GPU Parallelization of Random Forest Algorithm in CUDA

Rakshith Kamath
Electrical Engineering
Columbia University
New York City, United States
rk3165@columbia.edu

Trevor Gordon
Electrical Engineering
Columbia University
New York City, United States
tjg2148@columbia.edu

***Abstract*—Since the conception of machine learning, it has come a long way to be an integral part of our lives now. With the increasing use of its algorithms in various fields and also the increasing amount of data that is being generated and collected in this era, it becomes indispensable that the performance of such techniques improve. It has been seen that GPU parallelism for speed up has been used for various methods to improve its performance. In this project, we aim to do the same for one of the methods called random forest algorithm by introducing parallelism within each tree. We show that for larger datasets it's faster than the naive implementations and shows comparable results to the current libraries.**

***Index Terms*—Random Forests, CUDA, Parallel Computing, Machine Learning**

I. Introduction

A. Problem Description

Random forests or random decision trees are ensemble learning method for classification and other tasks that operates by constructing many decision trees training on random subsets of data and classifying based on the consensus of all trees. For a classification problem the output is the class picked by the most trees [1]. Increasing the number of trees helps us in giving results which are more robust to the various test cases. The advantage of random forest algorithm is also that it avoids over-fitting of the data which was the downside of decision tree algorithm. Another idea introduced in these algorithms are the general concept of bagging of data in machine learning which helps prevent the over-fitting of the data.In bagging method, we create subsets of data by randomly sampling from the uniform distribution of the data.This creates datasets which are similar to each other yet different from one another. Each of them are used as the input to the tree.This algorithm also handles the problem of missing data as well. The main aim of this project is to parallelize the construction of trees in random forest by giving the intensive math tasks of calculating the decision at nodes to the GPU to speed up the process of training the trees.

GPUs (Graphical Processing Units) are constructed in a way to be suitable for massively parallel computing. While CPU (Central Processing Units) are designed to handle a wide variety of inputs and conditional execution, GPUs are designed to perform similar operations on large amount of input. This is referred to as SIMD (single instruction, multiple data) operations. The "single instruction" referenced here can be described by a "Kernel". GPU kernels are written as C/C++ functions with special variables that allow the kernel to understand which input / output data it should be operating on. Every instance of this kernel running is described as a single thread in CUDA language. Consider the simple scenario where a user has an extremely large image represented by a 2 dimensional array of pixel values, and wishes to brighten the image by adding a fixed value to every pixel. A thread for every pixel will be instantiated. During the execution of each thread, it will have reference to the specific input/output pixel it should be working on. By computing the output for every pixel at the same time using the GPU the operation typically performs faster. The GPU advantage over the CPU increasing as the dataset becomes larger.

B. Prior Work

Some of the earliest work on parallelism of random forest can be seen in making parallelization in decision tree algorithms. [6] They delve into the concepts of task parallelism and data parallelism which is the basic essence of the use of GPU's. Furthermore studies were done more recently by Grahn et.al. [5] and Liao et.al. [4] in terms of implementing the random forest in CUDA. The former has developed a CUDA based algorithm called CUDA RF which was compared to the other standard algorithms like Fast-RF and LibRF. The latter focused on various strategies that were used to build trees using GPU's and explored the ideas of how to build trees using depth-first, breadth-first and hybrid concepts. All this gives us a very good insight so as to which tasks can be parallelized for our within a tree approach.

II. Description

This section starts with the talk on the aim and intent of the project, followed by a detailed guide on the mathematical basis and formulation of the same. Further sections delve into the system specifications and the design of the project itself.

A. Goals and Objectives

The main goals and objectives for our project are:

- Implementation of parallel forest algorithm with the help of GPU's for speed up within a tree

- Experiment with different types of blocksizes and memory allocations
- Compare the model with other models in terms of accuracy and timing for varying sets of data
- Detailed study of the GPU utilization for various kernel calls

B. Problem Formulation and Design

The core part of random forest still remains the decision tree structure itself which is shown below in figure 6.

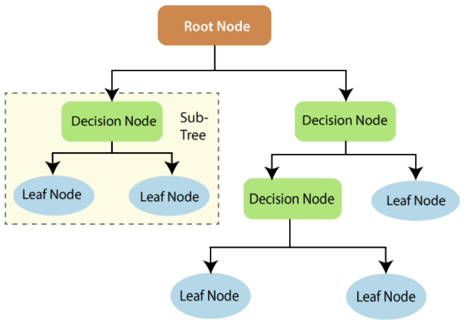


Fig. 1. Basic Decision Tree [2]

We see that the the node at the top is called the root node and it is the point of entry to the tree. All the intermediate nodes are called decision nodes which help us navigate to the leaf nodes which are the terminal of the tree. The leaf nodes have classes assigned to it and if any data reaches that node during prediction the following class is assigned to that data by that particular tree. The decision nodes are trained by taking only certain features in the case of random forest and these are based on the information theory. There are various metrics like entropy, oob score, Gini score etc which can be used to find the split or which can be used to make the decision. All of them aim to reduce the impurity/ intermixing of the data by finding the best split. In this project Gini Index was used for calculation in decision nodes. Gini Index is calculated for the given subset of data presented to that node as follows- Firstly, we find out the number of classes which are going to be the bins for the data to be put into. Each data is binned into the various classes based on the given value and dimension which we are testing. Then we calculate the Gini score using the following formula-

$$GiniScore(value, dim) = \sum_{n=1}^{n} p_i^2 \tag{1}$$

where, p_i is the probability of an element being classified as the right class. And we sum them up for all n classes to get the overall score for multi-class scenario. A small example as shown in figure 3 shows that for various classes the shaded region along the diagonal which shows the p_i^2 values which are being summed up for different classes. This gives us a geometric interpretation of the problem.

To calculate these values for various set of values and for various dimensions we make use of the GPU to speed up the process.

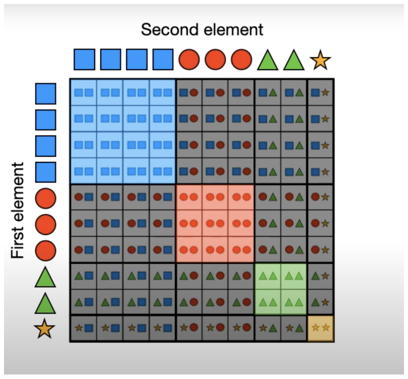


Fig. 2. Geometric Interpretation of the Gina score calculation [3]

Given these values we now need to find the max value of the Gini score which gives us the best split. For this we make use of the reduction kernels in the GPU which are done by blocked approach. Since there are large set of values we have used the hierarchical approach of using auxiliary values to store and find the maximum from the results of each block. This mathematically equates to-

$$B(Value, dim) = \underset{value \in l, dim \in d}{\text{argmax}} \quad GiniScore(value, dim) \tag{2}$$

where we go across all d dimensions and across all l values to find the best pair of value and dimension which gives the maximum value which is called the B(value,dim). Given the split condition and value we need to process the data into two children left and right for the further processing. This requires splitting the dataset into two parts which may not be equal in most cases. handling such large datasets can be an issue which can be handled using the GPU's again where they can handle such large stream of data.

C. System and Software Design

1) Overview of the python training code: We constructed our software in a modular way such that it was easy to speed up our native python implementation by incrementally replacing high workload functions with a GPU alternative. Our top level RandomForestFromScratch class gives an interface similar to the sklearn implementation. The RandomForestFromScratch class trains either DecisionTreeNativePython's or DecisionTreeCudaBase's based on a configuration parameter. Both DecisionTreeNativePython and DecisionTreeCudaBase inherit from DecisionTreeBase which contains the bulk of the logic for training trees. As described in the definition of DecisionTreeBase, it is meant to be inherited from and for the subclass to implement three functions: `calculate_split_scores`, `choose_best_score`, and `split_data`. DecisionTreeCudaBase references a separate Cuda utils class so that our cuda specific functions are in a separate file.

As described in the theory section, random forests involves training multiple decision trees on bagged data. The description below applies for training a single decision tree. The logic for our main fitting function is shown in Figure 3. Our implementation is based on a type of work queue. The algorithm is initiated by putting the entire bagged training dataset on the queue. A leaf node is processed one at a time from the queue. First, there is a check to determine if the leaf node should be processed. Only leaf nodes below a desired depth are processed. Also, if there is only a single type of label in the leaf, it is not processed any further. For nodes that need processing, the first operation is the Gini score calculation. Next, the best score is found for this split. Then the split occurs. The details of these kernels will be discussed in detail in the following sections. The two new child nodes are put on the end of the queue to be processed. By using a first in, first out approach on our data queue, we are training the tree in a breadth first manner. When the tree is finished training our model has a mechanism to quickly sort any new training data into it's likely class.

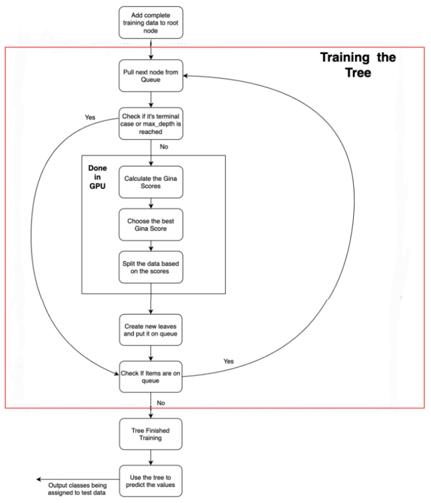

Add complete training data to root node
Training the Tree
Pull next node from Queue
Yes
Check if it's terminal case or max_depth is reached
No
Done in GPU
Calculate the Gina Scores
Choose the best Gina Score
Split the data based on the scores
Create new leaves and put it on queue
Yes
Check If Items are on queue
No
Tree Finished Training
Output classes being assigned to test data
Use the tree to predict the values

Fig. 3. Block diagram of the proposed implementation

2) Overview of the kernels: As said previously, there are 3 main kernels which are called for processing large data quickly. The first one `calculate_split_scores`, is a 2D kernel which takes in the all data points and all its dimensions as well. We calculate the Gini impurity score for each data point at each dimension and store this in a resultant 2D array. The calculation was not optimized to use shared memory since data sets were much larger than the size of shared memory. Here we bin the data into different classes based on the split value which is being considered. These values are then compared with the original landmarks to check if they were rightly classified. The score is calculated as shown in equation 1.

The returned 2D array is now reshaped into a 1D array which will be the input to the 2^{nd} kernel i.e., `choose_best_score`. This takes in the reduced 1D array as the input and applies reduction to it to fetch the maximum value and its corresponding index. We have used shared memory approach with parallel load as well to optimize the results in this kernel. Since the datasets were really large we had to store the results of each block into auxiliaries. Similar reduction kernels were applied to them to fetch the best score and the corresponding index. At the end of this we get the dimension and the value for the split.

The last kernel `split_data` is called to split the data based on the split value into left and right child nodes. These are used to split the huge dataset into 2 smaller parts which necessary wont be equal. Both the X value i.e., the data points and the class labels attached with them are split and returned back to the host for further processing.

3) Overview of testing structure: To ensure we could test our implementation on varying data complexities, we opted to generate our own synthetic data sets at run-time. We implementation a data generator which constructs gaussian mixture models with configurable number of data rows and data predictor columns. Furthermore, the data can have varying covariance between dimensions. In our experiments for each set of experiment parameters, we generate the corresponding data, train using a sklearn random forests model, then train on our own python implementation as well as our GPU implementation. We compare model accuracy to the sklearn implementation to ensure that our model is training correctly. Timing is recording at various stages across all models.

III. Results and Discussion

A. System and platform specifications

All the code was run and tested in the virtual machine that was set up at the start of the course in Google Cloud Platform. The following are it's specifications:

1) Machine Type: n1-standard-4
2) vCPU's: 4
3) Memory: 15GB
4) Number of CUDA devices available: 1
5) Device Name: NVIDIA Tesla T4
6) Compute Capability: 7.5

B. Performance of the model

We have done a comparison of our model with the sci-kit python model [7] to check the accuracy for datasets which are varying from 1-10. As the variance increases it will be difficult to predict the data and hence accuracy is expected to reduce.The following study is done on a dataset having 100,000 points each having 10 dimensions each which were divided into 3 classes. As seen in Figure 4, the accuracy of the model resembles to that of the sci-kit python implementation. It must be noted that the timing of these experiments also should be mentioned to get the exact picture. The sci-kit model was able to implement in 0.9s whereas our model took roughly 2.4s for each of the runs. Thus, Even-though our model cannot match the timing of the sci-kit learn model it is able to match the accuracy. This also shows the robustness of the model against noisy data as well and is up to the mark of standard libraries. The timing comparisons are discussed in the following section.

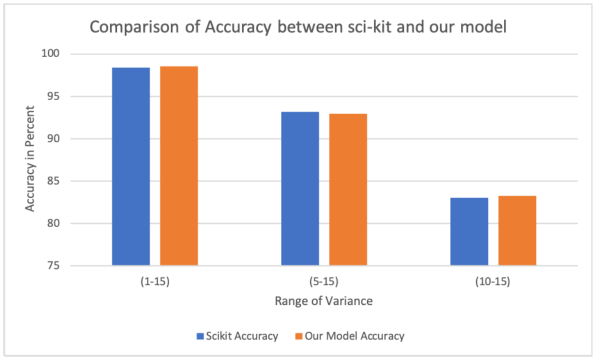


Fig. 4. Comparison of accuracy of our model with sci-kit for increasing variance in data

C. Top level timing of random forest fit method

The results of timing the random forest fit method are shown in Figure 5. Our naive python implementation took the longest amount of time. The timing for the serial implementation increases exponentially as the number of rows in our training set increases. This makes sense as we know splitting a node requires D*N Gini impurity calculations and each calculation requires iterating over N rows. So, for a tree with a fixed max depth, the timing to train increases by a factor of N squared. Figure 6 only shows the timing for the serial implementation up to 1000 rows. Comparatively, our parallel GPU implementation increases linearly in this range. Based on these results it would be nearly impossible to train large trees without leveraging parallel computing using a GPU. Furthermore, our implementation performs comparably to the popular sklearn implementation. However, the sklearn function performs slightly faster than our implementation all at values. It is possible that our implementation lags behind sklearn because our memory transfer isn't optimized. This will be discussed in the following section.

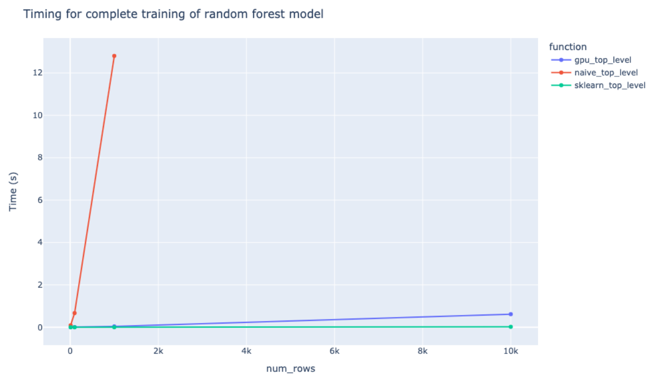


Fig. 5. Comparison total training time

D. Timing with and without memory allocation

Detailed timing of the kernels are shown below. It can be seen that the time to transfer the data set to the GPU takes a significant amount of time. The kernels to split data and find the max score are dominated by memory transfer time. As discussed in previous sections, each leaf node is processed by the running three kernels sequentially. Each kernel requires the output from the previous kernel. In our current implementation each of our kernel calls require transferring data back to the CPU where it is handled by the python run time before being transferred back to the GPU. This is an area where we could speed up our implementation significantly.

The timing for calculating the Gini score shows that while memory transfer plays a significant role, it is not the largest contributor to timing. From this we know that further optimizing this kernel would results in significant speedup of our algorithm.

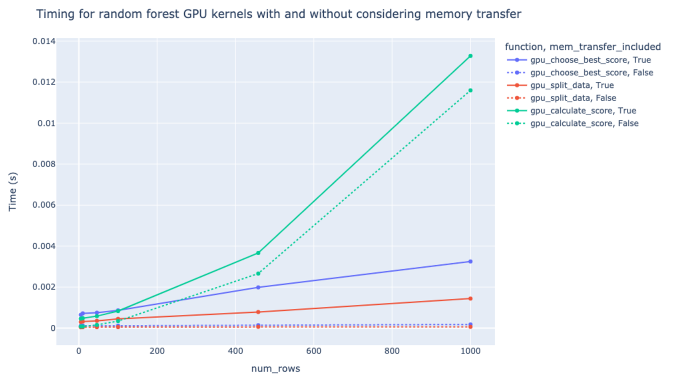


Fig. 6. Comparison of kernel execution time with and without memory transfer

E. Affect of Changing BlockSize

We can see an improvement in the performance of the kernels once we reduce its blocksize. The study was done on a dataset having 100,000 points each having 10 dimensions and which were having 3 different classes. The following table mentions the improvement in utilization of the GPU's as we decrease the blocksize for various kernels. Here in tables I , III and IV are the utilization for each level of reduction scan kernel call. As the blocksize increases the utilization of the lower or the 1st sweep of scan increases. This is because of the fact that threads are not getting wasted and are used effectively when the blocksize becomes smaller. For example, for running even 3 threads if the blocksize is 1024 we have a large number of threads that are getting wasted and not used for effective computation. That's why we see an increase in the utilization in 1st sweep. Due to this, the knockoff effect we see in the subsequent sweeps i.e., the 2nd and 3rd kernel calls is the decrease in the the SM and memory percentage. Table II talks about the affect of changing blocksizes in terms of threads and the Waves/SM. The size of the shred memory decreases accordingly as the blocksize decreases. Also we see that the number of threads that are being wasted decreases. A special note to be made here is in the case of blocksize 16, it's seen that the Waves/SM percentage is really high i.e., 97.66.

TABLE I
UTILIZATION FOR THE REDUCTION KERNEL $1^{st} scan$

Blocksize	Reduction kernel-$1^{st} scan$	
	SM Percentage	***Memory Percentage***
1024	31	31
256	52	52
64	67	67
16	64	64

TABLE II
THREADS AND WAVES PER SM DATA FOR CALCULATE REDUCTION KERNEL

Blocksize	*No. of threads*	*Waves/SM*	*Shared Memory/block*
1024	1000448	24.42	16384
256	1000192	24.42	4096
64	1000064	24.42	1024
16	1000016	97.66	256

The Split data kernel call was seen to have no much effect with change in blocksize and remained roughly constant irrespective of the blocksize. This can be seen in Table V.

Lastly we look at the 2D kernels which are being used in this project i.e., calculate Gini scores. We looked into the following two settings for this case which are 32x32 and 16x16. This can be seen in figure 7. This again shows us that similar to reduction the decrease in the blocksize helps in the increase of utilization due to efficient use of threads. The idea of the threads that are being generated is mentioned in table VI.The number of extra threads that are being generated is really large and thus the delay in calculation is seen.

To give an idea of the speed in time that we observed just by changing the blocksize, we see that when the sizes are maxed out i.e., 1024 and 32x32 the time to build the tree for 100,000 data points of 10 dimensions each took 3.4s. Where the time taken for the sizes 16 and 16x16 blocks was only 2.6s. This difference of 0.8s just by changing the blocksize is a huge improvement seen and helps us understand the importance of setting the right blocksize for the optimal performance of the GPU cores.

TABLE III
UTILIZATION FOR THE REDUCTION KERNEL $2^{nd} scan$

Blocksize	Reduction kernel-$2^{nd} scan$	
	SM Percentage	***Memory Percentage***
1024	20	20
256	7	7
64	3	4
16	2	2

TABLE IV
UTILIZATION FOR THE REDUCTION KERNEL $3^{rd} scan$

Blocksize	Reduction kernel-$3^{rd} scan$	
	SM Percentage	***Memory Percentage***
1024	0.6	0.6
256	0.25	0.25
64	0.1	0.3
16	0.06	0.23

TABLE V
UTILIZATION FOR THE SPLIT DATA KERNEL

Blocksize	Split Data Kernel	
	SM Percentage	***Memory Percentage***
1024	4	73
256	4	77
64	4	77
16	8	74

TABLE VI
THREADS AND WAVES PER SM DATA FOR CALCULATE GINI SCORE KERNEL

Blocksize	*Number of threads*	*Waves per SM*
32x32	3201024	78.15
16x16	1600256	39.07

IV. DEMONSTRATION

Figure 7 shows an example of a tree that was built for a dataset having 100,000 values with 10 dimensions each which needed to be classified into 3 classes. The depth of this algorithm for this case was set to 4 to avoid the overfit. The confusion matrix for the given tree is given below in figure 9.

V. FUTURE WORK AND DISCUSSION

A. *Optimizing memory transfer*

As described in the section above, the time for memory transfer significantly slows down the operation of the kernel. We could optimize the process by reducing the transfer from the device to host just by loading it once and then giving us results after the whole process. We would like to implement a more cohesive way of doing more implementation on the GPU which would result in better performance.

B. *Parallelized computation of leaf nodes*

The calculation here is happening for each of the nodes sequentially with the help of stacks. We would like to improve this method by parallelly running all the nodes in the stack so

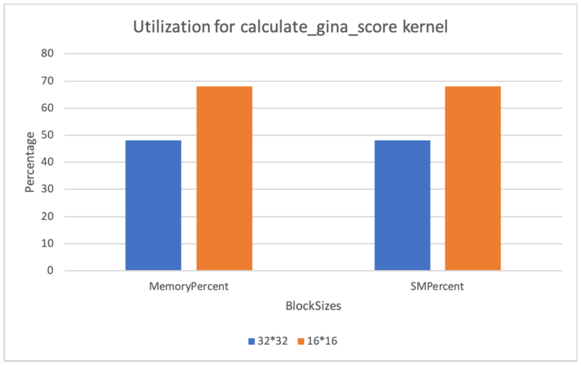
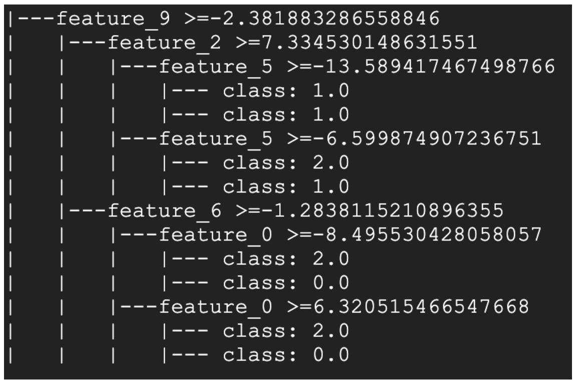


Fig. 7. Calculate Gini score kernel's utilization



```
|---feature_9 >=-2.381883286558846
|   |---feature_2 >=7.334530148631551
|   |   |---feature_5 >=-13.589417467498766
|   |   |   |--- class: 1.0
|   |   |   |--- class: 1.0
|   |   |---feature_5 >=-6.599874907236751
|   |   |   |--- class: 2.0
|   |   |   |--- class: 1.0
|   |---feature_6 >=-1.2838115210896355
|   |   |---feature_0 >=-8.495530428058057
|   |   |   |--- class: 2.0
|   |   |   |--- class: 0.0
|   |   |---feature_0 >=6.320515466547668
|   |   |   |--- class: 2.0
|   |   |   |--- class: 0.0
```

Fig. 8. One of the trees that is being created for the dataset mentioned above.

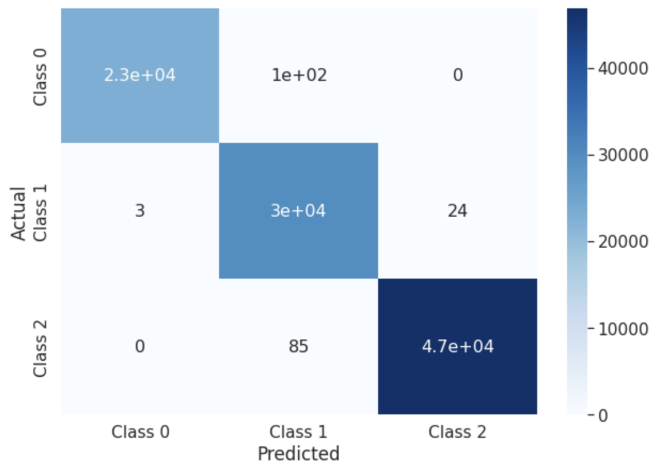


Fig. 9. Confusion matrix for the figure 8 example

as to save more time while training. This would lead to a much larger speed up since a bottleneck will be resolved.

Another approach we would have liked to work more on was trying to parallelize across trees as well.

VI. Conclusion

We have demonstrated one of the ways to implement the random forest algorithm parallelly within a tree by sending the math intensive functions to the GPU for calculation. Our experimental tests with varied data types shows us that the accuracy of the model was equal if not slightly better than the sci-kit library. It had comparable times as well. In terms of speed up we see that reducing the block size to 16 for 1D kernels and 16x16 for 2D kernels gave the optimal performance. Compared to the naive implementation in python this had a huge speed up due to parallelism.

Parallelism using GPU's opens new horizons in terms of implementing these algorithms in multiple GPU's and even multiple machines as well. This can be used to aid other algorithms such as boosting, ensemble decision trees etc.

Acknowledgment

We thank Arjun Ahuja for assistance with debugging of a small piece of code in one of the kernels which helped us in the improvement of the project.

		Contributor	
		Rakshith	Trevor
Contribution Area	Python code	40%	60%
	Cuda kernels	60%	40%
	Report / FIgure	50%	50%

Contributions

The contribution table outlines the rough contributions by each member.

References

[1] Ho, Tin Kam (1995). Random Decision Forests (PDF). Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, 14–16 August 1995. pp. 278–282. Archived from the original (PDF) on 17 April 2016. Retrieved 5 June 2016.

[2] https://www.tutorialandexample.com/wp-content/uploads/2019/10/Decision-Trees-Root-Node.png, Decision Tree Image,12-15-21.

[3] https://www.youtube.com/watch?v=u4IxOk2ijSs,Gina Score,12-15-21.

[4] H. Grahn, N. Lavesson, M. H. Lapajne and D. Slat, "CudaRF: A CUDA-based implementation of Random Forests," 2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), 2011, pp. 95-101, doi: 10.1109/AICCSA.2011.6126612.

[5] Yisheng Liao, Alex Rubinsteyn, Russell Power and Jinyang Li,Learning Random Forests on the GPU,Department of Computer Science,New York University

[6] Amado N., Gama J., Silva F. (2001) Parallel Implementation of Decision Tree Learning Algorithms. In: Brazdil P., Jorge A. (eds) Progress in Artificial Intelligence. EPIA 2001. Lecture Notes in Computer Science, vol 2258. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45329-6-4

[7] sklearn.ensemble.RandomForestClassifier,https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html,scikit-random-forest,12-15-21.

Appendix

Here are some of the plots that were made from tables I to V collated together which look like as shown in figure 10.

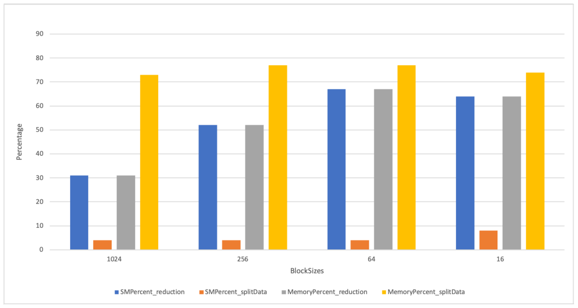


Fig. 10. Summary of split data and reduction kernel 1st sweep utilization