instance. We organize blocks using a two-dimensional grid of blocks. The first dimension represents the number of instances for histogram construction, and the second dimension represents how many blocks are needed for each instance. The calculation formula for the second dimension is as follows:

$$n\_block = min\{\lceil \frac{nnz \div n\_instance}{block\_size} \rceil, max\_block\} \tag{6}$$

where n_block stands for the number of blocks for each instance. $n\_instance$ is the instance number used to construct the histogram, and $block\_size$ is a predefined thread number in one block. We calculate the average feature number and divide it by the number of threads in a block to determine the required block numbers. Rounding up is used to ensure that the number is never zero. We predefine max_block to represent the maximum number of blocks that can be used by each instance and choose the smaller one.

6.2 Global Optimal Split Point Reduction

After completing the histogram construction for each candidate split point, we need to obtain the split point with the maximum gain for a given node. The traditional method typically calculates gains of split points and evaluates them to determine the local optimal split points for each feature. Then it aggregates all local optimal split points to get the global best one. To reduce unnecessary storage of local optimal split points and accelerate this process, we treat candidate split points in different features as a collective entity, traversing to compute their gains using a fixed number of blocks and employing a global reduction approach instead of local reduction for a node to obtain the optimal split point.

6.3 Other framework optimizations

6.3.1 Updating instance position by bin indices. After splitting nodes within a layer, updating the leaf node for each instance is essential. This involves determining if an instance goes to the left or right child node based on the split feature and its value. Instead of comparing feature values, we compare global bin indices from the bin mapper array. In CSR-like format, although bin indices cannot be accessed directly, all bin indices for a given instance are sorted in ascending order. So we use a binary search to find the corresponding bin indices.

Table 2: Datasets for performance evaluation

Dataset	Cardinality	Dimension	Density	Size
epsilon	400,000	2,000	100%	12GB
real-sim	72,309	20,958	0.24%	87MB
rcv1	677,399	47,236	0.15%	1.2GB
e2006	16,087	150,360	0.82%	485MB
kddb-raw	19,264,097	1,129,522	< 0.1%	2.1GB
news20	19,996	1,355,191	< 0.1%	134MB
url	2,396,130	3,231,961	< 0.1%	3.9GB
log1p	16,087	4,272,227	0.14%	2.2GB
web-trigram	350,000	16,609,143	< 0.1%	24GB
kdda	8,407,752	20,216,830	< 0.1%	2.5GB
kddb	19,264,097	29,890,095	< 0.1%	4.8GB
kdd12	149,639,105	54,686,452	< 0.1%	21GB

6.3.2 Handling possible OOM issues. During the initialization phase, potential OOM issues may arise in the generation of histogram bins. If GPU memory is not enough, ScalaGBM generates histogram bins on the CPU and transfers the results to GPU memory via PCIe. In the training phase, if the GPU memory cannot accommodate all feature histograms for one node during the histogram construction. ScalaGBM iteratively constructs histograms and evaluates the local optimal split point for one feature at a time. This process is iterated to complete the construction of histograms for all features on a node, along with the evaluation of split points.

7 Experimental Study

In this section, we empirically evaluate ScalaGBM against existing GBDT frameworks in terms of training time, GPU memory usage, and result quality. We also compare performance across different feature dimensions, tree depths, and dataset densities.

7.1 Experimental Setup

Platforms. All the experiments have been conducted on a platform running Rocky Linux 8.8 with dual Intel(R) Xeon(R) Platinum 8373C 36-core CPUs (144 threads in total), 512GB of RAM, and an NVIDIA RTX A6000 GPU of 48GB VRAM. All the source codes were compiled by GNU G++ compiler version 11.4, Clang++ compiler version 12.0.1 and 16.0.6, and NVCC compiler version 11.7. The NVIDIA GPU driver version is 530.30.02. We implemented our GBDT framework based on CUDA C++, Thrust, and CUB.

Datasets. We conducted experiments on 12 publicly available high-dimensional datasets[1]. Table 2 presents information about the datasets sorted by their feature dimensions. We used the *e2006* and *log1p* datasets for regression tasks and the remaining datasets for binary classification tasks.

Baselines. We used the latest C++ version of all baselines. Specifically, XGBoost version 2925ceb and ThunderGBM version 5062a3a, as the major baselines. We also compared ScalaGBM with LightGBM (version 5dfe716) and CatBoost (version 32b5faf). All frameworks were set to use histogram-based methods to build trees. Since training on GPUs is faster than on CPUs [38, 39], all training was

[1]http://www.csie.ntu.edu.tw/cjlin/libsvmtools/datasets

Table 3: End-to-end training time (seconds) comparison among XGBoost, LightGBM, CatBoost, ThunderGBM, and ScalaGBM.

Dataset	epsilon	real-sim	rcv1	e2006	kddb-raw	news20	url	log1p	web-trigram	kdda	kddb	kdd12
XGBoost	14.60	1.49	4.52	2.66	22.50	17.94	50.30	36.39	OOM	280.04	OOM	OOM
LightGBM	34.05	36.67	67.15	24.07	21.17	322.94	201.02	1295.59	OOM	OOM	OOM	OOM
CatBoost	**3.61**	6.44	38.59	19.21	OOM	181.64	OOM	445.39	OOM	OOM	OOM	OOM
ThunderGBM	5.05	2.57	OOM	1.71	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
ScalaGBM	3.99	**0.57**	**1.71**	**0.93**	**5.08**	**1.41**	**3.69**	**2.09**	**33.55**	**7.09**	**13.57**	**38.12**
Min. speedup	0.90	2.11	2.44	1.84	4.17	12.72	13.63	17.41	N.A.	39.50	N.A.	N.A.

Table 4: Average training time (seconds) per tree.

Dataset	XGBoost	Light GBM	CatBoost	Thunder GBM	Scala GBM
epsilon	0.32	0.24	0.04	0.10	0.09
real-sim	0.03	0.85	0.06	0.05	0.01
rcv1	0.11	1.37	0.20	OOM	0.04
e2006	0.06	0.36	0.10	0.03	0.02
kbbd-raw	0.54	0.34	OOM	OOM	0.11
news20	0.44	4.42	0.45	OOM	0.03
url	1.22	3.45	OOM	OOM	0.08
log1p	0.84	3.59	0.35	OOM	0.04
kdda	6.80	OOM	OOM	OOM	0.16

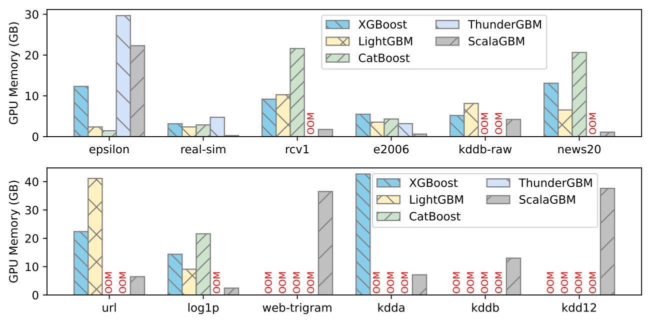


Figure 7: GPU memory consumption comparison on Nvidia RTX A6000 of 48GB VRAM.

conducted on GPUs. To conduct an end-to-end comparison, all the statistical time includes model initialization and training time.

Training Parameter Settings. According to the existing work [39], We fixed the tree depth at 6, boosting rounds at 40, maximum histogram bins at 255, learning rate at 1, minimum instances in a leaf node at 1, minimum sum of instance weight at 1, L2 regularization weight at 1, minimum split gain at 1, and training device at "CUDA". Due to an error with 144 CPU threads in CatBoost, it uses 120 threads, while other frameworks use all 144 threads by default. The remaining training parameters were kept at their default values. For binary classification tasks, we used logistic regression as the loss function. For regression tasks, we used mean squared error (RMSE) as the loss function.

7.2 Training Efficiency Evaluation

7.2.1 Training Time Comparison. Table 3 shows the training time (in seconds, to two decimal places) for different GBDT frameworks on 12 high-dimensional datasets.

Compared to existing frameworks, ScalaGBM handles higher-dimensional data more effectively. XGBoost struggles with data dimensions in the tens of millions, such as in the *web_trigram*, *kkb*, and *kdd12* datasets, due to the large memory demands of histograms and candidate split points. LightGBM faces similar issues and even has trouble with *kdda*. CatBoost cannot train on half of these datasets due to its lack of support for sparse data. ThunderGBM performs the worst with high-dimensional data, successfully training on only three datasets.

As the feature dimensions increase, the efficiency advantage of ScalaGBM becomes more significant. The bottom of Table 3 illustrates the minimum speedup of ScalaGBM relative to others. For small datasets, the speedup over XGBoost is about 2-3 times, but it rises to around 39 times on the *kdda* dataset as size and dimensions grow. ScalaGBM is also faster than LightGBM, with a 619 times speedup on the *log1p* dataset due to the lengthy CPU initialization of LightGBM (1151 seconds). In comparison with CatBoost, ScalaGBM is slightly inferior on the *epsilon* dataset, primarily because ScalaGBM is optimized for large-scale sparse datasets, and *epsilon* is dense. On other datasets, our training time is significantly shorter than CatBoost. Compared with ThunderGBM, ScalaGBM exhibits an average acceleration of 2.36 times, reaching 4.15 times on *real-sim*. Table 4 presents the average time for training a single tree excluding the initialization time, and ScalaGBM consistently exhibits the shortest training time across the sparse datasets.

Using the same hyperparameters, we also evaluated ScalaGBM and existing frameworks on two low-dimensional datasets, SUSY (5 million samples, 18 features) and HIGGS (11 million samples, 28 features). ThunderGBM achieves the highest efficiency, with runtimes of 0.7 seconds and 1.7 seconds, respectively, while ScalaGBM requires 1.7 seconds and 4 seconds. The training times for all frameworks are very short. While ScalaGBM focuses on high-dimensional data, it also performs competitively on low-dimensional data.

7.2.2 GPU Memory Usage Comparison. Figure 7 further analyzes the GPU memory requirements of each framework. Except for the *epsilon* dataset, ScalaGBM has the least GPU memory requirements on all the other datasets. On the *epsilon* dataset, the maximum memory requirement of ScalaGBM occurs during the histogram bin generation phase in the initialization. After that, the maximum memory usage during training is only 3.34GB. For the *kddb* dataset, where other frameworks fail to train due to limited memory, ScalaGBM

Table 5: Training quality comparison. We use accuracy for binary classification and RMSE for regression. "N.A." indicates that the framework cannot work correctly.

Dataset	Metric	XG Boost	Light GBM	Cat Boost	Thunder GBM	Scala GBM
epsilon	ACC	0.8274	0.8311	0.8484	0.8238	0.8267
real-sim		0.9411	0.9587	0.9225	0.9385	0.9430
rcv1		0.9619	0.9738	0.9546	N.A.	0.9618
kbbd-raw		N.A.	0.8849	N.A.	N.A.	0.8775
news20		0.9102	0.9285	N.A.	N.A.	0.9113
url		0.9850	0.9936	N.A.	N.A.	0.9861
kdda		0.8745	N.A.	N.A.	N.A.	0.8759
e2006	RMSE	0.4129	0.4114	N.A.	0.4019	0.4215
log1p		0.4078	0.4081	N.A.	N.A.	0.4076

Table 6: Parameter settings, speedup, and results when ScalaGBM has similar quality to LightGBM.

Dataset	Metric	Epoch	Depth	Speedup	Result	Light GBM
epsilon	ACC	100	6	3.54	**0.8384**	0.8311
real-sim		100	10	20.04	**0.9598**	0.9587
rcv1		100	12	4.25	**0.9743**	0.9738
kddb-raw		40	13	1.51	0.8818	0.8849
news20		100	10	69.45	0.9280	0.9285
url		100	10	9.81	**0.9936**	0.9936
e2006	RMSE	10	10	17.83	0.4140	0.4114
log1p		40	6	619.90	**0.4076**	0.4076

not only successfully trains but also consumes only 13GB of memory. This implies that ScalaGBM can run such large-scale datasets even on GPUs with smaller memory.

7.3 Model Quality Comparison

Table 5 compares the training quality after 40 epochs of training. For ease of comparison, we only show results on the datasets where at least two frameworks can train. The results are conducted on the test sets. XGBoost and CatBoost fail to predict when the feature dimensions of the test set differ from those of the training datasets.

ScalaGBM has comparable results to XGBoost. This similarity is because the tree construction approach of ScalaGBM resembles that of XGBoost, but slight differences arise due to variations in histogram bin generation and floating-point computation precision. Except for the *e2006* dataset, our training quality is also better than ThunderGBM. CatBoost performs better only on the *epsilon* datasets and fails to work on most datasets.

LightGBM's GPU version does not strictly constrain the tree depth, and as a result, improves the model quality. For example, on the *real-sim* dataset, trees of LightGBM can reach a depth of 30. By increasing tree depth and training rounds, ScalaGBM achieves comparable quality in less time. Table 6 shows the parameter settings and speedup of ScalaGBM compared with LightGBM when results are similar.

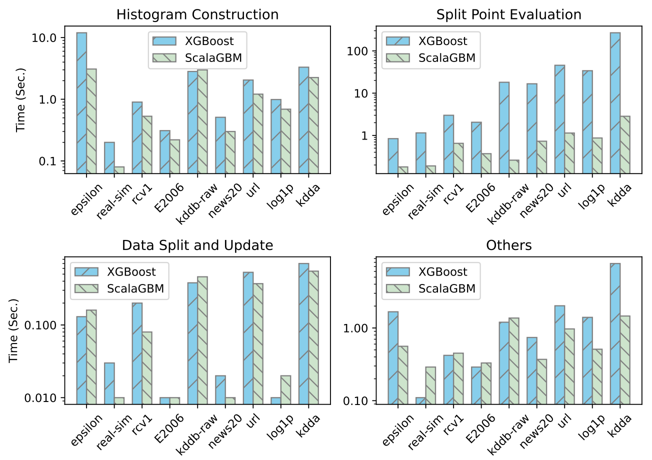


Figure 8: Time breakdown comparison with XGBoost.

Table 7: GPU memory consumption breakdown (GB).

Breakdown	Framework	rcv1	news20	url	kdda
Training data	XGBoost	2.22	0.88	2.77	2.08
	ScalaGBM	0.19	0.04	1.03	1.14
Histogram	XGBoost	3.99	7.06	7.28	9.98
	ScalaGBM	0.51	0.83	1.38	2.31

7.4 Training Time Breakdown Comparison

As XGBoost performs better than other existing works , we compare the time spent on each component with XGBoost on Figure 8 to further illustrate why ScalaGBM achieves faster training speeds. During histogram construction and data updates, ScalaGBM and XGBoost exhibit similar speed. However, in the split point evaluation phase, ScalaGBM is significantly faster than XGBoost. In XGBoost, the split point evaluation phase accounts for more than 80% of the total training. This is because XGBoost uses 64-bit arithmetic operations and allocates a block with a fixed 32 threads on the GPU for each histogram to find the local best one. If the number of bins in a histogram is less than 32, it will cause a waste of computing. Instead, ScalaGBM uses the global optimal split point reduction to avoid this problem.

7.5 GPU Memory Breakdown Comparison

In Table 7, we compare the GPU memory consumption of input training data and histogram storage between ScalaGBM and XGBoost. ScalaGBM consumes less memory than XGBoost in both parts, particularly in histogram storage. The CSR-like data format reduces the storage of input data, while the reorganization of the training flow and the double buffering structure reduce the memory requirements for histograms.

7.6 Sensitivity Study

This section examines the changes in GPU memory consumption and training time for ScalaGBM and XGBoost by varying feature dimensions, tree depth, and dataset density. For feature dimensions, we fixed the number of instances and varied the feature percentages

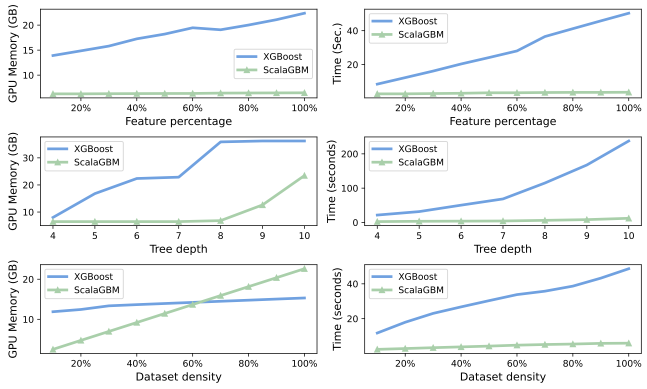


Figure 9: GPU memory usage and training time change with different feature dimensions, tree depth, and dataset density.

in 10% increments on the *url* dataset. Following [39], we tested tree depths from 4 to 10 using the *url* dataset. For dataset density, we synthesized datasets with 10,000 instances and 100,000 features, with densities from 10% to 100%. All experiments ran for 40 epochs.

7.6.1 Varying Feature Dimension. The top of Figure 9 shows the impact of feature dimensions. Both metrics increase almost linearly for XGBoost and ScalaGBM as feature dimensions grow, with a greater impact on XGBoost due to more histogram and candidate split storage requirements.

7.6.2 Varying Tree Depth. The middle of Figure 9 illustrates the impact of tree depth on GPU memory and training time. ScalaGBM consistently uses less GPU memory than XGBoost, with constant memory usage for depths of 7 or less, as maximum memory usage occurs during initialization. the GPU memory consumption of XGBoost stabilizes beyond a tree depth of 8 due to its predefined processing of 32 nodes at a time.

7.6.3 Varying Dataset Density. The bottom of Figure 9 indicates that increasing dataset density leads to linear growth in memory consumption and training time. ScalaGBM consumes less GPU memory for densities below 60%, thanks to its sparse data format, and consistently outperforms XGBoost in training speed due to better parallelization implementation.

8 Conclusion

In this paper, we first highlighted how the training input data and histograms significantly impact memory consumption when training GBDTs on high-dimensional data. Then, we developed a system named ScalaGBM to support GBDT training for higher dimensional data and optimize efficiency. To reduce the training input data storage, we employed a CSR-like data format and used CSR-based methods for histogram construction. To minimize histogram data storage, we reorganized the training process and used the double buffer to store histograms. Additionally, we optimized training efficiency by implementing multi-dimensional parallel histogram construction and global optimal split point evaluation. Experimental results demonstrate that ScalaGBM has better support for high-dimensional data and is faster to train than existing works.

Acknowledgments

This work is funded by the National Key R&D Program of China (No. 2019YFA0709400), the Ministry of Education AcRF Tier 2 grant, Singapore (NO. MOE-T2EP20121-0016), the NSFC Project (No. 62306256), the Guangzhou Science and Technology Development Projects (No. 2023A03J0143 and No. 2024A04J4458), and the Guangzhou Municipality Big Data Intelligence Key Lab (NO. 2023A03J0012).

References

[1] Firas Abuzaid, Joseph K Bradley, Feynman T Liang, Andrew Feng, Lee Yang, Matei Zaharia, and Ameet S Talwalkar. 2016. Yggdrasil: An optimized system for training deep decision trees at scale. *Advances in Neural Information Processing Systems* 29 (2016).

[2] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(SC '09)*. Association for Computing Machinery, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/1654059.1654078

[3] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[4] ML Contests. 2023. *Tabular Data Competitions - Winning Strategies.* https://mlcontests.com/tabular-data/

[5] Darren Cook. 2016. *Practical machine learning with H2O: powerful, scalable techniques for deep learning and AI.* " O'Reilly Media, Inc.".

[6] David L Donoho et al. 2000. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS math challenges lecture* 1, 2000 (2000), 32.

[7] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. 12, 9 (may 2019), 1044–1057. https://doi.org/10.14778/3329772.3329780

[8] Fangcheng Fu, Jiawei Jiang, Yingxia Shao, and Bin Cui. 2019. An Experimental Evaluation of Large Scale GBDT Systems. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1357–1370. https://doi.org/10.14778/3342263.3342273

[9] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-Based Quantile Sketches for Efficient High Cardinality Aggregation Queries. *Proc. VLDB Endow.* 11, 11 (jul 2018), 1647–1660. https://doi.org/10.14778/3236187.3236212

[10] Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. 2021. Revisiting Deep Learning Models for Tabular Data. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 18932–18943. https://proceedings.neurips.cc/paper_files/paper/2021/file/9d86d83f925f2149e9edb0ac3b49229c-Paper.pdf

[11] Michael Greenwald and Sanjeev Khanna. 2001. Space-Efficient Online Computation of Quantile Summaries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (Santa Barbara, California, USA) *(SIGMOD '01)*. Association for Computing Machinery, New York, NY, USA, 58–66. https://doi.org/10.1145/375663.375670

[12] Shihao Gu, Bryan Kelly, and Dacheng Xiu. 2020. Empirical Asset Pricing via Machine Learning. *The Review of Financial Studies* 33, 5 (02 2020), 2223–2273. https://doi.org/10.1093/rfs/hhaa009 arXiv:https://academic.oup.com/rfs/article-pdf/33/5/2223/33209812/hhaa009.pdf

[13] Leonid Iosipoi and Anton Vakhrushev. 2022. SketchBoost: Fast Gradient Boosted Decision Tree for Multioutput Problems. *Advances in Neural Information Processing Systems* 35 (2022), 25422–25435. https://openreview.net/forum?id=WSxarC8t-T

[14] Jiawei Jiang, Bin Cui, Ce Zhang, and Fangcheng Fu. 2018. DimBoost: Boosting Gradient Boosting Decision Tree to Higher Dimensions. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1363–1376. https://doi.org/10.1145/3183713.3196892

[15] Jie Jiang, Jiawei Jiang, Bin Cui, and Ce Zhang. 2017. TencentBoost: A Gradient Boosting Tree System with Parameter Server. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 281–284. https://doi.org/10.1109/ICDE.2017.87

[16] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. 2016. Field-aware factorization machines for CTR prediction. In *Proceedings of the 10th ACM conference on recommender systems.* 43–50.

[17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 3149–3157.

[18] Guolin Ke, Zhenhui Xu, Jia Zhang, Jiang Bian, and Tie-Yan Liu. 2019. DeepGBM: A Deep Learning Framework Distilled by GBDT for Online Prediction Tasks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 384–394. https://doi.org/10.1145/3292500.3330858

[19] Pan Li, Zhen Qin, Xuanhui Wang, and Donald Metzler. 2019. Combining Decision Trees and Neural Networks for Learning-to-Rank in Personal Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 2032–2040. https://doi.org/10.1145/3292500.3330676

[20] Guang Liu, Yuzhao Mao, Qi Sun, Hailong Huang, Weiguo Gao, Xuan Li, Jianping Shen, Ruifan Li, and Xiaojie Wang. 2020. Multi-scale Two-way Deep Neural Network for Stock Trend Prediction. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, Christian Bessiere (Ed.). International Joint Conferences on Artificial Intelligence Organization, 4555–4561. https://doi.org/10.24963/ijcai.2020/628 Special Track on AI in FinTech.

[21] Qi Meng, Guolin Ke, Taifeng Wang, Wei Chen, Qiwei Ye, Zhi-Ming Ma, and Tie-Yan Liu. 2016. A Communication-Efficient Parallel Algorithm for Decision Tree. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 1279–1287.

[22] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7. http://jmlr.org/papers/v17/15-237.html

[23] Rory Mitchell and Eibe Frank. 2017. Accelerating the XGBoost algorithm using GPU computing. *PeerJ Computer Science* 3 (2017), e127. https://doi.org/10.7717/peerj-cs.127

[24] Paul Mooney. 2022. *Kaggle Survey 2022: All Results.* https://www.kaggle.com/code/paultimothymooney/kaggle-survey-2022-all-results/notebook

[25] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 85 (2011), 2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html

[26] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: Unbiased Boosting with Categorical Features. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 6639–6649.

[27] Zhen Qin, Le Yan, Honglei Zhuang, Yi Tay, Rama Kumar Pasumarthi, Xuanhui Wang, Michael Bendersky, and Marc Najork. 2021. Are Neural Rankers still Outperformed by Gradient Boosted Decision Trees?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Ut1vF_q_vC

[28] Elissaios Sarmas, Evangelos Spiliotis, Nikos Dimitropoulos, Vangelis Marinakis, and Haris Doukas. 2023. Estimating the Energy Savings of Energy Efficiency Actions with Ensemble Machine Learning Models. *Applied Sciences* 13, 4 (2023). https://doi.org/10.3390/app13042749

[29] Yu Shi, Guolin Ke, Zhuoming Chen, Shuxin Zheng, and Tie-Yan Liu. 2022. Quantized Training of Gradient Boosting Decision Trees. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 18822–18833. https://proceedings.neurips.cc/paper_files/paper/2022/file/77911ed9e6e864ca1a3d165b2c3cb258-Paper-Conference.pdf

[30] Ravid Shwartz-Ziv and Amitai Armon. 2022. Tabular data: Deep learning is not all you need. *Information Fusion* 81 (2022), 84–90. https://doi.org/10.1016/j.inffus.2021.11.011

[31] Yapeng Su, Dan Yuan, Daniel G Chen, Rachel H Ng, Kai Wang, Jongchan Choi, Sarah Li, Sunga Hong, Rongyu Zhang, Jingyi Xie, et al. 2022. Multiple early factors anticipate post-acute COVID-19 sequelae. *Cell* 185, 5 (2022), 881–895. https://doi.org/10.1016/j.cell.2022.01.014

[32] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: a design space exploration and a comparative evaluation. 15, 1 (sep 2021), 85–97. https://doi.org/10.14778/3485450.3485459

[33] Xiaolei Sun, Mingxi Liu, and Zeqian Sima. 2020. A novel cryptocurrency price trend forecasting model based on LightGBM. *Finance Research Letters* 32 (2020), 101084. https://doi.org/10.1016/j.frl.2018.12.032

[34] Kapil Vaidya, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2021. Leveraging query logs and machine learning for parametric query optimization. 15, 3 (nov 2021), 401–413. https://doi.org/10.14778/3494124.3494126

[35] Farui Wang, Weizhe Zhang, Shichao Lai, Meng Hao, and Zheng Wang. 2022. Dynamic GPU Energy Optimization for Machine Learning Training Workloads. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 2943–2954. https://doi.org/10.1109/TPDS.2021.3137867

[36] Zeyi Wen, Bingsheng He, Ramamohanarao Kotagiri, Shengliang Lu, and Jiashuai Shi. 2018. Efficient Gradient Boosted Decision Tree Training on GPUs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 234–243. https://doi.org/10.1109/IPDPS.2018.00033

[37] Zeyi Wen, Qinbin Li, Bingsheng He, and Bin Cui. 2021. Challenges and Opportunities of Building Fast GBDT Systems. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Zhi-Hua Zhou (Ed.). International Joint Conferences on Artificial Intelligence Organization, 4661–4668. https://doi.org/10.24963/ijcai.2021/632 Survey Track.

[38] Zeyi Wen, Hanfeng Liu, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. 2020. ThunderGBM: Fast GBDTs and Random Forests on GPUs. *Journal of Machine Learning Research* 21, 108 (2020), 1–5. http://jmlr.org/papers/v21/19-095.html

[39] Zeyi Wen, Jiashuai Shi, Bingsheng He, Jian Chen, Kotagiri Ramamohanarao, and Qinbin Li. 2019. Exploiting GPUs for efficient gradient boosting decision tree training. *IEEE Transactions on Parallel and Distributed Systems* 30, 12 (2019), 2706–2717.

[40] Hsiang-Fu Yu, Hung-Yi Lo, Hsun-Ping Hsieh, Jing-Kai Lou, Todd G. McKenzie, Jung-Wei Chou, Po-Han Chung, Chia-Hua Ho, Chun-Fu Chang, Yin-Hsuan Wei, Jui-Yu Weng, En-Syu Yan, Che-Wei Chang, Tsung-Ting Kuo, Yi-Chen Lo, Po Tzu Chang, Chieh Po, Chien-Yuan Wang, Yi-Hung Huang, Chen-Wei Hung, Yu-Xun Ruan, Yu-Shi Lin, Shou-De Lin, Hsuan-Tien Lin, and Chih-Jen Lin. 2011. Feature Engineering and Classifier Ensemble for KDD Cup 2010. In *JMLR Workshop and Conference Proceedings*. To appear.

[41] Xiaoming Yuan, Jiahui Chen, Kuan Zhang, Yuan Wu, and Tingting Yang. 2022. A Stable AI-Based Binary and Multiple Class Heart Disease Prediction Model for IoMT. *IEEE Transactions on Industrial Informatics* 18, 3 (2022), 2032–2040. https://doi.org/10.1109/TII.2021.3098306