Random Forest Implementation on GPU

Rahul Vashisht (CS18D006)
Shahbaz Husain (CS18M049)
Ankur Yadav (CS18M063)

1 May 2019

Contents

1 Problem Definition **3**

2 Why Random Forests ? **3**

3 How it works ? **3**

4 Serial Implementation **3**

5 Parallel Implementation **3**

6 Initial ideas for Parallelization **4**

7 Current Progress **5**

7.1 Data Preprocessing 5

7.2 Serial Code Implementation 5

7.3 Parallel Code Implementation 7

7.3.1 Naive Parallel: 7

7.3.2 Optimized Parallel 8

8 Observations and Results **9**

9 Future Work **10**

10 Code Link **10**

1 Problem Definition

Random forest is an ensemble learning method for classification and regression. It operates by constructing multiple decision trees during training and makes inference by using all decision trees. The prediction of class is done by taking majority of all decision trees. Henceforth, it helps in reducing the over fitting in decision tree.These methods are easily parallelizable due to their inherent parallel nature as training of one decision tree does not effects the training of other decision trees. The parallel implementation gives better efficiency than it's sequential implementation.

2 Why Random Forests ?

A decision tree is a graphical representation of all the possible solutions to a decision based on certain conditions. It is called a decision tree because it starts with a root node, which then branches off into a number of solutions, just like a tree. The decision tree suffers massively because of over fitting (low bias and high variance). Random forest avoids over fitting by building multiple decision trees on sub sample (bootstrap sample) of data. It give more accurate and stable predictions. The problem with random forest is it can be slow, thus the time efficiency of random forest can be improved using parallel implementation.

3 How it works ?

Random Forest is a supervised learning algorithm. It adds additional randomness to the model, while growing the tree. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.
Therefore, in Random Forest, only a random subset of the features is taken into consideration by using some criterion for splitting a node.

4 Serial Implementation

There is no serial code available in C. Benchmark will be decided by Mentor/ Course Instructor.

5 Parallel Implementation

There is no parallel code available.
There are many high-quality CPU implementations of Random Forests, such as wiseRF, SPRINT, PyCuda[1] and Random Jungle. Random Forests may appear to be an ideal candidate for GPU parallelization.

6 Initial ideas for Parallelization

- The naive parallel implementation in which each thread block is used to generate one decision tree.
- The data level parallelism in which each thread block finds a threshold for splitting a sub sample of data from a single attribute. Each thread block is responsible for subset of the samples from single feature. We can calculate count of unique values feature is taking to calculate entropy in parallel. After this we can determine which feature has highest information gain. This implementation is known as Depth First tree construction. This algorithms works very fast in early stages of creating a decision tree. As depth of tree increases number of data points per node decreases and overhead of invoking kernel dominates.
- The task level parallelism in which each thread block creates one level of the decision tree. Instead of constructing single node per kernel launch we can create one level of decision tree. Each Cuda thread block is tasked with computing the optimal split for a single tree node in the current level. This implementation is known as Breadth first tree construction. This method is less efficient at the top of a tree but can significantly decrease the kernel launch overhead by processing many sites on a tree at the same time.
- The building of decision tree can further be parallelized by splitting the work of finding the optimal threshold for splitting of node. We can also achieve speed up by considering a hybrid technique combining both breadth first construction with depth first construction.

7 Current Progress

7.1 Data Preprocessing

We are using "Mental Health in Tech Survey" data set as a classification data set here. This data-set contains 27 attributes and 1259 data points. We are doing the reprocessing of data imputing missing values by mode of the variable for categorical variable and imputing missing values by mean of the variable. Also we are removing the attribute "Timestamp" as it's entropy is always very less because each point has one distinct value for this attribute hence it will always be chosen as for as a splitting attribute. Also we are removing the attribute "Comments" because number of missing values are large and it is not possible to impute these missing values also. The prediction label considered for training of data is "Treatment" attribute which takes fifty one percent yes values and forty nine percent no values. **Also we have converted all categorical attributes into numerical attributes**. We have divided the data into three sets train set, validation set and test set. After doing all the preprocessing steps we have 944 training points with 25 attributes which will be further divided into 70:30 ratio into training and validation set.

7.2 Serial Code Implementation

The random forest algorithm requires us to bootstrap the data (sampling with repetition) and we have to sample the features also. This set will be used to create the decision tree. We are repeating the process of sampling data and creating the decision tree to create multiple trees. This algorithm is called Iterative Dichotomiser 3(ID3) algorithm for classification using decision tree. We have used entropy and information gain for choosing the best attribute and best value of that attribute for split.

The number of decision trees we are considering are 150. The classification is done based on the majority of the decision trees.

The plot for number of decision trees vs accuracy is shown in figure 1
The plot for number of decision trees vs training time is shown in figure 5

The accuracy shown in the plots are for validation data. The validation set accuracy is increasing as the number of decision trees are increased. This implies that the variance of the single decision tree is decreasing (over fitting reducing). The best results obtained are for number of decision trees equal to 150. The accuracy on validation set is 78.09 %. The time taken for training of decision trees for best accuracy is 4340.92 milliseconds.We have run our serial code on GUP server.

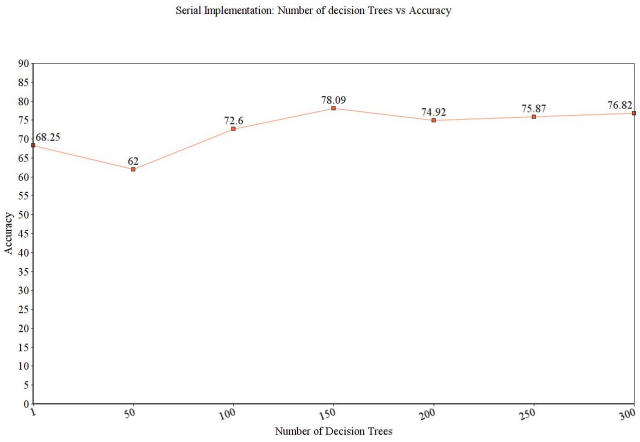

Figure 1: Number of Decision Trees Vs Accuracy

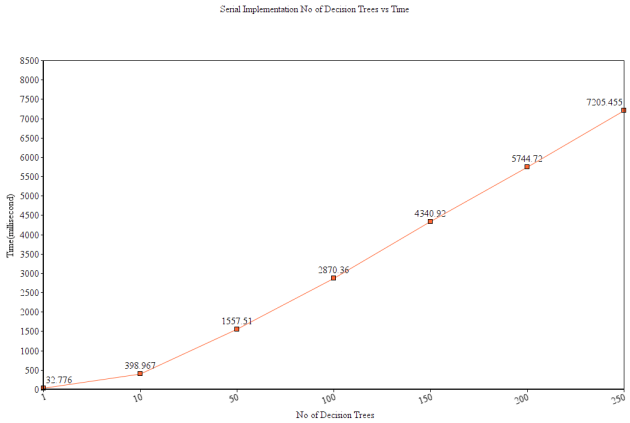

Figure 2: Number of Decision Trees Vs Time Serial

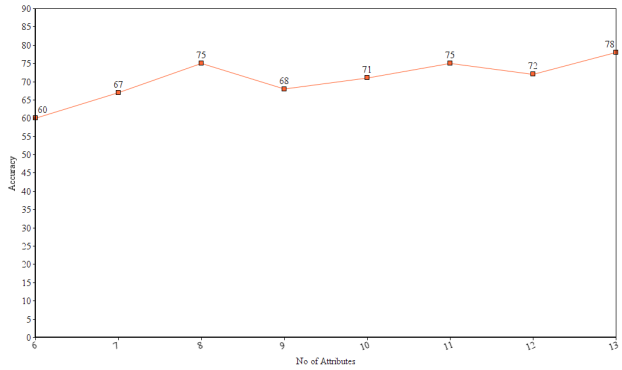

Figure 3: Number of Attributes Vs Accuracy

7.3 Parallel Code Implementation

7.3.1 Naive Parallel:

we are trying to implement **Task Level Parallelism**. in my serial program there are four functions implemented
(i) entropy: This function is used for calculation of entropy and frequency of each unique element of each attribute in training data. In serial code time complexity of this function was O(n) but for CUDA kernel it is O(1) and it's naive implementation(parallel) is using one block and it contains the number of threads equals to number of objects(unique values) in one attribute.There was problem in kernel call **(i)** We have to copy host memory to device memory and **(ii)** We have to use atomic add to avoid race condition so it is taking a lots of time.In this parallel version code no of thread per block is no of objects and each thread is calculating entropy and frequency of each unique element of each attribute in training data.
(ii) findmax: It is used to find the split point in decision tree by finding the maximum element in an array (to find which attribute value pair has maximum information gain). In serial and parallel version this function's time complexity is O(n) but we are trying to reduce to O(log(n)) in parallel version in next phase of this project.
(iii) findsum: To find the summation of all entropy corresponding to one attribute. In serial and parallel version this function is taking time complexity as O(n) but we will reduce findsum function to O(log(n)) in parallel version in next phase of this project
(iv) findk: To find the number of distinct elements in each attributes of training data.In serial code time complexity of this function was O(n^2) but for Cuda kernel it is O(n) and it's naive implementation(parallel) is using one block

and it contains the number of threads equals to number of attribute.There was problem in kernel call we have to copy host memory to device memory and we have to call this function many time.In this parallel version code no of thread per block is no of attributes and each thread is finding the number of distinct elements in particular attribute of training data.

- The fraction of attributes used for each decision tree is 50 percent of total number of attributes(24 attributes).
- The attribute used as Class label is treatment.
- The fraction of data points used for bootstrap sampling is also 50 percent.
- **The work done in Week 3 is serial code implementation, profiling of serial code, optimization of various tasks in parallel and parallel code implementation.**

All the possible optimizations like parallel reduce operations in max and sum , atomics in double,shared memory etc in entropy, findmax, findsum, findk are successfully applied. Still time taken by optimized naive parallel is twenty times more than serial code. After that we tried dynamic parallelism at task level in training a decision tree, but we did not acheived speed up. We have written all these codes from scratch.

7.3.2 Optimized Parallel

We have used task level parallelism where number of blocks is equal to number of attributes and number of threads in a block is equal to the number of data points. The time taken by Cuda API calls is far greater than Cuda kernels, So during calculation of time we have used profiling results of time used by Cuda kernels to show speed up. The speed up acheived is 3.3.[1]

[1]Note: We have followed some github repository for optimization of decision tree.

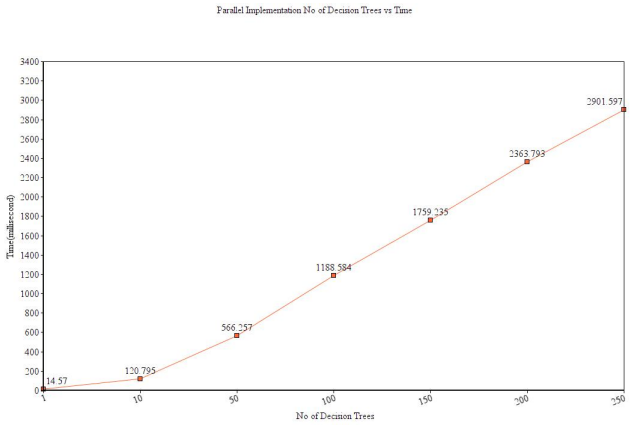

Figure 4: Number of Decision Trees Vs Time Parallel

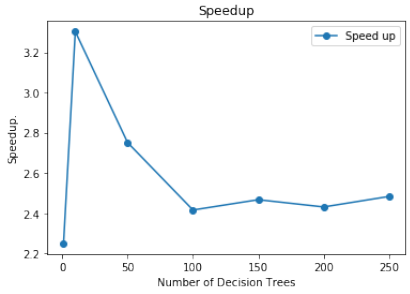

Figure 5: Speedup

8 Observations and Results

- We have observation is that use of task level parallelism for the small task is not good.
- Serial code in C does not required memory copy during function call we can simply pass data as an argument, so for small task it is faster than gpu.
- Cudamemcpy and some API calls(cudaEventCreate, cudaEventSynchronize, cudaFree) taking more time than kernel execution.
- As the number of tree increases, the time taken for training of random forests increases linearly.

- The speed up obtained over the serial code is 3.33.
- As we increase the no of attributes the accuracy some time increases and some time decreases but we get best accuracy when no of attributes is 13. We are calculating this when no trees in random forest is 150 and we have taken this because we get highest accuracy at 150 tree when we vary the no of decision trees.

9 Future Work

- We can combine task level parallelism with data level parallelism in a hybrid technique to obtain the better speed up.
- The trade off between task level parallelism and data level parallelism can also be studied further.
- The decision trees with pruning can also be considered for improvement in generalization.
- The pruned decision trees

10 Code Link

https://bitbucket.org/RahulVashisht/random-forest-implementation-on-gpu/src/master/

References

[1] Yisheng Liao and Alex Rubinsteyn and Russell Power and Jinyang Li “Learning Random Forests on the GPU", 2013