JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue

JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)

An International Scholarly Open Access, Peer-reviewed, Refereed Journal

EFFICIENT PARALLEL PROCESSING ON DECISION TREES USING GPU

[1]Thanasekhar B, [2] Balaji. C

[1]Associate Professor, Department of Computer Technology, Anna University, MIT Campus, Chennai, India.

[2]UG Scholar, Department of Computer Technology, Anna University, MIT Campus, Chennai, India.

Abstract: Decision trees trained in GPUs that followed batch processing overcame the insufficient memory to allocate the entire data by restricting the repeated computation of a split value in a node. This was achieved by implementation of the Node parallel processing method along with shared memory and synchronization among GPU cores. But this model suffered from a major load imbalance which leads to high communication cost. In order to minimize the imbalance in workload, a new model is proposed. The proposed model uses a histogram construction approach to compress the input data. This compression of data allows for faster transfer of data and less communication time between GPU cores. A Hybrid parallel architecture combining data and feature parallelism is proposed. Data parallelism is used in the compress function to compress the training data for computation. For split value calculation and entropy gain value calculation, Feature parallelism technique is used. This hybrid approach reduces the workload among worker nodes significantly thus leading to lower communication cost. The results of the proposed model with ijccn1 dataset indicate that it achieved a lower training time when compared with other parallel models. The evaluation of the proposed model against different datasets reflects the training performance is 10x speedup of the sequential model and 6x speedup of other models such as Sequential, OpenMP and OpenMPI models. The output of this phase can be used to determine the parameters which are significant in training the proposed model.

Keywords: Decision trees, Batch Processing, Histogram construction, Data parallelism, openMP, MPI

1. INTRODUCTION

An impressive supervised learning algorithm that is suitable for the classification problems is the decision tree. The motive of choosing decision trees is to create a training model that can be used to predict a class of value of target variable by reading simple decision rules based on prior data. Decision trees are used for handling non-linear data sets effectively. Decision trees also provide additional conditional control statements which can provide exceptional results. By The decision trees can be very quickly and accurately trained in real-time with the help of GPU providing all these benefits, the decision tree plays a major role in various fields of machine learning classification problems. The main motivation was the fact that the GPU has limited memory space to perform complex computations. Our major task is to employ parallelism in its limited memory space. Use of a single type of parallelism is not sufficient for our project. Hence, we propose a unique parallelism technique which combines two parallelism approaches. When training the model, computing the values without repetition for the same input values poses a great challenge. When updating the values of histogram, there is also the challenge of data conflict between different processors updating the same histogram. This issue needs to be solved to avoid any race conditions. In this project, we mainly focus on increasing the effectiveness of the parallel processing in GPUs at its limited memory space. In addition to achieving parallel processing, we also focus on resource utilization. Parallel processing ought to be implemented by the use of hybrid parallel processing methods, batch processing and synchronization. Our main objective is to reduce the workload imbalance among the processors while consuming low memory and processing at a faster time. Furthermore, the results of the proposed model can be used to derive a machine learning mathematical model which can be used to identify the significant attributes which affect the performance of GPUs against the proposed model. Using the compression algorithm, we might modify the histogram construction such that compressed data can be used by our proposed model and transmitted with low communication cost. The modification done to the algorithm is to count frequency of the output class for each feature attribute rather than counting overall output class as mentioned in the algorithm. The modification done to the algorithm is to count frequency of the output class for each feature attribute rather than counting overall output class as mentioned in the algorithm.

2. RELATED WORK

Svantesson et al.. (2018) proposes Streaming Parallel Decision Tree algorithm. In which, to increase the speed, a probabilistic approach is used. Instead of using the traditional method, histograms are used to make the appropriate decisions. The SPDT algorithm makes use of the workers on a network combined with a dynamic histogram approximation of the data. They used a predetermined number of data samples per layer to accelerate the GPU. It is found that the GPU implementation is as accurate as the CPU implementation. The GPU implementation was also found to be faster than the CPU implementation in one of the tested dataset. This opens up the possibility of parallelizing decision trees in the area of streaming data. Zhang et al.. (2017) proposed a novel algorithm for accelerating the building of decision trees using GPU. They tend to overcome the scalability and performance issues of other tree building algorithms, which use parallel multi-scan and radix sort to find the best split. They have shown that, In

GPU, using a histogram based algorithm which is scalable and more efficient to find the best split point. Haem et al.. (2010) proposes a hybrid algorithm which combines horizontal and vertical parallelism for tree construction. In compression, each processor has to only transmit the approximate description of the data with low communication complexity. Gehrke et al.. (2000) presented a framework for scaling up existing decision tree construction algorithms. This framework, which they call Rainforest, closes the gap between the limitations to main memory datasets of algorithms in the machine learning and statistics literature and the scalability requirements of a data mining environment. These data access algorithms that scale with the size of the database, adapt gracefully to the amount of main memory available, and are not restricted to a specific classification algorithm.

Ben et al.. (2010) proposes a new algorithm to construct a streaming parallel decision tree exclusively designed for the classification of larger datasets and for streaming datasets. In this type of classification histograms will be rapidly constructed to compress the data to a fixed memory size. A processor which will be appointed as master uses this information to find the nearest split points to the terminal nodes of the tree. Jin et al.. (2003) proposed a new method for decision tree construction. It is called SPIES (Statistical Pruning of Intervals for Enhanced Scalability). This work is derived from the Rainforest approach for scaling decision tree construction, as well as the one pass decision tree algorithm for streaming data. Haem et al.. (2010) proposes a hybrid algorithm which combines horizontal and vertical parallelism for tree construction. In compression, each processor has to only transmit the approximate description of the data with low communication complexity. Fang et al.. (2003) produces a prediction model that yields a set of weak prediction models in the form of decision trees. Gradient boosting is a machine learning technique for regression problems, which produces a prediction model in the form of an ensemble of weak prediction models. Its high accuracy means that almost half of the machine learning contests is won by GBDT models. The general concept of this gradient boosting decision tree is additive training. In each of the iterations, the decision tree learns the gradients of the residuals between the target values and the last predicted values, later the gradient descent is applied with the help of the learned gradients. The algorithm can't be parallelized like Random Forest. It can only parallelize the algorithm in the tree building step.

In summary, after performing the Literature Survey, all the solutions presented for parallel decision trees were an algorithm specific optimization that focused only on increasing the performance of decision trees rather than exploiting the full scale power of GPU. Although several approaches to a histogram based method was inferred from the survey, there was no hybrid architecture that proposed the histogram construction. The hybrid architecture could exploit the full power of GPU both in terms of memory and speedup. Combined with histogram based method, hybrid architecture can effectively train decision trees with high speed on live and streaming data. Thus, we conclude to propose a GPU backed parallel processing technique for training decision trees without any reduction in performance.

3. PROPOSED MODEL

To achieve kernel value sharing and reusing while parallel processing each node of the decision trees, a shared memory matrix is used. Shared memory matrix acts a memory block that is shared by all the threads in the block. All the concurrent processes can access this memory buffer. This shared memory is used to store the repeated computational values so that the computational time can be decreased. The computed value for each level of the decision tree is stored so as to enable other processes to access them, thus reducing the computational time.

3.1. System Architecture

Decision trees are flowchart like tree structure, which is a widely used tool for classification and prediction. In the decision tree, the terminal node or the leaf node holds a class label, the branch represents the outcome of the test and the internal node denotes the test of an attribute. The internal nodes are for decision making, where the decision making takes place after computation in each node. Calculating the attribute gain and information gain takes place in each node. The architecture of the proposed model is shown in Figure 1. The architecture mainly comprises two major functions which are compress and the split functions. The system architecture is comprised of three modules Compress Module, Split Node Module and Machine Learning Module. The above mentioned modules are explained in the below sections. Batch processing is the process by which the computing takes place in batches or in which the computation is completed in batches of jobs. Batch processing takes at all places of the implementation. During histogram construction, they are processed as batches of data rather than individual rows. Batch processing of the nodes of each level of the decision tree assists the better performance. Using batch processing, reduced processing time for large datasets is achieved and also the idle time of GPU cores is also reduced. In implementation, data are passed as batches during parallel processing.

3.2 Workflow of the proposed model

There are two functions in the tree building process that are computationally expensive. *Compress* function and *find_best_split* function are the costly functions that need to be parallelized. The *compress* function transforms the input data into an array of histogram values. The *find_best_split* function calculates the gain of a specific split feature and split values for the node. Our proposed model follows a hybrid approach where two parallel processing methods are implemented to these two functions since they are independent of each other.

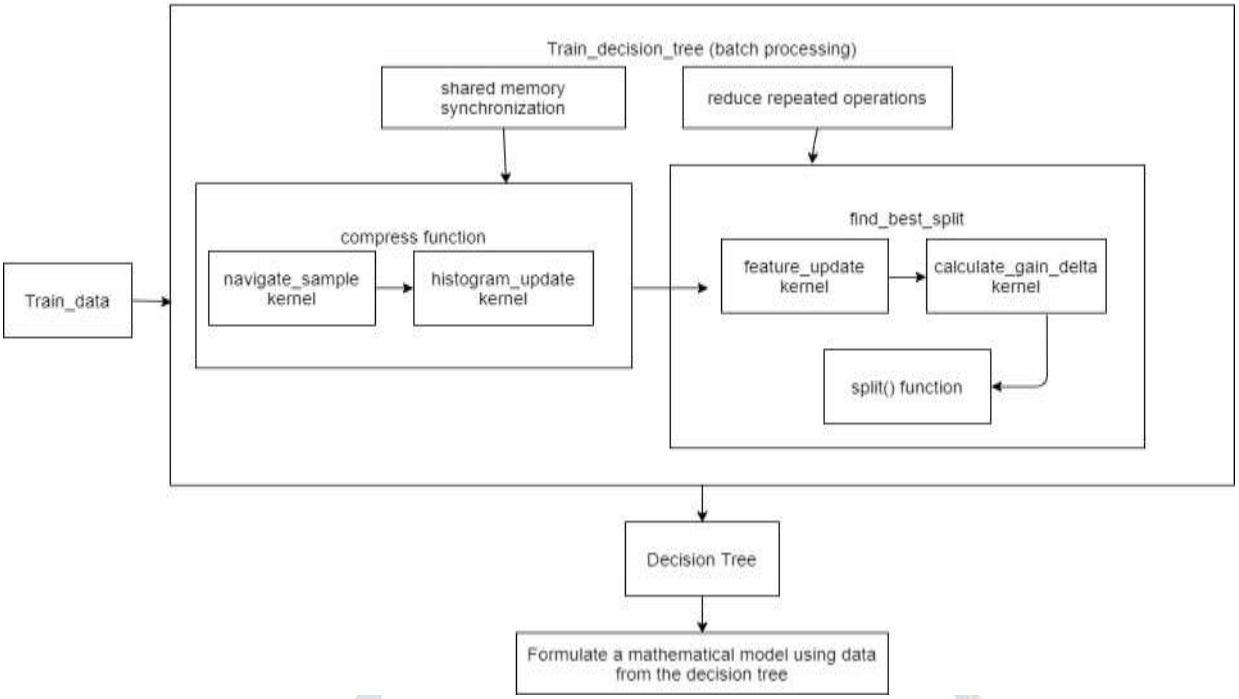

Figure 1 Architecture Diagram of the proposed model

4. COMPRESS MODULE

The main function of this compress module is to compress the given data to the bins or histograms which are to be used to find the best split points in the find_best_split function. The compress module initially creates predefined bins. This module takes input from the training dataset and compresses the given training data and returns an output set of histograms which will be parallel given to the find_best_split for further computation. To reduce the time complexity, we compress the data into predefined number of bins and then we iterate each bin for one feature so that it would reduce the time complexity into O (bins* features). There are two kernels involved in these functions which are the navigate sample kernel and the histogram update kernel. This module employs data parallelism to increase the performance of the module. We use data parallelism in this module because the data parallel approach creates parallelization across multiple processors in a parallel computing environment.

ALGORITHM 1: COMPRESS THE DATA

Input: 1 / W of the training set, where W is the number of processors
Output: histograms to be transmitted to the master processor
procedure Compress ()
    Initialize an empty histogram h (v, i, j) for every unlabeled leaf v, attribute i and class j
    for all observed training samples (x_k, y_k), where $x_k = (x_k^{(1)}, \ldots x_k^{(d)})$
    if the sample is directed to an unlabeled leaf then
        for all attributes i do
            Update the histogram h (v,i,y_k) with the point $x_k^{(i)}$, using the update procedure
        end for
    end if
    end for
end procedure

In the *compress* procedure in Algorithm 1, each data point in the input data is traversed with a loop. This loop can be parallelized in a way to all the worker nodes in a data parallel manner. When data parallel processing is applied, there is a big challenge. Whenever a data point is updated for a histogram, there is a huge contention for the histogram method. This may lead to a race condition which affects the performance. So to overcome this problem, the function is reordered in such a way that there is no race condition. The *compress* procedure is implemented in the GPU by using a shared memory model. However, using this approach will cause each leaf to contain an unequal number of data points. But this could be overcome by following dynamic scheduling.

4.1 Data Parallelism

Data parallelism is a parallelization technique which is implemented across multiple processors in different parallel processing environments. To operate on the data in parallel, this data is distributed across the nodes. Data parallelism is achieved when each processor performs the same task on different data. It is achieved in a multiprocessor system that executes a single set of instructions which is also called single instruction multiple data architecture. Data parallelism is implemented in four different phases. They are the decomposition, assignment, orchestration and mapping. In the decomposition phase the dataset is decomposed into small fragments of data to distribute the data among different processors. In the next phase, assignment is a process of assigning the data to the processes for computation. In the orchestration phase, the data is accessed, each process communicates with each other followed by the synchronization of the processes. In the last phase, which is the mapping phase, the processes are mapped to the processors.

The main advantage of data parallelism is that data parallelism allows efficient usage of a parallel machine's resources, while providing a straightforward programming style that avoids many of the difficulties of task-oriented concurrent programming. The compress module comprises two sub modules each with their own task and operations with the same objective to compress the data. Navigate sample kernel is the kernel which is dedicated to assign the data to the leaf nodes. The number of threads is assigned in a way that the number of threads is equal to the number of data. The sum of the number of data and one less than the number of threads divided by the number of threads gives the block number. For each thread it checks the assignment of the data pointer in the dataset. If the data is assigned to every leaf node in a level, the level is marked as labeled. To initialize the number level of unlabeled leaves of histograms this function is called every time. Histogram update kernel is the function that is used to update histogram with the given (key, value) pair. There are two versions in this kernel. The first implementation is the data-parallel implementation. In this function the block number is the number of data, the thread number is the number of features. For each thread, it uses the corresponding feature value in the corresponding datagram to update the histogram. Even though this version could achieve high distribution works, we found that there is competition for different threads to update the same histogram in the system, causing the wrong result. While the paper Svantesson et al.. (2003) believes that conflict of updating the same histogram affects little to the correctness, tasks are designed differently.

5. SPLIT NODE MODULE

Decision trees use multiple algorithms to determine the division of a node into two or more sub-nodes. The formation of sub-nodes increases the compatibility of sub-nodes. In other words, we can say that the purity of the node increases with respect to the target variable. The decision tree separates the nodes from all available variables and selects the partition that leads to the most compatible sub-nodes.

ALGORITHM 2: Updating the elements in histogram

```
Input: histogram to be updated, feature id and corresponding label of the feature and the value to be updated
Output: Updated histogram with the specified values
procedure Update(histogram_id,feature_id,label,value)
    histo get histogram (histogram_id, feature_id, label)
    bin_size get bin size (histo)
    for all samples in the histogramdo
        if the bin value of each sample – value < 1e-5 then increase the frequency of the sample by 1
        end if
    end for
    index  bin_size
    for all samples in the histogram do
        if bin_value of sample > value then
                idx = i
                break
        end if
    end for
    for all samples in range bin_size to idx in the histogram do
        shift sample in i-1th index in  to ith index
    end for
    increment bin_size of histo by 1
    histo[idx].bin_value  value
    histo[idx].freq  1
    return histo
end procedure
```

In Algorithm 2, it is used to update the histogram with the specified value. The first loop is used to identify if there are bin values which are equal to the specified value and increment the bin value by 1. The next 2 loops are used to put the new element with the specified value into the correct place in histogram. After inserting the value at the correct index, the histogram is updated in memory.

5.1. Gini index

Gini Index is the cost function which is used to evaluate splits in the dataset. It favors large partitions and is easy to implement whereas information gain favors smaller partitions with distinct values. It performs only binary splits. Classification and Regression Trees use the Gini method to create binary splits. Gini index is the sum of squares of probability of success and failure. Gini Index = (p^2+ q^2), Where p is probability of success and q is the probability of failure. Calculate Gini for sub-nodes and then calculate Gini for split using the weighted gini score of each node in the split.

5.2. Chi square

Chi-Square finds the statistical significance between the difference between the sub-nodes and the parent node. It can perform two or more splits. Higher the Chi-Square values higher the statistical significance of the difference between the sub-nodes and the parent node. It generates a tree called CHAID (Chi-Squared Automatic Interaction Detector). Chi-Square is calculated using the formula,

Chi-Square = ((Actual - Expected) ^{2} / Expected) ^{2}.

Chi-Square for each node is calculated by calculating the deviation of success and failure. Then, the Chi-Square of the split is calculated using the sum of all Chi-Square of success and failure of each node of the split.

5.3. Information gain

Entropy is the measure of the randomness in the information being processed. It is the impurity or the disorder in the given dataset. Higher the entropy, harder it is to draw any conclusions from the dataset. Information Gain is the reduction in entropy. It is calculated by comparing the entropy of the dataset before and after a transformation. It determines how well a given attribute separates the training examples according to their target classification. If the sample is completely homogenous, the entropy is zero. If the sample is equally divided, the entropy is one. Entropy can be calculated using the formula, Entropy = p log_2 q - q log_2 p, Here, p is the probability of success and q is the probability of failure. The entropy for a split is calculated as the following steps. First, the entropy of the parent node is calculated. And then, the entropy of each individual node of split is calculated and then the weighted average of all sub-nodes available in the split is calculated.

5.4. Reduction in variance

Reduction in Variables is an algorithm used in regression problems. This algorithm uses variance to find the best split. The split with the lower variance is selected.

$$\text{Variance} = \frac{\sum(X-\overline{X})^2}{n}$$

Here, $\overline{X}$ is the mean of the values. X is the actual value and n is the number of values.

The main advantage of task parallelism is that the data can be executed for different tasks concurrently. It enables multiple portions of a visualization task to be executed in parallel. The main disadvantage of this technique is that the number of independent tasks that can be identified, as well as the number of CPUs available, limits the maximum amount of parallelism. It can be difficult to load-balance the tasks, and therefore it can often be very challenging to take full advantage of the available resources. The task parallelism has become impractical for various reasons because we cannot increase either the clock speed or the instructions per clock for a single core. The proposed split function of a node comprises two sub modules. This kernel is used to calculate the promising split point's features and values for the splitting process in a leaf node. This kernel uses two functions namely, CUDA_merge_array_pointers and CUDA_uniform_array functions. With the help of CUDA_merge_array_pointers function, this kernel merges the two possible histograms and then unifies the array by calling CUDA_uniform_array function. The thread number is 128 and the block number is given by, (num_features + thread_no. - 1)/thread_no. We can say that, we're paralleling over different features of the leaf nodes. The total thread number is equal to the number of features. For each thread, this kernel comes up with a possible split pair (feature, value) and then stores this result in an array. The function representation of the kernel is as follows, __global__ void calculate_feature_value_kernel (int histogram_id, int* cuda_feature_value_num, int* cuda_feature_id, float* cuda_feature_value) { … }__global__ denotes that the function performs parallel processing. The function also receives the histogram id,, cuda_feature_value_num, cudoa_feature_id and cuda_feature_value. The kernel finds the feature id and then initializes the buf_merge array. It finds the histogram for class 0 and class 1 using CUDA_get_histogram_array function. And it merges both the histograms using CUDA_merge_array_pointers function. At last, this kernel writes the result of possible splits into CUDA arrays.

6. MACHINE LEARNING MODULE

We implemented a machine learning mathematical model that measures the performance of GPU (training time) to train the decision tree. A linear regression model is used to find out the significant attributes. Using a Correlation matrix, we have identified the most significant attributes that determine the training time. The results of the implementation in the previous module for various input data are used as a dataset for this mathematical model. For this implementation, we only use the attributes which have coefficient value >0.5. The attributes which satisfy the condition are NET_COMPRESS_TIME and NET_SPLIT_TIME. A Machine learning Linear Regression model is used to formulate the mathematical model. The tensor flow package is used for implementation. Before training, we normalize our input data since the data consists of different ranges and scales. To normalize, we use preprocessing. Normalization () method so that the data is not confined only to ranges in the training set.

In the Algorithm 3 explains the construction of trees to *find_best_split* function, a loop is called for each feature in the input data. This loop can be parallelized in a feature parallel manner such that each feature is assigned to a worker node. The challenge is that best split value information needs to be manually synchronized among all worker nodes. One way is to introduce a local variable for each worker node to store the split information and then merge the results afterwards. Each worker node has a local cache in the GPU memory which it can use to store the local best split value. The master node handles a global cache memory which receives all the individual local best split values from each worker node and calculates the overall best split. Since each thread only needs to send its best split feature to the master, the communication cost is O (P) where P is the number of worker nodes.

ALGORITHM 3: CONSTRUCTION OF DECISION TREE

```
Input: histograms with feature id and label frequency values
Output: decision tree
procedure TreeBuilding()
    Initialize T to be a single unlabeled node
    foreach batch_data do
        Reinitialize every leaf in T as unlabeled
        while there are unlabeled leaves in T do
            Navigate batch_data to the leaves
            Construct histogram h(v, i, c)  by calling Compress() procedure
            for all unlabeled leaves v in T do
                if there are no samples reaching v  then
```

```
                Label v
            else
                find_best_split()
                    for all features i do
                        Merge h (v,i,1) .. h(v,i, c) and get h(v,i)
                        Determine the candidate splits
                        Estimate the information gain of each candidate
                    endfor
                    Split v with the highest gain.
            endif
            endfor
        endwhile
    endforeach
end procedure
```

6.1. Parallelization Of Split Function

In the *find_best_split* function, a loop is called for each feature in the input data. This loop can be parallelized in a feature parallel manner such that each feature is assigned to a worker node. The challenge is that best split value information needs to be manually synchronized among all worker nodes. One way is to introduce a local variable for each worker node to store the split information and then merge the results afterwards.

Each worker node has a local cache in the GPU memory which it can use to store the local best split value. The master node handles a global cache memory which receives all the individual local best split values from each worker node and calculates the overall best split.Since each thread only needs to send its best split feature to the master, the communication cost is O (P) where P is the number of worker nodes.

6.2. Operations Handled At Master Node

All the above mentioned algorithms for compress and find_best_split modules are executed concurrently by individual threads or worker nodes. The proposed approach is essentially streaming and needs explicit synchronization. The schedule and assignment of tasks to the threads is done bya centralized process which is the master node. The Master node handles the process of synchronization and broadcasting.

In the compress function, each worker node completes their work, all the histogram data is transferred to the master node and this node does the merging process. After merging completes, the results is broadcasted to all the worker nodes.
So in summary the entire time taken or cost by the proposed model is,

- At most N/W operations by each processor in the updating phase. This means there will be no workload imbalance in all the processors.
- O(W * L * c * d) for communication cost of histograms.
- O(W * L * c * d) for merging
- O(P) for best split features.

where W is number of workers, L is number of unlabeled leaves in current iteration, c is number of labels and d is number of features.

7. EXPERIMENTAL SETUP AND RESULT ANALYSIS

7.1. Dataset Description

The dataset used to evaluate our proposed model are IJCNN dataset and a1a dataset. Big_size_small_feature and Middle_size_small_feature are synthetic datasets generated by a python script. These two are also used for evaluation purposes. IJCNN1 is a dataset of 49,990 observations and 22 features. It is taken from the LIBSVM dataset. A1a is a dataset that's originally from the UC Irvine Machine Learning Repository, another open source data repository The original Adult data set has 14 features, among which six are continuous and eight are categorical. In this data set, continuous features are discretized into quantiles, and each quantile is represented by a binary feature. Also, a categorical feature with m categories is converted to m binary features.

Table 1 Description of the Training Dataset

Dataset	Feature Size	Train Data	Test Data
IJCNN1	22	49990	91701
A1A	123	1605	30956
Big_size_small_feature	50	10,00,000	1,00,000
Middle_size_small_feature	50	1,00,000	10,000

7.2. Storing The Dataset

A specially constructed class *Dataset* is used to store the input training dataset. Each data in the dataset is stored using a class named *Data* from the same file in the class *Dataset*. The *Data* class structure is made up of an int *label* which marks the label or output class of the data, and an unordered map*values* which stores the map between the feature id and the feature values. Since the

dataset does not contain all features values and only sparse features, a map is used to store specified feature ids and their values. This approach ignores all the 0s of other feature ids.

The class *Dataset* contains a vector of class *Data* that stores all the possible input data. Since a vector is used, it is pretty easy to store all types of datasets of various sizes. Other than the class Data vector, the Dataset class also stores other types of information such as number of features and number of data.

Table 2 Configuration of CPU and GPU used

Configuration	CPU	GPU
Name	Intel(R) i5-8250	Nvidia GeForce MX110
Cores	4	256 CUDA cores
Memory	8 GB	2GB
Base Clock	2400 MHz	978 MHz
L1 cache	256 KB	64 KB
L2 cache	1024 KB	1024 KB

7.3. Serialization Of Datasets For GPUs

When the dataset is stored using class Dataset, it is not possible for the GPU to use the vector version of the dataset. The Data must be passed in such a way that the GPU must be able perform computation on the data. Therefore, we transform the input data into several serialized 1-D arrays. The class Dataset is stored in two 1-D arrays. The size of label_ptr is the number of data in the dataset. The size of value_ptr is number of data * number of features. If a read or write operation is need to performed at (data_id, feature_id), it is done as label_ptr [data_id] and for value it is done as value_ptr[data_id * num_features + feature_id].The limitation of serialized arrays dataset is that it's difficult to manage this dataset and provide operations in GPU. The memory usage can be very large if a large dataset is used. While the memory on GPU is limited, it's difficult for some GPUs with low memory size to run the parallel version. The evaluation of the proposed model is done against three computational models. Each computational model runs different types of parallel algorithms. Open MPuses Data Parallelism, MPI uses Feature Parallelism, Sequential computation uses single core with a single thread. The above mentioned models can serve as an evaluation tool to measure the increase in performance of the proposed model when compared with the above models.The computing environment in which the decision trees are trained is listed in Table 5.2. It provides the configuration information of both CPU and GPU.

7.4. Implementation Of Ml Model

The Machine learning mathematical model is implemented in Google Colab Platform. In the implementation, a Linear Regression model is used to find the significant attributes that measure the performance of the GPU against the proposed model. The Tensor flow package available in Colab is used to design and compile the linear mathematical model. The input data obtained as result of training the datasets in the proposed model is used in the Linear Regression. The *mean absolute error* is used as a loss function during training since the training set consists of a range of numeric values. For optimization, *Adam* optimizer is used. The Machine Learning model achieves an accuracy of 94%.

7.5. Evaluation Metrics

The proposed solution can be evaluated by comparing it with existing parallel processing solutions that are available. The proposed model can be compared against a standard GPU baseline which performs basic GPU execution. The libraries and methods that are used to compared are Compress time, Split time, Net train time, Compress communication time, and Split communication time.

The model was trained with a different number of dataset as mentioned above and the results are displayed. The results are displayed as a comparison done against other parallel processing models as mentioned earlier. In Figure 6.1, we can see that the compress time taken for CUDA proposed model is higher than other models even though the train_time is significantly smaller. This is because of the high data transfer rate between GPU memory and secondary storage.

Table 3 Time taken to train ijcnn1 dataset by each parallel model

MODEL	Compress time	Split_time	Net_train_time
Sequential	2.1385	0.2987	0.3651
OpenMP	1.6421	0.5890	0.2689
OpenMPI	0.7632	0.08142	0.3554
CUDA	3.1527	0.03678	0.3605

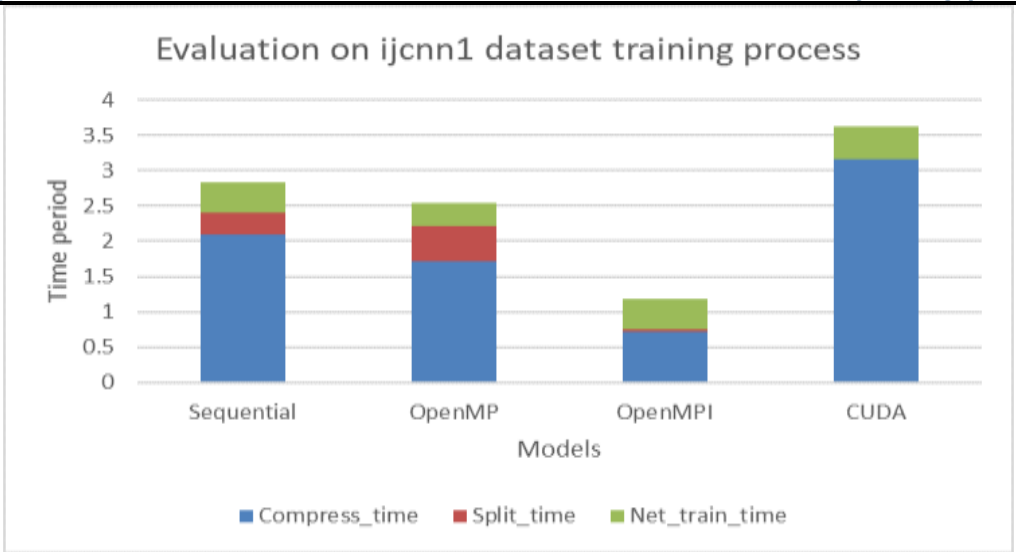

Figure 2 Time taken to train ijcnn1 dataset by each parallel model

In Figure 2 and 3, we notice that the proposed model only requires smaller training time than the other parallel processing models thus showing that the proposed model has achieved its goals.

Table 4 Time taken to train big_size small feature dataset by each parallel model

MODEL	Compress_time	Split_time	Net_train_time
Sequential	177.2985	2.588265	27.18194
OpenMP	84.33631	3.148343	27.32264
OpenMPI	67.78642	0.091329	29.95133
CUDA	72.28389	0.043678	21.2121

Table 5 Time taken to train middle_size small feature dataset by each parallel model

MODEL	Compress_time	Split_time	Net_train_time
Sequential	17.4874	2.28675	2.704877
OpenMP	9.791004	2.621128	2.808156
OpenMPI	6.619429	0.175356	3.065417
CUDA	7.358692	0.044515	2.379483

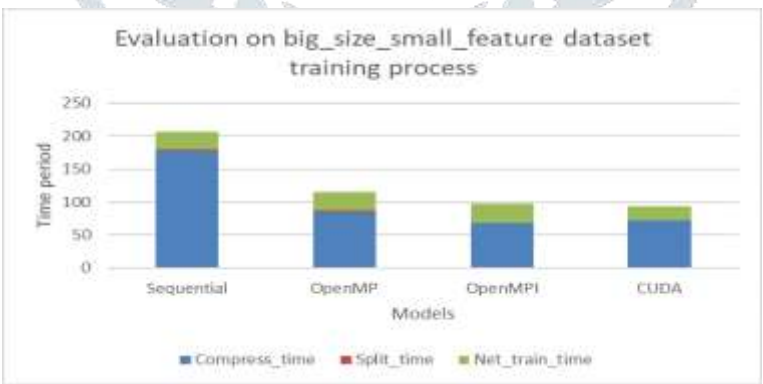

Figure 3 Time taken to train big_size small feature dataset by each parallel model

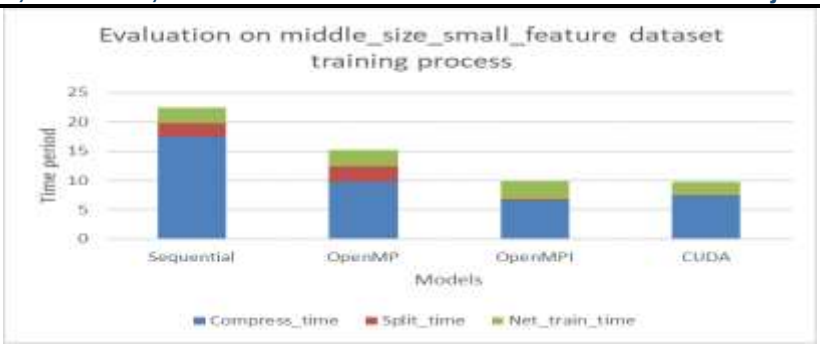

Figure 4 Time taken to train middle_size small feature dataset by each parallel model

The evaluation of the proposed model shows a significant decrease of communication time and net split time in training decision trees. This decrease shows that we have balanced the workload among CUDA cores for the majority of time.

7.6. Implementation Output Of Ml Model

The Machine Learning model achieved an accuracy of 94%.The weighted coefficients of the individual attributes are calculated using the regression model.

```
linear_model.layers[1].kernel
```

```
<tf.Variable 'dense_1/kernel:0' shape=(2, 1) dtype=float32, numpy=
array([[34.942],
       [15.018]], dtype=float32)>
```

Figure 5 Output of the weighted kernels indicating the significant attributes values

The loss function of the model is displayed in Figure 5. From the graph, it is observed that as epoch reaches 100 the loss function turns into a flat line indicating convergence of the model.

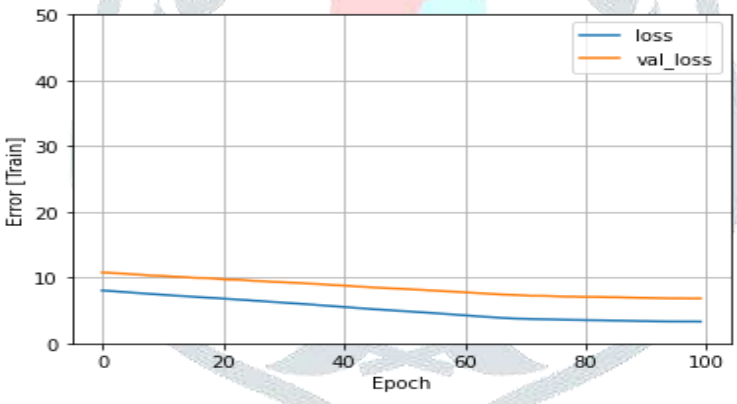

Figure 6 Loss function of the Linear Regression Model

The mathematical model can be used to determine the training time of the proposed model with the calculated weighted coefficients of COMPRESS_TIME and SPLIT_TIME of the model. This indicates that performance of the proposed model depends on the time taken for COMPRESS and SPLIT operations.

8. CONCLUSION

In this work, we have proposed a decision tree based solution with probabilistic output. The challenges of developing a highly parallelized GPU based solution for decision trees are (i) repeated accesses to the high latency GPU memory (ii) the requirement of much larger memory footprint than the GPU memory. Our proposed model reduces high latency memory accesses and memory consumption through shared memory, both data and feature parallelism. Experimental results have shown that our decision tree outperforms the CPU version by an order of magnitude. Our proposed model shows significant improvement in terms of performance as we have balanced the workload among all CUDA cores thus significantly not having any impact on processing speed and memory. As the results indicate, data transfer rate is large for smaller datasets but when the dataset begins to expand our proposed model reduces the influence of data transfer rate to negligible amount.

REFERENCES

[1] Anurag Srivastava, Eui-Hong Sam Han, Vipin Singh and Viney Kumar (1998) "Parallel formulations of decision-tree classification algorithms", Proceedings on International Conference on Parallel Processing (Cat.No.98EX205), Minneapolis, MN, USA, 1998, pp. 237-24.

[2] David Svantesson, (2018) “Implementing Streaming Parallel Decision Trees on Graphic Processing Units”, Degree Project in Computer Science and engineering, 2nd cycle, Sweden.

[3] Huan Zhang, Cho Jui-Hsieh (2017) “GPU-acceleration for Large-scale Tree Boosting”, June 2017, arXiv:1706.08359[stat.ML].Reference contd.

[4] Johannes Gehrke, Raghu Ramakrishnan, Venkatesh Ganti (2000) “RainForest—A Framework for Fast Decision Tree Construction of Large Datasets”. Data Mining and Knowledge Discovery 4, 127–162.

[5] John C.Shafer, Rakesh Agrawal, Manish Mehta (1996) “SPRINT: A Scalable Parallel Classifier for Data Mining”. Proceedings on the 22nd International Conference on Very Large Data Bases (VLDB '96). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 544–555.

[6] Manish Mehta, Rakesh Agrawal, Jorma Rissanen (1996) ”SLIQ: A fast scalable classifier for data mining” In: Apers P., Bouzeghoub M., Gardarin G. (eds) Advances in Database Technology — EDT '96. Lecture Notes in Computer Science, vol 1057, Springer, Berlin, Heidelberg.

[7] Mitchell, Rory & Adinets, Andrey & Rao, Thejaswi & Frank, Eibe (2018) “XGBoost: Scalable GPU Accelerated Learning”, arXiv:1806.11248 [cs.LG].

[8] Ping Li, Christopher J.C. Burghes and Qiang Yu (2007) "Learning to rank using classification and gradient boosting", NIPS ’07, Proceedings of the 20th international Conference in Neural Processing systems, December 2007,897-904.

[9] Qi Meng, Guolin Ke, R. Taifeng Wang, Wei Chen, QIwei Ye, Zhi-Ming Ma and Tie-Yan liu (2016) "A communication-Efficient Parallel algorithm for Decision tree" NIPS’’16 Proceedings of the 30th international conference on neural information processing systems, December 2016,1279-1287.

[10] Ruoming Jin, Gagan Agrawal (2003) ”Communication and Memory Efficient Parallel Decision Tree Construction”, 2003 SIAM International Conference on Data Mining, May 2003, DOI: 10.1137/1.9781611972733.11.

[11] Yael Ben haem, Elad Tom-Tov (2010) "A streaming parallel decision tree algorithm," in Journal on Machine Learning research 11(02/2010) 849-872.