Optimizing Random Forests on GPU

Derrick Cheng
John F. Canny

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-205
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-205.html

December 1, 2014

Acknowledgement

I want to thank Professor Canny for his tremendous help throughout this project, and for being a great advisor. I enjoyed working together with him on this project and have learned a lot from doing so. I also wanted to thank Professor Trevor Darrell, for taking the time to read my paper and for giving me great advice and feedback.

Optimizing Random Forests on GPU

by Derrick Cheng

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor J.F. Canny
Research Advisor

(Date)

* * * * * * *

Professor T. Darrell
Second Reader

(Date)

Abstract

We have designed *BIDMachRF* – an implementation of Random Forest with high CPU and GPU throughput and with full scalability. It is based on parallelism, maximal work by each datum, reduction of unnecessary data access, sorting, and data compression. *BIDMachRF* is optimized for GB sized large datasets, and our goal is to be 10-100x faster than SciKit-Learn Random Forests and CudaTree on these large datasets. *BIDMachRF* is currently a work in progress. This paper describes the current state of our implementation as well as points for improvement, which we have identified through benchmarks on classical datasets. Our current in progress version has already shown to be 5x faster than implementations such as SciKit-Learn on large sized GBs of data and is estimated to be at least 20x faster than those implementations when complete.

Optimizing Random Forests on GPU

Derrick Cheng & John F. Canny
derrick@cs.berkeley.edu & jfc@cs.berkeley.edu

1 Introduction

In this paper we focus on a high CPU and GPU throughput implementation of random forests optimized for large GB sized data sets. This implementation should have high performance and be fully scalable (i.e. it can process arbitrarily sized data sets). Our implementation, *BIDMachRF*, is a work in progress. This paper describes in detail the state our implementation currently, along with several benchmarks, as well as next steps requisite to improve its performance.

2 Background and Importance

Random forest is one of the most widely used classifiers in a variety of domains due to its high accuracy and ease of use [2] [3] [6]. *BIDMachRF*, is part of the BID Data Suite ("a collection of hardware, software and design patterns that enable fast, large-scale data mining at very low cost. It allows for single-machine performance levels that equal or exceed reported cluster implementations for common benchmark problems" [1]).

3 Related Work

There has been work done on large datasets to make Random Forests run on the CPU. WiseRF [6] and SciKit-Learn [1], which have Random Forest implementations, are two examples. WiseRF can successfully model GB+ worth of data in seconds. There have been a number of recent works on running Random Forests on the GPU. However, out of implementations that use the GPU, CudaTree [5] is of the recent leading implementations. CudaTree's implementation is highlighted by its hybrid approach of constructing trees. It first starts tree construction using a depth-first based algorithm; however, it completes by using a Breadth-First algorithm to grow smaller subtrees. It determines the crossover point for this switch using a linear model.

Our ultimate goal is to have *BIDMachRF* achieve at least a 10X magnitude gain on top of implementations such as CudaTree, WiseRF, and SciKit-Learn on large datasets.

4 The Implementation

4.1 Overall Design

In order to allow for high GPU/CPU throughput for large datasets, we base our design on the following principles:

[1] `http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`

1. Find parallelism
2. Each datum in the input does maximum work
3. Avoid unnecessary data access
4. Use sorting to organize data
5. Condense data where possible to remove redundancy

In random forest, many decision trees are made. To allow for high parallelism, *BIDMachRF* builds the trees, breadth first, one level at a time for all trees simultaneously. This allows us to process much larger chunks of data to attain full throughput. Making a forest of these trees allows us to vote across the result of each tree during classification. A decision tree is a binary tree, where each node represents which feature number to split on and what threshold value for that feature to split on. Decision trees are trained level by level. Data points are advanced throughout the tree, moving from one node to another between each level iteration. For each node, the data points currently at each node are used to determine the best feature and best feature value to split on based on impurity reduction algorithms such as Gini Impurity Reduction and Information Gain [4].

Our system consists of the following main components: packing, sorting, compression, aggregation, impurity calculation, updating data for trees regarding best thresholds, and iterating the node position.

4.2 Key Terms

$BIDMachRF$ = Our implementation of Random Forest
G = GPU Spaced Used in Bytes
t = # trees
ns = # potential of features to test per node
c = # of categories
d = depth
fv = bits allocated toward the feature value for packed values
nnz = # of non-zero entries in the sparse representation of the c x n categories matrix
n = # of data points

4.3 Overall Implementation Algorithm

Data: treenodes = t x n to store node position number; treesData = 4 x $t * 2^d$ matrix which stores information such as which feature to split on, what value to split on, majority category for node, and whether it is a leaf
Result: treesData now has complete information about which feature and thresholds to split on
while *current depth level is less than d* **do**
  for *block in blocks* **do**
    Pack values;
    Sort packed values;
    Compress sorted packed values;
  end
  Aggregate packed values from the blocks;
  Find min impurities;
  Update treesData using minImpurity information;
  Iterate each data point to next node;
  Increment current depth by 1;
end

Algorithm 1: The overall training algorithm

4.4 Packing values

Given which node each data point is on and which tree each data point is on, we generate an array of packed values, where each packed value is represented by 6 values: *tree index, node index, feature sample index, feature index, feature value, category value*. This packed value can be represented by 64 bits, sufficient bits to encode this information. In order to allow for histogramming, we can easily tune how many bits are given to the *feature value*, because the *feature value* stored in this packed long is a value scaled to fit in its alloted number of bits. This is because we later compress the values outputted from this step. (See Section 4.6). The *feature index* is determined by hashing the *tree index*, *node index*, and *feature sample index*.

When running this step on the GPU, we use the following amount of space to contain the input and output matrices for this step with this equation:

$$G = [(4 * f * n) + (3 * 4 * n) + (4 * t * n)] + [(8 * t * ns * n)] = [4f + (8 * t * ns) + 12] * n$$

This is determined given:
Input: Feature Matrix (f x n), Sparse Category Matrix (c x n), and a node position Matrix (1 x n)
Output: Long Array (t x ns x nnz)

On an examples dataset such as MNIST where:

$$f = 784,\ n = 60000,\ t = 64,\ ns = 32$$

$$G_{digits} = [4f + (8 * t * ns) + 12] * n = [4 * 784 + (8 * 64 * 32) + 12] * 60000 = 1.09144GB$$

Though meant to be run on the GPU, the current single threaded CPU version has a runtime of:

$$O(n * t * ns)$$

We compute the packing of many trees at once to fully exercise the data. On the GPU we achieve a throughput of 14 GB/s, while on the CPU we achieve a throughput of 0.366 GB/s.

4.5 Sorting

We can run a simple sort on the the Long array outputted from the packing step. This organizes the packed values in such a way that allows easy impurity calculation. This sort is very fast, because it uses the radix sort from Thrust [2].

Space usage on the GPU to contain the Long t x ns x nnz Array outputted from the previous packing step:

$$G = 8 * t * ns * nnz$$

This GPU sort has a throughput of 2.2 GB/s, while if done instead on the CPU has a throughput of 0.067 GB/s.

4.6 Compression

The result of tree pack will potentially have many overlaps when there is a large data set, thus we can represent this array as just a list of unique values along with a count of the number of occurrences of each value.

When run on the GPU, the amount of GPU sapce used would be (worst case):

$$G = 2 * (8 * t * ns * nnz) + min(8 * t * ns * nnz, 4 * n)$$

We confirmed in actuality that it compresses from 10 - 100 times, from running on the SpamBase dataset, leading to a more realistic GPU usage of:

$$G = 2 * (8 * t * ns * nnz) + min(0.1 * (8 * t * ns * nnz), 4 * n)$$

The throughput on the GPU is 15.8 GB/s.

4.7 Aggregation Step

We need the whole data set to train the trees, but the steps described above – pack, sort, and compression – are not always guaranteed to fit onto the GPU. We resolve this by mini-batching the data set into blocks. The size of the block is determined by what can maximally fit onto the GPU. We base the size of each block off of our calculations of space on the GPU needed for the compression, sorting, compression steps.

Due to mini-batching, we add another step (an aggregation) where we merge the resultant arrays produced by the compression together on the CPU. This merge is quick because each

[2] `http://docs.thrust.googlecode.com/hg/group__sorting.html`

of the arrays are already sorted. This merge step is currently done on the CPU sequentially; however, in the future, can be done more efficiently with multi-threading.

4.8 Impurity Reductions

After the previous steps, given that our compressed packed values are sorted and grouped according to *tree, node index, and feature sample index*, and within each group all the packed values are sorted by threshold value, it makes it easy to identify which threshold is the best. *BIDMachRF* calculates the impurity reductions for both Entropy and Gini impurity reductions.

Entropy Impurity:

$$i(N) = -\sum_j P(\omega_j) log_2 P(\omega_j)$$

Gini Impurity:

$$i(N) = \sum_{i \neq j} P(\omega_i) P(\omega_j) = 1 - \sum_j P^2(\omega_j)$$

Impurity Reduction:

$$\delta i(N) = i(N) - P_L i(N_L) - (1 - P_L) i(N_R)$$

GPU space usage assuming worse case for the size of the compressed packed values:

$$G = [(8+4) * min(t * 2^d * ns * 2^{fv} * c, 4 * n)] + [5 * (ns * t * 2^d)] = O(2^{d+fv})$$

This is based on:
Input: The keys and counts produced by the merge step of the compressed packed values, at the worse case when all values are unique (therefore the 2^{fv})
Output: Four ns by $t * 2^d$ matrices, which stores data about the best impurity for all possible features that could be selected for each node of the tree, which currently has data points residing on them.

Example for reasonably sized data set:

$$f = 784,\ n = 60000,\ t = 64,\ ns = 32,\ fv = 8,\ c = 10,\ d = 11$$

$$G = [(8+4) * min(64 * 2^{11} * 32 * 2^8 * 10, 4 * 60000)] + [5 * (32 * 64 * 2^{11})]$$

$$= 12 * min(10737418240, 240000) + 20971520 = 0.0222135 GB$$

From the example calculation it is easy to identify that the majority of the space usage on the GPU is from the compressed packed values.

On the GPU, this method runs at 2.79 GB/s. On the CPU it runs at 0.198 GB/s.

With the output from the Impurity Reduction step, for each node, we mark the feature and threshold that corresponds to the best impurity gain at that node. Nodes, whose best split impurity gains are not larger than a preset threshold, are marked as leaves.

This is done on the CPU, and the single threaded run time of this algorithm on the CPU is:

$$\theta(ns * t * 2^d)$$

4.9 Tree Steps

This is the final step. It is to traverse the data points down to the next level of the tree, as determined by the new information aggregated at the previous step from Section 4.8. For example, if for a specific data point, if its feature value uses less than or equal a specified value, it will iterate to a child node on the left, else it will iterate to the child node on the right.

This is fast enough to done on the CPU, since the run time is $\theta(n)$ where $n = \#$ data points.

4.10 Classification

Classification is done completely on the CPU, and has a runtime of: $\theta(d*n+t*n)$ to account for tree traversal as well as voting.

5 Benchmarking

We ran our implementation on a variety of classical data sets. We measured the accuracy, speed, and GPU space used. Through this benchmarking, we have identified what is working well in our in progress *BIDMachRF* implementation as well as points to improve on.

5.1 Expected Target Time

Given the GPU/CPU throughput of each of the steps described in Section 4, we can calculate the time that *BIDMachRF* (when in final form) is expected to run by considering the sum of the runtimes of each step given a specific data set. Let us consider MNIST Large Data set from Table 1. We run with 8 trees, 128 samples, and a max depth of 11, and we expect $64.89s$ runtime, given that the most overpowering steps such as sort has an aggregate runtime of $16.06s$ on the GPU, and the currently single threaded CPU aggregation steps take an aggregate of approximately $38s$.

5.2 Test Machine Specifications

BIDMachRF was benchmarked on a our data engine prototype, which has an 8-core CPU (Intel E5-2660 @ 2.2GHz) with 64GB ram, two dual-GPUs (Nvidia GTX-690), i.e. 4 independent GPUs., and 20 x 2TB Sata Disks.

5.3 Test Datasets

Data Source	# Train Examples	# Feats	# Categories	# Test Examples
MNIST[3]	60000	784	10	10000
MNIST Large[4]	1000000	784	10	1100000
Spambase[5]	3065	57	2	1536
Covertype[6]	290506	54	7	290506

Table 1: Datasets used for benchmarking

DataSet	Classifier	Acc	Train Time	Test Time	d	# t	ns
MNIST	SciKit-Learn	0.932	5.60s	0.036s	11	8	32
	CudaTree	0.948	3.94s	0.17s	N/A	8	32
	BIDMachRF	0.925	**5.208s**	0.117	11	8	32
	Target *BIDMachRF*	—	1.67s	—	11	8	32
Large MNIST	SciKit-Learn	0.919	1134.184s	8.21s	11	8	128
	CudaTree	fail	fail	fail	N/A	8	128
	BIDMachRF	0.894	**264.2s**	56.794s	11	8	128
	Target *BIDMachRF*	—	64.89s	—	11	8	128
SpamBase	SciKit-Learn	0.939	1.76s	0.012s	11	64	32
	CudaTree	0.946	2.98s	0.091s	N/A	64	32
	BIDMachRF	0.928	6.63s	0.126s	11	64	32
Covertype	SciKit-Learn	0.877	138.17s	1.66s	15	32	32
	CudaTree	0.960	30.47s	3.06s	N/A	32	32
	BIDMachRF	0.71	**67.27s**	**2.289s**	15	32	32

Table 2: Results on running SciKit-learn, CudaTree, and *BIDMachRF* on each data set from Table 1 with similar parameters. All was run on our same data engine prototype. Target *BIDMachRF* is the goal runtime of our system. Bolded is where our implementations is either quicker than SciKit-Learn or CudaTree.

5.4 Results and Comparison

As can be seen from Table 2, we notice that *BIDMachRF* is in general lacking in accuracy. For smaller datasets, if BIDMachRF is faster, it is not much faster. However, on larger datasets, its runtimes are faster than both SciKit-Learn and CudaTree.

In terms of accuracy, CudaTree is the clear lead in terms of accuracy, with 1-2% better accuracy than SciKit-Learn for SpamBase and MNIST. CudaTree is more than 8% more accurate than SciKit-Learn for the CoverType data set. *BIDMachRF* is less accurate than SciKit-Learn and CudaTree for all the dataset, especially on CoverType where it has a 71% accuracy, while CudaTree has a 96% accuracy. Inspection of the CoverType dataset reveals that CoverType has a category distribution amongst its training set that is uneven. *BIDMachRF*'s accuracy can be improved by incorporating better sampling methods.

BIDMachRF is in general slower than both SciKit-Learn and CudaTree the smaller datasets, such as the smaller MNIST dataset and SpamBase. However, on larger sized datasets *BIDMachRF* triumphs as it should. *BIDMachRF* is faster than SciKit-Learn on CoverType as well as the large MNIST. SciKit-Learn takes 1134s, while *BIDMachRF* takes 264.2s. This is promising because it shows that *BIDMachRF* is optimized for large sized datasets.

[3] `http://yann.lecun.com/exdb/mnist/`

[4] mnist8m from `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html`

[5] `https://archive.ics.uci.edu/ml/datasets/Spambase`

[6] `https://archive.ics.uci.edu/ml/datasets/Covertype`

Also notable is that CudaTree fails for the Large MNIST dataset, which has millions of training examples. This failure is due to the fact that CudaTree can only support dataSet size that fits in GPU memory[7]:

$$G_{memory} = dataset + 2 * samples * features * ceil(log2(samples)/8) + samples * features$$

Both *BIDMachRF* and SciKit-Learn are not limited by this. As explained in Section 4.7, *BIDMachRF* uses blocking to allow data to fit onto the GPU.

5.5 Improvements and Next Steps

To identify where we can make improvements to the current *BIDMachRF* implementation, we ran *BIDMachRF* on the MNIST dataset. This profiling was done on a run of *BIDMachRF*, using the GPU instead of CPU whenever possible. In this profiling, we took note of several aspects of each step of training: allocation time (time taken to allocate and transfer data on to the GPU), run time (time taken to run each step on the GPU or CPU), and and deallocation time (time to free memory on the GPU).

The step that took unnecessary amount of time was allocation for the packing step. We saw that there was notable time taken in allocation time. Meaning that between each step we allocate data on the GPU, transfer data to the GPU, process it, transfer it back to the CPU, and then free the data on the GPU. There are many points in which we could just leave data in the GPU to avoid having to re-transfer it. This inefficiency was done in the interest of ease of implementation and will be resolved in *BIDMachRF*'s continued final state.

Many of the steps that are run solely on the CPU can be multi-threaded. As mentioned before in Section 5.4 sampling can be included to increase accuracy. In addition, sampling can also be used to reduce the amount input data for very large datasets. Eventually, this implementation will be part of the BIDMach `https://github.com/BIDData/BIDMach`. Thus, it will be incorporated into the learner framework. Doing so will allow *BIDMachRF* to use memory caching, thus allowing us to recycle both CPU and GPU memory to reduce the total amount of memory used. This will also reduce the amount of unnecessary allocation of GPU memory, which has shown to be the largest time sink.

6 Conclusion

We have described in detail *BIDMachRF*, a high performance and highly scalable Random Forest implementation built on the principles of parallelism, maximal work by each datum, reduction of unnecessary data access, fast sorting, and data compression. As can be seen in this paper, following these principles has allowed for high GPU/CPU throughput. Our current in progress version of *BIDMachRF* has already shown to be 5x faster than implementations such as SciKit-Learn on GBs of data. We estimate it to be at least 20x faster than SciKit-Learn when in its complete state. Other GPU implementations such as CudaTree, fail at handling indefinitely sized data sets. In addition, we have run a series of benchmarks

[7] `https://github.com/EasonLiao/CudaTree`

on a wide range of data sets that has allowed us to identify points of improvements and next steps.

References

[1] Canny, J., and Zhao, H. Big data analytics with small footprint: Squaring the cloud. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, ACM (New York, NY, USA, 2013), 95–103.

[2] Chen, X., and Ishwaran, H. Random forests for genomic data analysis. *Genomics 99*, 6 (2012), 323–329.

[3] Cutler, D. R., Edwards Jr, T. C., Beard, K. H., Cutler, A., Hess, K. T., Gibson, J., and Lawler, J. J. Random forests for classification in ecology. *Ecology 88*, 11 (2007), 2783–2792.

[4] Duda, R. O., Hart, P. E., and Stork, D. G. *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000.

[5] Liao, Y., Rubinsteyn, A., Power, R., and Li, J. Learning random forests on the gpu.

[6] Richards, J. W., Eads, D., Bloom, J. S., Brink, H., and Starr, D. Wiserftm: A fast and scalable random forest.