ThunderGBM: Fast GBDTs and Random Forests on GPUs

Zeyi Wen[†] wenzy@comp.nus.edu.sg

Jiashuai Shi[†‡] shijiashuai@gmail.com

Bingsheng He[†], **Qinbin Li**[†] {hebs,qinbin}@comp.nus.edu.sg

Jian Chen[‡] ellachen@scut.edu.cn

[†] *School of Computing, National University of Singapore, 117418, Singapore*

[‡] *School of Software Engineering, South China University of Technology, Guangzhou, 510006, China*

Editor:

Abstract

Gradient Boosting Decision Trees (GBDTs) and Random Forests (RFs) have been used in many real-world applications. They are often a standard recipe for building state-of-the-art solutions to machine learning and data mining problems. However, training and prediction are very expensive computationally for large and high dimensional problems. This article presents an efficient and open source software toolkit called *ThunderGBM* which exploits the high-performance Graphics Processing Units (GPUs) for GBDTs and RFs. ThunderGBM supports classification, regression and ranking. It uses identical command line options and configuration files as XGBoost—one of the most popular GBDT and RF libraries. ThunderGBM can be used through multiple language interfaces including C/C++ and Python, and can run on single or multiple GPUs of a machine. Our experimental results show that ThunderGBM outperforms the existing libraries while producing similar models, and can handle high dimensional problems which existing GPU based libraries fail. Documentation, examples, and more details about ThunderGBM are available at `https://github.com/xtra-computing/thundergbm`.

Keywords: Gradient Boosting Decision Trees, Random Forests, GPUs, efficiency

1. Introduction

Gradient Boosting Decision Trees (GBDTs) and Random Forests (RFs) are widely used in advertising systems, spam filtering, sales prediction, medical data analysis, and image labeling (Chen and Guestrin, 2016; Goodman et al., 2016; Nowozin et al., 2013). For ease of presentation, we use GBDTs as a representative in the remaining of this article, rather than repeatedly mentioning both GBDTs and RFs. In contrast with deep learning, GBDTs are simple and relatively easy to explain. The wide use of GBDTs are largely due to the user-friendly open source toolkits such as XGBoost (Chen and Guestrin, 2016), LightGBM (Ke et al., 2017) and CatBoost (Prokhorenkova et al., 2018). Additionally, the GBDT has won many awards in recent Kaggle data science competitions. However, training GBDTs is often very time-consuming, especially for large and high dimensional problems.

GPUs have been used to accelerate many real-world applications (Dittamo and Cisternino, 2008), due to their abundant computing cores and high memory bandwidth. In this article, we propose a GPU-based software tool called *ThunderGBM* to improve the

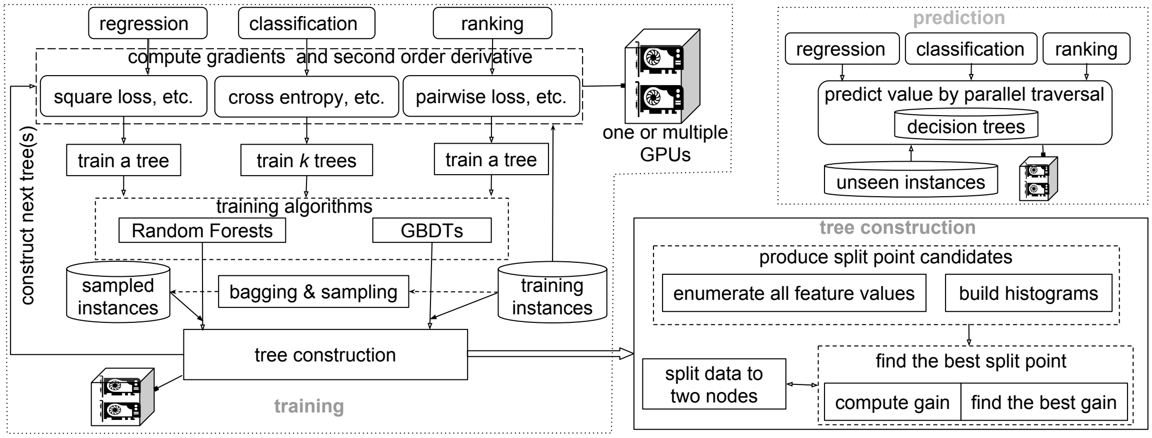

Figure 1: Overview of training and prediction in ThunderGBM

efficiency of GBDTs and RFs. ThunderGBM supports binary and multi-class classification, regression and ranking. It uses the same command line input options and configuration files as XGBoost—arguably the most popular library for GBDTs. Moreover, ThunderGBM supports multiple interfaces such as C/C++ and Python, and can run on single or multiple GPUs of a machine. Our experimental results show that when the existing libraries running on CPUs, ThunderGBM is 6.4 to 12 times, 4.6 to 26.4 times, and 10.3 times faster than XGBoost, LightGBM and CatBoost, respectively, while producing similar models. In comparison with them on GPUs, ThunderGBM is 1 to 6.6 times, 9.6 to 19.5 times and 1.5 times faster than XGBoost, LightGBM and CatBoost, respectively. More importantly, ThunderGBM can handle high dimensional problems which existing GPU based libraries fail. The full version of ThunderGBM, which is released under Apache License 2.0, can be found on GitHub at `https://github.com/xtra-computing/thundergbm`.

2. Overview and Design of ThunderGBM

Figure 1 shows the overview and software abstraction of ThunderGBM. The training algorithms for different tasks (i.e., classification, regression and ranking) are built on top of a generic tree construction module. This software abstraction allows us to concentrate on optimizing the performance of tree constructions. Different tasks only require different ways of computing the derivatives of the loss functions. Notably, the multi-class classification task requires training k trees where k is the number of classes (Bengio et al., 2010; Chen and Guestrin, 2016), while regression and ranking only require training one tree per iteration. The prediction module is relatively simple, and is essentially computing predicted values by concurrent tree traversal and aggregating the predicted values of the trees on GPUs. Here, we focus on the training on single GPU. More details about using multiple GPUs and the prediction are in Appendix B. We develop a series of optimizations for the training. For each module that leverages GPU accelerations, we propose efficient parallel algorithmic design as well as effective GPU-aware optimizations. The techniques are used to support two major components in ThunderGBM: (i) computing the gradients and second order derivatives, and (ii) tree construction.

2.1 Computing the Gradients and Second Order Derivatives on GPUs

The gradients and second order derivatives are computed by the predicted values and the true values by $g_i = \frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i}$ and $h_i = \frac{\partial^2 l(y_i, \hat{y}_i)}{\partial \hat{y}_i^2}$, where the gradient and second order derivative of the loss function are denoted by g_i and h_i, respectively; $l(y_i, \hat{y}_i)$ denotes the loss function, and y_i and $\hat{y}_i$ denote the true and predicted target value of the i-th training instance, respectively. ThunderGBM supports common loss functions such as mean squared error, cross-entropy and pairwise loss (De Boer et al., 2005; Cao et al., 2007; Lin et al., 2014). More details on loss functions and derivatives are in Appendix A.

Computing g_i and h_i requires the predicted value $\hat{y}_i$ of the i-th training instance, ThunderGBM computes $\hat{y}_i$ based on the intermediate training results. This is because the training instances are recursively divided into new nodes and are located in the leaf nodes at the end of training each tree. Thus, ThunderGBM can obtain the predicted values for each training instances by reusing this intermediate results, and avoids traversing the trained trees to obtain the predicted values. To exploit the massive parallelism of GPUs, we create a sufficient number of threads to efficiently use the GPU resources. Each GPU thread keeps on pulling a training instance and computes g and h for the instance.

2.2 Tree Construction on GPUs

Tree construction is a key component and time consuming in the GBDT and RF training. We adopt and extend many novel optimizations in our previous work (Wen et al., 2018) to improve the performance of ThunderGBM. Tree construction contains two key steps: (i) producing the split point candidates, and (ii) finding the best split for each node.

Step (i): ThunderGBM supports two ways of producing the split point candidates: one based on enumeration and the other based on histograms. The former approach requires the feature values of the training instances to be sorted in each tree node, such that it can enumerate all the distinct feature values quickly to serve as the split point candidates. However, the number of split point candidates may be huge for large data sets. The later approach considers only a fixed number of split point candidates for each feature, and each feature is associated with a histogram containing the statistics of the training instances. Each bin of the histogram contains the values of the accumulated gradients and second order derivatives for all the training instances located in the bin. In ThunderGBM, each histogram is built in two phases. Firstly, a partial histogram is built on the thread block level using shared memory. Secondly, all the partial histograms of a feature are accumulated to construct the final histogram. ThunderGBM automatically chooses the split point candidate producing strategies based on the data set density, i.e., histograms based approach for dense data sets and enumeration based approach for the others. The density is measured by $\frac{\text{total \# of feature values}}{\text{\# of instances} \times \text{\# of dimensions}}$. If the ratio is large than 20%, we choose the histogram based approach; choose the enumeration based approach otherwise.

Step (ii): Finding the best split is to look for the split point candidate with the largest gain. The gain (Chen and Guestrin, 2016) of each split point candidate is computed by $gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R-\lambda}\right]$, where G_L and G_R (resp. H_L and H_R) denote the sum of g_i (resp. h_i) of all the instances in the left and right node, respectively; λ is the regularization constant. In ThunderGBM, one GPU thread is dedicated to compute the gain

data set			on cpus (sec)			on the gpu (sec)				speedup (cpu)			speedup (gpu)		
name	card.	dim.	xgb	lgbm	cat	xgb	lgbm	cat	ours	xgb	lgbm	cat	xgb	lgbm	cat
higgs (reg)	11M	28	44.6	77.4	67.9	9.9	80.8	10.1	6.6	6.8	11.7	10.3	1.5	12.2	1.5
log1p (reg)	16K	4M	oom	676	oom	oom	337	oom	25.6	n.a.	26.4	n.a.	n.a.	13.2	n.a.
cifar10 (clf)	50K	3K	521	456	lerr	124	785	lerr	81.5	6.4	5.6	n.a.	1.5	9.6	n.a.
news20 (clf)	16K	62K	198	222	oom	109	321	oom	16.5	12	13.5	n.a.	6.6	19.5	n.a.
yahoo (rnk)	473K	700	18.8	11	n.a.	2.4	29.4	n.a.	2.4	7.8	4.6	n.a.	1.0	12.3	n.a.

Table 1: Comparison with XGBoost, LightGBM and CatBoost

of each split point candidate. The split point candidate with the largest gain is selected as the best split point for the node, which is done by a parallel reduction on GPUs. Once the best split point is obtained, the training instances in a node is divided into two child nodes. For producing the split point candidates by enumeration, ThunderGBM adopts the novel order preserving data partitioning techniques on GPUs proposed in our previous work (Wen et al., 2018), i.e., the feature values of the child nodes are naturally sorted. For producing the split point candidates using histograms, a GPU thread is dedicated to determining which child node a training instance should go to based on the best split point. ThunderGBM repeats Step (i) and (ii) until the termination condition is met (e.g., the tree reaches the maximum depth).

3. Experimental Studies

We conducted experiments on a workstation running Linux with two Xeon E5-2640 v4 10 core CPUs, 256GB memory and a Tesla P100 GPU of 12GB memory. The tree depth is set to 6 and the number of trees is 40. More results on experiments and descriptions about the data sets can be found in Appendix C. Five representative data sets are used here (cf. Table 1). The data sets for testing regression, classification and ranking are marked with “reg”, “clf” and “rnk”, respectively. We used the latest versions of XGBoost (Chen and Guestrin, 2016), LightGBM (Ke et al., 2017) and CatBoost (Prokhorenkova et al., 2018) on 10 Jan 2019, respectively.

The results are shown in Table 1, where “oom” stands for “out of memory”, “lerr” stands for “large training error” and “n.a.” stands for “not applicable”. When the existing libraries are running on CPUs, ThunderGBM is 6.4 to 12 times, 4.6 to 26.4 times, and 10.3 times faster than XGBoost, LightGBM and CatBoost, respectively. When they are on GPUs, ThunderGBM is 1 to 6.6 times, 9.6 to 19.5 times, and 1.5 times faster than XGBoost, LightGBM and CatBoost, respectively. Moreover, ThunderGBM can handle high dimensional problems (e.g., *log1p*) which the existing libraries fail or run slowly. ThunderGBM has smaller or comparable errors to the existing libraries (cf. Appendix C).

4. Conclusion

In this article, we present *ThunderGBM* which supports classification, regression and ranking. To be easy to use, ThunderGBM uses identical input command line options and configuration files as XGBoost, and supports the Python interface. Our experimental results show that ThunderGBM outperforms the existing libraries while producing similar models, and can handle high dimensional problems which the existing libraries often fail.

Appendix A. Loss functions of regression, classification and ranking

In this section, we describe different loss functions used in ThunderGBM in order to support regression, classification and ranking. We also derive the gradient and second order derivative of each loss function, because the gradients and the second order derivatives play key roles in the GBDT training.

ThunderGBM supports the following loss functions: mean square error, logistic loss (Collins et al., 2002), cross-entropy loss (Moher and Gulliver, 1998), pairwise loss and NDCG loss (Ravikumar et al., 2011). Please note that for providing more information about the loss functions and to keep consistent with XGBoost, we use a special case of cross-entropy for logistic regression, and call the corresponding loss "logistic loss" similar to XGBoost (Chen and Guestrin, 2016).

A.1 Mean square error

The mean square error is used in regression in ThunderGBM. The option for the objective function is "reg:linear" following the convention of XGBoost which is arguably the most popular library for GBDTs and Random Forests. The goal of the training is the same as linear regression. The mean square error is defined as follows.

$$l(y_i, \hat{y}_i) = \frac{1}{2}(y_i - \hat{y}_i)^2$$

where y_i and $\hat{y}_i$ are the true target value and the predicted target value of the i-th training instance, respectively. Then, the gradient and second order derivative are $(y_i - \hat{y}_i)$ and 1, respectively.

A.2 Logistic loss

Logistic loss can be used in binary classification or regression for applications with target values between 0 and 1 (i.e., $y_i \in [0, 1]$). The option for the objective function is "reg:logistic" in ThunderGBM similar to XGBoost. The goal of the training is the same as logistic regression, and aims to minimize the logistic loss which is defined as follows.

$$l(y_i, \hat{y}_i) = -y_i \log(p_i) - (1 - y_i)\log(1 - p_i)$$

where $p_i = \frac{1}{1+e^{-\hat{y}_i}}$. To help compute the gradient and second derivative of the loss function, we first compute the derivative of p_i below.

$$\frac{\partial p_i}{\partial \hat{y}_i} = \frac{e^{-\hat{y}_i}}{(1 + e^{-\hat{y}_i})^2} = (1 - p_i)p_i$$

The derivative of the loss function is derived as follows.

$$\frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i} = -y_i \cdot \frac{1}{p_i} \cdot p_i' - (1 - y_i) \cdot \frac{-1}{1 - p_i} \cdot p_i'$$

By substituting the derivative of p_i into the above equation, we obtain $\frac{\partial l(y_i,\hat{y}_i)}{\partial \hat{y}_i} = p_i - y_i$. The second order derivative of the loss function is $\frac{\partial (p_i - y_i)}{\partial \hat{y}_i} = (1 - p_i)p_i$.

A.3 Cross-entropy loss

This loss is similar to logistic loss, but is used in the multi-class classification problems in ThunderGBM. The option for choosing this objective function is "multi:softmax" or "multi:softprob". The "multi:softprob" option has the same objective function as "multi:softmax", but the predicted value is a probability instead of a class label. Next, we present the loss function and derive its derivative. The cross-entropy loss is defined as follows.

$$l(y_i, \hat{y}_i) = -\sum_k y_i^k \log(p_i^k) \tag{1}$$

where k is the identifier of the k-th class and i is the identifier of the i-th training instance. Similar to the logistic loss, to help derive the gradient and second order derivative of the loss function, we first derive the derivative for the softmax function defined below.

$$p_i^k = \frac{e^{\hat{y}_i^k}}{\sum_{m=1}^K e^{\hat{y}_i^m}}$$

where K is the total number of classes. For ease of presentation, we ignore the subscript i which is the identifier of the i-th training instance. Then, we can write the above softmax function as $p^k = \frac{e^{\hat{y}^k}}{\sum_{m=1}^K e^{\hat{y}^m}}$. The derivative of the function is shown below.

$$\frac{\partial p^k}{\partial \hat{y}^m} = \frac{\partial \frac{e^{\hat{y}^k}}{\sum_{m=1}^K e^{\hat{y}^m}}}{\partial \hat{y}^m}$$

There are two cases for computing the derivative of the softmax function: $k = m$ and $k \neq m$. We first consider $k = m$.

$$\frac{\partial p^m}{\partial \hat{y}^m} = \frac{\partial \frac{e^{\hat{y}^m}}{\sum_{m=1}^K e^{\hat{y}^m}}}{\partial \hat{y}^m} = \frac{e^{\hat{y}^m}\sum_{m=1}^K e^{\hat{y}^m} - e^{\hat{y}^m}e^{\hat{y}^m}}{(\sum_{m=1}^K e^{\hat{y}^m})^2} = \frac{e^{\hat{y}^m}}{\sum_{m=1}^K e^{\hat{y}^m}} \cdot \frac{\sum_{m=1}^K e^{\hat{y}^m} - e^{\hat{y}^m}}{\sum_{m=1}^K e^{\hat{y}^m}}$$

As $k = m$, the derivative derived from above can be written as $\frac{\partial p^k}{\partial \hat{y}^m} = p^k(1 - p^m)$.

If $k \neq m$, the derivative of the softmax function p^k is shown below.

$$\frac{\partial p^k}{\partial \hat{y}^m} = \frac{0 - e^{\hat{y}^k}e^{\hat{y}^m}}{(\sum_{m=1}^K e^{\hat{y}^m})^2} = -\frac{e^{\hat{y}^k}}{\sum_{m=1}^K e^{\hat{y}^m}} \cdot \frac{e^{\hat{y}^m}}{\sum_{m=1}^K e^{\hat{y}^m}} = -p^k \cdot p^m$$

The two cases (i.e., $k = m$ and $k \neq m$) can be written together as follows.

$$\frac{\partial p^k}{\partial \hat{y}^m} = p^k(\delta^{km} - p^m) \tag{2}$$

where

$$\delta^{km} = \begin{cases} 1, & \text{if } \ k = m \\ 0, & \text{otherwise} \end{cases}$$

Now, we derive the gradient for the cross-entropy loss.

$$\frac{\partial l(y,\hat{y}^m)}{\partial \hat{y}^m} = -\sum_k y^k \cdot \frac{\partial log(p^k)}{\partial \hat{y}^m} = -\sum_k y^k \cdot \frac{1}{p^k} \cdot \frac{\partial p^k}{\partial \hat{y}^m}$$

From the derivative of the softmax function in Equation (2), we have the following.

$$\begin{aligned}\frac{\partial l(y,\hat{y}^m)}{\partial \hat{y}^m} &= -y^m(1-p^m) - \sum_{k\neq m} y^k \frac{1}{p^k}(-p^k \cdot p^m) = -y^m(1-p^m) + \sum_{k\neq m} y^k p^m \\ &= -y^m + y^m p^m + \sum_{k\neq m} y^k p^m = p^m(y^m + \sum_{k\neq m} y^k) - y^m\end{aligned}$$

As $\sum_k y^k = 1$, so we have $\frac{\partial l(y,\hat{y}^m)}{\partial \hat{y}^m} = p^m - y^m$. The second order derivative is $\frac{\partial^2 l(y,\hat{y}^m)}{\partial^2 \hat{y}^m} = p^m(1-p^m)$. In our implementation, a common normalization technique is used in computing p_i.

$$p_i^k = \frac{e^{\hat{y}_i^k}}{\sum_{m=1}^K e^{\hat{y}_i^m}} = \frac{Ne^{\hat{y}_i^k}}{N\sum_{m=1}^K e^{\hat{y}_i^m}} = \frac{e^{\hat{y}_i^k + \log(N)}}{\sum_{m=1}^K e^{\hat{y}_i^m + \log(N)}}$$

The term $\log(N)$ is computed by $\log(N) = -\max(\hat{y}_i^k)$.

A.4 Pairwise loss and NDCG loss

Here, we present the loss functions used for ranking in ThunderGBM. In order to describe the loss functions, we define the true probability of a pair of training instances with indices i and j as follows.

$$P_{ij} = \frac{1}{2}(1 - S_{ij})$$

where $S_{ij} = -1$ if the i-th training instance is less relevant than the j-th training instance, $S_{ij} = +1$ if the i-th training instance is more relevant than the j-th training instance, and $S_{ij} = 0$ if the two training instances are the same.

We define the predicted probability of the pair of instances as follows.

$$\hat{P}_{ij} = \frac{1}{1 + e^{-\sigma(s_i - s_j)}}$$

where σ is a hyper-parameter, and s_i and s_j are the predicted scores of the ranking functions for the i-th and j-th training instance, respectively.

Following the existing literature (Burges, 2010), ThunderGBM also uses the cross-entropy loss defined below for ranking problems.

$$C_{ij} = -P_{ij}\log(\hat{P}_{ij}) - (1 - P_{ij})\log(1 - \hat{P}_{ij}) = \frac{1}{2}(1 - S_{ij})\sigma(s_i - s_j) + \log(1 + e^{-\sigma(s_i - s_j)}) \quad (3)$$

The derivatives over s_i and s_j are shown below.

$$\frac{\partial C}{\partial s_i} = -\frac{\partial C}{\partial s_j} = \sigma(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}})$$

Please note that the derivatives over s_i and s_j have the same absolute value, and the only difference is their signs. For ease of presentation, we define λ_{ij} as follows.

$$\lambda_{ij} = \frac{\partial C}{\partial s_i} = \sigma(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}})$$

Existing studies (Burges et al., 2007) show that adding the learning metric into the derivative helps achieve good results. The λ_{ij} defined above can be simulated using the following equation.

$$\lambda_{ij} = -\frac{\sigma}{1 + e^{\sigma(s_i - s_j)}} \cdot |\Delta Z_{ij}|$$

where $|\Delta Z_{ij}|$ is the value of the metric (e.g., NDCG). Please note that only $S_{ij} = 1$ is considered in the above definition of λ_{ij}.

The second order derivative of the loss function (cf. Equation 3) is as follows.

$$\lambda'_{ij} = \frac{\partial \lambda_{ij}}{\partial s_i} = \frac{-\sigma \cdot e^{\sigma(s_i - s_j)} \cdot \sigma}{(1 + e^{\sigma(s_i - s_j)})^2} \cdot |\Delta Z_{ij}| = -\sigma^2 \cdot |\Delta Z_{ij}| \cdot \frac{1}{1 + e^{\sigma(s_i - s_j)}} \cdot (1 - \frac{1}{1 + e^{\sigma(s_i - s_j)}})$$

Then the gradient of the i-th instance is computed as follows.

$$g_i = \sum_{\{i,j\} \in I} \lambda_{ij} - \sum_{\{j,i\} \in I} \lambda_{ji}$$

where I is the set of all pairs of the training instances. Similarly, we can compute the second order derivative for the i-th instance $h_i = 2 \cdot \sum_{\{i,j\} \in I} \lambda'_{ij}$. In ThunderGBM, $|\Delta Z_{ij}|$ equals to 1 when the objective function is pairwise loss; $|\Delta Z_{ij}|$ equals to the change of NDCG when the objective function is NDCG loss.

Appendix B. Multiple GPU Support and the Prediction Algorithm

ThunderGBM supports multiple GPUs and parallel prediction. Next, we elaborate the details for these two components.

B.1 Training 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. A machine nowadays can have multiple GPUs, and commonly can host two to four GPUs. ThunderGBM can leverage multiple GPUs to train models on larger data sets that can fit into multiple GPUs. It supports a simple and effective approach for GBDT training on multiple GPUs. In particular, we partition the training data by features to handle large data sets (i.e., column based partitioning). There are two advantages of the feature based partitioning. First, both enumeration based and histogram based techniques for split point candidate production are natively supported. Producing the split point candidates of a feature requires accessing all the values of the feature. Storing all the feature values of a feature in one GPU helps perform finding the split points more communication efficiently. The second advantage is that the GPUs do not need to exchange the partial histograms in order to find the approximate split points,

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

Table 2: Information of data sets used in the experiments

since all the feature values of a feature are stored locally and the GPU can build the whole histogram. Hence, the GPUs only need to exchange the local best split point candidates in order to obtain the global best split point candidates for the tree nodes. This reduces the communication cost from $\mathcal{O}(N \times F \times B)$ to $\mathcal{O}(N \times F)$, where N is the number of tree nodes needed to split, F is the number of features 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.

B.2 The Parallel Prediction Algorithm

The prediction module in ThunderGBM is relatively simple, because the prediction mainly involves tree traversal. ThunderGBM supports different types of tasks including classification, regression and ranking. These tasks are designed in a unified prediction algorithm: traversing the decision trees to obtain the predicted value of an input instance. In ThunderSVM, we perform the prediction by concurrently traversing multiple decision trees for multiple input instances.

Appendix C. Additional experimental results

Experimental setup. We used six more publicly available data sets as shown in Table 2, and *higgs* and *log1p* have been used in our main text. The data sets were downloaded from the LibSVM website. The data sets cover a wide range of the cardinality and dimensionality. The experiments were conducted on a workstation running Linux with two Xeon E5-2640v4 10 core CPUs, 256GB main memory and one Pascal P100 GPU 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 (Chen and Guestrin, 2016), LightGBM (Ke et al., 2017) and CatBoost (Prokhorenkova et al., 2018). We used the latest versions of XGBoost (Chen and Guestrin, 2016), LightGBM (Ke et al., 2017) and CatBoost (Prokhorenkova et al., 2018) on 10 Jan 2019, 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

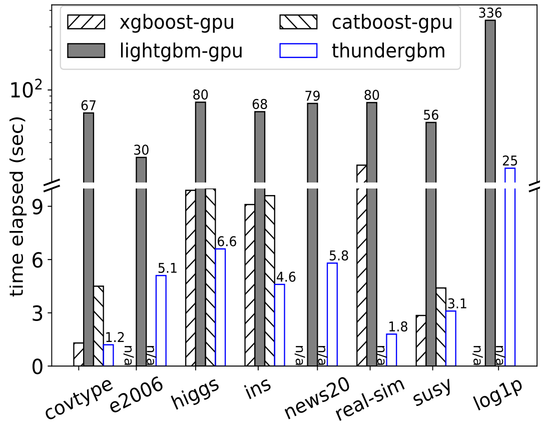

Figure 2: Comparison with XGBoost, LightGBM and CatBoost on the GPU

loss function in our experiments for all the libraries (including ThunderGBM) is the mean squared error: $l(y_i, \hat{y}_i) = \sum_i (y_i - \hat{y}_i)^2$.

C.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 them on CPUs. We show that ThunderGBM running on the GPU significantly outperforms the three existing libraries on the GPU or the CPU. Finally, we compare the Root Mean Squared Error (RMSE) of the libraries against ThunderGBM to study the quality of the trained models. Some results shown here are from our previous work (Wen et al., 2019).

C.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 point candidates are found using the histogram based method, as LightGBM and CatBoost only support producing the split point candidates using histograms.

The results of the four GPU implementation of GBDTs are shown in Figure 2. The first observation is that ThunderGBM 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 40 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, LightGBM

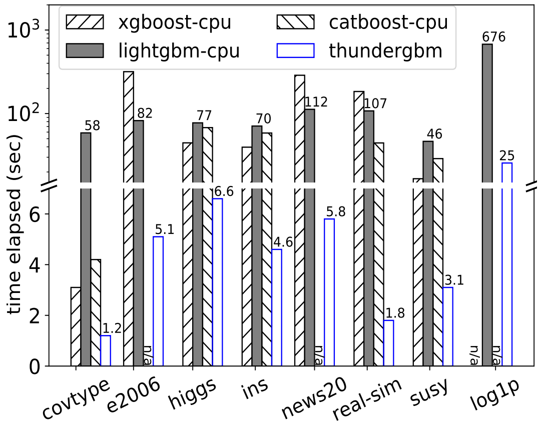

Figure 3: Comparison with XGBoost, LightGBM and CatBoost on CPUs

on the GPU is inefficient, and ThunderGBM outperforms it by more than 10 times on the data sets.

C.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 CPU threads is chosen automatically by the libraries.

The results of the three libraries on CPUs are shown in Figure 3, 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 more than 10 times faster than LightGBM on all the data sets. For the *real-sim* data set, ThunderGBM achieves nearly 60 and 100 times speedup over LightGBM and XGBoost, respectively.

C.2 Handling high dimensional data

Here, we further investigate the memory consumption of ThunderGBM in comparison with XGBoost, LightGBM and CatBoost on *e2006*, *news20* and *log1p*. As XGBoost and CatBoost run out of memory for the data sets, we analyze the memory consumption of ThunderGBM and the other libraries. The memory for storing the training data sets are shown in Table 3. XGBoost and CatBoost require much more memory than LightGBM and ThunderGBM, because XGBoost and CatBoost use dense data representation while LightGBM and ThunderGBM use sparse data representation. Although LightGBM is as memory efficient as ThunderGBM, ThunderGBM is an order of magnitude faster than LightGBM as shown in Figure 2 and 3.

data set	XGBoost	LightGBM	CatBoost	ThunderGBM
e2006	9GB	76MB	9GB	76MB
log1p	256GB	0.4GB	256GB	0.4GB
news20	101GB	4.9MB	101GB	4.9MB

Table 3: Memory consumption comparison

data set			xgboost	lightgbm	catboost	thundergbm	measure
name	card.	dim.					
higgs (reg)	11M	28	0.42	0.42	0.43	0.42	rmse
log1p (reg)	16,087	4,272,228	n.a.	0.29	n.a.	0.24	
cifar10 (clf)	50,000	3,072	0.83%	2.18%	45.22%	0.82%	prediction error rate
news20 (clf)	15,935	62,061	4.04%	5.68%	n.a.	3.78%	
yahoo (rnk)	473,134	700	0.882	0.903	n.a.	0.884	ndcg

Table 4: Training result comparison

C.3 Training error comparison

We study the quality of trees learnt by different libraries in this set of experiments. In this set of experiments, we first investigate the training errors for the data sets listed in the main text. Then, we study the training errors using more data sets shown in Table 2 beyond the listed data sets.

We compare the training errors for the data sets shown in the main text. The results are shown in Table 4, where “n.a.” stands for “not applicable”. We used RMSE to measure the regression tasks (marked with “reg”), used prediction error rate to measure the classification tasks (marked with “clf”), and used NDCG to measure the ranking task (marked with “rnk”). ThunderGBM obtains better or comparable results to the existing libraries.

Next, we study the differences among different libraries using eight data sets in total. We use RMSE to measure the training error of different models. Table 5 shows the results. ThunderGBM produces similar RMSE as XGBoost and LightGBM. CatBoost tends to have higher training errors.

data set	xgboost	lightgbm	catboost	thundergbm
covtype	0.72	0.64	0.86	0.72
e2006	0.25	0.29	n.a.	0.25
higgs	0.42	0.42	0.43	0.42
ins	38.70	38.70	38.80	38.70
log1p	n.a.	0.29	n.a.	0.24
news20	0.49	0.35	n.a.	0.49
real-sim	0.47	0.37	0.52	0.47
susy	0.37	0.37	0.37	0.37

Table 5: RMSE comparison with XGBoost, LightGBM and CatBoost

data set	elapsed time (sec)			RMSE	
	xgboost (cpu)	xgboost (gpu)	thundergbm	xgboost	thundergbm
covtype	1.97	0.48	1.29	1.09	1.04
higgs	54.61	9.21	6.54	0.46	0.45
ins	52.99	10.47	5.3	38.99	38.98
susy	21.59	lerr	3.16	0.39	0.39

Table 6: Comparison on training Random Forests

C.4 Training Random Forests

Only XGBoost and ThunderGBM support RFs. Therefore, we only compare ThunderGBM with XGBoost in terms of RMSE and execution time. The results are shown in Table 6, where "lerr" stands for "large error". ThunderGBM produces similar or better RMSE than XGBoost on CPUs. XGBoost on GPUs results in large RMSE in some data sets (e.g., RMSE is 0.9 on the *susy* data set). In terms of efficiency, ThunderGBM is 1.5 to 10 times faster than XGBoost on CPUs. The improvement of ThunderGBM is more notable for large data sets such as *higgs* and *ins*. ThunderGBM is generally faster and more stable than XGBoost on GPUs as we can see from the results.

C.5 Prediction

ThunderGBM has similar efficiency as the other existing libraries.

References

Samy Bengio, Jason Weston, and David Grangier. Label embedding trees for large multiclass tasks. In *NeurIPS*, pages 163–171, 2010.

Christopher J Burges, Robert Ragno, and Quoc V Le. Learning to rank with nonsmooth cost functions. In *NeurIPS*, pages 193–200, 2007.

Christopher JC Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *ICML*, pages 129–136. ACM, 2007.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *SIGKDD*, pages 785–794. ACM, 2016.

Michael Collins, Robert E Schapire, and Yoram Singer. Logistic regression, adaboost and bregman distances. *Machine Learning*, 48(1-3):253–285, 2002.

Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67, 2005.

Cristian Dittamo and Antonio Cisternino. GPU White paper, 2008.

Katherine E Goodman, Justin Lessler, Sara E Cosgrove, Anthony D Harris, Ebbing Lautenbach, Jennifer H Han, Aaron M Milstone, Colin J Massey, and Pranita 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*, 63(7):896–903, 2016.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A highly efficient gradient boosting decision tree. In *NeurIPS*, pages 3149–3157, 2017.

Guosheng Lin, Chunhua Shen, and Jianxin Wu. Optimizing ranking measures for compact binary code learning. In *ECCV*, pages 613–627. Springer, 2014.

Michael Moher and T Aaron Gulliver. Cross-entropy and iterative decoding. *IEEE Transactions on Information Theory*, 44(7):3097–3104, 1998.

Sebastian Nowozin, Carsten Rother, Shai Bagon, Toby Sharp, Bangpeng Yao, and Pushmeet Kohli. Decision tree fields: An efficient non-parametric random field model for image labeling. In *Decision Forests for Computer Vision and Medical Image Analysis*, pages 295–309. Springer, 2013.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. CatBoost: unbiased boosting with categorical features. In *NeurIPS*, pages 6637–6647, 2018.

Pradeep Ravikumar, Ambuj Tewari, and Eunho Yang. On NDCG consistency of listwise ranking methods. In *AISTATS*, pages 618–626, 2011.

Zeyi Wen, Bingsheng He, Ramamohanarao Kotagiri, Shengliang Lu, and Jiashuai Shi. Efficient gradient boosted decision tree training on GPUs. In *International Parallel and Distributed Processing Symposium*, pages 234–243. IEEE, 2018.

Zeyi Wen, Jiashuai Shi, Bingsheng He, Jian Chen, Ramamohanarao Kotagiri, and Qinbin Li. Exploiting GPUs for efficient gradient boosting decision tree training. *under review*, 2019.