Computer tools in education, 2022
№ 2: 59–73
http://cte.eltech.ru
doi:10.32603/2071-2340-2022-2-59-73

CATBOOST GPU IMPLEMENTATION

Ershov V. A.[1,2], Postgraduate, ✉ noxoomo@yandex-team.ru

[1]Yandex LLC, 16, Leo Tolstoy Street, 119021, Moscow, Russia
[2]Saint Petersburg State University, 7/9 Universitetskaya nab., 199034, Saint Petersburg, Russia

Abstract

In this paper we discus GPU implementation of open-sourced gradient boosting library CatBoost. This implementations shows the state-of-the-art performance among openly-available libraries and we want to share design insights and used algorithms.

Keywords: *NVIDIA, GPU, Gradient boosting, Decision trees, GBM, Categorical features.*

Citation: V. A. Ershov, "CatBoost GPU Implementation," *Computer tools in education*, no. 2, pp. 59–73, 2022; doi: 10.32603/2071-2340-2022-2-59-73

1. INTRODUCTION

Gradient boosting is a powerful machine-learning technique that achieves state-of-the-art results in a variety of practical tasks. For a number of years, it has remained the primary method for learning problems with heterogeneous features, noisy data, and complex dependencies: web search, recommendation systems, weather forecasting, and many others [2, 10–12]. It is backed by strong theoretical results that explain how strong predictors can be built by iterative combining weaker models (*base predictors*) via a greedy procedure that corresponds to gradient descent in a function space. Most popular implementations of gradient boosting use decision trees as base predictors (gradient boosted decision trees, GBDT).

There exist several gradient boosted decision tree libraries: XGBoost [4], LightGBM [7], H20, CatBoost [9]. CatBoost (for "categorical boosting") — high-quality open-source[1] gradient boosting decision trees library that successfully handles both numerical and categorical features.

CatBoost has several advantages compared to other gradient boosting libraries. First of all, it could efficiently handle categorical features (those ones having a discrete set of values that are not necessary comparable with each other, e.g. user ID or name of a city). In practice, many datasets include such feature, but there was a bad support for such type of predictors in open-source libraries and CatBoost fills the gap: CatBoost was compared with with XGBoost, LightGBM and H2O and outperforms competitors on several publicly available dataset. The results of this comparison are shown in table 1, and more information and detailed experiment setup could be found in [9].

CatBoost is one of the most popular gradient boosted decision trees framework today. This library is used to train model on a really big datasets. To scale this library on real-world data, one usually use GPU version of this library. This implementation is highly efficient and could

[1] https://github.com/catboost/catboost

utilize multiple GPU on one or several hosts. Despite having been already released, we realized that CatBoost GPU implementation was not adequately described and are therefore looking to correct this oversight in this article.

Table 1. Comparison with baselines. Tuned algorithms. Logloss

	CatBoost	LightGBM	XGBoost	H2O
Adult	**0.269741**	0.276018 (+2.33 %)	0.275423 (+2.11 %)	0.275104 (+1.99 %)
Amazon	**0.137720**	0.163600 (+18.79 %)	0.163271 (+18.55 %)	0.162641 (+18.09 %)
Appet	**0.071511**	0.071795 (+0.40 %)	0.071760 (+0.35 %)	0.072457 (+1.32 %)
Click	**0.390902**	0.396328 (+1.39 %)	0.396242 (+1.37 %)	0.397595 (+1.71 %)
Internet	**0.208748**	0.223154 (+6.90 %)	0.225323 (+7.94 %)	0.222091 (+6.39 %)
Kdd98	**0.194668**	0.195759 (+0.56 %)	0.195677 (+0.52 %)	0.195395 (+0.37 %)
Kddchurn	**0.231289**	0.232049 (+0.33 %)	0.233123 (+0.79 %)	0.232752 (+0.63 %)
Kick	**0.284793**	0.295660 (+3.82 %)	0.294647 (+3.46 %)	0.294814 (+3.52 %)

The rest of the article is structured as follows:

1. We will briefly discus decision tree learning on numerical dataset in the section 2.
2. The section 3 gives insight about how we should build decision trees on GPU.
3. In the section 4 we will discus novel way of dealing with categorical features, introduced in CatBoost, and how we implemented it in on GPU
4. In the section 5 we will talk about decision tree learning in distributed setting.
5. Finally, in the section 6 we will provide some benchmarks of CatBoost GPU implementation

2. DECISION TREE LEARNING IN GRADIENT BOOSTING

For the rest of article we will used the following notation: we have set of n learning samples $\mathscr{D} = \{(d_i, y_i)\}_{i=1}^{n}$, where $d_i \in \mathbb{R}^m$ is a sample features vector and y_i is associated label. By y we will denote labels vector: $y = (y_1, \ldots, y_n)$. We have set of all decision trees $\mathscr{F}$ — functions from $\mathbb{R}^m$ to $\mathbb{R}$. We will use capital letter F_k (or F, if ensemble length is not specified) to denote ensemble of k decision trees and non capital letter f_i (or f) to denote single decision tree. One defines loss function $L(u, v)$ and we need to find ensemble F such empirical loss function is as small as possible:

$$F = \operatorname{argmin} \frac{1}{n} \sum_{i=1}^{n} L(F(d_i), y_i). \tag{1}$$

On the figure 1 visualized the basic idea of gradient boosting: one fits a simple decision tree to the data. This tree is usually not too deep, and, as a result, has a big error. To improve model quality the one more tree is trained, in a such way that new tree will correct errors of the first one. Two trees will decrease error, but they are still pretty bad. This procedure is iteratively repeated until a strong tree ensemble is built.

This idea is formalized in the following way. Boosting algorithms solves the problem in a greedy manner: we have ensemble F_N of N decision trees, we need to build ensemble of size $N+1$, then:

$$F_{N+1} = F_N + \operatorname{argmin}_{f \in \mathscr{F}} \frac{1}{n} \sum_{i=1}^{n} L\big(F_N(d_i) + f(d_i), y_i\big) \tag{2}$$

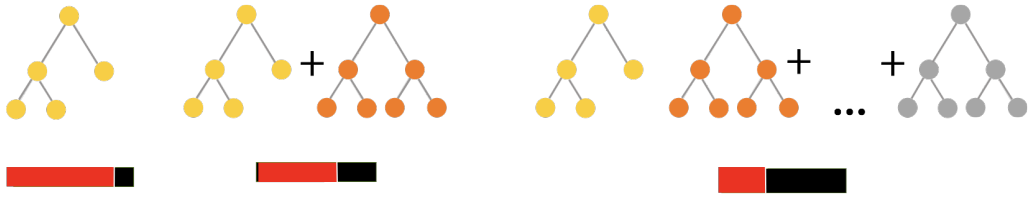

Figure 1. Gradient Boosting

There exist several techniques how to approximately solve equation 2. This techniques are based on similar ideas of approximation of L via first and / or second order derivatives. For our purpose we just need to know, that this problem eventually reduces to the next one: we have label vectors $t = (t_1, \dots, t_n)$, weights vector $w = (w_1, \dots, w_n)$ and some similarity measure $S(u, v, w)$ defined on $u, v, w \in \mathbb{R}^n$. Decision tree is searched by solving next optimization problem:

$$f(d) = \operatorname{argmin}_f S(t, f(d), w), \tag{3}$$

where $f(d) = (f(d_1), \dots, f(d_n))$ is vector of predictions for fitted decision tree. We refer reader to the literature for more details [6].

Classical CART algorithm uses as a similarity measure l_2-distance:

$$S(u, v) = \sum_{i=1}^{n} w_i (f(d_i) - t_i)^2, \tag{4}$$

while CatBoost similarity measure is based on vector correlation:

$$S(u, v) = 1.0 - \frac{\sum_{i=1}^{n} w_i f(d_i) t_i}{\sqrt{\sum_{i=1}^{n} w_i t_i^2} \sqrt{\sum_{i=1}^{n} w_i f(d_i)^2}} \tag{5}$$

Optimization of equation 3 is usually done by greedy algorithm (see CART [1] for more details): decision tree is build iteratively on each iterations by splitting leaf (or leaves) in a way that gives the best similarity measure improvement.

Assume, for example, we need to build a decision tree of depth 1 (decision stump). Assume, for simplicity, that we have just one feature $x = (x_1, \dots, x_n)$ (it could be, for example, height) and that all values of x are unique. Then to build decision tree we need to evaluate each possible split conditions c and chose one that after splitting points with it into two sets $\{i : x_i \le c\}$ and $\{i : x_i > c\}$ we will obtain the best score.

For decision trees with common similarity functions (like l_2-distance or correlation) to evaluate best splits we need to compute just two statistics[2]:

$$\begin{gathered} \sum_{i=1}^{n} t_i [x_i > c], \\ \sum_{i=1}^{n} w_i [x_i > c], \end{gathered} \tag{6}$$

where $[\cdot]$ denotes Iverson brackets, i.e., $[x_j > c]$ equals 1 if $x_j > x$ and 0 otherwise.

[2] Or, sometimes, three: one count of samples in each leaf — $\sum [x_i > c]$.

This two statistics, known for each split candidate, allows us to compute similarity measures S for each split and choose the best one.

For solving real-world problems evaluation of all possible splits is not necessary: usually we could consider restricted set of splits selected in advance in unsupervised way: for example, based on quantiles of feature vector x. This approach, in spirit similar to using 4-bit or 8-bit floats in neural networks, leads to efficient scalable decision tree learning algorithms. If we are using restricted set of splits, then, without loss of generality, we could assume, that all features (numeric ones) are 8-bit integers. So for evaluation of best split we'll need to compute at most (feature count) $\times 255 \times 2$ statistics:

$$\begin{gathered} \sum_{i=1}^{n} t_i [x_i = b] \forall b = 1 \dots 255, \\ \sum_{i=1}^{n} w_i [x_i = b] \forall b = 1 \dots 255. \end{gathered} \tag{7}$$

Equation 7 is well-known histogram primitive: we just aggregate some float statistic based on feature values. Histogram primitives could be efficiently parallelized between several cores, hosts, GPU devices or GPU streaming multiprocessors.

Implementation note

Here we discus building of decision stump. For a deeper tree algorithm is essentially the same: one iteratively split some leaf (or leaves) to two ones. In each leaf there are some subsets of observations and the task reduces to build a decision stump conditioned on observations at hand. The only one problem appears: this should be done for subset of observations and, as a result, requires a random access to data (while first level could be done via sequential data loads)[3].

CatBoost specific note

CatBoost, at the same time with classical boosting scheme, described above, support so-called *Ordered*-boosting scheme. This scheme, designed to reduce overfitting, is more computationally expensive, but, computation primitives for it are essentially the same.

3. DECISION TREES ON GPU

To effectively find a new decision tree on each step of GBDT algorithm we need to solve following tasks: firstly, we need to store entire dataset in GPU RAM; econdly, as we discussed in previous section, we need a fast way to compute histograms 7.

To take all advantages of GPU we need to solve this two problems simultaneously: dataset layout in GPU memory should be done in a such way, that we could efficiently compute histogram primitives. At the same time, dataset in GPU RAM should have low memory consumption.

Histograms computation for decision trees has their own peculiar properties. Firstly, we need to compute histograms for many features at once. Secondly, we knew, that each histogram has limited number of bins (255 at most, but for production usage it's usually sufficient to use just 32). And, finally, we need to compute several statistics.

For efficient computation on GPU we should make as little global memory access, as possible. So, we should compute all statistics in one pass through data. And, as GPU uses $\geq$ 32-bit loads,

[3] By the way, for specific type of tree (oblivious trees), which are used in CatBoost, random access to data could be avoided on CPU. Sadly, but on GPU we still need to do random access because of hardware memory restrictions. So, for this article, we don't specify type of trees one should use.

we should ensure, that all memory access are at least 32-bit wide. Also, all aggregation should be done in fast shared memory.

One way to satisfy this conditions — group several features in one integer. For CatBoost we group several numerical features in one 32-bit integer in following way:

— binary features are grouped 32 features per integer;
— 4 bits (or less) are grouped 8 features per integer;
— 8 bits features are grouped 4 features per integer.

And histogram primitives are designed to work on group of several features: 32 features and 2 statistics at one for binary once, 8 features and 2 statistics for 4-bit ones and 4 features and 2 statistics for 8-bit ones. Plus we specialize kernels for different bin count (discretization of data to 32 bins, 64 bins, 128 bins and 255 bins).

3.1. 32-bin histogram

CatBoost kernels were initially designed to work with 32-bin histograms. This bin count provides a good speed vs quality tradeoff.

So, here we'll discus only this specialization. All other CatBoost kernels are similar in spirit, but less clear to describe.

So we have gradient values $g[i]$, associated weights $w[i]$ and feature groups $(f_1, f_2, f_3, f_4)[i]$. We have samples $i_1, \dots, i_l$ in current leaf and need to find the best condition to split this leaf to two ones. To evaluate all possible splits of leave we need to compute 4×2 histograms:

$$\begin{aligned} \text{hist}_g[j][b] &= \sum_{k: f_j[i_k]=b} g[i_k], \\ \text{hist}_w[j][b] &= \sum_{k: f_j[i_k]=b} w[i_k]. \end{aligned} \tag{8}$$

CatBoost builds partial histogram per each warp and use sample i_k[4]. We will describe work which is done by one warp on first 32 samples. Thread with index k processes sample i_k. Since we are building 2×4 histograms at once we need $32 * 32 * 4$ bytes of shared memory per warp. We are using the following layout: for each warp we allocate 4 partial histograms in shared memory (so $4 \times 2 \times 32 = 1024$ floats). First 8 threads will work with first histogram, second 8 ones with second and so on. To avoid bank conflicts we group histograms by bin value: for each bin value we store histogram block visualized on figure 2.

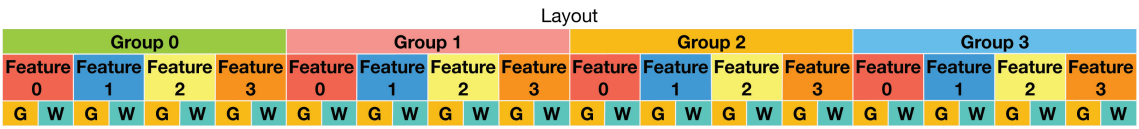

Figure 2. Histogram layout

To update histograms all 32 threads load sample labels and grouped features to registers. Then warp performs updates of shared-memory histogram simultaneously in 4 iterations: see listing 3.1 for histogram pseudo-code.

With describe we need 4KB shared memory per warp. So we could run at most 384 threads per one block. On modern GPUs with 96KB of shared memory this allows us to run 2 blocks per

[4] g and w vectors could be gathered in advance and be accessed via direct index, because they are the same for all feature groups.

streaming multiprocessor and achieve 38 % occupancy. For Kepler (and Fermi) hardware we are able to run one block with 19 % occupancy.

Listing 1. Histogram aggregation

```
void AddPoint(const ui32 featuresGroup,
              const float t,
              const float w) {
    //target or weight to use first
    const bool flag = threadIdx.x & 1;

    //Warp Layout:
    // feature: 0 0 1 1 2 2 3 3 1 1...
    // stat:    t w t w t w t w t w...
    histGroupId = (threadIdx.x & 31) / 8;
    hist = WarpHist + 8 * histGroupId;

    for (int i = 0; i < 4; i++) {
        int f = (threadIdx.x / 2 + i) % 3;
        const int shift = 28 - 4 * f;
        int bin = featuresGroup >> shift;
        bin &= 31;

        int offset0 = 32 * bin + 8 * f + flag;
        int offset1 = 32 * bin + 8 * f + !flag;

        hist[offset0] += (flag ? t : w);
        hist[offset1] += (flag ? w : t);
    }
}
```

For group of binary features, and 4-bit features, almost the same layout allows to run 768 % threads per one block and achieve 75 % occupancy.

3.2. Avoid atomics

Atomic operation in shared memory are expensive even on modern GPU with support on hardware level. So this operations should be avoided, if possible. Currently, CatBoost computational kernels does not use atomic operations in shared memory at all: we need to be able to run fast even on Kepler GPUs.

We have made a simple test to evaluate atomic operation performance: CatBoost kernel uses 384 threads and achieves just 38% percent occupancy on GPU with 96KB of shared memory, and 19% occupancy on old-generation Kepler devices. We could run ×2 more blocks (or threads) in exchange for atomic operations in shared memory: first 6 warps could share histograms with the second 6 ones. We compared this histogram implementation with one which does not use atomics. For benchmark we choose the most computationally expensive histogram — the histogram on the first level of tree (splits after first depth are much cheaper).

Results are presented in figure 3. Atomics are awfully slow on NVIDIA K40 — indeed, Kepler does not have hardware atomics, and computation with atomics are x1.5-x2.5 times slower even on more modern hardware despite bigger occupancy.

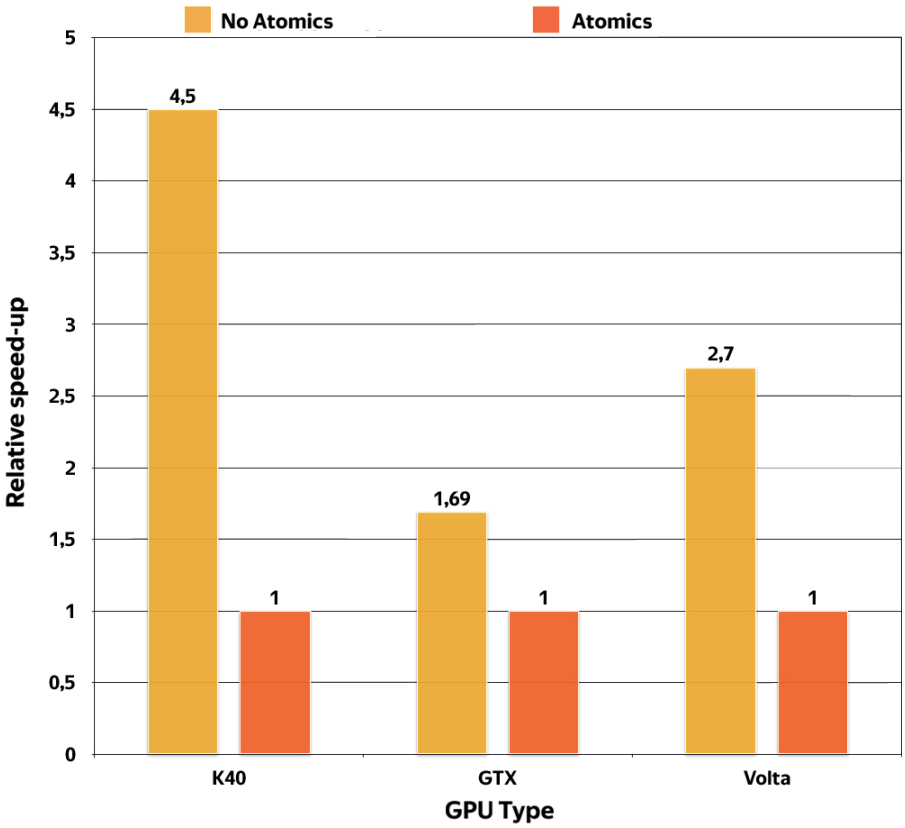

Figure 3. Atomic vs Non-Atomic performance

4. WORKING WITH CATEGORICAL FEATURES

Categorical features have a discrete set of values called *categories* which are not necessary comparable with each other; thus, such features cannot be used in binary decision trees directly. A common practice for dealing with categorical features is converting them to numbers at the preprocessing time, i.e., each category for each example is substituted with one or several numerical values. These values can be compared with each other, thus, they can be further used in decision trees.

There are several ways to calculate numerical values to replace categorical feature. The most widely used technique which is usually applied to low-cardinality categorical features is *one-hot encoding*: the original feature is removed and a new binary variable is added for each category [8]. One-hot encoding can be done during the preprocessing phase or during training, the latter can be implemented more efficiently in terms of training time and is implemented in CatBoost. Computationally, dealing with one-hot is essentially the same as the numerical ones and the problem reduces to one described in previous section.

Another way to deal with categorical features is to compute some statistics based on the label values of the examples. Namely, assume that we are given a feature $x = (x_1, \dots, x_n)$ and $y \in \mathbb{R}^n$ is a *label value*. The simplest way is to substitute the category with the average label value on the whole train dataset. So, x_i is substituted with

$$\frac{\sum_{j=1}^{n} [x_j = x_i] \cdot y_j + \alpha}{\sum_{j=1}^{n} [x_j = x_i] + \alpha + \beta}, \tag{9}$$

where $[\cdot]$ denotes Iverson brackets, i.e., $[x_j = x_i]$ equals 1 if $x_j = x_i$ and 0 otherwise; α and β are prior parameters [3, 8] used to reduce the noise obtained from low-frequency categories. For regression tasks standard technique for calculating prior is to take the average label value in the dataset. For binary classification task a prior is usually an a priori probability of encountering a positive class [8].

This procedure leads to overfitting. For example, if there is a single example from the category x_i in the whole dataset then the new numeric feature value will be equal to the label value on this example. A straightforward way to overcome the problem is to partition the dataset into two parts and use one part only to calculate the statistics and the second part to perform GBDT training. This reduces overfitting but it also reduces the amount of data used to train the model and to calculate the statistics.

CatBoost uses a more efficient strategy, inspired by online-learning algorithms, which allows to use the whole dataset for training. We perform a random permutation of the dataset and for each example compute average label value for the example with the same category value placed before the given one in the permutation. Let $\sigma = (\sigma_1, \ldots, \sigma_n)$ be the permutation, then x_{σ_p} is substituted with

$$\frac{\sum_{j=1}^{p-1} [x_{\sigma_j} = x_{\sigma_p}] y_{\sigma_j} + \alpha}{\sum_{j=1}^{p-1} [x_{\sigma_j} = x_{\sigma_p}] + \alpha + \beta}, \tag{10}$$

where we again use a prior parameters α, β.

Replacing categorical features with such type of statistics is a straightforward: we just compute everything in preprocessing stage and use reduce problem to learning of model on numerical dataset — this type of problem, as we discussed in previous section, could be efficiently done on GPU.

4.1. Feature combinations

Note that any combination of several categorical features could be considered as a new one. For example, assume that the task is music recommendation and we have two categorical features: user ID and musical genre. Some user prefers, say, rock music. When we convert user ID and musical genre to numerical features according to 10, we loose this information. A combination of two features solves this problem and gives a new powerful predictor. However, the number of combinations grows exponentially with the number of categorical features in dataset and it is not possible to build all of them on preprocessing stage. So we are doing them during learning in following way. When constructing a new split for the current tree, CatBoost considers combinations in a greedy way. No combinations are considered for the first split in the tree. For the next splits CatBoost combines all combinations and categorical features present in current tree with all categorical features in dataset. Combination values are converted to numbers on the fly.

This part of algorithm is computationally very expensive. Construction of feature combinations on the fly requires us to store each categorical feature in GPU ram and substitute features combination to numbers via equation 10.

Firstly, lets deal with memory consumption and categorical feature storage: memory is limited, categorical features are, originally, not a numbers at all: they are, in general, some text. So for each categorical feature we build perfect hash function and use at most 32-bits to store one feature value. It's still too much, so CatBoost use bit-compression to store perfect hashes for features which are not currently used in computations. To achieve fast access to this features we

are doing compression by blocks: we store several values in one 64-bit integer and compress/decompress via block of 128 threads.

Secondly, we need to build features combinations from several features. One of techniques, which could be used on GPU, is rebuilding of perfect hash: we use perfect hash to store each feature. To combine feature to new one we need to combine two 32-bit hashes in 64-bit one (first feature takes first 32 bits, second feature next 32 bits), sort samples with respect to obtained hash: for this we could use RadixSort, which could be efficiently done on GPU. After that, consecutive runs of the same 64-bit hash will provide us a new 32-bit perfect hash for feature combination (standard GPU technique: mark borders of segments with the same 64-bit hash value and scan vector of borders masks).

Summing up, to maintain perfect hash for feature combinations we need:

— make 64-bit hash from two 32-bit ones,
— RadixSort new hash,
— mark borders,
— scan borders mask.

All this operations could be efficiently done with the help of CUB library. One more implementation note: we don't need to build 64-bit hashes explicitly, as well as do radixSort on all 64 bits.

Now, we'll assume, that we have perfect hash for feature combinations. How we could compute equation (10) on GPU? From equation we see, that it's essentially a segmented primitive: we need to split observations two groups based on category value. Then we need to perform prefix sums for numerator and denominator with respect to time in each group and, based on obtained values, substitute features.

The most convenient and efficient way to split observations in groups on GPU is *radix sort*. Fortunately, this primitives also maintains order of observations in each group. So, if we have x is sorted with respect to σ, that after radix sort of x we will obtain groups of one category runs with maintained time constraint.

After this, just run segmented scan primitive to compute all necessary statistics. CatBoost use segmented scan primitive implemented on top of CUB library scan implementation [5] via operator transformation.

5. BEYOND ONE GPU

CatBoost on GPU support multi-GPU learning as well ass distributed learning on several GPU machines. Our distributed version uses MPI as a fast transport layer to transfer data between several hosts. CatBoost architecture, unlike most of GPU-based MPI applications, is hybrid: we use several GPU for one MPI process. Also we don't need any additional libraries to run on single machine with several GPU.

There are two common ways to learn histogram-based decision trees in distributed setting: we could split samples between devices (so-called sample-parallel learning) or we could split features between devices. Each mode has their own advantages and disadvantages. CatBoost support both modes of parallelization and in the next two paragraphs we will briefly discuss this modes, their pros and cons.

5.1. Sample-parallel learning

Usually, sample parallel learning is the most efficient way to learn standard decision trees[5]. Pros of sample-parallel learning:

[5] If we care about memory consumption and don't store copy of the input data on each GPU devices.

- Dataset and all buffers are splitted between devices.
- Gradient computation and splitting of samples between leaves are parallel: it's too costly to split gradient computations between devices and then reshard data in a such way that every device has all gradient values. This is especially useful for problems with computationally expensive gradient operations, like ranking problems.

Cons of sample-parallel learning

- If features count is much bigger than samples count, than reduce-scatter operation will dominate all computations.
- Samples-parallel mode can't be used for dynamic feature combinations. Categorical features has high cardinality and for efficient build of perfect hashes on GPU we need to store all samples on one device.
- This type of parallelization could not be used with introduced in CatBoost *ordered* boosting approach. It's not a big issue: the most significant improvements for this mode are achieved for dataset with small sample count. Such tasks does not needed several GPUs.

For sample-parallel learning the main network communication are reduce-scatter operation on each level: we need to do reduction of partial histograms 7 stored on each devices, scatter them between devices and use reduced histgorams to obtain the best split condition. For decision trees (oblivious trees), used in CatBoost, it gives us $(1 << \text{level}) \times \text{Bin-factor} \times \text{feature count} \times \text{sizeof(float)} \times 2$ bytes to transfer on each level of tree.

5.2. Feature-parallel learning

The second approach to distributed learning is split data by features. Pros of this approach are:

- Support for dynamic feature combinations: every GPU has access to every sample, so we could use RadixSort to rebuild perfect hashes for feature combintations.
- Support for *Ordered*-boosting scheme.
- Efficient for cases, when sample count is much smaller, then features count.

But, this approach has significant cons:

- Redundant computation: we need to maintain indices of leaves for every sample on every devices, we should compute gradient on each device.
- More memory pressure (duplicate memory usage by labels, weights, indices and temporary buffers).

For feature parallel learning the main network communication are:

- Leaf split: Broadcast for 1 bit per sample from one GPU device to all other
- Greedy feature combinations: broadcast selected categorical feature perfect hash values from one device to all other. Memory need for transfer depends on feature cardinality.

So for datasets with only numerical features we need to transfer (tree depth) × (sample count) bits. For datasets with categorical features and enabled search for feature combinations the data size is upper-bounded by tree depth × sample count × $(1 + 32)$ bits.

5.3. Distributed communication primitives

CatBoost utilize several communication primitives to build search trees: reduce-scatter, all-reduce, broadcast. Currently we use our custom implementation of all this primitives: for reduce-scatter we use tree-based reduce, broadcast is done via our implementation of ring algorithm.

For CatBoost all-reduce is done on very small vectors (like 64 floats), and this data also need to be send to master host CPU ram as well, so we just use CPU for this primitive.

The main advantage of our custom solutions is support for CUDA stream semantics (we could run associate this operation with CUDA stream and overlap memory operations with computation), as well as support all major operation system: CatBoost could work on Linux, Windows, MacOS in single-host, as well as multi-host modes.

6. PERFORMANCE

We made several benchmarks of CatBoost GPU library. Firstly, we compared CatBoost CPU and GPU speed on datasets with only numeric features and on datasets with also categorical ones. Secondly, we test our scalability on Yandex internal dataset for distributed setting: we measure time we need to learn part of production formula on different GPU count with different GPU connections (PCI, 1Gbs network, InfiniBand). Finally, we compare CatBoost with competitors.

6.1. CPU vs GPU speed

For CPU version we used dual-socket server with 2 Intel Xeon CPU (E5-2650v2, 2.60GHz) and 256GB RAM and run CatBoost in 32 threads (equal to number of logical cores). GPU implementation was run on several servers with different GPU types. Our GPU implementation doesn't require multi-core server for high performance, so different CPU and machines shouldn't significantly affect GPU benchmark results. As you can see from figure 4 you could find speed-up one could obtain from GPU compared to CPU, GPU always outperforms CPU: even old K40 almost 6 times faster than dual-socket CPU, while latest NVIDIA Tesla V100 is more then 40 times faster.

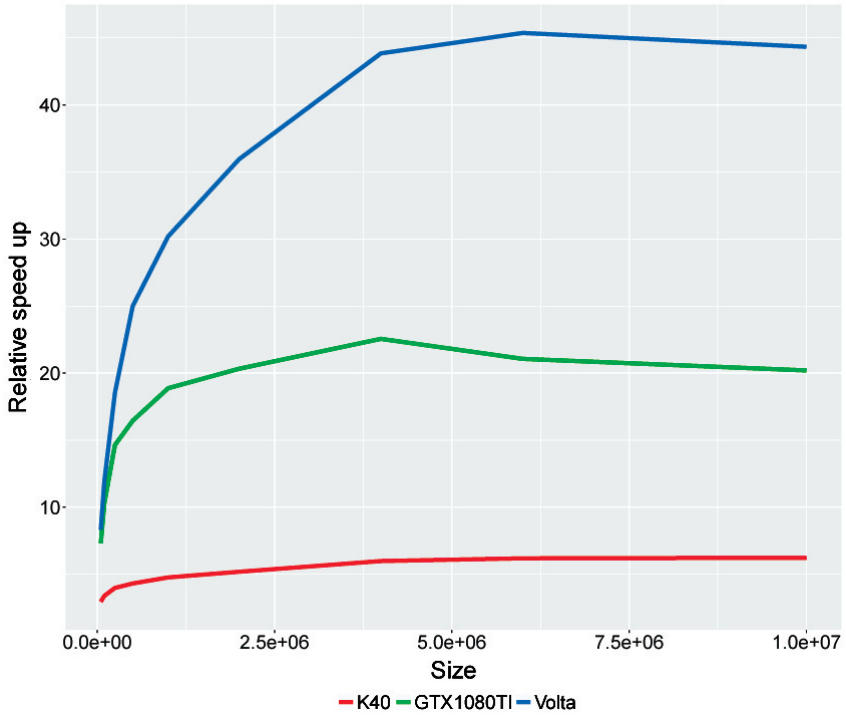

Figure 4. Speed-up from moving to GPU

6.2. Distributed performance

Next, we CatBoost scaling beyond one GPU. We used dual-socket servers with 8 NVIDIA M40 per each host. This servers are connected with Mellanox InfniBand ConnectX-4 and standard 1GBs network. On figure 5 you can CatBoost performance gains from using several GPUs instead of default 4 ones for different connection types.

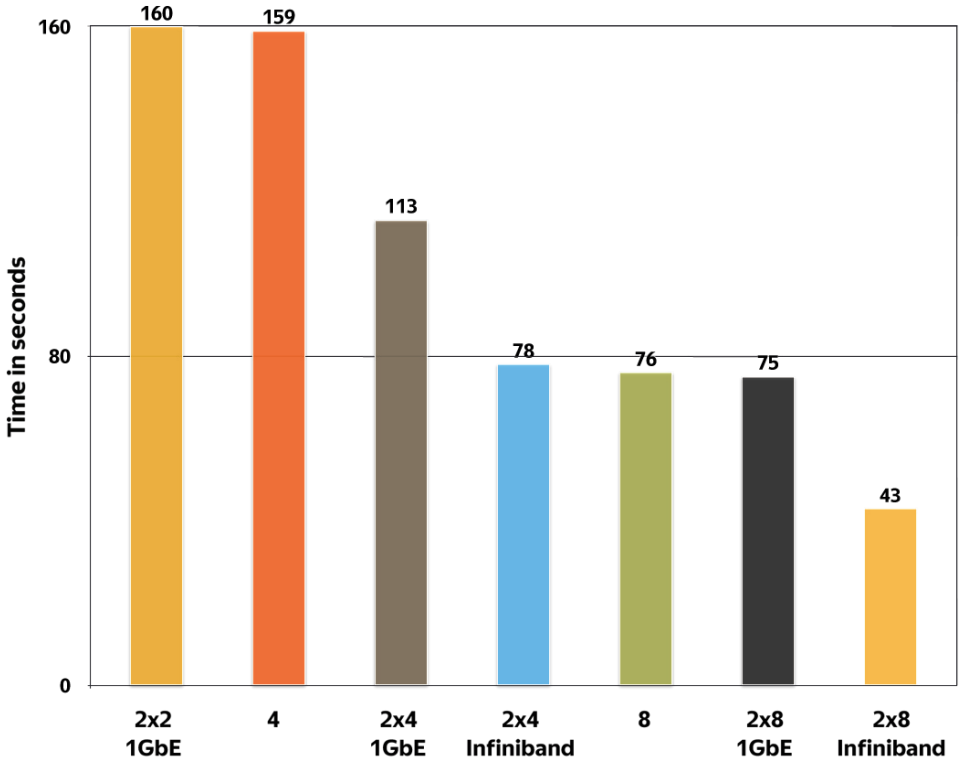

Figure 5. Distributed CatBoost performance

As you can see, with fast interconnection we could achieve significant speed-up for decision tree training on multi GPUs. Even more — if we have enough data (e.g. use more than half of 24GB available RAM on each GPU), we could have performance gains even on slow 1Gbs networks.

6.3. Comparison with competitors

Its very hard to compare different boosting libraries in terms of training speed. Every library has a vast number of parameters which affect training speed, quality and model size in a non-obvious way. Every library has its unique quality/training speed trade-off's and can't be compared without domain knowledge (e.g. is 0.5% of quality metric worth it to train model 3-4 times slower?). Plus for each library it is possible to obtain almost the same quality with different ensemble sizes and parameters. As a result, we can't compare libraries by time we need to obtain certain level of quality. And, for a fair comparison, we should account for time we need to tune parameters, which is also hard to measure.

So we could give only some insights of how fast our GPU implementation could train a model of fixed size. We take the same dataset, which was used during CPU vs GPU benchmarks and train XGBoost, CatBoost and LightGBM on 6×10^6 lines. We run all experiments on the same

machines with NVIDIA GTX1080Ti accelerator, dual-core Intel Xeon E5-2660 CPU and 128GB RAM. For XGBoost and CatBoost we use default tree depth equal to 6, for LightGBM we set leafs count to 64 to have more comparable results. We set bin to 32 for all 3 methods. All algorithms were run with 16 threads, which is equal to hardware core count. We measure time to train ensembles of 500 trees on three datasets: Higgs[6], Epsilon[7], MSLR[8]. Results are presented in figure 6.

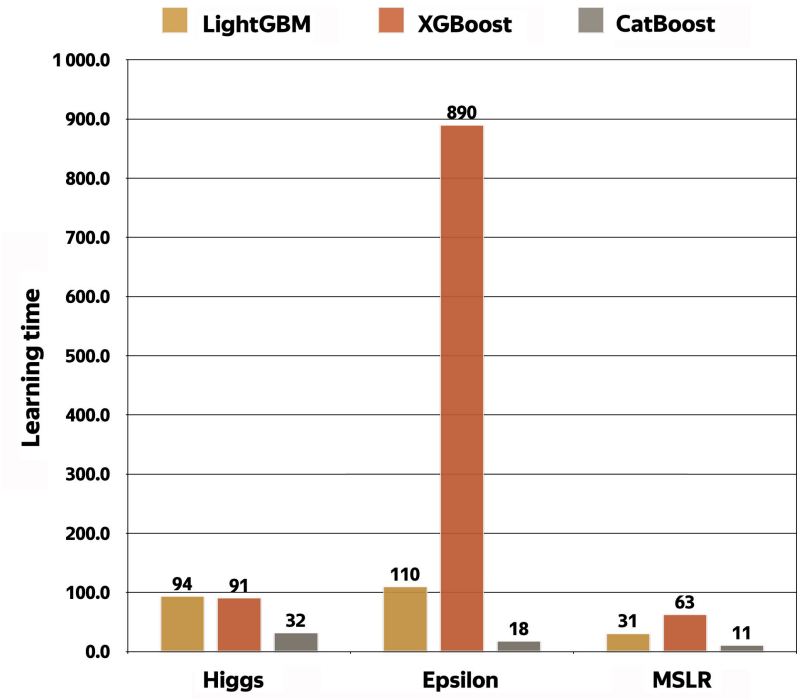

Figure 6. CatBoost vs Competitors performance

7. CONCLUSION

In this article we've briefly describe GPU version of CatBoost library. This library design implementation are based on years of working experience with decision trees on GPU: we had successfully trained decision trees on Fermi GPU when the was no XGBoost at all, and now our efficient implementation is available to everyone to use.

Here discussed basic computation ideas, as well as provide benchmark comparison. There are, definitely, much more in our implementation, than we discuss: custom memory allocators with memory defragmentation designed for decision tree boosting, opaque memory pointer on master thread to allow efficient distributed tree learning, CUDA-stream-like semantic for launching kernel on several GPU not necessary on the one host, sophisticated caching scheme. We hope this insights about GPU implementation will help to design and implement other efficient machine learning libraries.

[6] https://archive.ics.uci.edu/ml/datasets/HIGGS, 11000000 samples, 28 float features.
[7] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html, 2000 float features, 400000 samples.
[8] https://www.microsoft.com/en-us/research/project/mslr/ 136 float features, 3M samples.

References

1. L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, “Classification and Regression Trees,” *Biometrics*, vol. 40, no. 3, p. 874, 1984; doi: 0.2307/2530946
2. R. Caruana and A. Niculescu-Mizil, “An empirical comparison of supervised learning algorithms,” in *Proc. of the 23rd international conference on Machine learning. ICML’06, ACM*, pp. 161–168, 2006; doi: 10.1145/1143844.1143865
3. B. Cestnik, “Estimating probabilities: A crucial task in machine learning,” in *Proc. 9th European Conference on Artificial Intelligence, ECAI’90*, vol. 90, pp. 147–149, 1990.
4. T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proc. of the 22nd ACM acm sigkdd int. conf. on knowledge discovery and data mining*, pp. 785–794, 2016; doi: 10.1145/2939672.2939785
5. D. Merrill, M. Garland, and NVIDIA Corporation, “Single-pass Parallel Prefix Scan with Decoupled Look-back,” [Technical Report], in *Research.nvidia.com*, 2016. [Online]. Available: https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back
6. J. H. Friedman, “Stochastic gradient boosting. Computational Statistics,” *Data Analysis* vol. 38, no. 4, pp. 367–378, 2002; doi: 10.1016/S0167-9473(01)00065-2
7. G. Ke, Q. Meng, Th. 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 (NIPS 2017)*, vol. 30, pp. 3146–3154, 2017.
8. D. Micci-Barreca, “A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problem,” *ACM SIGKDD Explorations Newsletter*, vol. 3, no. 1, pp. 27–32, 2001; doi: 10.1145/507533.507538
9. L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, “CatBoost: unbiased boosting with categorical features,” in *Advances in Neural Information Processing Systems (NIPS 2018)*, vol. 31, pp. 6638–6648, 2018.
10. B. P. Roe, H.-J. Yang, J. Zhu, Y. Liu, I. Stancu, and G. McGregor, “Boosted decision trees as an alternative to artificial neural networks for particle identification,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 543, no. 2-3, pp. 577–584, 2005; doi: 10.1016/j.nima.2004.12.018
11. Q. Wu, C. J. C. Burges, K. M. Svore, et al., “Adapting boosting for information retrieval measures,” *Inf Retrieval*, vol. 13, no. 3, pp. 254–270, 2010; doi: 10.1007/s10791-009-9112-1
12. Y. Zhang and A. Haghani, “A gradient boosting method to improve travel time prediction,” *Transportation Research Part C: Emerging Technologies*, vol. 58, pp. 308–324, 2005; doi: 10.1016/j.trc.2015.02.019

Received 02-04-2022, the final version — 26-05-2022.

Vasily Ershov, Postgraduate at St. Petersburg State University, Head of Cloud ML Tools in Yandex LLC, ✉ noxoomo@yandex-team.ru

Компьютерные инструменты в образовании, 2022
№ 2: 59–73
УДК: 004.62
http://cte.eltech.ru
doi:10.32603/2071-2340-2022-2-59-73

Детали реализации GPU версии библиотеки CatBoost

Ершов В. А.[1,2], аспирант, ✉ noxoomo@yandex-team.ru

[1]ООО «Яндекс», ул. Льва Толстого д. 16, 119021, Москва
[2]Санкт-Петербургский государственный университет,
Университетская набережная, д. 7/9, 199034 Санкт-Петербург

Аннотация

В этой статье мы обсуждаем реализацию графического процессора открытой библиотеки градиентного бустинга CatBoost. Реализация обеспечивают наиболее эффективную производительность на GPU среди общедоступных библиотек, и мы хотим поделиться идеями проектирования и используемыми алгоритмами.

Ключевые слова: *NVIDIA, GPU, градиентный бустинг, деревья решений, GBM, категориальные признаки.*

Цитирование: Ершов В. А. Детали реализации GPU версии библиотеки CatBoost // Компьютерные инструменты в образовании. 2022. № 2. С. 59–73. doi: 10.32603/2071-2340-2022-2-59-73

Поступила в редакцию 02.04.2022, окончательный вариант — 26.05.2022.

Ершов Василий Алексеевич, аспирант СПбГУ, руководитель службы инструментов машинного обучения в ООО "Яндекс", ✉ noxoomo@yandex-team.ru