Parallelizing Testing of Random Forest

Vardaan Kishore Kumar
Department of Computer Science
Stevens Institute Of Technology
Hoboken, NJ 07030
vkishore@stevens.edu

Abstract

Random forest is a collection of decision trees built to exploit the concept of bagging to give the best result on classification datasets. Here we try to parallelize the testing of random forest, in accordance with amdhal's law we can argue that random forest can be completely parallelized. There are two versions presented here with more than 10x speed in the CUDA version with different configuration of threads and blocks in both,

1 Introduction

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of over fitting to their training set[7] [5] [3].
The training algorithm for random forests applies the general technique of bootstrap aggregating, or bagging, to tree learners. Given a training set X = x1, ..., xn with responses Y = y1, ..., yn, bagging repeatedly (B times) selects a random sample with replacement of the training set and fits trees to these samples.This bootstrapping procedure leads to better model performance because it decreases the variance of the model, without increasing the bias. This means that while the predictions of a single tree are highly sensitive to noise in its training set, the average of many trees is not, as long as the trees are not correlated. Simply training many trees on a single training set would give strongly correlated trees (or even the same tree many times, if the training algorithm is deterministic); bootstrap sampling is a way of de-correlating the trees by showing them. different training sets.

2 Problem Tackled and Amdhlas's Law

Here the problem being tackled is that of the testing of random forest. In a real world scenario training can take as much time as needed but testing in real time should be as fast as possible. In addition to this there is an integration to the Scikit-Learn package. This can help in fast prototyping of the algorithm with support on CUDA. The scikitlearn package currently does not support CUDA and this can be a small step in helping test algorithms in cuda. The decision trees in the forest are built independently of each other and the data points being passed into the trees are also independent of each other thus this allows us to argue that we can completely parallelize the algorithm.

3 DataSet

Here a synthetic dataset is being used to generate dataset of 1,000,000 points with 50 features and 3 classes using the inbuilt make-classification function of scikit learn. Out of this we set 40 percent of

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

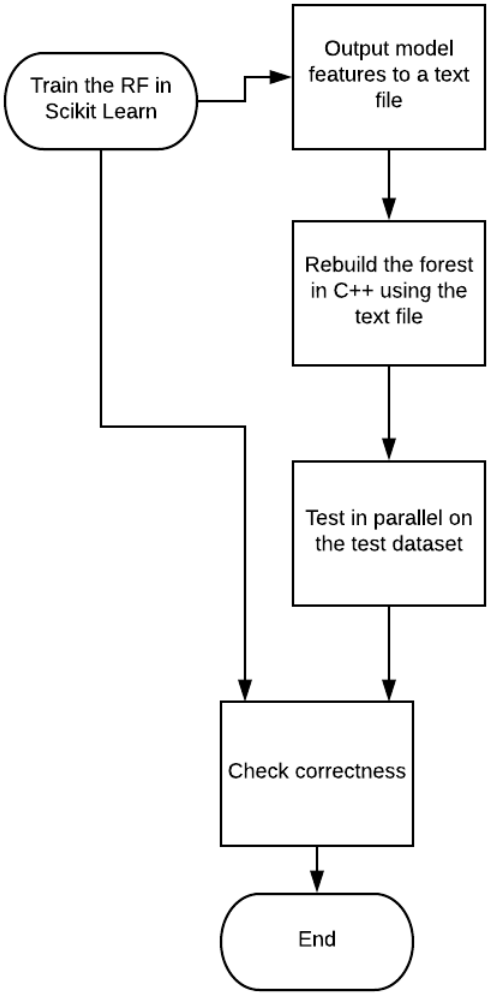

Figure 1: How the algorithm works

the data for training and the rest for testing of the algorithm.
The data can be visualized as seen in figure 2.

4 Python Side

On the scikit learn side, we first train the dataset mentioned on forest's with 50 and 100 estimators respectively, the main goal is to make sure the algorithm is as fast as possible so there is not a lot of focus on the accuracy of the algorithm. We output the values of the most important features to help us build the tree on the C++ side[6][2]. These features are namely.

- Feature the node is split on.
- Threshold of the node.
- the number of children on left and right.
- The division of classes at each level.

All these are printed to a text file.

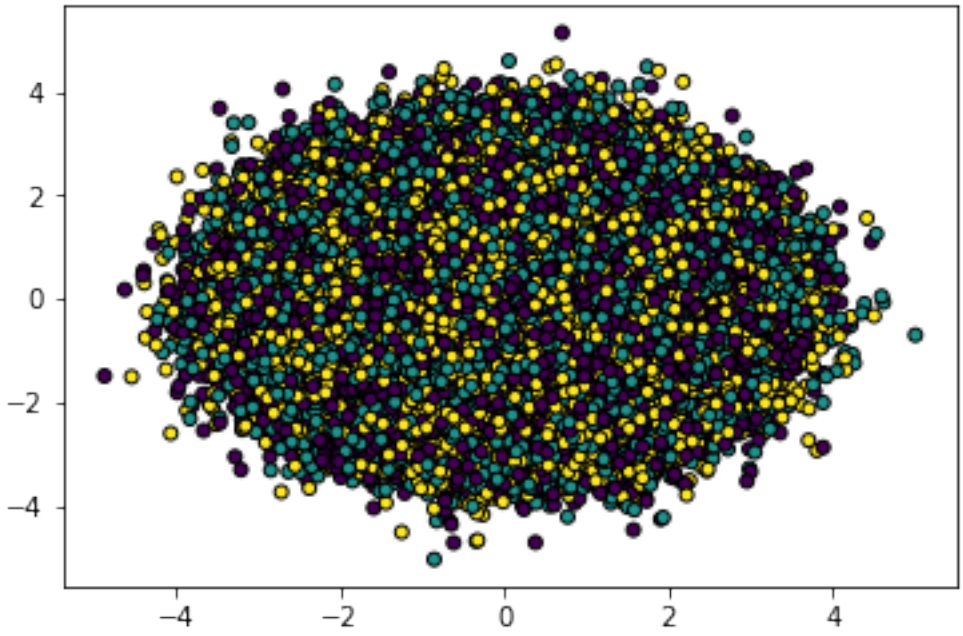

Figure 2: Dataset

5 C++ Side

5.1 CPU Version

Initially everything was built on the CPU with no CUDA involved, this will help us make the move to CUDA faster and easier. The first step is to create a few helper functions like the input reader using the ifstream. We also define functions for getting the number of rows and the columns in the test data, this can help build a level of abstraction independent of the data. Then we create a Decision tree class with the the features mentioned above. This will help us create many objects of the class namely the different trees of the forest independently. We initialize the left and right nodes, and also declare a function to load the model from the text file, this are contained in the Decision tree class. Next we define a random forest class with the decision tree class included in it along with the two parameters, namely the number of estimators and number of classes in the data. We open an ifstream read in the first two lines of the text file which have the two previously mentioned parameters. Next based on the number of estimators we create objects of the decision tree class. After this is done we go on to reconstruct the tree by now reading into each object its own tree and the parameters of the tree iteratively(this is one of main bottle necks as the size of the file is close to half a gb).After the tree has been constructed on the we move on to loading the test dataset generated from the scikitlearn function.
After this we go on to the prediction part we first create a empty two dimensional array to store the results, we pass this along with the data in a for loop to the predict function, this predict function is very similar to figure two. What happens here is that we check if the value of the feature from the dataset is greater or lesser than the threshold of the feature the tree is built on, if greater we move to the right else we move to the left and when we reach the leaf node we just return the class of that node. This procedure is followed for each datapoint for each tree, the max(majority)is stored to be later written to a file to check correctness.

So from this we can quiet clearly see how this can be ported to a GPU, we can assign one thread to one point and load a tree and test the points in parallel. This is what is done in the GPU verion. The run time is mentioned in Table 1 .

```
# Make a prediction with a decision tree
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']
```

Figure 3: The predict function of Random Forest built from scratch in python,image from https://machinelearningmastery.com/implement-random-forest-scratch-python[1]

5.2 GPU Version 1

From the above version we can quiet clearly see how this can be ported to a GPU. In the GPU verison we keep all of the functions from above and try to parallelize the function from figure 3.

Coming to the work allocation of the threads we assign one thread per point and also initialize the number of threads and number of blocks to ensure the same, this ensures that we are running the gpu at maximum occupancy. In the gpu we load one tree at a time into gpu memory(not shared as it still has memory issues). After the results are computed for one tree we discard it and then load the next in the memory and again compute the result. This is done till we exhaust the number of estimators in the forest. This version is not very different from the cpu version other than copying trees from cpu to gpu one by one and assigning on thread per point to be calculated in parallel.

- Blocks =391
- Threads = 1024
- Samples per thread =1

The occupancy and the memory results are presented in figure 5 and figure 4. As we can see the gpu

```
vardaan@vardaan-Blade:~/Downloads/cuda_rf/rf_cuda_2 (copy)$ nvcc -Xptxas -v Rand
omForest.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z19_DDT_computePredictP4NodePdS1_iiii
Pi' for 'sm_30'
ptxas info    : Function properties for _Z19_DDT_computePredictP4NodePdS1_iiiiPi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 21 registers, 368 bytes cmem[0]
```

Figure 4: xptxas output for first version

is running at maximum occupancy.

5.2.1 Issues with the current version

In the above version we are chasing pointers and also there also is a bottleneck of the loading time of the model, these can be an issue if this system is put to use in real time. We can solve the first issue by flattening the trees into arrays and this leads us to the second version, there is still the bottleneck of loading the model into memory.

5.3 GPU Version 2

In this version we flatten all the trees into arrays and load them all into memory at once and then we change the allocation of threads and blocks as well. In the first version we load one tree at a time

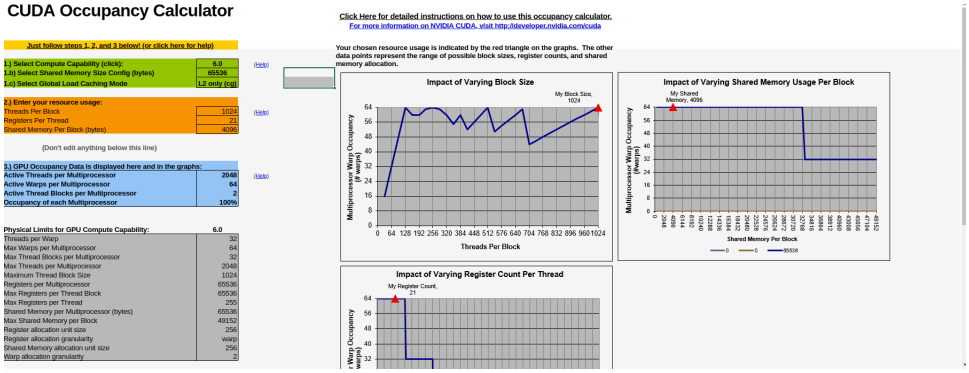

Figure 5: Occupancy results

and use all threads for testing in parallel. Here we assign one block to one tree and we assign 40 points for one thread and ensure that the blocks are running at full capacity. This helps minimize the loading and unloading time of trees into memory.Another improvement done is that the votes are also calculated in parallel here.The memory results are presented in figure 6 , the occupancy is similar to figure 5.

- Blocks=10
- Threads = 1024
- Samples per thread = 40

```
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_ZN71_GLOBAL__N__47_tmpxft_0000103e_00000000_8_RandomForest_cpp1_ii_ab31213817_ddt_computeVotesEiiiiPiS0_' for 'sm_30'
ptxas info    : Function properties for _ZN71_GLOBAL__N__47_tmpxft_0000103e_00000000_8_RandomForest_cpp1_ii_ab31213817_ddt_computeVotesEiiiiPiS0_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 25 registers, 352 bytes cmem[0]
ptxas info    : Compiling entry function '_ZN71_GLOBAL__N__47_tmpxft_0000103e_00000000_8_RandomForest_cpp1_ii_ab31213819_ddt_computePredictEP12DecisionTreePfiiiiiPi' for 'sm_30'
ptxas info    : Function properties for _ZN71_GLOBAL__N__47_tmpxft_0000103e_00000000_8_RandomForest_cpp1_ii_ab31213819_ddt_computePredictEP12DecisionTreePfiiiiiPi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 21 registers, 368 bytes cmem[0]
```

Figure 6: xptxas output for second version

5.3.1 Issues with version 2

There is still the issue of loading the tree into shared memory, that needs to be analyzed, the first bottleneck of loading trees into memory still remains.

5.4 Final Results

Here we present the results of the 4 versions averaged over 10 runs
The accuracy's of the trees are also presented in table 2.

Results(s)	100 Trees	50 Trees	Ratio
Python	15.6	7.6	2.0
CPU	12.27	7.05	1.74
GPU-1	1.36	0.75	1.8
GPU-2	1.24	0.69	1.8

Table 1: Run Time of all the version in Seconds

5.5 Conclusion

In this report we present a parallel version of the random forest algorithm and see that we are able to speed it up by close to 12x on version two and 11x on the first version of the gpu on the 100 trees. This show the suitability of random forest algorithm for the gpu and can be used in testing. The code for this will be available with full integration in a few days after cleaning on the following link https://github.com/naadvar

Accuracy	100 Trees	50 Trees
scikit learn	94.0	85.2

Table 2: Accuracy of the different tree configurations

5.5.1 How this is different from current CUDA random Forest

The current version present on github have no integration with scikit learn and they also build everything from scratch including the training of the algorithm [4].Their allocation of threads is also different, they use one thread for one tree.

5.6 Future Work

The future work on this can be done by reading the file in parallel,if it is possible, also need to see how much of an improvement shared memory gives to the original version. We can also try to make the training of random forest in paralle and try to integrate it to scikit learn.

References

[1] Jason Brownlee. https://machinelearningmastery.com/implement-random-forest-scratch-python/. 2016.

[2] core. https://github.com/1core2life/random$_forests$. 2016.

[3] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[4] Håkan Grahn, Niklas Lavesson, Mikael Hellborg Lapajne, and Daniel Slat. Cudarf: A cuda-based implementation of random forests. pages 95–101, 12 2011.

[5] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[6] ishita. https://github.com/ishitatakeshi/randomforest. 2016.

[7] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, Aug 1998.