Efficient Gradient Boosted Decision Tree Training on GPUs

Zeyi Wen[†], Bingsheng He[†], Kotagiri Ramamohanarao[‡], Shengliang Lu[†], Jiashuai Shi[†§]
[†]*National University of Singapore*, [‡]*The University of Melbourne*, [§]*South China University of Technology*
{wenzy,hebs}@comp.nus.edu.sg, kotagiri@unimelb.edu.au, lusl@nus.edu.sg, shijiashuai@gmail.com

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 wide use of the open sourced XGBoost library, 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, there are a series of key challenges of developing a GPU-based GBDT algorithm, including irregular memory accesses, many small sorting operations and varying data parallel granularities in tree construction. To tackle these challenges on GPUs, we propose various novel techniques (including Run-length Encoding compression and thread/block workload dynamic allocation, and reusing intermediate training results for efficient gradient computation). Our experimental results show that our algorithm named *GPU-GBDT* is often 10 to 20 times faster than the sequential version of XGBoost, and achieves 1.5 to 2 times speedup over a 40 threaded XGBoost running on a relatively high-end workstation of 20 CPU cores. Moreover, GPU-GBDT outperforms its CPU counterpart by 2 to 3 times in terms of performance-price ratio.**

I. Introduction

The recent advancement of machine learning technologies is not only because of new algorithms to improve accuracy, but also new algorithms 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. 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, the GBDT has the advantage of simplicity, effectiveness, and a user-friendly open source toolkit called XGBoost. Additionally, the GBDT has won many awards in recent machine learning and data mining competitions (e.g., Kaggle competitions). However, training effective GBDTs is often very time-consuming, especially for training a large number of deep trees using large datasets. In this paper, we propose a novel GPU-based algorithm called *GPU-GBDT* to improve GBDT training.

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 [6], the trees of GBDTs are dependent. Thus, it is a challenging task to develop an efficient parallel GBDT training algorithm. Particularly, there are a number of key challenges on the efficiency of GPU accelerations for GBDTs, such as irregular memory accesses, many small sorting operations and varying data parallel granularities in tree construction (more details are presented in Section 3.1). We have managed to develop GPU-GBDT, a highly efficient GPU-based training algorithm, to address the challenges. GPU-GBDT 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, we extend GPU-GBDT with Run-length Encoding (RLE) compression, since RLE compression is able to (i) reduce memory consumption so that the GPU can handle larger datasets, (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.

To summarize, our contributions are listed as follows.

- We propose GPU-GBDT, a GPU-based training algorithm for GBDTs. The source code is available at GitHub[1]. This is particularly important for a wider use of GBDTs in more machine learning and data mining applications.
- To make GBDT training efficient to perform on GPUs, we propose various techniques including RLE compression for datasets with high compression ratio, dynamic thread/block workload allocation in different stages of the GBDT training, and reusing intermediate training results for efficient gradient computation.
- We conduct extensive experiments in comparison with the state-of-the-art implementation on the CPU and on the GPU. The results show that GPU-GBDT is often 10 to 20 times faster than the sequential version of XGBoost, and achieves around 1.5 to 2 times speedup over a 40 threaded XGBoost running on a relatively high-end workstation of 20 CPU cores. Moreover, GPU-GBDT outperforms its CPU counterpart by 2 to 3 times in terms of performance-price ratio. We also compare GPU-GBDT with XGBoost using GPUs, and have found that (i) XGBoost using

[1] https://github.com/zeyiwen/gbdt

TABLE I

DENSE AND SPARSE DATA REPRESENTATION

	Dense	Sparse
$\boldsymbol{x}_1$	$\langle 0.0, 0.0, 0.1, 0.0\rangle$	$(a_3\colon 0.1)$
$\boldsymbol{x}_2$	$\langle 1.2, 0.0, 0.1, 0.6\rangle$	$(a_1\colon 1.2); (a_3\colon 0.1); (a_4\colon 0.6)$
$\boldsymbol{x}_3$	$\langle 0.5, 1.0, 0.0, 0.0\rangle$	$(a_1\colon 0.5); (a_2\colon 1.0)$
$\boldsymbol{x}_4$	$\langle 1.2, 0.0, 2.0, 0.0\rangle$	$(a_1\colon 1.2); (a_3\colon 2.0)$

GPUs cannot handle large datasets due to its large GPU memory consumption whereas GPU-GBDT can, and (ii) the execution time of our algorithm is comparable to XGBoost with GPUs for smaller datasets.

II. PRELIMINARIES

A. Dense and sparse data representation

GBDTs are trained using a set of labeled instances (a.k.a. data points), and the set is called a training dataset. We can represent the training dataset 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. In comparison, the sparse representation stores only the non-zero elements, which is more memory efficient, but more expensive to locate the attribute value of an instance.

Suppose we have a training dataset which has four instances: $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$ and $\boldsymbol{x}_4$. We have a dense representation and a sparse representation as shown in Table I.

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. To facilitate enumeration through all the split points, the matrix (e.g., Table I) is transposed and the attribute values are stored in sorted order. This is a common and efficient approach used in training decision trees [3], [7]. The sorted results on each attribute of Table I are shown below.

$a_1 = (\boldsymbol{x}_2\colon 1.2); (\boldsymbol{x}_4\colon 1.2); (\boldsymbol{x}_3\colon 0.5)$

$a_2 = (\boldsymbol{x}_3\colon 1.0)$

$a_3 = (\boldsymbol{x}_4\colon 2.0); (\boldsymbol{x}_2\colon 0.1); (\boldsymbol{x}_1\colon 0.1)$

$a_4 = (\boldsymbol{x}_2\colon 0.6)$

The sorted results are useful when computing the quality (i.e., gain as defined in Section II-B) 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.

Missing values: An additional advantage of sparse representation is that missing values of attributes are naturally supported. We can consider the missing value as either $-\infty$ or $+\infty$, which can be decided during learning. In the dense presentation, missing values need to be filled (e.g., treated as 0) to allow sorting attribute values.

B. 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 target value of $\boldsymbol{x}_i$, respectively. The common loss functions include mean squared error and cross-entropy loss [8]. 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 $\boldsymbol{x}_i$ in the left and right node, respectively; similarly, H_L and H_R denote the sum of h_i of all the instances $\boldsymbol{x}_i$ in the left and right node, respectively; λ is the regularization constant.

C. 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 memory (e.g., 12 GB memory in Titan X). 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, because the number of useful values of a cache line is small. 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 datasets, and reduce data transferring between CPUs and GPUs. Previous studies have demonstrated the efficiency of GPUs in many irregular data-intensive applications [9], [10]. The recursive nature of GBDTs has posed new technical challenges. In this paper, we address the limitation of GPU memory and take advantage of GPU massive computing capability.

D. The GBDT training and the XGBoost library

GBDTs: The GBDT training algorithm is summarized in Algorithm 1. The GBDT training algorithm consists of two key components: (i) finding a split point to a node (Line 7 to 18 of Algorithm 1), and (ii) splitting a node (Line 19 to 23).

XGBoost: The parallel XGBoost on CPUs: The key idea of parallelism in XGBoost is to find the best splits 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

Algorithm 1: The GBDT training algorithm

```
   Input: $\mathcal{I}$: a set of instances; $d$: the maximum depth;
          $\gamma$: a threshold for valid splits; $T$: the number of trees.
   Output: $\mathcal{T}$: a set of decision trees.
1  $\mathcal{T} \leftarrow \phi$
2  repeat
3      InitTree($t$)            /* tree $t$ has only one node */
4      $P \leftarrow \phi$, $\mathcal{N} \leftarrow$ GetRootNode($t$)
5      foreach $n \in \mathcal{N} \wedge depth(n) < d$ do
6          $\mathcal{I}_n \leftarrow$ InstanceInNode($n$)          /* ins. in $n$ */
7          foreach $a \in \mathcal{A}$ do      /* for an attribute */
8              $g \leftarrow 0, p \leftarrow 0$            /* gain & split */
9              $\mathcal{V}_n \leftarrow$ AttributeValue($a$, $n$)
10             foreach $v \in \mathcal{V}_n$ do          /* each split */
11                 $gain \leftarrow$ ComputeGain($\mathcal{I}_n$, $v$, $a$)
12                 if $g < gain$ then
13                     $g \leftarrow gain$, $p \leftarrow v$
14             $P \leftarrow P \cup (a, g, p)$
15         $(a^*, g^*, p^*) \leftarrow \mathbf{0}$        /* best split point */
16         foreach $(a, g, p) \in P$ do          /* get best $a$ */
17             if $(g^* < g) \wedge (g > \gamma)$ then
18                 $(a^*, g^*, p^*) \leftarrow (a, g, p)$
19         if $g^* = 0$ then      /* node won't be split */
20             RemoveLeafNode($n$, $\mathcal{N}$)
21         else
22             $(n_1, n_2) \leftarrow$ SplitNode($n$, $a^*$, $p^*$)
23             UpdateLeafNodes($\mathcal{N}$, $n_1$, $n_2$)
24     $\mathcal{T} \leftarrow \mathcal{T} \cup t$                /* store the tree */
25 until $|\mathcal{T}| > T$;
```

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 dataset. Therefore, the XGBoost on GPUs requires too much GPU memory and cannot handle large datasets. 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).

III. Our GPU-GBDT algorithm

A. Challenges and design rationale

Challenges: The key challenges of designing GPU-GBDT 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 and the number of sorting operations may be huge. Sorting operations are 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 contains 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 granularity. 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 the duplicated split points is expensive on GPUs, because we need to access the neighboring elements which requires extensive memory accesses.

Design rationale: In order to better take advantage of GPU accelerations, we have the following design rationales. First, 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 (also used in XGBoost), we propose fine-grain parallelism by parallelizing the gain computation of each split point (cf. Line 10 of Algorithm 1).

Second, due to the GPU memory limitation, we look for more memory efficient representation than the dense and sparse representations introduced in Section II-A. Particularly, we extend GPU-GBDT with RLE compression, 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 when handling datasets with high compression ratio), because it helps reduce the PCI-e traffic.

Lastly, based on the fine-grained multi-level parallelism and RLE compression, we further address all the technical challenges. First, to reduce the irregular memory access, we propose to reuse the intermediate training results to compute gradients and avoid traversing the trees (cf. Section III-B). Second, to keep the values sorted of each attribute in every node, we propose to use the order preserving partitioning (i.e., histogram based partitioning in Section III-B), powered by techniques to control memory consumption. Third, 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 (cf. the end of Section III-B). Lastly, to avoid the same split point having different gains, we exploit the RLE compression and develop novel techniques to split an RLE element (cf. Section III-C).

B. Training GBDTs using sparse representation

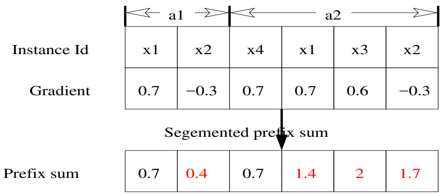
Finding the best split point for a node: There are three steps in finding the best split point for a node: (i) compute the gain for each possible split point, (ii) reset the gain of repeated split points to 0, and (iii) select the best split point (i.e., the split point with the maximum gain).

(i) Compute the gain of a split point: As discussed in Section II-B, we need to compute g_i and h_i for computing the gain of each possible split point (cf. Equation 2). Although

our algorithm supports user defined loss functions, we suppose the mean squared error is used as the loss function[2]. 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 large amount 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 where leaf node an instance belongs, and perform prediction by 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 II-A, 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 [11]. Figure 1 gives an example of the segmented prefix sum for the first order gradient g_i. In the example, we have two segments: one segment corresponds to attribute a_1 and the other corresponds to attribute a_2. The second row labeled as "Gradient" is the values where we need to perform segmented prefix sum. Then, we obtain the results shown in the bottom row.



Instance Id	x1	x2	x4	x1	x3	x2
Gradient	0.7	−0.3	0.7	0.7	0.6	−0.3
Prefix sum	0.7	0.4	0.7	1.4	2	1.7

Fig. 1. Example results of segmented prefix sum

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 node, depending on which way results in larger gain.

(ii) Reset gain of repeated split points: We need to compute the gains of all possible split points of an attribute (e.g., a_1 in Section II-A) in parallel. However, some split points may be repeated in the attribute (e.g., $a_1 = 1.2$ in Section II-A). The split points with the same value next to each other may have different gains which are computed using Equation 2. The different gains are due to different values of G_L, G_R, H_L and H_R 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 node and the right node. In reality an instance should belong to only one node (either left or right 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 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 [12] 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 datasets 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 dataset. 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 datasets, as we will show in Section IV.

Splitting a node: After we have found the best split point, we split the node using that split point. 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. The most challenging task in splitting the node is to maintain values of each attribute in the new nodes in sorted order. We propose to extend the histogram-based method [13] for order preserving partitioning. Figure 2 gives an example of the order preserving partitioning. In the example, the second row labeled as "Attribute values" is the content that we need to partition. Before partitioning,

[2] Our implementation can support other loss functions by customizing the functions for computing g_i and h_i, and nothing else needs to be changed.

the attribute values are in two partitions (i.e., Node 1 and Node 2); after the partitioning, the attribute values are in four partitions (i.e., Node 3 to Node 6) as shown in the bottom row. The key component of this example is the third row labeled as "Scatter" which is used to preserve the sorted order. This "Scatter" is computed by the histogram based partitioning which is discussed next.

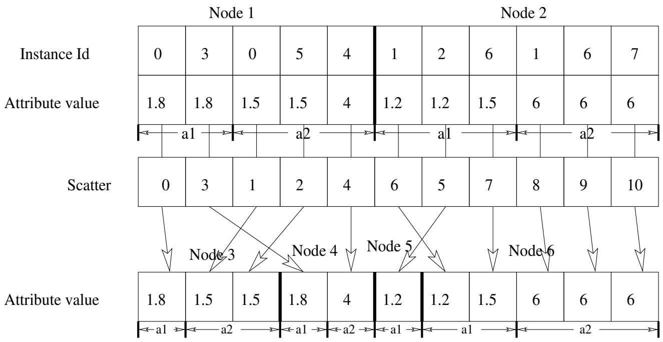


Fig. 2. Partitioning instances of a node to two nodes

Histogram-based data partitioning: Figure 3 shows an example of how the histogram-based data partitioning works. In the example, each thread (denoted by t_i) handles two elements and requires two counters. Originally, the data are in one partition. We want to partition the data into two partitions. So the total number of counters is: (# of threads) × (# of partitions). The first counter records the number of elements that goes to partition 1 and the second counter records the number of elements that goes to partition 2. The counters of the threads are written into the row labeled as "counters". Then, we perform the prefix sum on the counters to determine the first location that each thread will write their elements to. Finally, the offsets (forming the "Scatter" in Figure 2) of the elements are used by the threads to relocate the elements for achieving the final partitioning.

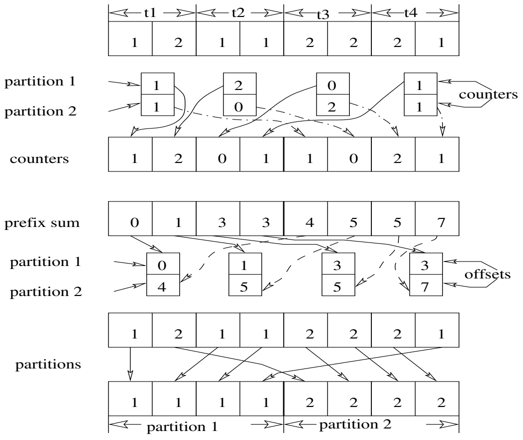


Fig. 3. Histogram-based partitioning

More generally, 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) × (# 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 datasets, 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 [13], 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. The basic idea is that we allocate more workload to a thread when the number of partitions (i.e., # of attribute values × # 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).

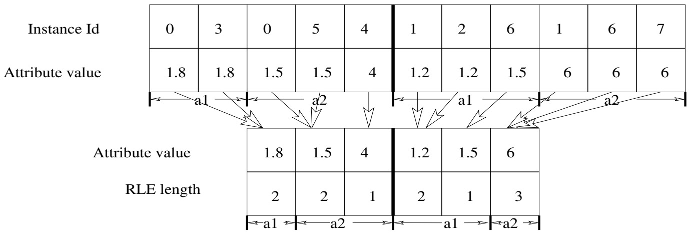


Fig. 4. Sorted attribute values to RLE

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

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

C. GBDT training with Run-length Encoding

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) [14]. 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: $(1.2, 3), (3.4, 4)$. The overview of RLE compression on the sorted attribute values is shown in Figure 4.

This compression has the following two advantages. (i) Reduce memory consumption: some datasets which originally cannot fit into the GPU memory now can be stored in the GPU memory; the memory traffic for transferring the training dataset 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.

The RLE compression is very effective for datasets with high repetition. For attributes that have little repetition, the RLE compression may not be useful. To make our GBDT training algorithm adapt to different datasets, we use a formula to decide whether to perform RLE compression based on the estimated compression ratio: $\frac{\text{dimensionality}}{\text{cardinality}}$. If the ratio is

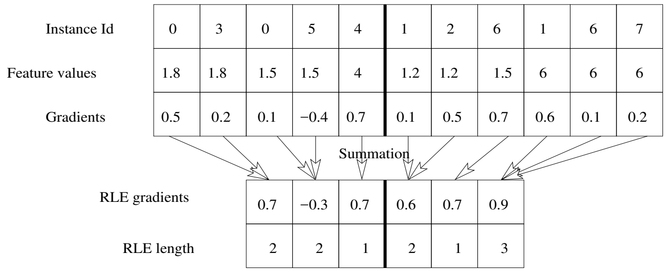


Fig. 5. Computing g'_i for each RLE element

large than R (a user defined constant), we perform RLE compression; otherwise, we do not compress the dataset. It is worth pointing out that the cost of RLE compression is low. This is because the attribute values are already sorted and we only need linear time to compress the data. The compressed data can be used for multiple times (i.e., the number of times equals to the number of trees to construct).

Since the GBDT with RLE compression is similar to the GBDT with sparse representation (discussed in Section III-B) in selecting the best split, the major difference is how to compute gain of split points using RLE and how to split RLE elements. In the following, we highlight the techniques that are different from the GBDT with sparse representation.

Finding the best split point of a node: Finding the best split point is similar to that using sparse representation (i) compute the gain for each possible split point, and (ii) select the best split point for each node.

Compute gain of split points with RLE compression: 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 (the second order derivative h'_i) for each RLE element is to compute the sum of g_i (the sum of h_i) of each instance in the node. Figure 5 gives an example of computing g'_i of each RLE element. The key idea is that the gradients of the instances with the same attribute values are added together.

Splitting a node: Despite several advantages with RLE compression, splitting nodes becomes much more challenging. Next, we present two approaches to split nodes: splitting RLE with decompression, and directly splitting RLE elements.

Splitting RLE with decompression: A simple approach for splitting RLE elements is that we first decompress the RLE elements, then we use the approach discussed for sparse data representation to partition the node, and finally we compress the attribute values on each node to obtain the new RLE elements. Figure 6 shows an example of splitting RLE elements by decompression. Given RLE elements, we decompress the RLE elements back to the original form as shown in the middle of the figure. Then, we apply the order preserving partitioning to the decompressed values, such that we partition the original data. After that, we compress the partitioned data to obtained the new RLE elements as shown in the bottom of the Figure 6.

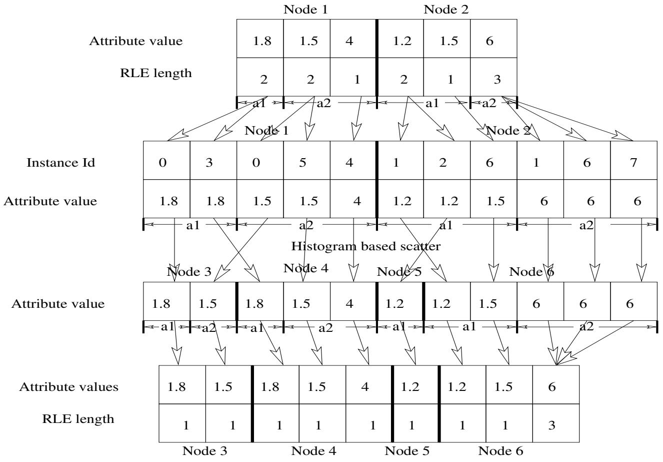


Fig. 6. Splitting RLE elements with decompression

Directly splitting RLE elements: The above approach requires performing decompression and compression every time the nodes are split, which leads to repeated computation. We propose another technique to directly split an RLE element. First, each RLE element is potentially split into two RLE elements, and hence we preallocate two RLE elements in the GPU memory for each RLE element. Then, we compute the length of each new RLE element using the instance to node mapping information. As a result, some preallocated RLE elements may have length of 0. We use prefix sum to remove the RLE element with length of 0 and obtain the final new RLE elements. Figure 7 gives an example of directly splitting RLE elements. First, each RLE element is potentially split into two RLE elements, and hence we preallocate two RLE elements in the GPU memory for each RLE element as shown in the middle of Figure 7. Then, we compute the length of each new RLE element using the instance to node mapping information. As a result, some preallocated RLE elements may have length of 0 (shown by shaded grids in the figure). We use prefix sum to remove the RLE element with length of 0 and obtain the final new RLE elements as shown at the bottom of Figure 7.

D. GPU-GBDT prediction algorithm

In GBDT training, training the next tree is based on the results of the previous trees. Hence, the prediction algorithm is 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

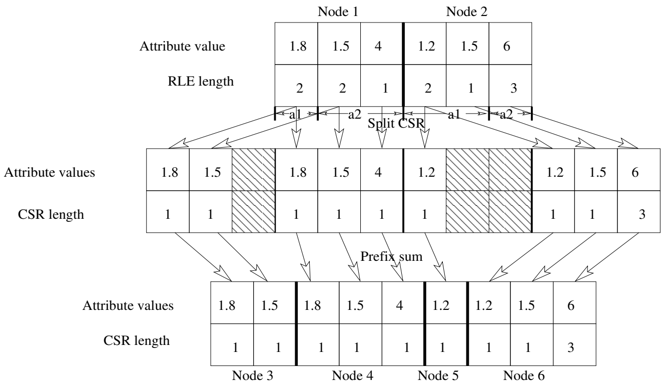


Fig. 7. Direct split RLE elements

independent and all the trees are independent. 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 (right) child if the condition is true (false). Other optimizations for decision tree prediction can be applied to our prediction algorithm [15], but they are out of the scope of this paper, because prediction can be totally avoided in the GBDT training as we have discussed earlier.

IV. Experimental studies

Experimental setup. We used 8 publicly available datasets which were downloaded from the LibSVM website[3], as listed in Table II. The datasets 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 a Titan X Pascal GPU of 12GB memory. The program was compiled with -O3 option. We have also tested GPU-GBDT on Tesla P100 and K20, and the speedup is almost sublinear in the number of cores of the GPUs. Our GPU-GBDT algorithm was implemented in CUDA-C.

Comparison. We compare our algorithm with a well-known GBDT training algorithm named XGBoost[4] which exploits multi-core CPUs and GPUs. We compare three versions of XGBoost: sequential XGBoost (denoted by "xgbst-1"), parallel XGBoost with 40 CPU threads (denoted by "xgboost-40"), and parallel XGBoost with 40 CPU threads and GPUs (denoted by "xgbst-gpu"). Note that the number of threads (i.e., 40 threads) in the parallel 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 datasets. Although GPU-GBDT supports other loss functions, the loss function in our experiments for both XGBoost and GPU-GBDT is mean squared error: $l(y_i, \hat{y}_i) = \sum_i (y_i - \hat{y}_i)^2$.

A. Overall performance study

To study the performance of GPU-GBDT, we set the tree depth to 6 and the number of trees to 40 (corresponding to d and T, respectively in Algorithm 1). Although GPU-GBDT performs better when tree depth is 2 in our experiments, we use tree depth of 6 as it is more common in real world applications to have good accuracy. We measured the total time (including data transfer from CPU to GPU) of training all the trees for both XGBoost and GPU-GBDT. During training, the split points are found without approximation. The results are shown in Table II.

Execution time comparison: We compare the execution time of GPU-GBDT with that of XGBoost. As can be seen from the "speedup over" columns, our algorithm achieves 1.5 to 2 times speedup over xgbst-40, and is often 10 to 20 times faster than xgbst-1. We report the results of xgbst-1, because we hope readers can gain the expected speedup on their CPU (e.g., with 4 cores). The most expensive operation is finding the best split point: around 75% of total training time for XGBoost and around 95% of that for GPU-GBDT in our experiments. This gives insight for further improvement of GPU-GBDT. Further improving the performance of finding the best split is indeed a challenging task due to the extensive memory access and recursive nature of the splitting process. The GPU enabled XGBoost (i.e., xgbst-gpu) is quite unstable: xgbst-gpu cannot process most of the datasets tested either because of out of memory or because of very large different RMSE. For the *susy* dataset that xgbst-gpu can process, the execution time of our algorithm is comparable to xgbst-gpu. If we further analyze the performance difference between these two GPU implementation, we find that xgbst-gpu is slightly faster than our algorithm, because xgbst-gpu uses dense data representation and is faster when looking up which attribute a split point belongs to. That also leads to a potential optimization to our algorithm for small datasets, and GPU-GBDT can use dense data representation if the data set is small. We have performed the experiments, and confirmed this finding. The results are omitted here due to lack of space.

RMSE comparison: We have compared the trees constructed by GPU-GBDT and the CPU-based XGBoost, and found that the trees are identical. Here, we show the rooted mean squared error (RMSE) on the training datasets for the trees trained by GPU-GBDT and XGBoost. As can be seen from the last three columns (i.e., "rmse" column) of Table II, our algorithm produces exactly the same RMSE as XGBoost (both xgbst-1 and xgbst-40; the RMSE of xgbst-1 is omitted as it is identical to xgbst-40). For the GPU version of XGBoost (i.e., xgbst-gpu), RMSE is different from its CPU counterpart and our algorithm (cf. *real-sim* and *covetype*). The large RMSE of xgbst-gpu is probably because of dense representation which considers missing values as 0.

B. Sensitivity studies

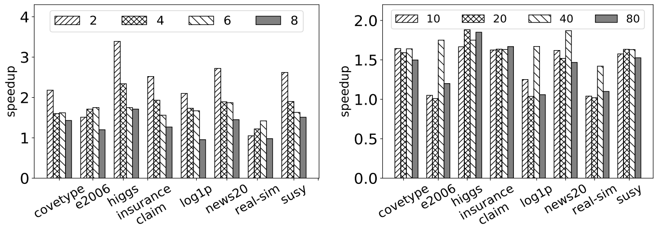
Varying tree depth: We set the number of trees to 40, and varied the tree depth from 2 to 8. Figure 8a shows the effect of tree depth on speed up of GPU-GBDT over xgbst-40. As we

[3] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

[4] https://github.com/dmlc/xgboost

TABLE II
OVERALL COMPARISON BETWEEN OUR GPU ALGORITHM WITH XGBOOST

dataset			elapsed time (sec)				speedup over		rmse		
name	cardinality	dimension	ours	xgbst-40	xgbst-1	xgbst-gpu	xgbst-40	xgbst-1	ours	xgbst-40	xgbst-gpu
covetype	581012	54	3.65	5.93	36.11	diff. rmse	1.62	9.89	0.31	0.31	$3.3\text{x}10^{15}$
e2006	16087	150361	12.47	21.80	227.79	out of mem.	1.75	8.27	0.23	0.23	N.A.
higgs	11000000	28	220.01	386.47	2189	out of mem.	1.75	9.95	0.42	0.42	N.A.
insurance claim	13184290	35	216.6	339.55	2085	out of mem.	1.56	9.63	38.22	38.22	N.A.
log1p	16087	4272228	39.93	66.58	544.88	out of mem.	1.67	13.65	0.24	0.24	N.A.
news20	19954	1355191	7.03	13.12	139.78	out of mem.	1.87	19.88	0.49	0.49	N.A.
real-sim	72201	20958	2.97	4.23	44.87	diff. rmse	1.42	15.11	0.48	0.48	7.29
susy	5000000	18	78	127.18	869.11	60.11	1.63	11.14	0.37	0.37	0.37



(a) Varying tree depth from 2 to 8 (b) Varying the number of trees

Fig. 8. Speedup of GPU-GBDT over xgbst-40

can see from the figure, our algorithm consistently outperforms xgbst-40. Our algorithm performs best when the tree depth is 2, but the speedup is relatively stable when the tree depth increases further.

Varying the number of trees: We set the tree depth to 6, and varied the number of trees from 10 to 80. Figure 8b shows the result. The speedup when varying the number of trees is rather stable as the number of trees increases. This is due to the fact that the trees in GBDTs are dependent, and hence the increase of the number of trees does not bring better parallelism.

C. Impact of individual optimizations

As we have discussed in Section III, we have some optimizations specifically for our GPU algorithm. Here, we study their individual impacts on GPU-GBDT. The techniques include (i) Customized SetKey: to automatically set key for the segmented prefix sum discussed in Section III-B; (ii) Customized IdxComp Workload: to decide thread workload depending on datasets discussed in Section III-B; (iii) RLE: to compress datasets with RLE compression discussed in Section III-C; (iv) SmartGD: to compute g_i and h_i using the intermediate training results discussed in Section III-B; (v) Directly Split RLE: to directly split RLE elements as discussed in Section III-C. We switch off each individual optimization, and investigate the execution time change to the entire algorithm. We set the tree depth to 6 and the number of trees to 40 in this set of experiments.

Figure 9 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 SetKey technique helps improve 10% to 20% for the execution time in datasets (e.g., *log1p* and *news20*) of high dimensionality. 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.

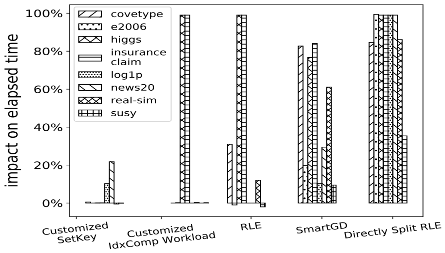
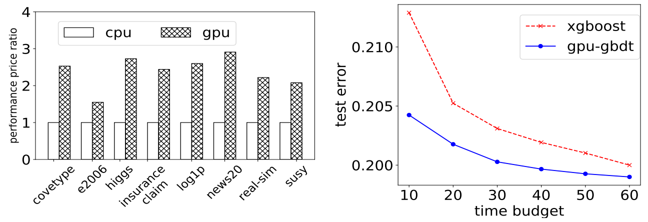


Fig. 9. Impact of individual optimizations



(a) Economic cost of CPUs and GPUs (b) Test error given a time budget

Fig. 10. Economic cost and test error per time budget comparison

D. Economic cost comparison

Finally, we study the performance per dollar. We define the performance-price ratio as: $\frac{1/time}{price}$. The price of NVIDIA Titan X is 1,200 USD [16], and that of two Xeon E5-2640v4 CPUs is 1,878 USD [17] at the time of writing this paper. Figure 10a shows the performance ratio normalized by the performance ratio of CPUs (xgbst-40 is used for CPU). The performance-price ratio of GPU-GBDT consistently outperforms its CPU counterpart by 1.5 to 3 times. This result confirms that GPU-GBDT is more cost-effective than its CPU counterpart.

E. Case studies

Our algorithm is useful to three key application scenarios: (i) applications require updating the predictive model frequently (i.e., online learning); (ii) applications require training models in real-time and (iii) applications require minimizing the test error given a time budget.

(i) Applications that require updating the predictive model frequently: One example of such applications is using GBDT for credit risk prediction [18] where updating model timely

is important, because a large number of transactions occurs every minute. However, the number of credit card transaction records used in the training dataset is large and the training is expensive. The work [18] uses 211,357 training instances each of which has 8,990 features. Training the model (described in the work [18]) requires about 27 minutes using XGBoost on our workstation. In comparison, GPU-GBDT can respond new credit risk and prevent invalid transactions more timely.

(ii) Applications that require training models in real-time: One example of such applications is using GBDT for malware detection [19]. Responding to newly emerged malware in real-time is crucial, because malware spreads so quickly and every second counts in the prevention of malware spreading. XGBoost took 43 seconds to update the model for malware detection [19], while our GPU-GBDT requires only about 20 seconds to update the model so as to detect recent outbreak malware. This is important to mitigate the spread of malware.

(iii) Applications that require training an effective model in a given time budget: One example of such applications is using GBDT for data mining and machine learning competitions (such as Kaggle.com, a popular data science competition platform). We have studied a use-case from Kaggle.com on product recommendation[5]. The solution with ranking at the third place[6] uses XGBoost. The solution uses 142 features to represent a training instance, and the dataset has 17 million training instances in total. The hyper-parameters of gradient boosting decision trees include the number of trees T, the depth of trees d, the regularization constant γ and learning rate η. In practice, competition participants will look for the best settings for hyper-parameters in order to achieve an effective model. We consider the following search space where $T \in \{500, 1000, 2000, 4000\}$, $d \in \{2, 4, 6, 8\}$, $\gamma \in \{0, 0.1, 0.2\}$ and $\eta \in \{0.2, 0.3, 0.4\}$. In total, we train 144 models each of which has a unique four tuple of hyper-parameters. The solution (with the best accuracy) winning the competition has 1000 trees of depth 4 with $\eta = 0.3$ and $\gamma = 0$ and takes 2.95 hours to train the model. Training the 144 models for best hyper-parameter selection takes about 535 hours (about 22.3 days) on a workstation with 20 CPU cores. In comparison, our GPU-GBDT finishes the training in about 10 days, which saves practitioners significant amount of time and is especially critical for winning a competition.

As a sanity check, Figure 10b shows the test error of XGBoost and of our GPU-GBDT on the workstation that we used for our experimental study. The dataset used here is *susy*. For the same time budget measured in seconds, GPU-GBDT obtains the model that clearly has smaller test error than XGBoost.

In addition, we achieve the speedup using the GPU which is 2 to 3 times cheaper than CPUs with 20 cores (cf. Figure 10a) in terms of price-performance ratio.

V. Related work

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 [20]. 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 [21]. 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 [22], 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 replaced by reusing intermediate training results (cf. Section III-B).

GPU accelerated decision tree training: Grahn et al. proposed to use a GPU thread to train one decision tree for the random forest training [23]. 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 [24]. Lo et al. [25] 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 [26] 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. 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 split points without approximation. In contrast, our algorithm utilizes data compression techniques to train GBDTs more efficiently and to support larger datasets.

Gradient Boosting Decision Trees: Gradient Boost Machines were first introduced by Friedman [27], and have shown great potential in many real world applications [28], [29]. Panda et al. [7] proposed a MapReduce-based learning algorithm for decision trees that ensembles with approximation when finding split points for large datasets. Tyree et al. proposed parallel CPU boosting regression trees for webpage ranking problems [30]. Si et al. developed a GBDT training algorithm for high dimensional sparse output [31]. Chen and Guestrin proposed an efficient GBDT algorithm which is

[5] https://www.kaggle.com/c/santander-product-recommendation

[6] www.kaggle.com/c/santander-product-recommendation/discussion/26899

implemented in XGBoost [3]. Mitchell and Frank proposed to use GPUs to accelerate the finding split point procedure of XGBoost [1]. 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 datasets due to the large memory consumption. LightGBM [32] is an alternative implementation of GBDTs, but it only supports finding the best split points approximately.

VI. Conclusion and future work

GPU accelerations have become a hot research topic for improving the efficiency of machine learning and data mining algorithms. This paper develops a novel parallel implementation named GPU-GBDT 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 our GPU algorithm is often 10 to 20 times faster than the sequential version of XGBoost, and achieves 1.5 to 2 times speedup over XGBoost running on a relatively high-end workstation with 20 CPU cores. Moreover, GPU-GBDT outperforms its CPU counterpart by 2 to 3 times in terms of performance price ratio. Thus, we demonstrate that GPU-GBDT can be a more efficient and cost effective alternative to its CPU-based counterparts.

Our algorithm is naturally applicable to multiple GPUs or GPU clusters, and we consider this direction as our future work. Moreover, GPU-GBDT spends most of its time on finding the best split point. We will further investigate this process for improving GPU-GBDT.

Acknowledgement

This work is supported by a MoE AcRF Tier 2 grant (MOE2017-T2-1-122) and an NUS startup grant in Singapore.

References

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

[2] J. Bhimani, M. Leeser, and N. Mi, "Accelerating k-means clustering with parallel implementations and GPU computing," in *High Performance Extreme Computing Conference (HPEC)*, 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] A. Liaw, M. Wiener, *et al.*, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[7] 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.

[8] 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.

[9] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 1543–1552, June 2014.

[10] 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.

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

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

[13] 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.

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

[15] 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.

[16] P. Graham, "The price of Titan X: www.vrfocus.com/2017/04/titan-x-vs-titan-xp-comparison-guide," 2017.

[17] Intel, "The price of E5-2640v4: ark.intel.com/products/92984/Intel-Xeon-Processor-E5-2640-v4-25M-Cache-2_40-GHz," 2017.

[18] X. Yu, "Machine learning application in online leading credit risk prediction," *arXiv preprint arXiv:1707.04831*, 2017.

[19] M. Yousefi-Azar, L. Hamey, V. Varadharajan, and M. D. McDonnell, "Fast, automatic and scalable learning to detect android malware," in *International Conference on Neural Information Processing*, pp. 848–857, Springer, 2017.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

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

[31] 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.

[32] 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 *Advances in Neural Information Processing Systems*, pp. 3149–3157, 2017.