Exploiting GPUs for Efficient Gradient Boosting Decision Tree Training

Zeyi Wen, Jiashuai Shi, Bingsheng He, Jian Chen, Kotagiri Ramamohanarao and Qinbin Li

Abstract—In this paper, we present a novel parallel implementation for training Gradient Boosting Decision Trees (GBDTs) on Graphics Processing Units (GPUs). Thanks to the excellent results on classification/regression and the open sourced libraries such as XGBoost, GBDTs have become very popular in recent years and won many awards in machine learning and data mining competitions. Although GPUs have demonstrated their success in accelerating many machine learning applications, it is challenging to develop an efficient GPU-based GBDT algorithm. The key challenges include irregular memory accesses, many sorting operations with small inputs and varying data parallel granularities in tree construction. To tackle these challenges on GPUs, we propose various novel techniques including (i) Run-length Encoding compression and thread/block workload dynamic allocation, (ii) data partitioning based on stable sort, and fast and memory efficient attribute ID lookup in node splitting, (iii) finding approximate split points using two-stage histogram building, (iv) building histograms with the aware of sparsity and exploiting histogram subtraction to reduce histogram building workload, (v) reusing intermediate training results for efficient gradient computation, and (vi) exploiting multiple GPUs to handle larger data sets efficiently. Our experimental results show that our algorithm named *ThunderGBM* can be 10x times faster than the state-of-the-art libraries (i.e., XGBoost, LightGBM and CatBoost) running on a relatively high-end workstation of 20 CPU cores. In comparison with the libraries on GPUs, ThunderGBM can handle higher dimensional problems which the libraries become extremely slow or simply fail. For the data sets the existing libraries on GPUs can handle, ThunderGBM achieves up to 10 times speedup on the same hardware, which demonstrates the significance of our GPU optimizations. Moreover, the models trained by ThunderGBM are identical to those trained by XGBoost, and have similar quality as those trained by LightGBM and CatBoost.

Index Terms—Graphics Processing Units, Gradient Boosting Decision Trees, Machine Learning.

1 Introduction

The recent advancement of machine learning technologies is not only because of new algorithms to improve accuracy, but also new algorithms and systems to exploit the high-performance hardware (e.g., GPUs and FPGAs) to improve efficiency. Nowadays, many companies (e.g., Amazon, Google and Microsoft) are providing GPU clouds as an integral component in computing infrastructure. More and more researchers are exploring GPU clouds for machine learning algorithms [1], [2].

Recently, Gradient Boosting Decision Trees (GBDTs) are widely used in advertising systems, spam filtering, sales prediction, medical data analysis, and image labeling [3], [4], [5]. In contrast with deep learning, GBDTs have the advantage of simplicity, effectiveness, and user-friendly open source toolkits such as XGBoost [3], LightGBM [6] and CatBoost [7]. Additionally, the GBDT has won many awards in recent machine learning and data mining competitions (e.g., Kaggle competitions). However, training GBDTs is often very time-consuming, especially for training a large number of deep trees on large data sets. In this article, we propose a novel GPU-based algorithm called *ThunderGBM* to improve GBDT training performance.

- *Z. Wen, B. He and Q. Li are with SoC, National University of Singapore. E-mail: {wenzy, hebs, qinbin}@comp.nus.edu.sg*
- *J. Shi and J. Chen are with South China University of Technology. E-mail: shijiashuai@gmail.com, ellachen@scut.edu.cn*
- *R. Kotagiri is with The University of Melbourne, Australia. E-mail: kotagiri@unimelb.edu.au*

The GBDT is essentially an ensemble machine learning technique where multiple decision trees are trained and used to predict unseen data. A decision tree is a binary tree in which each internal node is attached with a yes/no question and the leaves are labeled with the target values (e.g., "spam" or "non-spam" in spam filtering). Unlike random forests where individual decision trees are independent [8], the trees of GBDTs are dependent. Thus, it is challenging to develop an efficient parallel GBDT training algorithm. There are a number of key challenges on the efficiency of GPU accelerations for GBDTs, such as irregular memory accesses, many sorting operations with small inputs and varying data parallel granularities in tree construction (more details are presented in Section 3.1).

We have developed ThunderGBM, a highly efficient GPU-based training algorithm, to address the challenges. ThunderGBM is powered by many techniques specifically designed for GPUs. Notably, to exploit the massive thread parallelism of GPUs, we develop fine-grained multi-level parallelism for GBDTs, from the node level, the attribute level parallelism to parallelizing the gain computation of each split point. Moreover, ThunderGBM exploits Run-length Encoding (RLE) compression, since RLE compression is able to (i) reduce memory consumption so that the GPU can handle larger data sets, (ii) improve the efficiency of finding the best split point due to the avoidance of repeated attribute values, (iii) retain efficiency in splitting nodes without a total decompression. We also propose various novel techniques to find approximate split points and exploit multiple GPUs to handle large data sets.

This article is an extension of our previous conference

paper [9]. Our major new contributions are summarized in the following three aspects.

- First, we propose new techniques which lead to further two times speedup over the implementation in our conference paper. The experimental results can be found in Section 5.1.3. The new techniques include using a stable sorting based method instead of a histogram based method for more efficiently partitioning data in parent nodes to child nodes (cf. Section 3.2.2), and improving the node splitting process using fast and memory efficient attribute ID lookup (cf. Section 3.5).
- Second, we extend ThunderGBM to support finding approximate split points which is not considered in our previous work. We propose a series of novel techniques for exploiting GPUs (cf. Section 3.4). Specifically, to reduce the shared memory consumption, we propose sparsity aware techniques to build histograms for efficient handling both low and high dimensional data. When building the histograms, we first build a partial histogram on GPU thread block-level using shared memory, and then build the global histogram by aggregating them. We also exploit histogram subtraction to significantly reduce the workload in building histograms.
- Third, this extension also enables the support for multiple GPUs in order to handle larger data sets which cannot be entirely stored in a single GPU. Through the attribute based training data partitioning, we avoid exchanging partial histograms among GPUs, and hence substantially reduce the communication cost among GPUs (cf. Section 4.1).

We conduct comprehensive experiments to compare our algorithm with the state-of-the-art libraries—XGBoost [3], LightGBM [6] and CatBoost [7]—on both CPUs and GPUs. The experimental results show that ThunderGBM can achieve 10x times speedup over the state-of-the-art libraries running on a relatively high-end workstation on CPUs with 20 cores. In comparison with the libraries on GPUs, ThunderGBM can handle high dimensional data sets which the existing libraries on GPUs fail. For data sets the existing libraries on GPUs can handle, ThunderGBM achieves up to 10 times speedup over the libraries on GPUs. This also implies that GPU optimizations are challenging, and our careful optimizations outperform those in existing libraries on GPU. Furthermore, the models produced by ThunderGBM are identical to those produced by XGBoost, and have similar quality as those trained by LightGBM and CatBoost. The source code of ThunderGBM is available on GitHub at https://github.com/xtra-computing/thundergbm.

2 Background and Related Work

In this section, we first present some background on GBDTs and GPUs, and then discuss the related studies GBDTs.

2.1 Background

Figure 1 gives an example of GBDTs and how the prediction works. The key idea is that multiple decision trees are trained for GBDTs, and the prediction result is the accumulated values of the individual prediction of each tree. A decision tree is a binary tree, where the internal nodes are associated with decision rules (e.g., "salary>100k") and the leaf nodes are associated with values (e.g., probability of buying a house). More details about decision trees can be found from this reference [3] and its related material. The decision trees in GBDTs are dependent, because the later tree corrects the error of the previous trees.

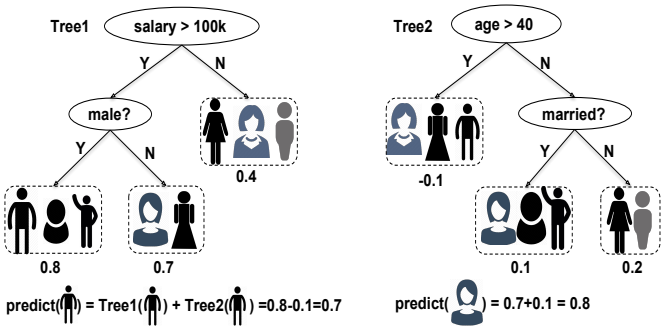


Fig. 1. An example of trees in the GBDT and its prediction

TABLE 1
Dense and sparse data representation

Dense	Sparse
$y_1 = 0.0, \boldsymbol{x}_1 = \langle 0.0, 0.0, 0.1, 0.0\rangle$	$y_1 = 0.0, \boldsymbol{x}_1 = (a_3 : 0.1)$
$y_2 = 0.4, \boldsymbol{x}_2 = \langle 1.2, 0.0, 0.1, 0.6\rangle$	$y_2 = 0.4, \boldsymbol{x}_2 = (a_1 : 1.2); (a_3 : 0.1); (a_4 : 0.6)$
$y_3 = 1.0, \boldsymbol{x}_3 = \langle 0.5, 1.0, 0.0, 0.0\rangle$	$y_3 = 1.0, \boldsymbol{x}_3 = (a_1 : 0.5); (a_2 : 1.0)$
$y_4 = 0.2, \boldsymbol{x}_4 = \langle 1.2, 0.0, 2.0, 0.0\rangle$	$y_4 = 0.2, \boldsymbol{x}_4 = (a_1 : 1.2); (a_3 : 2.0)$

2.1.1 Dense and sparse data representation

GBDTs are trained using a set of instances (a.k.a. data points), and the set is called a training data set. We can represent the training data set in either a dense or a sparse form. The dense representation is basically a matrix, which is efficient for accessing the value of an attribute given an instance. For example, the third attribute of the fourth instance (i.e., a_3 of $\boldsymbol{x}_4$) can be easily retrieved at the third column of the fourth row in the matrix. However, the disadvantage is huge memory consumption for high dimension and sparse data sets. In comparison, the sparse representation stores only the non-zero elements, which is more memory efficient for data sets with many attribute values of zero, but more expensive to locate the attribute value of an instance. Suppose we have a training data set which has four instances: $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$ and $\boldsymbol{x}_4$ with their associated target values y_1, y_2, y_3 and y_4, respectively. We have a dense and sparse representation as shown in Table 1. Each attribute of an instance can be associated with a value. For example, a_3 of $\boldsymbol{x}_4$ is 2.0, where a_3 can be the "score" attribute of the "student" $\boldsymbol{x}_4$ and 2.0 is the value the "score" attribute takes.

In decision tree training, we need to enumerate all the possible split points of each attribute, such that we can split a node using the best split point. This approach is also called finding the exact split point for an attribute. To facilitate enumeration through all the split points, the matrix (e.g., Table 1) is transposed and the attribute values are stored in sorted order. This is a common and efficient approach used in training decision trees [3], [10]. The sorted results on each attribute of Table 1 are shown below.

$a_1 = (\boldsymbol{x}_2 : 1.2); (\boldsymbol{x}_4 : 1.2); (\boldsymbol{x}_3 : 0.5)$
$a_2 = (\boldsymbol{x}_3 : 1.0)$
$a_3 = (\boldsymbol{x}_4 : 2.0); (\boldsymbol{x}_2 : 0.1); (\boldsymbol{x}_1 : 0.1)$
$a_4 = (\boldsymbol{x}_2 : 0.6)$

We note that any missing instance value in the list is zero. The sorted results are useful when computing the quality (i.e., gain as defined in Section 2.1.3) of each possible split point, because we can easily obtain the number of instances on the left/right side of the split point under evaluation. For large data sets, the number of possible split points of each attribute is very large. To reduce the number of possible split points, histograms with a fixed number of bins are used in GBDT training. The key idea is that the domain of each attribute is divided into B parts and the attribute values in the same bin are considered as the same. Thus, the number of possible split points of each attribute is B.

2.1.2 Missing values

An additional advantage of sparse representation is that missing values of attributes are naturally supported. The missing values are treated as either $-\infty$ or $+\infty$ in GBDT training [3], which can be decided during learning. More specifically, the missing values may be put in the left child node (i.e., treated as $-\infty$) or right child node (i.e., treated as $+\infty$), depending on which way of putting the missing values results in better reduction of loss. In the dense presentation, missing values need to be filled (e.g., treated as 0) to allow sorting attribute values for efficiently finding the exact split points of nodes.

2.1.3 Loss function and gain of a split point

Training GBDTs is to reduce the value of a loss function denoted by $l(y_i, \hat{y}_i)$ where y_i and $\hat{y}_i$ are the true and predicted value of $\boldsymbol{x}_i$, respectively. The common loss functions include mean squared error and cross-entropy loss [11]. The first order and second order derivatives of the loss function are denoted by g_i and h_i which are computed as follows.

$$g_i = \frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i}, h_i = \frac{\partial^2 l(y_i, \hat{y}_i)}{\partial \hat{y}_i^2} \quad (1)$$

where g_i is also called gradient. The first order and second order derivatives are used to compute the quality, i.e., gain, of a split point using the following formula [3].

$$gain = \frac{1}{2}\Big[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R - \lambda}\Big] \quad (2)$$

where G_L and G_R denote the sum of g_i of all the instances in the left and right node, respectively; similarly, H_L and H_R denote the sum of h_i of all the instances in the left and right node, respectively; λ is the regularization constant.

2.1.4 Graphics Processing Units

A GPU contains a large number of (e.g., thousands of) cores which are grouped into streaming multiprocessors (SMs). In the NVIDIA *Compute Unified Device Architecture* (CUDA), GPU threads are grouped into blocks which are also called *thread blocks*. Each thread block is executed in an SM. At any timestamp, an SM can only execute instructions of one thread block. Compared with main memory, GPUs have relatively small global memory (e.g., 12 GB memory in Tesla P100). Accessing the GPU global memory is much more expensive than computation, so we should avoid accessing the GPU global memory as much as possible. Irregular accesses to global memory is even more expensive. The data transfer between CPUs and GPUs is through PCI-e which is one order of magnitude slower than accessing the GPU global memory. Therefore, we should make full use of the GPU memory to efficiently handle large data sets, and reduce data transferring between CPUs and GPUs. Previous studies have demonstrated inefficiency of GPUs in several data-intensive applications with irregular memory accesses [12], [13]. The recursive nature of GBDTs has posed new technical challenges. In this article, we propose methods to overcome the limitation of GPU memory and take advantage of GPU massive computing capability.

2.2 Related studies on decision trees and GBDTs

2.2.1 GPU accelerated decision tree prediction

In the studies of GPU accelerated decision trees, most of the work focuses on the decision tree prediction process. Sharp proposed to use GPUs for accelerating the decision forest prediction [14]. Sharp's key idea is to use a GPU thread to predict the target value of one instance in order to take advantage of the massive thread parallelism on the GPU. Similar to Sharp's algorithm, Birkbeck et al. presented a GPU-based algorithm for the decision tree prediction [15]. Their algorithm stores the decision tree in the texture memory of GPUs to improve efficiency. Van Essen et al. tried to find out which hardware (i.e., multi-core CPUs, GPUs and FPGAs) is the best for decision tree prediction [16], and their results show that FPGAs performs the best for prediction. Although the above proposed techniques can be used to accelerate the prediction module during GBDT training, our proposed approach is faster since the prediction can be totally avoided by reusing intermediate training results (cf. Section 3.2.1).

2.2.2 GPU accelerated decision tree training

Grahn et al. proposed to use a GPU thread to train one decision tree for the random forest training [17]. Thus, many decision trees can be trained in parallel, unlike the trees having dependency in GBDTs. Nasridinov et al. developed a GPU-based algorithm to compute the information gain when finding the best split point of a node [18]. Lo et al. [19] designed a GPU-based algorithm to train decision trees. Their key idea is to split one node at a time and sort the values of each attribute for all the instances in the node. One key limitation of the above discussed GPU-based algorithms for decision tree training is that the level of parallelism is low and the GPU can be severely underutilized. Strnad and Nerat [20] proposed a GPU-based algorithm with three levels of parallelism: evaluating multiple possible split points concurrently, finding the best split point for multiple attributes on a node concurrently, and finding the best attribute for multiple nodes concurrently. The bottleneck of their algorithm lies in launching too many kernels inside GPU kernels, and repeatedly sorting attribute values for every newly created node. For example, many small sorting operations degrade the GPU performance significantly. Most of the above-mentioned ideas for training decision

trees are implemented in the GPU version of XGBoost. However, the GPU version of XGBoost supports only dense data representation when finding exact split points. In contrast, our algorithm utilizes data compression techniques to train GBDTs more efficiently and to support larger data sets.

2.2.3 Gradient Boosting Decision Trees

Gradient Boost Machines (GBMs) were first introduced by Friedman [21], and have shown great potential in many real world applications [22], [23]. Panda et al. [10] proposed a MapReduce-based learning algorithm for decision trees that ensembles with approximation when finding split points for large data sets. Tyree et al. proposed parallel CPU boosting regression trees for webpage ranking problems [24]. Si et al. developed a GBDT training algorithm for high dimensional sparse output [25]. Chen and Guestrin proposed an efficient GBDT algorithm which is implemented in XGBoost [3]. Mitchell and Frank proposed to use GPUs to accelerate the finding split point procedure of XGBoost [26]. XGBoost with GPUs uses dense data representation for the ease of tracking back which attribute has the best gain, which makes it unable to handle large data sets due to the large memory consumption. LightGBM [6] is an alternative implementation of GBDTs, but it only supports finding approximate split points. XGBoost also supports approximation. The key difference of the approximation between XGBoost and LightGBM is that XGBoost adapts the breadth first node splitting and LightGBM uses the depth first node splitting. CatBoost [7] is the latest implementation of GBDTs and also only supports finding approximate split points. CatBoost trains much simpler trees which are called oblivious decision trees where a level of the tree has the same split point [27]. Table 2 summarizes the key differences between our ThunderGBM algorithm with the other popular GBDT implementations on GPUs.

2.3 The GBDT training and the existing libraries

The GBDT training consists of two key components: (i) finding a split point to a node and (ii) splitting a node.

2.3.1 The existing GBDT libraries

The most popular library for GBDT training is XGBoost [3]. Here, we first describe both the CPU implementation and GPU implementation of GBDTs in XGBoost. Then, we present LightGBM [6] and CatBoost [7].

The parallel XGBoost on CPUs: The key idea of parallelism in XGBoost is to find the best split points for multiple attributes of multiple nodes concurrently (i.e., Line 5 and 7 of Algorithm 1). In other words, XGBoost uses attribute level and node level parallelism. Parallelizing these two levels results in more than enough threads to occupy the CPUs.

The parallel XGBoost on GPUs: Similarly to the CPU version, XGBoost on GPUs also uses attribute level and node level parallelism. For attribute level parallelism (i.e., Line 7 of Algorithm 1), a GPU thread block is dedicated to compute the best split point of an attribute. For node level parallelism (i.e., Line 5), the algorithm uses a so-called "node interleaving" techniques which requires reserving many copies of memory for g_i and h_i of instance $\boldsymbol{x}_i$ (the number of copies equals to the number of nodes to split). Moreover, for the ease of tracking back which attribute the best split point belongs to, they use the dense data representation for the training data set. Therefore, the XGBoost on GPUs requires too much GPU memory and cannot handle large data sets. That motivates us to carefully examine the algorithm, and develop GPU-efficient parallelization as well as memory access patterns (as described in the next section).

LightGBM and CatBoost have similar parallelism principles as XGBoost. The key difference is that LightGBM eliminates some instances with small g_i and combines attributes that are correlated, and CatBoost trains the so-called oblivious trees where the whole level of a tree has an identical split point.

3 Our ThunderGBM algorithm

3.1 Challenges and design rationale

3.1.1 Challenges

The key challenges of designing ThunderGBM are in four aspects. First, the memory access pattern is irregular due to the nature of tree structures. The irregular memory accesses can significantly degrade the efficiency of GPU-based algorithms. Second, the values of the attributes of every node need to be sorted to facilitate the enumeration of possible split points, and the number of sorting operations with small inputs may be huge. Performing a large number of sorting operations is expensive on GPUs especially for a large number of small segments (each attribute of a node is stored as a segment). Third, the data parallel granularity changes as the tree grows. At the early stages, the nodes are large which contain many training instances (e.g., the root node contains all the training instances); at the later stages, the nodes become smaller but the number of nodes is large. This is challenging because the massive thread parallelism of the GPU needs to adapt to different parallel granularities. Fourth, the same attribute value appears in many instances which causes the same split point having different gains when the gains are computed in parallel. Removing duplicated split points is expensive, because we need to access the neighboring elements which requires extensive memory accesses.

3.1.2 Key design rationale

In order to better take advantage of GPU accelerations, we have the following design rationales. To begin with, according to the massive thread parallelism of GPUs, we develop fine-grained multi-level parallelism for GBDTs. In addition to the node level and attribute level parallelism, we propose fine-grain parallelism by parallelizing the gain computation of each split point. Due to the GPU memory limitation, we look for more memory efficient representation than the dense and sparse representations. Particularly, we exploit Run Length Encoding (RLE) compression in ThunderGBM, since RLE compression is able to (i) reduce memory consumption, (ii) improve the efficiency of finding the best split point due to the avoidance of repeated attribute values, (iii) retain efficiency in splitting nodes without a total decompression. RLE compression is particularly effective for our algorithm (especially for handling data sets with high compression ratio), because it helps reduce PCI-e traffic.

TABLE 2
Comparision between ThunderGBM and existing GPU implementations of GBDTs. XGBoost (GPU) has the option of finding exact split points, but it either runs out of memory or produces extremely large RMSE. Hence, we consider XGBoost (GPU) does not support finding exact split points.

GBDT GPU implementation	sparsity aware	multi-GPU support	find exact split points	find approximate split points with histograms	use data compression	train regular decision tree
XGBoost (GPU) [3]	✗	✓	✗	✓	✗	✓
LightGBM (GPU) [6]	✓	✗	✗	✓	✗	✓
CatBoost (GPU) [7]	✗	✓	✗	✓	✗	✗
ThunderGBM	✓	✓	✓	✓	✓	✓

Based on the fine-grained multi-level parallelism and compression, we further address all the technical challenges.

- To reduce the irregular memory access, we propose to reuse the intermediate training results to compute gradients and avoid traversing the trees. To avoid the same split point having different gains, we exploit the RLE compression and develop novel techniques to split an RLE element. We also design a memory friendly and fast attribute ID look up technique in node splitting.
- To keep the attribute values sorted in a node for ease of locating the best split points, we propose to use the order preserving partitioning, powered by techniques to control memory consumption. The order preserving partitioning is further enhanced by a stable sorting based method.
- To further improve the efficiency of finding the best split points, we extend ThunderGBM to support finding approximate split points using histograms and propose a series of novel techniques. For example, to reduce the shared memory consumption, we propose sparsity aware techniques to build histograms faster and more memory friendly. When building the histograms, we first build a partial histogram on GPU thread block-level, and then build the global histogram by aggregating the partial histograms. Through the exploitation of histogram subtraction, ThunderGBM can reduce the workload in building histograms by half.
- To handle the changing number of nodes and the increasing number of segments (the number of segments equals to the number of attributes times the number of nodes), we develop techniques to dynamically allocate the number of segments that each GPU thread block handles.
- Finally, we also enable the support for multiple GPUs in order to handle larger data sets which cannot be entirely stored in one GPU. Through the attribute based training data partitioning, ThunderGBM avoids exchanging partial histograms among GPUs, and hence significantly reduces the communication cost.

Our major contributions in this extension are in four aspects: (i) proposing data partitioning based on stable sort (cf. Section 3.2.2), and fast and memory efficient attribute ID lookup in node splitting (cf. Section 3.5), (ii) developing sparsity aware techniques to support finding approximate split points and build histograms efficiently (cf. Section 3.4), (iii) using two-stage histogram building to exploit shared memory and histogram subtraction to reduce histogram building workload, and (iv) designing mechanisms to support multiple GPUs in order to handle larger data sets which cannot be entirely stored in one GPU (cf. Section 4.1).

3.2 Training GBDTs using sparse representation

Here, we first provide the technical details of finding the exact split points in ThunderGBM when the training data is represented in sparse formate. Then, we explain the techniques of splitting a node and how to keep the attribute values in the new nodes sorted.

3.2.1 Finding the exact split point for a node

Finding the exact split point for a node is to find the split point with the maximum loss reduction. All the possible split points are enumerated based on the training data set. There are three steps in finding the exact split point for a node: (i) compute the gain for each possible split point, (ii) reset the gain of repeated split points to 0, and (iii) select the best split point (i.e., the split point with the maximum gain).

(i) Compute the gain of a split point: As discussed in Section 2.1.3, we need to compute g_i and h_i for computing the gain of each possible split point (cf. Equation 2). Although ThunderGBM supports user defined loss functions, for ease of presentation we use the mean squared error as the loss function[1]. Then, $g_i = 2(\hat{y}_i - y_i)$ and $h_i = 2$. As computing g_i and h_i requires the predicted value (i.e., $\hat{y}_i$) for each training instance, a naive approach is that we first use the trained decision trees to perform prediction and then compute g_i and h_i using Equation 1. This naive approach results in a large number of irregular memory accesses due to tree traversal. Next, we present optimizations to avoid the irregular memory accessess.

Computing g_i using intermediate training results: Before we present our optimization in computing the predicted values, we first discuss a simple optimization. The quick and simple optimization is that each time we need to compute g_i and h_i, we only predict the target value using the latest trained tree and reuse the predicted target value of the previous trees (i.e., predict a target value incrementally). This is because the predicted target value is the accumulated result of all the previous trees. However, traversing a tree on GPUs is very expensive while predicting the target values. This is because the tree traversal results in thread branch divergence and irregular memory access. Recall that during the training, the training instances are partitioned into new nodes. At the end of training a tree, all the training instances are in leaf nodes. Hence, we avoid traversing the tree to decide which leaf node an instance belongs to, and perform prediction by

1. ThunderGBM can support other loss functions by customizing the computing of g_i and h_i, and nothing else needs to be changed.

obtaining the weight of the leaf node where the instance belongs.

After we have obtained g_i and h_i, we can compute G_L, G_R, H_L and H_R. Because the values of each attribute are sorted as discussed in Section 2.1.1, we can consider all the instances on the left (right) part of the possible split point go to the left (right) node. Then, we can obtain the aggregated g_i and h_i of the left and right nodes (e.g., G_L and G_R) for computing the gain shown in Equation 2 relatively easily as follows. Computing G_L and H_L can be done by segmented prefix sum which is available in CUDA Thrust [28].

G_L of the i^{th} possible split point is the i^{th} element of the prefix sum result; G_R equals to $(G - G_L)$ where G is the total gradients of the node to split. The gains of all the possible split points are computed in parallel on GPUs. The instances with missing values on that attribute either go to the left or right child node, depending on which way results in a larger gain.

(ii) Reset gain of repeated split points: We need to compute the gains of all the possible split points of an attribute (e.g., a_1 in Section 2.1.1) in parallel. However, some split points may be repeated in the attribute (e.g., $a_1 = 1.2$ in Section 2.1.1). The split points with the same value next to each other may have different gains. The different gains are due to different values of G_L, G_R, H_L and H_R (cf. Equation 2) computed from the segmented prefix sum.

The interpretation of the different gains is that instances of equal attribute values to the split point can go to the left child node and the right child node. In reality an instance should belong to only one node (either left or right child node). To avoid the same split point having different gains, we set the gains after the first value to 0, i.e., forcing all the instances with the same attribute values going to only one child node.

(iii) Select the best split point for each node: After we have obtained the gain of all the possible split points, we first use the segmented reduction to obtain the best split point for each attribute of a node. Then, we use the GPU parallel reduction [29] to get the best split point for each node. When using segmented reduction, each segment needs to have its own key to distinguish one segment from another. A naive method to set key for each segment is using one block per segment. However, the granularity of parallelism varies as the tree grows. Specifically, the number of segments is increasing as the tree grows, and some data sets may have a large number of segments (due to high dimensionality and the large number of tree nodes). Using one block per segment results in low efficiency, due to the overhead of scheduling and launching a large number of GPU thread blocks.

We propose techniques to automatically decide how many segments a block should process depending on the data set. The simple and effective formula we use is: $1 + \frac{\text{\# of segments}}{(\text{\# of SM}) \times C}$ where C is a user defined constant and we set it to 1000 (i.e., one SM—GPU Stream Multi-processor—executes 1000 blocks). The basic idea of the formula is that we set the number of blocks created to handle the segments to a fixed number, such that the number of blocks does not explode when the number of segments is large. Although the formula is simple, it brings 10% to 20% performance improvement for some data sets [9].

3.2.2 Splitting a node

After we have found the best split point, we split the node using that split point. Splitting a node is essentially dividing the training instances in the node into two groups: one group to be relocated in the left child and the other group to be relocated in the right child. During splitting, an important task is to partition the training instances that belonging to the current node into two child nodes. For partitioning, we can use the sorted values on that attribute to directly partition the training instances, and we will present more details in Section 3.5. The most challenging task in splitting the node is to maintain values of each attribute in the new nodes in sorted order, for efficiently finding the exact split points. Here, we propose to extend the histogram based method [30] for order preserving partitioning, and further develop data partitioning using stable sort.

Histogram based data partitioning: In the histogram based data partitioning, suppose we want to partition the data into k partitions (i.e., creating k new nodes of a tree); each thread handles b elements and requires maintaining k counters (a counter for each partition). Based on the counters of each thread, we can build histograms and determine where an element should go to in the new partitions. So the total number of counters is: (# of threads) $\times$ (# of partitions). A naive approach is to set the workload of a thread to a constant (e.g., $b = 16$), but such an approach suffers from the uncontrollable amount of memory consumption and runs out of GPU memory for large data sets, because of the large number of counters. To control the memory consumption by the counters, we need to limit the number of threads. To address the limitation of the existing approach [30], we propose techniques to automatically decide the number of threads used in partitioning a node under the memory constraint. The formulas for computing the thread workload and the number of threads are shown below.

$$\text{thread workload} = \frac{(\text{\# of attribute values}) \times (\text{\# of nodes})}{\text{Maximum allowed memory size}}$$

$$\text{\# of threads} = \frac{\text{\# of attribute values}}{\text{thread workload}}$$

The basic idea is that we allocate more workload to a thread when the number of partitions (i.e., # of attribute values $\times$ # of nodes) is large, such that we avoid using a large number of counters and running out of GPU memory. The maximum allowed memory size is a user predefined value (e.g., 2^{30} for 2GB). This histogram based data partitioning is implemented in our previous version of ThunderGBM [9]. Next, we discuss the sorting based approach for data partitioning which is more efficient in terms of shared memory consumption.

Data partitioning using stable sort: Our key goal in this data partitioning is to preserve the sorted order for the attribute values. This goal can be accomplished by a stable sort operation. More specifically, we assign a key to each attribute value, where the key is computed based on the attribute ID and node ID of the attribute value. In ThunderGBM, we store the attribute ID in the higher bits and the node ID in the lower bits of the key. The underlying idea is that the attribute ID serves as the primary key, and the node ID serves as the secondary key. As a result, the sorting algorithm orders the attribute values first by their attribute ids and then by the node ids if the attribute ids are equal. The sorting algorithm we use in ThunderGBM is radix sort provided in NVIDIA CUB [31]. Using stable

sort instead of histogram based data partitioning reduces the shared memory consumption, because of the avoidance of maintaining the counters for each partition.

3.3 GBDT training with Run-length Encoding

Here, we present the Run-length Encoding (RLE) compression for storing the training data more memory efficiently. We also describe the property of RLE compression in GBDTs, where the same split point with different gains issue is avoided and the gain computation cost is reduced. The key intuition of RLE is that the instances with the same attribute values in a node are combined into one. Thus, the memory consumption can be improved and we can also avoid evaluating the gain of duplicated split points.

We have observed that there are many repeated values in each sorted attribute, and the repeated values can be compressed using Run-length Encoding (RLE) [32]. Given a sequence of values $1.2, 1.2, 1.2, 3.4, 3.4, 3.4, 3.4$, RLE represents the sequence using value-and-length pairs: $(1.2, 3), (3.4, 4)$.

This compression has the following two advantages. (i) Reduce memory consumption: some data sets which originally cannot fit into the GPU memory now can be stored in the GPU memory; the memory traffic for transferring the training data set through PCI-e is reduced. (ii) Improve the efficiency of finding the best split point: the same split point with different gains issue is naturally avoided and the number of split points to compute gains is reduced. Moreover, as we will see later in this section, we retain fast execution time in splitting nodes without the requirement of a total decompression.

Since the GBDT with RLE compression is similar to the GBDT with sparse representation (discussed in Section 3.2) in selecting the best split, the major difference is how to compute gain of split points using RLE. Computing the gain for each possible split point requires computing g_i and h_i for each instance in the node. In the sparse data representation, each possible split point corresponds to an attribute value of one instance; in data representation with RLE compression, each possible split point corresponds to a few instances with the same attribute value. Here, we denote the first order and second order derivatives for an RLE element by g'_i and h'_i, respectively. Then, g'_i and h'_i for the split point of RLE are the sum of the first order and second order derivatives, respectively. To calculate the first order derivative g'_i (resp. the second order derivative h'_i) for each RLE element is to compute the sum of g_i (resp. the sum of h_i) of each instance in the node.

3.4 Finding best split points using histograms

The number of possible split points may be very large for data sets with a large number of instances. The cost of finding the exact split points is high for such problems. Here, we elaborate the techniques of finding approximate split points using histograms, and the details of two-stage histogram building and building histograms by subtraction. Then, we present building histograms for high dimensional data with lock-free techniques. The key idea of finding approximate split points is that we only consider a fixed number of possible split points instead of enumerating all the possible split points from the training data.

Existing studies [3], [6], [7] use histograms with a fixed number of bins to tackle the problems of having too many possible split points. Then the number of possible split points of an attribute equals to the number of bins. The key idea is that the domain of the values of an attribute is "evenly" divided into B parts, where "evenly" means that each part has the similar number of attribute values. After the domain is divided into B parts, attribute values are mapped to their corresponding bin ids. Dividing the values into B parts is a well-studied problem in the database community. We suggest the interested readers to read the work [33] for more details. In ThunderGBM, we also use histograms to address the problems of having a large number of possible split points.

In ThunderGBM, we build multiple histograms concurrently for a node, where the number of histograms in a node equals to the number of attributes. More formally, suppose there are N nodes needed to be split, and the total number of attributes is A. Then, the total number of histograms we need to build is $(N \times A)$. In ThunderGBM, the histograms are built in two stages. In the first stage, each GPU block builds a partial histogram using shared memory, and then in the second stage the partial histograms on different GPU blocks are aggregated in the global memory to form the final histogram for an attribute of a node.

Another novel technique we make use of is building histograms for child nodes by subtraction. More specifically, when we build the histograms for two sibling nodes n_1 and n_2, we first build the histograms for one of the two sibling nodes. Without loss of generality, suppose we decide to build the histograms for node n_1. Then, we subtract the histograms of their parent by those of node n_1 to obtain the histograms of n_2. Thus, we reduce the histogram building workload by half with subtraction. Formally, we need to build $(2 \times A)$ histograms in total for n_1 and n_2. With subtraction, we only need to build A histograms. Moreover, when selecting which sibling node to build the histograms, we select the node with fewer attribute values. This is because building histograms for the node with few attribute values has lighter workload.

3.4.1 Building histograms for high dimensional data

As an attribute of a node corresponds to a histogram, the number of histograms may be extremely large for high dimensional data. For example, a public data set called *log1p* has more than four million attributes. Training trees for such data sets requires constructing billions of histograms (i.e., $N \times 4\ million$), which leads to running out of shared memory and performing a large number of locking operations. To tackle the problems of building a large number of histograms efficiently, we propose the sparsity aware histogram building technique. We consider each distinct attribute value as a cut point of the histogram for high dimensional data. Then, the same attribute values are stored in the same bin of the histogram. Thus, we can perform a sorting of the attribute values, and then we perform a reduction operation on GPUs to accumulate the same attribute values into the bins of the histograms. By doing so, we can build a large number of histograms without

requiring much shared memory. An intriguing property of this approach is that the process of histogram building is lock-free.

3.5 Applying the best split points to the nodes

After obtaining the best split point for a node, we split the node into two child nodes. This node splitting requires assigning the instances of the parent node to two child nodes. Assigning the instances to the child nodes is non-trivial, because the training data is stored in an attribute oriented manner. We cannot directly access to the attribute values of an instance without scanning the whole training data set. In this work, we use the associated instance ID of each attribute value to assign the instances to the child nodes. With the attribute values of an instance and the best split point of the parent node, we can figure out which child node the instance belongs to.

Another technical challenge is how to obtain the attribute ID of an attribute value. There are three ways to help find the attribute ID of an attribute value. The first way is to store an attribute ID for each attribute value, which requires a large amount of GPU memory. The second way is to perform a binary search to find the attribute ID of a given attribute value, which requires both computation and irregular memory access. It is worthy to note that all the attribute values are stored in a one-dimensional array together with the start position and length of each attribute, i.e., the Compressed Sparse Column (CSC) format. Hence, it is possible to perform binary search to locate the attribute ID of an attribute value in the array. The third way is to associate the attribute ID with the GPU thread block ID which avoids extra memory consumption and irregular memory access for the attribute ID lookup. In ThunderGBM, we use the third way to obtain the attribute ids.

4 Enhancement of ThunderGBM

In this section, we present some enhancement of ThunderGBM including supporting multiple GPUs and the prediction algorithm for completeness.

4.1 Training GBDTs on multiple GPUs

One of the key limitations of GPUs is that the global memory size is relatively small (e.g., 12GB) compared with the size of main memory. In this work, we propose to partition the training data by attributes to handle large data sets (i.e., column based partitioning). This fashion of data partitioning is different from the existing libraries, such as XGBoost and CatBoost, which partition the training data by instances (i.e., row based partitioning). Note that LightGBM does not support multiple GPUs.

There are two advantages of the attribute based partitioning. First, both finding exact split points and finding approximate split points by histograms are natively supported. This is because finding the split points of an attribute requires accessing all the attribute values. Storing all the attribute values of an attribute in one GPU helps perform finding the split points more communication efficiently. In comparison, XGBoost and CatBoost do not support multiple GPUs for finding the exact split points, because the request of accessing all the attribute values when finding a split point requires too much communication among GPUs. Therefore, XGBoost and CatBoost only support finding approximate split points by histograms, which has relatively lower communication cost compared with finding the exact split points.

The second advantage is that the GPUs do not need to exchange the partial histograms in order to find the approximate splits, since all the attribute values of an attribute are stored locally and the GPU has the whole histogram. Hence, the GPUs only need to exchange the local best split points in order to obtain the global best split points for the tree nodes. This reduces the communication cost from $\mathcal{O}(N \times A \times B)$ to $\mathcal{O}(N \times A)$, where N is the number of nodes needed to split, A is the number of attributes in the training data set and B is the number of bins of the histograms. The intuition is that a histogram is replaced by a local best split point when communicating to other GPUs.

4.2 ThunderGBM prediction algorithm

In GBDT training, training the next tree is based on the results of the previous trees. Hence, the prediction algorithm is a part of the GBDT training algorithm. Although this prediction algorithm can be used for other purposes (e.g., predict target values for unseen instances), we discuss it here for the completeness of our GBDT training algorithm. We need to predict the target values in order to compute derivatives (e.g., g_i) for training a new tree (i.e., splitting nodes). To perform prediction in parallel, we do both instance level and tree level parallelism (i.e., one GPU thread predicts the partial target value of an instance using one tree), since all the instances are independent and all the trees can be traversed independently. The prediction algorithm repeats the following two steps until a leaf node is reached: (i) examine the decision making condition (i.e., the information of the split point) of the current node for an instance, and (ii) go to the left (resp. right) child if the condition is true (resp. false). Other optimizations [34] for decision tree prediction can be applied to ThunderGBM, but they are out of the scope of this article, because prediction can be totally avoided in the GBDT training as we have discussed earlier.

4.3 Summary of computation steps and communication in the GBDT training

In the GBDT training, there are two key computation steps: (i) finding a split point for a node and (ii) splitting a node. In the following, we relate the challenges mentioned in Section 3.1 to the two key computation steps. The irregular memory access issue occurs in both of the steps, and the large number of sorting operations occurs in the first step. The varying data parallel granularity challenge appears in both steps. The same split point having different gain issue occurs in the first step.

There are two major communications during the GBDT training: (i) obtaining the global best split point and (ii) splitting the training instances in one node into two groups. Obtaining the global best split point is simple and is just aggregating the local best split points from different GPUs. In comparison, assigning a training instance to a new node is challenging, i.e., updating the mapping between node

TABLE 3
Information of data sets used in the experiments

data set	card.	dim.	data set	card.	dim.
covetype	581012	54	log1p	16087	4272228
e2006	16087	150361	news20	19954	1355191
higgs	1.1x10^7	28	real-sim	72201	20958
ins	13184290	35	susy	5x10^6	18

IDs and instance IDs. This is because the information of an instance is stored in multiple GPUs. Therefore, in our algorithm, each GPU constructs a partial mapping between node IDs and instance IDs, and then the GPUs need to exchange the partial mapping information between node IDs and instance IDs, in order to correctly assign a training instance to a new node.

5 Experimental Studies

Experimental setup. We used 8 publicly available data sets as shown in Table 3. The data sets were downloaded from the LibSVM website[2]. The data sets cover a wide range of the cardinality and dimensionality. The experiments were conducted on a workstation running Linux with 2 Xeon E5-2640v4 10 core CPUs, 256GB main memory and two Pascal P100 GPUs of 12GB memory. Each program was compiled with the -O3 option. ThunderGBM was implemented in CUDA-C. The default tree depth is 6 and the number of trees is 40. The total time measured in all the experiments includes the time of data transfer via PCI-e bus.

Comparison. We compare ThunderGBM with well-known GBDT libraries, namely XGBoost [3], LightGBM [6] and CatBoost [7]. The version number of XGBoost, LightGBM and CatBoost on GitHub is 85939c6, 2323cb3 and 503b3b8, respectively. The libraries support both CPUs and GPUs, hence we compare ThunderGBM with both versions of the libraries. Although ThunderGBM supports other loss functions, the loss function in our experiments for all the libraries (including ThunderGBM) is the mean squared error: $l(y_i, \hat{y}_i) = \sum_i (y_i - \hat{y}_i)^2$.

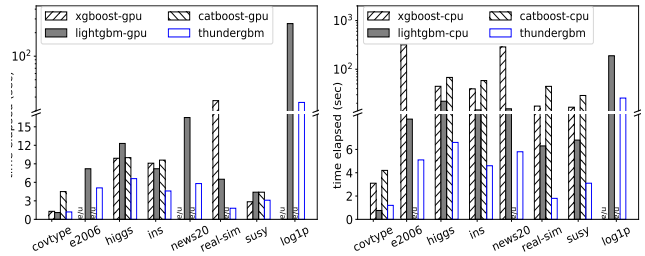
5.1 Overall performance study

This set of experiments aims to study the improvement of execution time of ThunderGMB over the existing libraries XGBoost, LightGBM and CatBoost. We first present the improvement of ThunderGBM over the three libraries on the GPU, and then we present the improvement of ThunderGBM over the libraries on CPUs. We show that ThunderGBM running on the GPU significantly outperforms the three existing libraries on the GPU or the CPU. After that, we study the efficiency improvement of ThunderGBM over our previous implementation [9]. Finally, we compare the Root Mean Squared Error (RMSE) of the libraries against ThunderGBM to study the quality of the trained models.

5.1.1 Execution time comparison on the GPU

We measured the total time (including data transfer from main memory to GPUs via PCI-e bus) of training all the trees for ThunderGBM, XGBoost, LightGBM and CatBoost. During training, the split points are found using the histogram based method. We study the execution time of finding the exact split points in Section 5.1.3, as the functionality is only supported by ThunderGBM and XGBoost.

2. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/



(a) Existing libraries on the GPU (b) Existing libraries on CPUs

Fig. 2. Comparison with existing libraries on the GPU and CPUs

The results of the four GPU implementation of GBDTs are shown in Figure 2a. The first observation is that our ThunderGBM algorithm can handle all the data sets efficiently, and outperforms all the existing libraries. In comparison, XGBoost and CatBoost cannot handle high dimensional data sets such as *e2006*, *news20* and *log1p* (marked with "n/a"). This is because the GPU versions of XGBoost and CatBoost do not make use of data sparsity, which leads to running out of GPU memory. Moreover, XGBoost took 27 seconds to handle *real-sim* which CatBoost cannot handle. ThunderGBM can handle *real-sim* 15 and 3.6 times faster than XGBoost and LightGBM, respectively. The GPU version of LightGBM is more reliable than XGBoost and CatBoost and can handle all the data sets. However, ThunderGBM can outperform LightGBM by up to 10 times on the data sets tested (e.g., *log1p*).

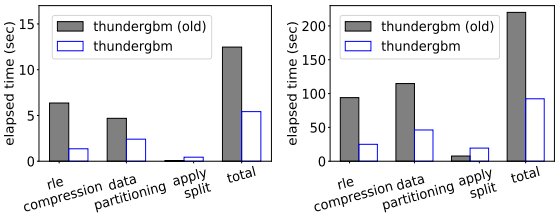
5.1.2 Execution time comparison on CPUs

We study the speedup of ThunderGBM on the GPU over XGBoost, LightGBM and CatBoost on CPUs. Note that the number of CPU threads (i.e., 40 threads) in XGBoost is automatically selected by the XGBoost library. We have also tried XGBoost with 10, 20, 40 and 80 threads, and found that using 40 threads results in the shortest execution time for XGBoost in the 8 data sets. Similarly for LightGBM and CatBoost, the number of threads is chosen by the libraries.

The results of the three libraries on CPUs are shown in Figure 2, in comparison with ThunderGBM running on the GPU. Among the three libraries, LightGBM is more reliable compared with XGBoost and CatBoost. XGBoost runs out of memory on the *log1p* data set, while CatBoost runs out of memory on *e2006* and *news20* besides *log1p*. ThunderGBM is often several times faster than LightGBM. For *e2006*, ThunderGBM achieves 60 times speedup over XGBoost.

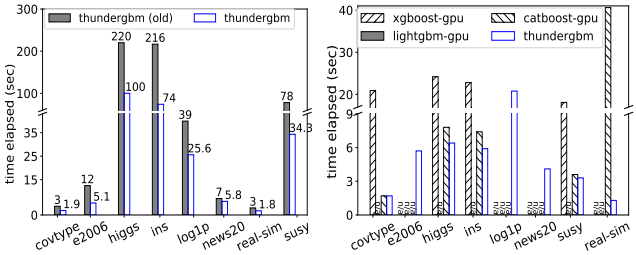
5.1.3 Comparison with our previous implementation [9]

Our previous implementation of GBDT training on GPUs only supports finding exact split points, so we compare it against our new implementation with finding exact split points. As we can see from Figure 4a, our new implementation denoted by "thundergbm" consistently outperforms our previous implementation [9] denoted by "thundergbm (old)". For example, the speedup of our new algorithm is 3.2 and 2.5 times over our previous implementation on *ins* and



(a) breakdown on *e2006* (b) breakdown on *higgs*

Fig. 3. Elapsed time breakdown on *e2006* and *higgs*



(a) Our old version [9] (b) Existing librairies on 2 GPUs

Fig. 4. Comparison with old version and existing libraries on 2 GPUs

susy, respectively. The improvement over our previous implementation mainly lies in (i) the use of stable sort instead of data partitioning to reduce shared memory consumption (cf. Section 3.2.2), and (ii) fast and memory efficient attribute ID look up in node splitting (cf. Section 3.5).

To further investigate why the new version of ThunderGBM is faster than the previous version. We use *nvprof* to obtain the detailed measurements for the two versions. The results are shown in Table 4. Overall, our new algorithm increases GPU utilization, GFLOPs and shared memory throughput; it decreases the PCI-e transactions and read operations to the caches and global memory. Figure 4 shows that the effect of our optimizations on different key components. The elapsed time for RLE compression and data partitioning is significantly reduced, which leads to the final improvement of the total elapsed time.

5.1.4 RMSE comparison

We have compared the trees constructed by ThunderGBM and XGBoost, and found that the trees are identical. Here, we show the Root Mean Squared Error (RMSE) on the training data sets for the trees trained by ThunderGBM, XGBoost, LightGBM and CatBoost. Due to the space limitation, we summarize the key results here. ThunderGBM produces the same RMSE as XGBoost. The RMSE of CatBoost is always higher than ThunderGBM on the data sets. This is because CatBoost uses the simplified version of decision trees called "oblivious decision trees" where a level of the tree has the same split point [27]. When comparing with LightGBM, ThunderGBM has comparable RMSE. In summary, the quality of the trees trained by ThunderGBM is the same or similar to XGBoost and LightGBM, and is better than CatBoost. This indicates ThunderGBM can train the trees much faster than the three libraries while having better or similar tree quality.

5.2 Scalability studies on multiple GPUs

We study the scalability of different GPU based GBDT implementations on multiple GPU environments. Figure 4b shows the results of the four implementations on two Pascal P100 GPUs. LightGBM does not support multiple GPUs, and the elapsed time is identical to Figure 2a on a single GPU. We do not duplicate the results of LightGBM here for ease of inspecting the other three implementations.

The training time of XGBoost on two GPUs (cf. Figure 4b) is even higher than that on single GPU (cf. Figure 2a). For example, XGBoost only needs less than 2 seconds to handle the *covtype* data set using one GPU, while it needs more than 20 seconds to handle the same data set using two GPUs. Similarly, using two GPUs does not help CatBoost much either, and it again fails to handle the data sets (e.g., *e2006*, *log1p* and *news20*) that it cannot handle using one GPU. As the other libraries do not support multiple GPUs well, we study the scalability of ThunderGBM only next.

5.2.1 Scalability of ThunderGBM on multiple GPUs

For investigating the scalability of ThunderGBM over multiple GPUs, we constructed larger data sets using the *higgs* data set. The original data set has about 300 million attribute values. We constructed five bigger data sets of 600, 900, 1200, 15000, and 18000 million attribute values, respectively, through attribute value duplication. We used four Tesla K80 GPUs in the experiments.

Figure 5a shows the results. ThunderGBM on one GPU can handle data sets of size up to 600 million attribute values, and that on two GPUs can handle data sets of size up to 1500 million attribute values. With three or four GPUs, ThunderGBM can handle all the six data sets. Therefore, ThunderGBM can handle very large data sets with more GPUs. Another observation from the figure is that ThunderGBM on more GPUs is often faster than that on fewer GPUs, especially for large data sets.

5.3 Sensitivity studies

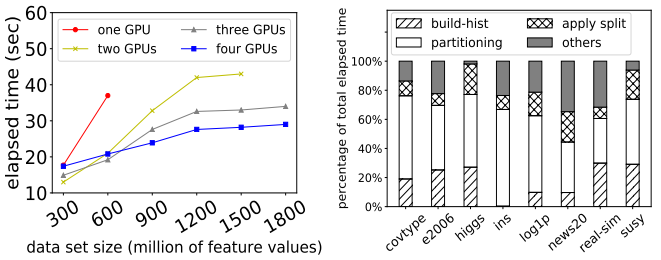
We perform sensitivity studies on two key parameters of GBDTs: tree depth and the number of trees. Due to the space limitation, here we briefly summarize the results. We vary the tree depth from 4 to 10 and the number of trees from 10 to 80. These sensitivity studies show that the speedup of ThunderGBM over other libraries is stable. The elapsed time of ThunderGBM on the GBDT training increases as the tree depth or the number of trees increases.

5.4 Elapsed time of the key components

Here we study the elapsed time of three key components: (i) building histograms, (ii) partitioning data to child nodes and (iii) applying the best splits. The elapsed time of the other components is denoted by "others". As we can see from Figure 5b, these three key components account for more than 65% of the total elapsed time. In the *higgs* data set, these three components even account for 98%. A further observation is that partitioning data to new nodes is the dominating component in the whole training. Hence, for the future improvement of ThunderGBM, the key challenge is to improve the efficiency of partitioning data to new nodes.

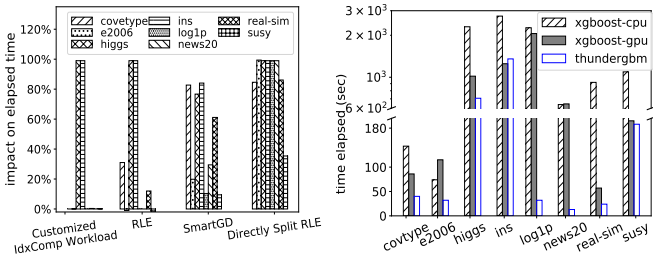
TABLE 4
Detailed comparison with the old version of ThunderGBM [9]

e2006	GFLOPs	Global memory read transactions	L2 read transactions	L1 read transaction	Shared memory load throughput (GB/s)	PCI-e transactions		GPU utilization
						Count	Total size (MB)	
thundergbm (old)	19.21	$8.8x10^9$	$6.5x10^{10}$	$1.2x10^{11}$	164	5928	119	0.81
thundergbm	90.24	$2.3x10^{10}$	$3.7x10^{10}$	$7.5x10^{10}$	230	1885	161	0.84
higgs								
thundergbm (old)	4.03	$2.2x10^{14}$	$7.5x10^{11}$	$1.0x10^{12}$	155	5928	1267	0.77
thundergbm	8.94	$1.0x10^{10}$	$1.7x10^{10}$	$3.8x10^{10}$	250	2211	4694	0.85



(a) Varying the number of GPUs (b) Key component cost

Fig. 5. Varying GPUs and key component cost



(a) Impact of the optimizations (b) Prediction time comparison

Fig. 6. Individual optimization and prediction efficiency

5.5 Impact of individual optimizations

As we have discussed in Section 3, we have some optimizations specifically for our GPU algorithm. Here, we study their individual impacts on the overall efficiency. The techniques include (i) Customized IdxComp Workload: to decide thread workload depending on datasets discussed in Section 3.2; (ii) RLE: to compress datasets with RLE compression discussed in Section 3.3; (iii) SmartGD: to compute g_i and h_i using the intermediate training results discussed in Section 3.2; (iv) Directly Split RLE: to directly split RLE elements as discussed in Section 3.3. We switched off each individual optimization, and investigate the execution time change to the entire algorithm. Figure 6a shows the change in the execution time of disabling each optimization. Two techniques (including SmartGD and Directly Split RLE) have quite significant impact on the overall algorithm. This demonstrates the important of our SmartGD and Directly Split RLE techniques. The Customized IdxComp Workload has significant improvement on execution time for large datasets, this is because more workload a thread does, more memory the algorithm saves.

5.6 Prediction execution time comparison

For completeness, we also evaluated the execution time of prediction of our algorithm in comparison with XGBoost. Figure 6b provides the results of prediction time of ThunderGBM, XGBoost running on CPUs and XGBoost running on the GPU. Our algorithm is often much faster than the XGBoost implementations. The trees trained by ThunderGBM and XGBoost are the same. Therefore, the predictive accuracy and generalization capability are the same and does not reveal any new information and hence the results are omitted here.

6 Conclusion and Future Work

GPU accelerations have become a hot research topic for improving the efficiency of machine learning and data mining algorithms. This article presents a novel parallel implementation named ThunderGBM for training GBDTs, which have become very popular in recent years and won many awards in machine learning and data mining competitions. Although GPUs have much higher computational power and memory bandwidth than CPUs, it is a non-trivial task to fully exploit GPUs for training GBDTs. We have addressed a series of technical challenges in training GBDTs on GPUs, including irregular memory access and order reserving node partitioning. Our experimental results show that ThunderGBM can be 10x times faster than the state-of-the-art libraries (i.e., XGBoost, LightGBM and CatBoost) running on a relatively high-end workstation with 20 CPU cores. ThunderGBM can handle high dimensional data sets which existing libraries on GPUs fail to handle, and achieves up to 10 times speedup over the libraries on GPUs for the data sets the libraries can handle.

Acknowledgements

This work is supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore. Prof. Chen is supported by the Guangdong special branch plans young talent with scientific and technological innovation (No. 2016TQ03X445), and the Guangzhou science and technology planning project (No. 201904010197). Bingsheng He and Jian Chen are corresponding authors.

References

[1] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, "ThunderSVM: a fast SVM library on GPUs and CPUs," *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 797–801, 2018.

[2] J. Bhimani, M. Leeser, and N. Mi, "Accelerating k-means clustering with parallel implementations and GPU computing," in *High Performance Extreme Computing Conference*, pp. 1–6, IEEE, 2015.

[3] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *SIGKDD*, pp. 785–794, ACM, 2016.

[4] K. E. Goodman, J. Lessler, S. E. Cosgrove, A. D. Harris, E. Lautenbach, J. H. Han, A. M. Milstone, C. J. Massey, and P. D. Tamma, "A clinical decision tree to predict whether a bacteremic patient is infected with an extended-spectrum β-lactamase–producing organism," *Clinical Infectious Diseases*, vol. 63, no. 7, pp. 896–903, 2016.

[5] S. Nowozin, C. Rother, S. Bagon, T. Sharp, B. Yao, and P. Kohli, "Decision tree fields: An efficient non-parametric random field model for image labeling," in *Decision Forests for Computer Vision and Medical Image Analysis*, pp. 295–309, Springer, 2013.

[6] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A highly efficient gradient boosting decision tree," in *NeurIPS*, pp. 3149–3157, 2017.

[7] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "CatBoost: unbiased boosting with categorical features," in *NeurIPS*, pp. 6637–6647, 2018.

[8] A. Liaw, M. Wiener, *et al.*, "Classification and regression by random forest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[9] Z. Wen, B. He, R. Kotagiri, S. Lu, and J. Shi, "Efficient gradient boosted decision tree training on GPUs," in *IPDPS*, pp. 234–243, IEEE, 2018.

[10] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "Planet: massively parallel learning of tree ensembles with mapreduce," *PVLDB*, vol. 2, no. 2, pp. 1426–1437, 2009.

[11] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of operations research*, vol. 134, no. 1, pp. 19–67, 2005.

[12] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *TPDS*, vol. 25, pp. 1543–1552, June 2014.

[13] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled CPU-GPU architecture," *Proc. VLDB Endow.*, vol. 6, pp. 889–900, Aug. 2013.

[14] T. Sharp, "Implementing decision trees and forests on a GPU," in *ECCV*, pp. 595–608, Springer, 2008.

[15] N. Birkbeck, M. Sofka, and S. K. Zhou, "Fast boosting trees for classification, pose detection, and boundary detection on a GPU," in *CVPR Workshops*, pp. 36–41, IEEE, 2011.

[16] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?," in *FCCM*, pp. 232–239, IEEE, 2012.

[17] H. Grahn, N. Lavesson, M. H. Lapajne, and D. Slat, "CudaRF: a CUDA-based implementation of random forests," in *International Conference on Computer Systems and Applications*, pp. 95–101, IEEE, 2011.

[18] A. Nasridinov, Y. Lee, and Y.-H. Park, "Decision tree construction on GPU: ubiquitous parallel computing approach," *Computing*, vol. 96, no. 5, pp. 403–413, 2014.

[19] W.-T. Lo, Y.-S. Chang, R.-K. Sheu, C.-C. Chiu, and S.-M. Yuan, "CUDT: a CUDA based decision tree algorithm," *The Scientific World Journal*, vol. 2014, 2014.

[20] D. Strnad and A. Nerat, "Parallel construction of classification trees on a GPU," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 5, pp. 1417–1436, 2016.

[21] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[22] P. Li, Q. Wu, and C. J. Burges, "Mcrank: Learning to rank using multiple classification and gradient boosting," in *NeurIPS*, pp. 897–904, 2008.

[23] E. J. Atkinson, T. M. Therneau, L. J. Melton, J. J. Camp, S. J. Achenbach, S. Amin, and S. Khosla, "Assessing fracture risk using gradient boosting machine (GBM) models," *Journal of Bone and Mineral Research*, vol. 27, no. 6, pp. 1397–1404, 2012.

[24] S. Tyree, K. Q. Weinberger, K. Agrawal, and J. Paykin, "Parallel boosted regression trees for web search ranking," in *WWW*, pp. 387–396, 2011.

[25] S. Si, H. Zhang, S. S. Keerthi, D. Mahajan, I. S. Dhillon, and C.-J. Hsieh, "Gradient boosted decision trees for high dimensional sparse output," in *ICML*, pp. 3182–3190, 2017.

[26] R. Mitchell and E. Frank, "Accelerating the XGBoost algorithm using GPU computing," *PeerJ Preprints*, vol. 5, p. e2911v1, 2017.

[27] L. Rokach and O. Z. Maimon, *Data mining with decision trees: theory and applications*, vol. 69. World scientific, 2008.

[28] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU computing gems Jade edition*, vol. 2, pp. 359–371, 2011.

[29] M. Harris, "Optimizing CUDA," *SC*, 2007.

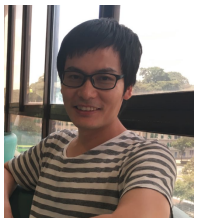
[30] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD*, pp. 511–524, ACM, 2008.

[31] D. Merrill, "CUB v1. 5.3: CUDA Unbound, a library of warp-wide, blockwide, and device-wide GPU parallel primitives," *NVIDIA Research*, 2015.

[32] S. Golomb, "Run-length encodings (corresp.)," *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399–401, 1966.

[33] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," in *ACM SIGMOD Record*, vol. 30, pp. 58–66, ACM, 2001.

[34] K. Jansson, H. Sundell, and H. Boström, "gpuRF and gpuERT: efficient and scalable GPU algorithms for decision tree ensembles," in *IPDPS Workshops*, pp. 1612–1621, IEEE, 2014.



Zeyi Wen is a Research Fellow at National University of Singapore. Zeyi received his PhD degree in Computer Science from University of Melbourne in 2015, and a bachelor degree of Software Engineering from South China University of Technology in 2010. Zeyi's areas of research include parallel computing, machine learning and data mining.

Jiashuai Shi received his bachelor degree in Software Engineering from South China University of Technology in 2016. He is currently a master student at School of Software Engineering, South China University of Technology. His research interests include machine learning and high-performance computing.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an Associate Professor in School of Computing of National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Jian Chen is currently a Professor of the School of Software Engineering at South China University of Technology where she started as an Assistant Professor in 2005. She received her B.S. and Ph.D. degrees, both in Computer Science, from Sun Yat-Sen University, China, in 2000 and 2005 respectively. Her research interests can be summarized as developing effective and efficient data analysis techniques for complex data and the related applications.

Ramamohanarao(Rao) Kotagiri received PhD from Monash University in 1980. He has been at Melbourne University since 1980 and was appointed as a professor in 1989. Rao was Head of CS and SE and Head of the School of EE and CS. He received Distinguished Contribution/Service Awards from ICDM, PAKDD, DASFAA, etc. Rao is a Fellow of the Institute of Engineers Australia, Fellow of Australian Academy Technological Sciences and Engineering and Fellow of Australian Academy of Science.

Qinbin Li is currently a Ph.D. student with National University of Singapore. His current research interests include machine learning algorithms and systems.