Master's Thesis

GPU Accelerated Adaptive Random Forest

Junji Watanabe

Supervisor Yasushi Inoguchi

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

Conferment March, 2023

Abstract

The rapid growth of mobile, wearable, and IoT devices is dramatically increasing the amount of data being generated every moment. Learning from a high-volume data stream is extremely challenging. Traditional machine learning methods are not suitable for online data stream applications because they are designed for offline batch processing, where the entire data set needs to be kept somewhere in the system. Not many researches has been done on data stream learning, especially with GPU acceleration.

This thesis examines the effectiveness of GPUs in data stream learning by accelerating Adaptive Random Forest (ARF) on GPUs. ARF is an ensemble learning method based on Hoeffding Tree, an incremental decision tree algorithm.

Experimental results show that my GPU ARF implementation outperforms the CPU ARF implementation in terms of execution time while maintaining prediction accuracy. It also studies the trade-offs between GPU and CPU by experimenting with various learning conditions.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In the era of Big Data, due to the rapid growth of mobile devices, wearable devices, IoT devices, etc., the amount of data being generated every moment is drastically increasing. However, learning from a high-volume and continuous data stream is extremely difficult. Over the past few decades, learning from big data using machine learning and deep learning has been studied and widely applied to real-world applications. However, many of these are designed for offline batch processing, where the entire data set needs to be kept somewhere in the system, and are therefore **not** suitable for data stream applications.

1.1 Problem Statement

Classification is a problem of predicting a class or label y from a data instance $\boldsymbol{x}$ with multiple attributes. In data stream classification, data instances for training are not always available. Instead, they are provided as a continuous stream of data at a fast pace. Prediction requests are expected to arrive at any time, and the model makes predictions based on the current state.

In this thesis, it is assumed that the target labels are provided at training time (supervised learning). Unsupervised or semi-supervised learning, in which no or partial labels are provided, is not covered.

1.2 Data stream classification methods

Algorithms for data stream learning have the following unique requirements that are not necessary for typical machine learning problems.

- Incremental update: The algorithm needs to update a model incremen-

tally as it observes a new data instance, since the whole dataset can't be retained.

- Fast: To handle a massive volume data stream, the computation needs to be done very fast.

1.2.1 Hoeffding Tree

Hoeffding tree algorithm [11] is an incremental decision tree algorithm that can learn from a data stream without having to keep all data instances locally. As a new data instance arrives, it updates the statistics of the training nodes and determines the partitioning of the nodes based on the Hoeffding bound, which probabilistically approximates the optimal split.

1.2.2 ARF: Adaptive Random Forest

Adaptive Random Forest or ARF [14] is a random forest algorithm for data stream learning based on the Hoeffding tree algorithm. While ARF showed good accuracy for for many types of data streams, it's computationally expensive since Random Forest algorithm trains many trees in parallel. Training Hoeffding trees in multiple CPU cores in parallel achieved speed up ARF without losing accuracy. However, further speed up would make it possible to apply ARF for massive volume data streams.

1.2.3 GPU Random Forest

GPUs (Graphics Processing Units) are massively parallel processors and are widely used to accelerate machine learning and deep learning in offline learning environments.

GPU Random Forest or GPU RF [18] utilizes GPU to speed up a Random Forest technique based on Hoeffding trees. GPU RF showed that GPU can speed up Random Forest and Hoeffding tree algorithms. However, the proposed techniques can be applied to limited situations. For example, only binary attributes are supported. Another example is that the max tree depth needs to be limited. The detailed reasons will be explained in the later in this thesis.

It's important to resolve those limitations to apply GPU to other data stream learning.

1.3 Objectives

The mission of this thesis is to improve the throughput of data stream learning by utilizing GPU. The larger capability for high-volume data stream, the wider applications it has.

This thesis proposes **GPU ARF**, a GPU version of ARF. GPU ARF has the following objectives:

1. Speed up ARF algorithm with GPU.

   The speed will be compared with ARF with multi-core CPU.

2. Resolve the max tree depth limitation that GPU RF has.

   Without the tree depth limitation, GPU will be applicable to more complex problems.

3. Keep the accuracy

   The speed up should not bring a large penalty on accuracy in comparison to ARF.

1.4 Contributions

The main contribution of this thesis is to apply a memory pool technique to allocate memory for tree nodes dynamically. Since there are many computations running in parallel, they may conflict with each other to obtain a memory block from the pool. However, the memory pool successfully handles parallel accesses and it doesn't lose performance. With the dynamic memory allocation, GPU ARF can grow for the data streams used in the experiments without causing out-of-memory.

Another contribution is that, to the best of my knowledge, other studies do not take a batching strategy. This thesis examines the trade off between speed and accuracy loss, and shows that batching data instances brings speed up while keeping the accuracy reasonably.

GPU ARF follow some parallel implementation techniques that are used in other studies which include assigning a GPU block for a tree and calculating split candidates in parallel with parallel reductions.

1.5 Thesis outline

Chapter 2 provides background for this thesis by reviewing related work. Chapter 3 describes the details of the proposed methods to accelerate ARF

on GPU. In Chapter 4, I evaluate the performance through experiments using GPU ARF. Chapter 5 describes the experimental results and future work. Chapter 6 summarizes this thesis.

Chapter 2

Background

This chapter explains the background of data stream learning, and related works. First, I will explain the difference between data stream learning and traditional batch learning. Second, Hoeffding tree, a foundation of data stream learning technique, and variants of Hoeffding Tree will be explained. Lastly, the concept of GPU acceleration will be explained.

2.1 Offline batch learning and online learning

Typical machines learning models are trained with batch learning. Batch learning generates the best model by learning from the whole data set repeatedly. The problem of the batch learning is that the whole data set may not always be available for training or not feasible due to the limit of storage or memory. Online machine learning is a learning method in which data becomes available sequentially and is used to update the best model at each step. Online machine learning algorithms can learn incrementally from data streams that generates infinite data instances, and can dynamically adapt to the new pattern in the data. Most of the time, online machine learning algorithms can forget the previous knowledge for the adaptation to the new pattern.

2.2 Decision trees

Decision tree is a traditional machine learning method for supervised classification problems.

A decision tree consists of intermediate nodes and leaf nodes. An intermediate node has conditions for an attribute of data instances. A data instance

traverses a decision tree by following the conditions of the intermediate nodes until a leaf node. The leaf node makes a prediction for the data instance.

Training a decision tree is done with an offline batching algorithm. Given a initialized root node, the decision tree is trained as follows:

1. Process all data instances to derive the leaf nodes.
2. Derive the best split for each leaf node by calculating all possible splits and merits.
3. Split the leaf nodes if the merits exceeds the threshold.
4. Repeat the steps above until the tree stops growing.

There are multiple ways to evaluate the merits of decision tree splits. In this thesis, information gain is used as the split criteria.

Information gain is based on the concept of entropy from information theory. The entropy for the node N is defined as Equation 2.1, where n_{total} is the total number of data instances, J is the number of classes, n_{y_i} is the number of data instances with the class y_i on the node. Given a node N_{parent} and a split condition a, the information gain is defined by Equation 2.2. The information gain is the difference between the entropy before the split and the entropy after the split, which is the sum of the entropy of all child nodes. The best split condition can be derived by calculating all possible split conditions, and choosing the one that maximizes the information gain.

$$\begin{aligned} n_{\text{total}} &= \sum_{i=1}^{J} n_{y_i},\ p_i = \frac{n_{y_i}}{n_{\text{total}}} \\ H(N) &= I(n_{y1}, n_{y_2}, ...) = -\sum_{i=1}^{J} p_i \log_2 p_i \end{aligned} \tag{2.1}$$

$$G(N_{\text{parent}}, a) = H(N_{\text{parent}}) - \sum^{\text{All children}} H(N_{\text{child}}|a) \tag{2.2}$$

Although the decision tree algorithm is simple, decision tree based algorithms are still used for many machine learning problems because of the simplicity and the accuracy. For example, Random Forest [8] and Gradient tree boosting [13] are based on decision trees. XGBoost [9], LightGBM [16] are often used in machine learning competitions such as Kaggle [3].

However, it's not possible to use decision trees for data stream learning because the algorithm requires the whole dataset to calculate information gain.

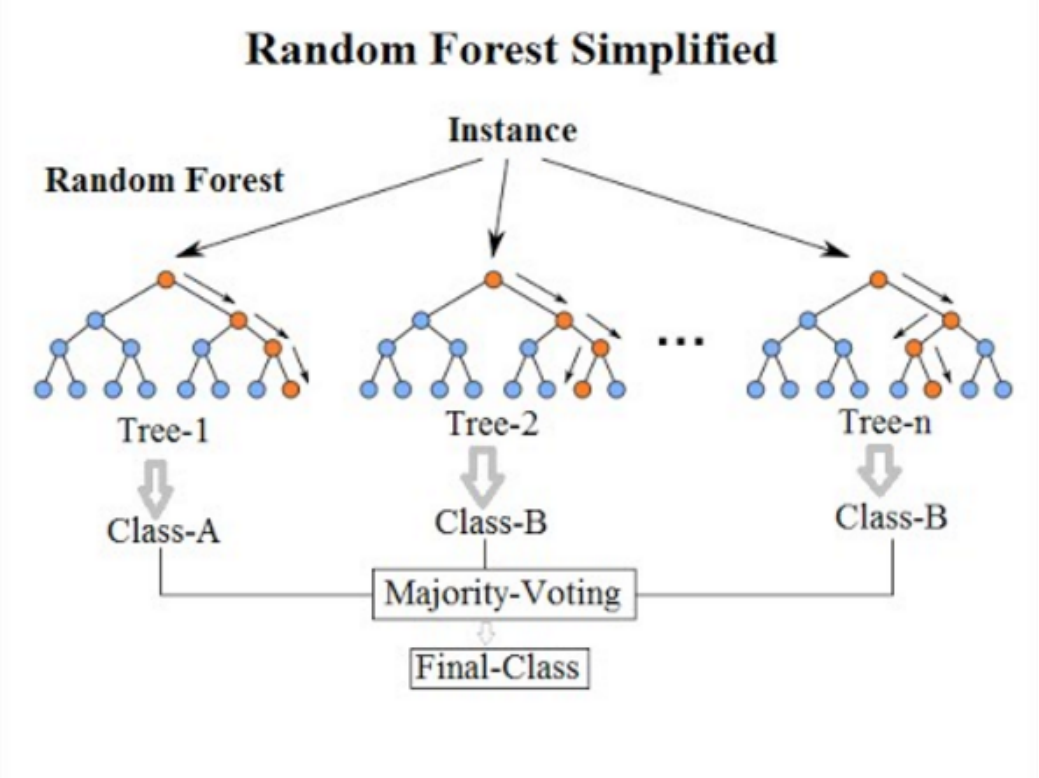


Figure 2.1: Random forest diagram by Venkata Jagannath, under a CC BY-SA 4.0 license, via Wikimedia Commons [15]

2.3 Random Forests

Random forests [8] is an ensemble learning method for classification and regressions problems. Figure 2.1 is a diagram of a random forest. A random forest consists of many decision trees, which are trained with a sub set of datasets and/or attributes independently each other. For classifications, all trees make predictions and the majority vote is chosen as the final predict.

For the same reason with decision trees, it's not possible to use random forest for data stream learning.

2.4 Hoeffding Tree: Very Fast Decision Tree

Hoeffding Tree or Very Fast Decision Tree (VFDT) [11] is an incremental decision tree that can learn from data stream, and is widely used in this area.

Algorithm 1 describes the procedures of a Hoeffding Tree. As opposed to

traditional decision trees, Hoeffding Trees don't require the whole data sets and can learn from data streams incrementally. As a data instance arrives, a Hoeffding tree processes it as follows:

1. Find the leaf node l for the data instance $\boldsymbol{x}$ by traversing the tree.
2. Increment attribute counters of l for the sets of (y, X_i, x_{ij}), where y is the target class, X_i is an attribute, x_{ij} is the value for attribute X_i.
3. Calculate the merits for all possible split candidates of the leaf node l.
4. Choose the best and the second best split candidates m_a, m_b.
5. Calculate the Hoeffding bound ϵ of the leaf node l.
6. Split the leaf node l, if the difference between m_a and m_b is greater than the Hoeffding bound ϵ.

The Hoeffding bound ϵ can be calculated by the equation 2.3 with $1 - \delta$ confidence where R is a range of split merits and n is the number of observed instances.

$$\epsilon = \sqrt{\frac{R^2 \ln 1/\delta}{2n}} \tag{2.3}$$

There are many variants of Hoeffding Tree. Hoeffding Adaptive Tree (HAT) [5] is proposed to adapt concept drifts. Manapragada et al.[17] made Hoeffding Tree more efficient. In this thesis, only Hoeffding Tree is considered since the same GPU parallelization techniques can be applied to those variants.

2.5 Distributed stream learning

Vertical tree (VHT) [10] is a distributed version of Hoeffding Tree that partitions data instances in terms of attributes. The distributed approach is a different strategy that parallelize computations. However, this thesis assumes edge computing rather than cloud computing. VHT is designed for centralized cloud systems with a cluster that consists of many machines. Therefore, the comparison between distributed strategies and GPU-based strategies is out of the scope of this thesis.

Algorithm 1: Hoeffding Tree

Input:

S is a sequence of data instances
$\boldsymbol{X}$ is a set of attributes.
$G(.)$ is a split criterion function

begin

Initialize HT as a Hoeffding tree.

Let $\boldsymbol{x}$ as attributes of a data instance
Let y as a target class of a data instance
foreach $(\boldsymbol{x}, y) \in S$ **do**

Find a leaf node l from HT for $\boldsymbol{x}$

`/* Update attribute statistics */`
foreach attribute value $x_{ij} \in \boldsymbol{x}$ **do**
IncrementCounter(l, y, X_i, x_{ij})
end
`/* Compute the merits of all possible splits */`
foreach attribute $X_i \in \boldsymbol{X}$ **do**
foreach attribute values $X_{ij} \in X_i$ **do**
Let s_{ij} be a split condition with (X_i, X_{ij})
Let m_{ij} be a merit for s_{ij} from $G(l, s_{ij})$
end
Let s_i, m_i be the best split and merit for the attribute X_i
end
Let m_a, m_b be the best and the second best merits for s_a and s_b respectively.

Let ϵ to be the Hoeffding bound of the leaf node l.
if $m_a - m_b \geq \epsilon$ **then**
Split(l, s_a)
end
end
end

2.6 Adaptive Random Forest: ARF

Random Forest [8] is a common technique to make decision tree more robust by learning many decision trees with different sub sets of data instances or

attributes. Adaptive Random Forest or ARF [14] applied the Random Forest technique with rees as base learners so that it can learn from data stream. Note that training the trees in ARF can be done independently in parallel.

2.6.1 Parallel Adaptive Random Forest

The original research of ARF [14] demonstrated that multi-core CPUs can speedup ARF with no degradation for the classification performance in comparison to a serial CPU implementation. The number of trees in an ARF can be large. For example, the ARF implementation of MOA [6] uses 100 trees by default. GPU has many streaming multiprocessors (SMs) which are capable of handling many trees. For example, NVIDIA a100 has 108 SMs. This is a motivation to parallelize ARF further in GPU.

2.6.2 MOA: Massive Online Analysis

MOA (Massive Online Analysis) [6] is an open source project for data stream learning written in Java. It includes many online learning algorithms such as classification, regression, clustering, and is used for scientific evaluation. This thesis uses the ARF implementation in MOA as a reference.

2.7 GPU acceleration for Decision Trees and Random Forests

Toby [20] proposed an implementation of decision tree and random forest on a GPU for image classification tasks using Direct 3D. And Marron et al.[18] proposed a random forest of Hoofing Tree on GPU (GPU RF) for evolving data stream. Wu [21] utilizes GPU parallelism for adapting concept drifts in data stream.

They all allocate memory to all possible tree nodes. This makes a limitation on the tree depth because the number tree nodes increases exponentially to the tree depth. This is problematic when learning complex problem where trees need to be large enough to represen the complexity.

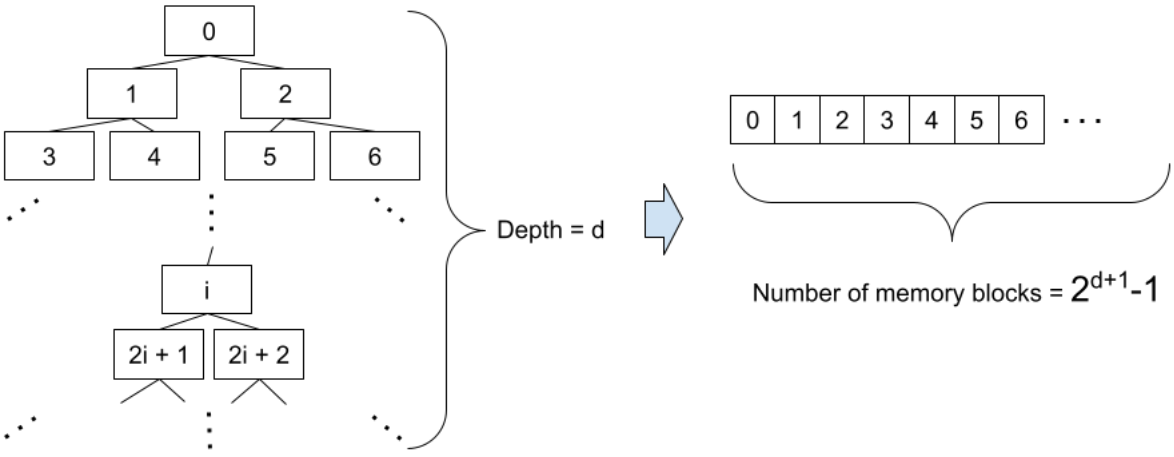


Figure 2.2: Array representation of a binary tree

2.8 Random Forests of Very Fast Decision Trees on GPU for Mining Evolving Big Data Streams

As mentioned before, Marron et al.[18] proposed a GPU version of VFDT based random forest (GPU RF), and is most similar to this thesis. However, GPU RF has a limitation of the max tree depth because it pre-allocates memory for all possible tree nodes in advance. GPU RF was compared only with a serial CPU implementation in Java. Thus, the overhead that comes from the programming language is unclear. Since it was not compared with a multi-core CPUs implementation, it is also unclear how large benefit GPU has over multi-core CPUs.

According to the paper, GPU RF completes the computation for Covertype dataset in 0.330 seconds, which is 1000 times faster than the CPU version of Random Forest. However, I was not able to reproduce this speed by myself since just reading the data instances and sending them to GPU take longer than 0.33 sec. Although, I tried to contact with an author to clarify what was measured as the execution time, I couldn't get any answer. Therefore, GPU ARF could not be compare with GPU RF.

The following sub sections summarize the techniques to utilize GPU in GVFDT and GPU RF.

2.8.1 Tree Layout

As described in Figure 2.2, GVFDT represents a binary decision tree in an array so that the offsets of child nodes can be easily calculated from the parent nodes. Given a node at offset i, its left child will be at offset $2i + 1$ and its right child will be at offset $2i + 2$. Similar techniques are used in other GPU accelerations, too [20] [21].

A drawback of this approach is that the memory for all nodes need to be allocated in advance regardless of whether used or not. The number of nodes is $2^{d+1} - 1$, where d is the max tree depth. As the required memory increases exponentially, the max tree depth needs to be limited to avoid exceeding the available GPU memory.

2.8.2 Leaf Nodes

The leave nodes in the tree array have IDs which are used to find counters stored in another array. The classes of leaf nodes are also stored in a different array. Decoupling the classes and counters of leaf nodes from the tree makes it easy to compute prediction and calculate information gains in parallel.

2.8.3 Node splits

Calculating information gain is also parallelized by using 2 parallel reductions. After getting information gain for all leaves, the candidates are sorted by parallel sort to find best splits to calculate Hoeffding bounds.
This thesis also uses similar techniques.

2.9 GPUs

GPUs are processors for highly parallel tasks, originally designed for graphic processing. GPUs began to be used for general-purpose processing in the early 21st century. There are many scientific tasks that are suitable for parallel processing such as fluid simulation, protein folding, and so on. GPUs also work well for machine learning and deep learning. Deep learning requires to train neural networks which consist of a massive amount of nodes connected with many layers.

2.9.1 GPU Architecture and CUDA Programming model

Figure 2.3 describes a simplified GPU architecture. To process data on a GPU, the data needs to be transferred from the host CPU memory to the

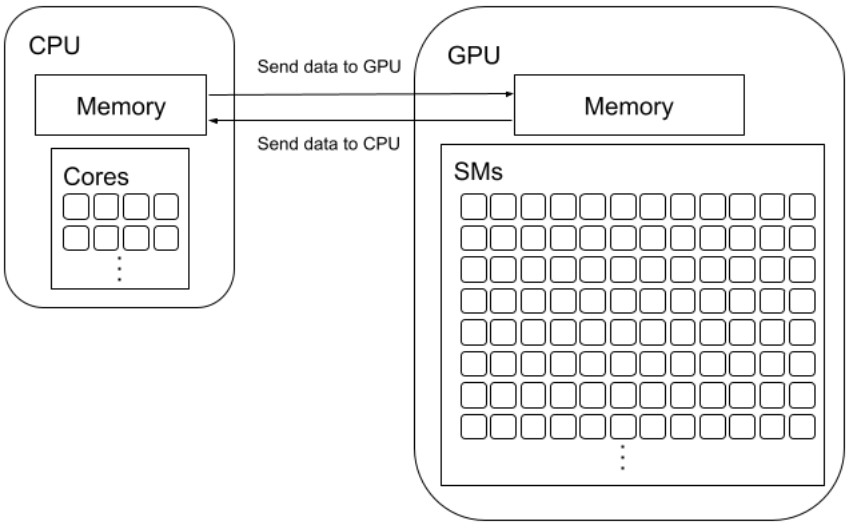


Figure 2.3: GPU Architecture

Figure 2.4: GPU kernel in CUDA

```
__global__ void SomeKernel(float *inputs, float *outputs) {
  ...
};

int main() {
  ...
  SomeKernel<<<NumBlocks, NumThreadsPerBlock>>>(inputs,
      outputs);
}
```

GPU memory. After processing, the results need to be transferred to the host CPU memory. A GPU consists of multiple Streaming Multi-processors (SMs). For example, A NVIDIA a100 [4] GPU has 108 SMs.

2.9.2 CUDA Programming model

A GPU **kernel** is a function that gets executed on GPU. A GPU kernel runs K times where K is the number of CUDA threads. As shown in Figure 2.4, a GPU kernel is declared with `__global__` specifier, and triggered with `<<<NumBlocks, NumThreadsPerBlock>>>` that specifies the number of CUDA blocks and the number of CUDA threads per block.

A CUDA **block** is a group of CUDA threads. CUDA blocks are grouped

Figure 2.5: Mapping between GPU hardware and CUDA programming model

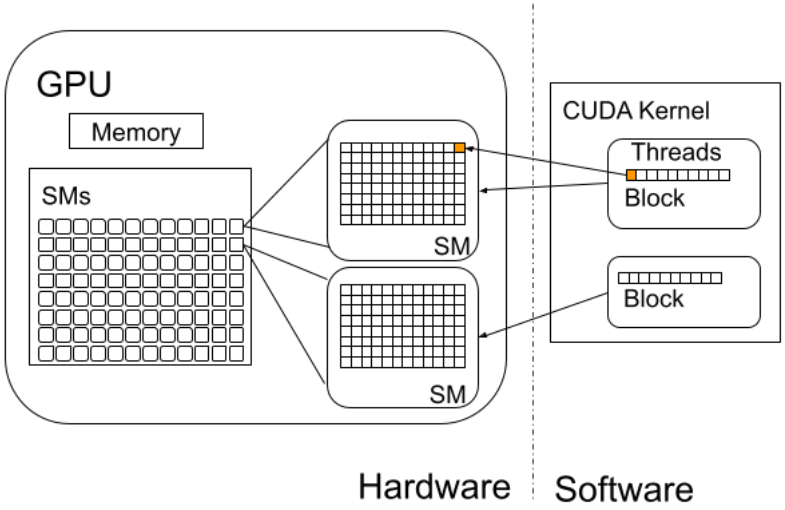
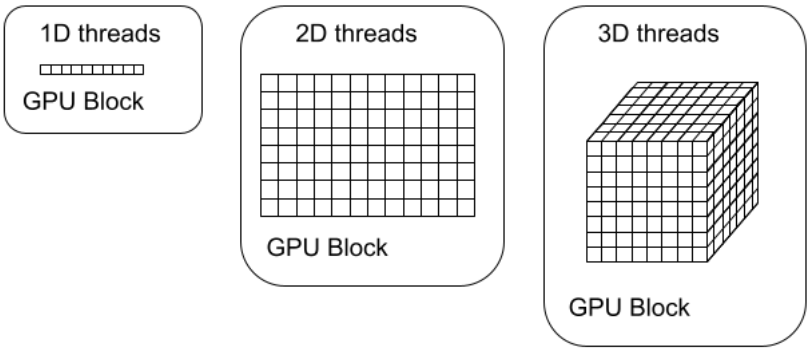


Figure 2.6: GPU Block dimensions



in to a CUDA **grid**. A CUDA kernel is triggered as a grid of blocks of threads. Figure 2.5 illustrates the mapping between GPU hardware and CUDA programming model. A CUDA blocks run on the same SM. In a CUDA block, each thread runs on a GPU core runs in parallel.

As described in Figure 2.6, a GPU Block can also have 2 dimensional or 3 dimensional threads. This is convenient when mapping multi dimensional problems to CUDA program.

2.9.3 GPUs for Random forests construction

cuML is a CUDA library developed by NVIDIA that supports many parallel algorithms for machine learning. cuML also supports the Random Forest

algorithm, which is 20 to 45 times faster than the CPU version of Random Forest for the training [19]. This speedup is achieved by training all trees independently and computing all candidate partitions in parallel. As mentioned above, the original random forest cannot be used for data stream learning. However, this suggests that accelerating ARF on a GPU can be effective.

2.10 Summary

In this chapter, the background of data stream learning and relevant works, the concept GPU acceleration were reviewed.

Decision trees and Random Forest are traditional machine learning techniques. However, they are not suitable for data stream learning because they require retaining the whole dataset. Hoeffding Trees and Adaptive Random Forest are algorithms that approximate Decision trees and Random Forest respectively to be able to learn from data streams.

There are relevant studies where GPU was utilized to accelerate Hoeffding Trees and Random Forest. However, they all take a strategy to allocate memory to all possible tree nodes. This makes a limitation on the max tree depth as the number of nodes increases exponentially to the tree depth.

Lastly, the concept of GPU acceleration and the CUDA programming model were reviewed.

Chapter 3

GPU ARF implementation

3.1 Introduction

In this thesis, I propose GPU ARF, a GPU version of ARF, that accelerates data stream learning by exploiting the massive parallelism of GPUs. In this chapter, I explain how to utilize a GPU for this algorithm, showing implementation points of GPU ARF.

First, I will explain an approach to resolve the existing constrains in GPU RF. Next, I will explain the speed up achieved by GPU's parallelism.

3.2 Approach to existing constrains

This section explains how GPU ARF resolves the max tree depth limitation, which exists in GPU RF [18] and other GPU Hoeffding Tree techniques.

3.2.1 Static tree node memory pre-allocation

In the related works with GPU [20] [18] [21], the algorithms pre-allocate the GPU memory for all possible tree nodes statically.

As Figure 2.2 describes in the previous chapter, a binary tree can be represented as a single array of tree nodes. This allows Hoeffding Trees to grow without allocating new memory blocks for child nodes at node splits.

The reason behind this strategy is that GPU is not good at dynamic memory allocation. Although CDUA has `cudaMalloc` API to allocate memory dynamically, it is an expensive operation. GPU applications should avoid calling `cudaMalloc` as much as possible. Since Hoeffding Trees need to split leaf nodes often, using `cudaMalloc` at every node split would have a substantial overhead. In my preliminary experiments, `cudaMalloc` didn't provide a

practical speed.

The drawback of this strategy is that the number of possible nodes in a tree increases exponentially $2^{d+1} - 1$, where d is the tree depth. Therefore, it needs to limit the max tree depth to avoid exceeding the available GPU memory.

3.2.2 Dynamic memory allocation

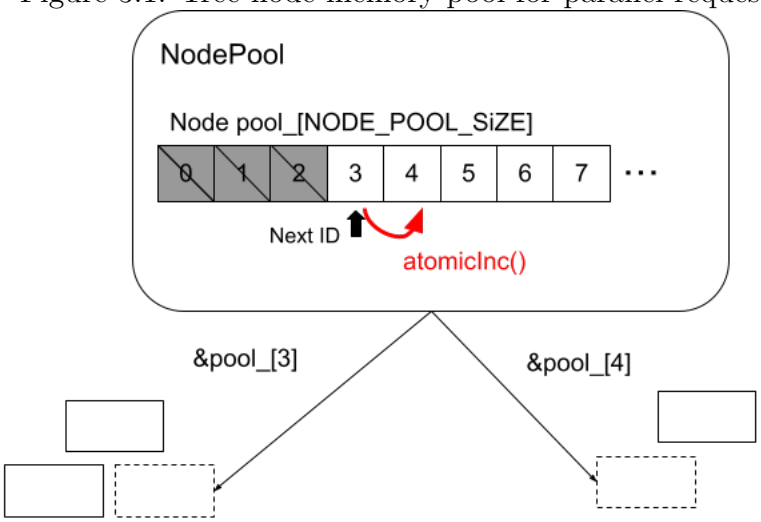
In GPU ARF, the memory allocation for tree nodes is done dynamically.

Memory pools

GPU ARF prepares dedicates memory pools that assigns memory addresses for node splits in parallel. Figure 3.2 are the structures of the memory pools, and Figure 3.1 is a simplified diagram for the tree node memory pool.

The pools consist of an ID for next allocatable memory address and an array of pre-allocated memory. Getting a memory address is done by incrementing `next_id_` atomically. It uses CUDA's `atomicInc` API to this operation. Thanks to the CUDA's atomic operation, this simple logic allows multiple GPU threads to take a memory address in parallel without conflicts.

Figure 3.1: Tree node memory pool for parallel requests



GPU ARF Node

Figure 3.3 partially describes the structures of tree node. A `Node` holds the pointers to the child nodes. In this case, this is a binary tree. Thus, there

Figure 3.2: Memory pools for GPU ARF Node

```
struct LeaningStatePool {
  uint32_t next_id_;
  LearningState pool_[LEARNING_STATE_POOL_SIZE];
}
struct NodePool {
  uint32_t next_id_;
  Node pool_[NODE_POOL_SiZE];
}
```

Figure 3.3: Data structures for GPU ARF Node

```
struct Node {
  LearningState *learning_state_;
  Node *child_nodes_[2];
  ...
};
struct LearningState {
  AttributeObserver attr_observers_[NUM_LEARNING_ATTRS];
  uint8_t learning_attrs_[NUM_LEARNING_ATTRS];
  ...
};
struct AttributeObserver {
  uint32_t stats_[NUM_CLASSES][NUM_ATTR_VALUES];
}
```

are two child nodes for a parent node. When splitting the node, it requests two memory addresses and stores in `child_nodes_`.

It also has a pointer to `LearningState`, which holds multiple `AttributeObserver`. `AttributeObserver` is just a struct that consists of counters to record the observed class and attribute values.

After node split, `LearningState` is not necessary anymore. Therefore, it can be returned to the memory pool. However, the memory pool doesn't have this advanced mechanism yet.

Learning attributes

Note that `NUM_LEARNING_ATTRS` is not equal to the number of attributes that a data instance has. In ARF, only a subset of attributes is randomly chosen for each node independently to improve the robustness. Therefore, the chosen attributes are stored to `learning_attrs_` property when initialized. This avoids allocating redundant memory to the attributes that are not used for

training.

3.3 Speed up with GPU parallelism

This sections explain how speed up of ARF can be obtained with GPU by following the whole routine. Similar techniques are used in the related works with GPU.

The overview of the GPU ARF implementation is shown in Algorithm 2. The functions that ends with `Kernel` refer to GPU kernel launches. After initialization steps, the following steps repeat as the data stream continues.

1. On CPU, data instances from the data stream are buffered until the batch size B.
2. The batch of data instances is sent to the GPU memory.
3. On GPU, `PredictAndTrainKernel` does prediction and training on the given data instances.
4. On GPU, `CalculateSplitMeritKernel` calculates split merits for all the candidates.
5. On GPU, `SplitNodeKernel` executes node splits when necessary.

The following sections explain the details of them.

Algorithm 2: GPU ARF

Input:

T is a number of trees
S is a sequence of data instances
B is a batch size of data instances

begin

Initialize $\boldsymbol{b}$ as an empty batch of data instances.
Initialize f as an ARF with T trees.

Let $\boldsymbol{x}$ as attributes of a data instance
Let y as a target class of a data instance
foreach $(\boldsymbol{x}, y) \in S$ **do**

`/* Collect data until the batch size */`
if Size($\boldsymbol{b}$) $< B$ **then**

`/* Append the data instance to the batch */`
Append($\boldsymbol{b}$, $\boldsymbol{x}$, y)
continue
end

Send $\boldsymbol{b}$ from CPU memory to GPU memory

`/* Do prediction and training */`
Initialize $\boldsymbol{p}$ as a list of predictions.
Initialize $\boldsymbol{l}$ as a list of reached leaf nodes.
PredictAndTrainKernel$<<<T, B>>>(f, \boldsymbol{b}, \boldsymbol{p}, \boldsymbol{l})$

`/* Calculate split merits for all the candidates */`
Initialize $\boldsymbol{s}$ as a list of split merits.
CalculateSplitMeritKernel
$<<<T, B \times N_{\text{attributes}} \times N_{\text{attribute values}}>>>(\boldsymbol{l}, \boldsymbol{s})$

`/* Split nodes based on the calculated merits */`
SplitNodeKernel$<<<T, B>>>(\boldsymbol{l}, \boldsymbol{s})$

Reset $\boldsymbol{b}$ to an empty batch of data instances
end
end

3.3.1 Data batching

Data transfer from host CPU memory to GPU memory is a well-known bottleneck for GPU applications. To reduce this delay, GPU applications often process large amounts of data at a time. As shown in Algorithm 2, the proposed method also sends T data instances as a batch, where T is up to 128 in my experiments.

In data stream learning scenarios, concept drifts may occur, and the model needs to keep updated as fast as possible. A large batch size may cause a delay in training, which may result in lower accuracy than a small batch size or no batching. However, the smaller the batch size or no batching, the slower the computational speed. This trade-off between speed and accuracy is important in this theme.

3.3.2 Prediction and training

`PredictAndTrainKernel` in Algorithm 2 is a GPU kernel that does predictions and training with the given data instances.

As introduced in the previous chapter, ARF consists of multiple decision trees. Since these trees are independent, most training and prediction steps can be done independently to other trees. As described in Figure 3.4, `PredictAndTrainKernel` assigns each tree to one GPU block. On the GPU block side, each data instance is assigned to one GPU thread.

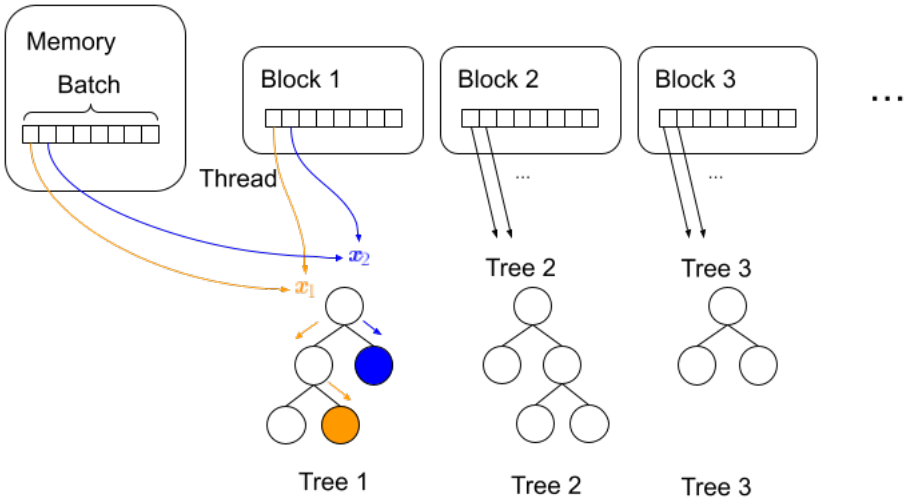


Figure 3.4: Block and thread assignments for PredictAndTrainKernel

Algorithm 3 shows the details of `PredictAndTrainKernel`.

First, each GPU thread traverses the tree to a leaf node of the given data instance. Each intermediate node has an evaluation rule to route a data instance to an appropriate child node. The evaluation result is stored as a boolean variable to avoid branches.

Next, the thread predicts the class of the assigned data instance based on the statistics of the leaf node. In this thesis, it uses a majority class strategy for prediction. That is, the most observed class of the leaf node is selected as the predicted class. The prediction results are stored to $\boldsymbol{p}$. Note that there are other threads in other GPU blocks for the same data instances. Therefore, updating $\boldsymbol{p}$ is done in an atomic manner. The arrived leaf node is also stored to $\boldsymbol{l}$ to use in the following kernels.

Lastly, the leaf node updates the statistics with the given instance $\boldsymbol{x}$ and the true class y, which is provided somehow from the outside of the system.

Algorithm 3: PredictAndTrainKernel

Input:

f is an ARF
$\boldsymbol{b}$ is a batch of data instances
$\boldsymbol{p}$ is a list of predictions to store prediction result
$\boldsymbol{l}$ is a list of leaf nodes to store tree traversal result

begin

Let b is the GPU block ID
Let t is the GPU thread ID

`/* Load the assigned tree from GPU global memory */`
$tree$ = GetTree(f, b)

`/* Load the assigned data instance from GPU global memory */`
$\boldsymbol{x}$ = GetDataInstance($\boldsymbol{b}$, t)

`/* Traverse the tree until a leaf node */`
Let n to be the root node of $tree$.
while n is not leaf **do**
    Let r to be a boolean.
    r = GoRight(n, $\boldsymbol{x}$)
    n = GetChildNode(n, r)
end
`/* Store the leaf node to GPU global memory */`
StoreLeaf(n, $\boldsymbol{l}$, t)

`/* Predict a class for` $\boldsymbol{x}$`, and stores the results to GPU global memory */`
Predict(n, $\boldsymbol{x}$, $\boldsymbol{p}$)

`/* Update the statistics of the leaf node. */`
Let y to be a true class for $\boldsymbol{x}$.
UpdateStats(n $\boldsymbol{x}$, y);

end

3.3.3 Split candidate calculation

`CalculateSplitMeritKernel` in Algorithm 2 is a GPU kernel that calculates split merits for all the possible candidates. That is, it calculates for all possible attributes and all attribute values of the reached leaf nodes. Algorithm 4 and Figure 3.5 shows the details of `CalculateSplitMeritKernel`.

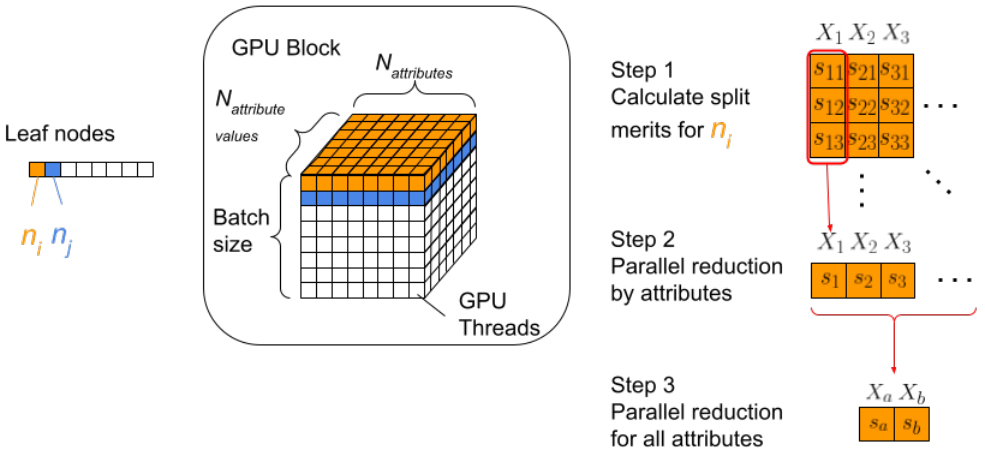


Figure 3.5: CalculateSplitMeritKernel

As same as `PredictAndTrainKernel`, each tree is assigned to one GPU block. On the other hand, `CalculateSplitMeritKernel` launches more GPU threads in a GPU block. As described in Figure 3.5, the number of threads per block is $B \times N_{\text{attributes}} \times N_{\text{attribute values}}$, where B is the number of data instances in a batch, $N_{\text{attributes}}$ is the number of attributes and $N_{\text{attribute values}}$ is the number of possible attribute values. In my experiments, $N_{\text{attributes}}$ is up to 44, and $N_{\text{attribute values}}$ is always 2.

After calculating all the candidates, it chooses the best split candidate and the second best candidate to make a decision for node split at the next stage. Choosing the best candidates is done by parallel reductions, which is a popular parallel technique to derive the max value. There are two phases of parallel max reduction. One is for deriving the best split for each attribute, and the other is for deriving the best splits among all the attributes.

Algorithm 4: CalculateSplitMeritKernel

Input:

l is a list of leaf nodes to store tree traversal result
s is a list of split candidates to store the calculated splits and merits

begin

Let b is the GPU block ID
Let t is the GPU thread ID

`/* Retrieve a leaf node derived by PredictAndTrainKernel */`
n = GetNode(***l***, b, t)
Let i to be an attribute index assigned to t
Let j to be an attribute value assigned to t

`/* Calculate a split merit for attribute` i `and value` j `*/`
s_{ij} = CalculateSplitMerit(n, i, j)

Let ***s*** be a global memory address to store calculated split candidates.
StoreSplit(***s***, s_{ij})

`/* Parallel reduction to choose the best split for the attribute` i `*/`
ParallelReductionForAttribute(***s***, i)

`/* Parallel reduction to choose the best split among all attributes */`
Let s_1 is the best split candidate
Let s_2 is the second best split candidate
s_1, s_2 = ParallelReduction(***s***)

end

3.3.4 Node splitting

`SplitNodeKernel` in Algorithm 2 is a GPU kernel that splits the leaf nodes when applicable. Algorithm 5 and Figure 3.6 describe the details of this GPU kernel.

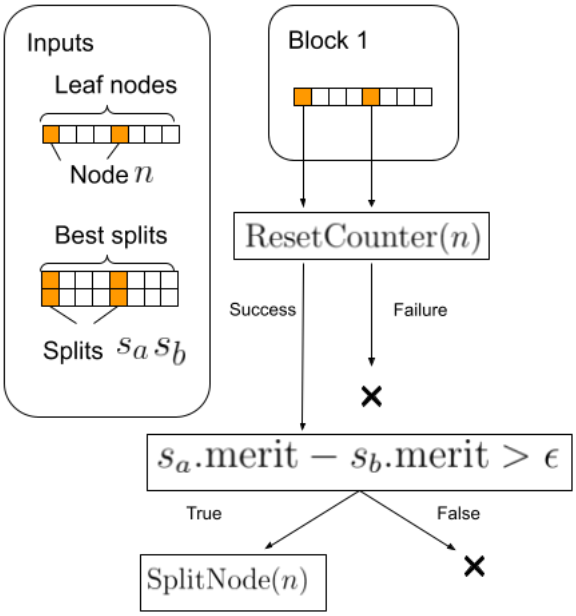


Figure 3.6: SplitNodeKernel

As same as other kernels, each tree is assigned to one GPU block. On a GPU block, each leaf node is assigned to one GPU thread, and executes node split.

Note that multiple data instances may arrive to the same data instance. In this case, only one split operation should happen to the leaf node. As the best split and the second split for each leaf node are derived in the previous kernel, it is possible to make a decision whether to split or not. Please refer to Algorithm 1 for the split decision logic in Hoeffding trees. In the `SplitNode` function, it obtains new memory spaces for child nodes from the memory pool explained earlier in this chapter.

Algorithm 5: SplitNodeKernel

Input:
l is a list of leaf nodes
s is a list of split candidates

begin
Let b is the GPU block ID
Let t is the GPU thread ID

```
/* Retrieve a leaf node derived by
   PredictAndTrainKernel                                    */
```
n = GetNode(***l***, b, t)

```
/* Atomic operation to reset counter.                       */
```
ok = ResetCounter(n)
if not ok **then**
```
   /* Only one thread per node executes split
      operations                                            */
```
   return
end

if MakeSplitDecision*(n, **s**)* **then**
   SplitNode(n)
end
end

3.4 Summary

In this chapter, I explained the approach to resolve the max tree depth limitation, and also explained the speed up of ARF obtained by GPU.

The memory pool strategy allows trees to grow incrementally without a wasting memory for unused nodes and attributes. Therefore, no tree depth limitation is necessary.

The routine of GPU ARF was explained to show how to speed up ARF algorithm. The data batching hides slow data transfer latency. The prediction and training is done independently to other trees with the data instance batch. Calculating split candidates is done further in parallel by launching more threads for all possible attributes and attribute values.

Chapter 4

Experiments

4.1 Introduction

The previous chapter explained how GPU can be utilized to implement ARF. In this chapter, I will explain the experiments to evaluate the GPU ARF performance in comparison to GPU ARF. The first section explains the experiment settings, and the second section explains the results of the experiments. The experiments also consist of two parts. The first part is for evaluating the individual techniques of GPU ARF, and the second part is for comparing the overall performance between GPU ARF and CPU ARF.

4.2 Experiment Settings

4.2.1 Classification problem for data stream

As mentioned in the chapter 1, I experimented GPU ARF in data stream classification settings.

In a typical classification problem, a set of data is given, and the dataset is split into training and validation sub datasets. A classification model is trained based on the training sub dataset including target classes until the model converges to the optimal state. And the trained model is evaluated against the validation sub dataset.

In data stream setting, it is assumed that each data instance is observed only once, and is not available afterwards. As described in Algorithm 6, a model needs to predict a target class for a data instance as it arrives. The true class is given in this scenario. The predicted class and the true class are stored for evaluation. After the prediction step, the model trains with the data instance.

Algorithm 6: Data stream classification

Input:

S is a sequence of data instances
m is a model

begin
  Let $\boldsymbol{x}$ as attributes of a data instance
  Let y as a target class of a data instance
  foreach $(\boldsymbol{x}, y) \in S$ **do**
    $\hat{y}$ = Predict(m, $\boldsymbol{x}$)
    Record(y, $\hat{y}$)
    Train(m, $\boldsymbol{x}$, y)
  end
end

4.2.2 Evaluation metrics

The evaluation metrics used in the experiments are accuracy and execution time.

Accuracy is simply calculated by Equation 4.1, where $N_{\text{correct predictions}}$ is the number of correct predictions, and $N_{\text{total predictions}}$ is the number of total predictions or the number of data instances. There are other evaluation metrics for classification problems such as precision, recall, f1 score. However, they are omitted since they don't provide different insights from accuracy in the experiments.

$$Accuracy = \frac{N_{\text{correct predictions}}}{N_{\text{total predictions}}} \tag{4.1}$$

Execution time is measured by wall-clock time to compute a given dataset. It includes initialization steps where memory pools are prepared.

4.2.3 Hardware

For the experiments, the host machine has AMD 3700X as the CPU. All procedures of CPU ARF (Java) and CPU ARF (C++) run on the CPU with 16 threads on 8 cores. Some parts of GPU ARF also run on the CPU, such as data loading, GPU kernel launches/memory syncing and evaluation. The machine has a NVIDIA A100, which has 108 SMs. The GPU kernels explained in the previous chapter run on this GPU.

4.2.4 Datasets

In this section, I explain 2 datasets used in the experiments. Those datasets are hosted in UCI Machine Learning Repository [12].

LED Display Domain Dataset

LED dataset is a dataset used by GVFDT [18] and ARF [14]. The data is generated synthetically by the generator. In the experiments, I generated 1,000,000 data instances. A data instance has 24 binary attributes and a target class. There are 10 classes with balanced distributions. 7 attributes are relevant to the target class and 10% of noise is added to them. 17 attributes are irrelevant. This dataset does not have concept drifts.

Covertype Dataset

Covertype [7] is a dataset used by GVFDT [18] and ARF [14] and other data stream leaning researches. It has 581,012 data instances. A data instance has 54 attributes and a target class. There are 7 classes with unbalanced distributions. This dataset has concept drifts.

4.2.5 CPU ARF implementations

MOA [6] has the original implementation of ARF written in Java. On the other hand, GPU ARF is written in CUDA C++. I assumed that there is a considerable overhead on Java in comparison to CUDA C++. Thus, I re-implemented ARF in C++. Those ARF implementations utilize multi-core CPUs to train trees in parallel.

Figure 4.1 shows the accuracy and execution time comparison between CPU ARF written in Java and CPU ARF written in C++. For LED dataset and Covertype dataset, the C++ implementation is 2.1 times faster and 5 times faster respectively, while keeping the accuracy at the same levels.

Dataset	Implementation	Accuracy (%)	Execution time (secs)
LED	CPU ARF (Java)	73.28	268.38
LED	CPU ARF (C++)	73.97	127.70
Covertype	CPU ARF (Java)	74.21	308.27
Covertype	CPU ARF (C++)	72.85	61.70

Figure 4.1: Performance comparison between Java CPU ARF and C++ CPU ARF

For the experiments later in this thesis, only the C++ implementation is used for the comparison with GPU ARF.

4.3 Experiment Results

The experiments consist of two parts. The first part is for evaluating the individual techniques of GPU ARF proposed in the previous chapter. The second part is for comparing the overall performance of GPU ARF with CPU ARF.

4.3.1 Part1: Evaluations for GPU ARF techniques

In this part, I will experiment with GPU ARF on the following factors to evaluate the proposed techniques individually.

- Max tree depth
- Scalability for the number of trees
- Batch size

Max tree depth

The limitation of max tree depth was a problem in GPU RF [18] and other studies that utilized GPU for Hoeffding Trees. GPU ARF avoids this limitation with the dynamic memory allocation explained in the previous chapter.

Figure 4.2 shows the accuracy of GPU ARF for different max tree depths. For the both of Covertype dataset and LED dataset, the accuracy increases as the max depth increases.

With the static tree node memory pre-allocation, the max depth was around 15. It would not be an issue for LED dataset as the accuracy already converges where max tree depth is under 10. However, this would cause a substantial regression for Covertype dataset.

With the proposed dynamic tree node memory allocation, it doesn't exceed the available GPU memory. This means this strategy doesn't add any penalty on the complexity of the mode and the accuracy.

This experiment shows that the dynamic tree node memory allocation strategy makes GPU ARF applicable for more complicated problems.

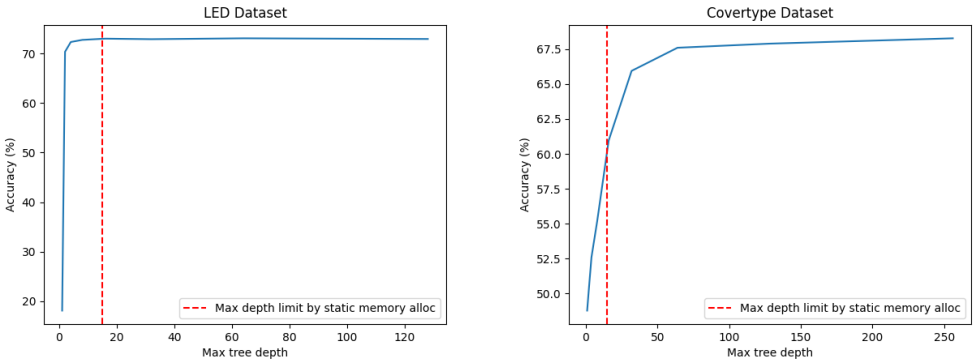


Figure 4.2: Accuracy of GPU ARF for different max tree depths

Scalability for the number of trees

As explained in the previous chapter, the trees of ARF can be trained independently to each other. Although the both CPU ARF and GPU ARF train trees in parallel, GPU has more cores than CPU. Therefore, GPU ARF should be capable of handling more trees than CPU ARF.

Figure 4.3 shows execution time for different number of trees. For the both of LED dataset and Covertype dataset, GPU ARF shows a capability to scale out for large number of trees. On the other hand, the execution time with CPU ARF increases more rapidly than GPU ARF.

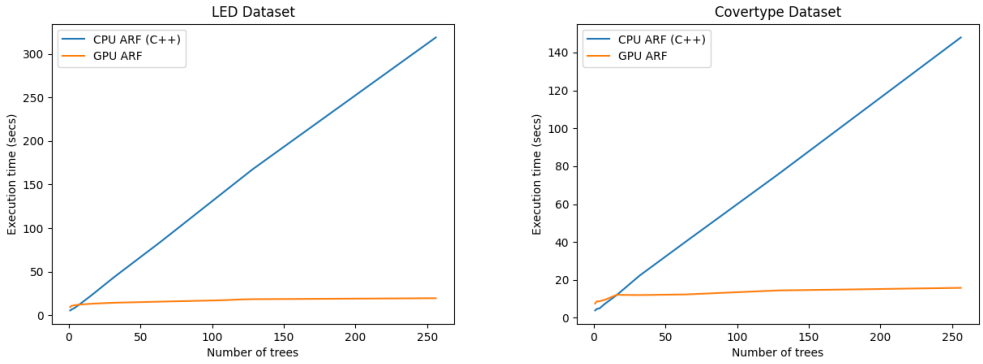


Figure 4.3: Execution time for different number of tree

CPU ARF is faster or parity with GPU ARF when the number of trees is small. The threshold seems to be 16, which is the number of available CPU threads. This indicates that GPU ARF doesn't have an advantage in terms of speed up when the CPU has more threads than the number of trees.

Batch size

As mentioned in the previous section, the batch size of GPU ARF brings a trade-off between execution time and accuracy.

Figure 4.4 shows the accuracy and execution time for different batch sizes. As expected, the execution speed shortens as the batch size increases. On the other hand, the accuracy decreases as the batch size increases.

However, some results for Covertype dataset don't align the expectation. For example, the accuracy at batch size 32 is better than the ones at batch size 8 and 16. It requires more experiments to understand why this configuration has better accuracy than the expectation.

LED Dataset

Batch size	Accuracy (%)	Execution time (secs)
1	72.12	51.52
2	72.14	30.54
4	72.14	18.06
8	71.89	11.00
16	71.97	7.62
32	71.60	5.73
64	71.80	4.66
128	70.65	4.14

Covertype Dataset

Batch size	Accuracy (%)	Execution time (secs)
1	73.30	300.10
2	73.31	163.47
4	73.19	91.55
8	65.65	107.03
16	66.76	48.05
32	72.84	16.36
64	68.43	13.36

Figure 4.4: Accuracy and Execution time of GPU ARF for different batch sizes

4.3.2 Part2: Performance comparison between GPU ARF and CPU ARF

Lastly, I compare the GPU ARF performance with the CPU ARF performance. The GPU ARF performance depends on the batch size as discussed earlier. I chose 64 for LED dataset and 32 for Covertype dataset as the batch size. The number of trees is 100 as it's the default value in MOA [6].

Figure 4.5 shows the accuracy and execution time comparison between GPU ARF and CPU ARF. For the both of LED dataset and Covertype dataset, GPU ARF is 7.3 times faster and 3.8 times faster respectively, while keeping the accuracy at the same levels with CPU ARF.

Dataset	Implementation	Accuracy (%)	Execution time (secs)
LED	CPU ARF (C++)	73.97	127.70
LED	GPU ARF	72.31	17.59
Covertype	CPU ARF (C++)	72.85	61.70
Covertype	GPU ARF	72.84	16.36

Figure 4.5: Performance comparison between GPU ARF and CPU ARF

4.4 Summary

This chapter explained the experiment settings and the experiment results.

For the evaluation metrics, execution speed and accuracy are used. As same as other data stream learning studies, LED dataset and Covertype dataset are used in the experiments.

I re-implemented the CPU ARF in C++ to avoid the overhead comes from Java, which is used for the original ARF implementation. The C++ CPU ARF shows 2 to 5 times faster execution speed with the same accuracy with the original implementation of ARF written in Java.

In the first part of the experiments, the proposed GPU ARF techniques were evaluated individually. The dynamic memory allocation allows GPU ARF to grow enough without being sacrificed by max depth limitation. GPU ARF shows the higher scalability for the number of trees than CPU ARF. The trade-off comes from batch size was discussed. And the best batch sizes were chosen for the datasets. However, in the batch size experiments, unexpected accuracy regressions were observed, and the root cause is still unclear.

In the second part of the experiments, the overall performance of GPU ARF was compared with CPU ARF. It shows that GPU ARF achieves the 3 to 7 times speed up without decreasing the accuracy.

Chapter 5

Discussions and Future works

5.1 Introduction

The previous chapter shows the experiments of GPU ARF to see how much speed up it has from the proposed approaches, and compare the performance with CPU ARF written in C++. Overall, GPU ARF achieved the considerable speed up while keeping the accuracy at the same level with CPU ARF. However, the experiments are done in limited scenarios.

There are some situations where GPU ARF can't take advantages from the parallelism of GPU. In this chapter, I discuss in which scenarios GPU ARF is suitable, or **not** suitable.

There are other type of problems that are not covered by this thesis, such as datasets with non-categorical attributes and regression problems. The last section describes these problems.

5.2 Hardwares

In the previous chapter, I showed that GPU ARF can scale out for the large number of trees in comparison to the 8 cores/16 threads CPU. However, the scalability of CPU ARF is limited by the number of CPU cores/threads. When the CPU has more cores and threads, GPU ARF would not have as large advantages as in the experiments.

5.2.1 Cloud and High performance computing

As of writing, high-end CPUs for cloud and high performance computing are capable of serving more threads than the CPU used in the experiments. For example, Intel© Xeon© Platinum 8380 [2] has 40 cores that can serve 80

threads. AMD EPYC™ 9654P [1] has 96 cores that can serve 192 threads. If the situation is in high performance computing or cloud computation, these high-end CPUs would be sufficient to handle 100 trees.

5.2.2 Embedded systems and mobile devices

On the other hand, a GPU usually has much more cores/threads in an embedded system or a mobile device, where small numbers of CPU cores/threads are available. It's also important to utilize all existing computation resources on the hardware. If the application can utilize a SIMD GPU, the host CPU can work on other tasks such as receiving data from sensors, networking, file I/O and so on.

5.3 Complexity of problem

Whether GPU ARF is suitable or not depends on the complexity of the problem. GPU ARF is suitable more for complex problems that requires many computations than simple problems for which a simple solution is sufficient. This section lists up the points related to GPU ARF performance.

5.3.1 Number of trees

For the experiment, I chose 100 as the number of trees. However, the appropriate number of trees depends on the problem: the dataset, the target accuracy and the required throughput and so on. If the number of trees can be less than the number of available CPU threads, CPU ARF should be sufficient.

5.3.2 Number of attributes and classes

LED dataset and Covertype dataset have a large number of attributes and target classes. These datasets are chosen for the experiments because the parallelism of GPU can work better for them. When the datasets don't have many attributes and target classes, The speedup of GPU ARF would not be as large as the experiments.

5.3.3 Batch size and concept drift

The speed up of GPU ARF gains from the large batch. If concept drifts happen quickly, GPU ARF would not be updated as quickly as CPU ARF.

In this situation, the accuracy would decrease more. GPU ARF is more suitable for handling large number of data instances at a time with slow concept drift.

5.4 Robustness for the order of data instances

Since Hoeffding Tree grows as incrementally, the trained model depends on in which order the model observes the data stream. However, this thesis experimented with the data streams only in the fixed orders. It would be important to evaluate how robust the training algorithm is against the order of the data instances, and identify in which scenarios the training doesn't work well.

5.5 Other type of problems

In this thesis, it experiments GPU ARF only with classification problems with binary attributes. It needs more experiments to apply GPU ARF to other type of problems.

5.5.1 Non-categorical attributes

As of writing, GPU ARF supports only categorical attributes, and only binary attributes are used in the experiments. It's also important to support numerical attributes to test different type of datasets.

Hoeffding tree for numerical attributes is different from categorical attributes. For categorical attributes, a Hoeffding tree stores the number of occurrences by attributes and target classes. For numerical attributes, it needs to store other statistics such as max, min, sum, mean, variance. It also requires more floating point operations since the observed values themselves are floating point, while categorical attributes only have integers.

Although it's feasible to support numerical attributes, mixing categorical attributes and numerical attributes may cause speed down due to branch divergence. In that case, more advanced GPU block/thread assignment strategies would need to be considered. For example, assigning the same attributes to the same GPU block would reduce the branch divergence. This is a future work of this study.

5.5.2 Regression problems

This thesis discusses only with classification problems. However, there are many regression problems, where the prediction result is numerical instead of categorical. The computation required for regression is different from classification. It needs to study with regression problems to understand how much speed up GPU ARF can achieve, and what type of regression problems GPU ARF is suitable.

Chapter 6

Conclusion

In the chapter 2, I reviewed the background of this thesis. The importance of data stream learning is explained in the context of the evolution of IoT devices and mobile devices. It also reviewed the related studies. Although there are many studies on data stream learning, there are not many studies with GPU. ARF [14] already showed that multiple CPU cores can parallelize the computation without sacrificing the accuracy. GPU RF [18] implemented a stream learning version of random forest using GPU. However, it had a limitation of the number of tree depth due to memory allocation mechanism.

In the chapter 3, I explained GPU ARF, a GPU version of ARF. Using memory pool allows GPU ARF to allocate memory for tree nodes at node splits. In the related works, there is the max depth limitation because the memory for nodes is allocated statically even for unused nodes. With static memory allocation, the amount of memory increases exponentially to the tree depth. GPU ARF removes this limitation. Training trees in parallel is a natural idea that comes from the multi-core CPU ARF [14]. Batching data is a common technique for GPU applications to hide the latency of data transfer between the host CPU and the GPU. However, in the stream learning scenario, the regression of the accuracy is expected. Calculating all possible split suggestions is parallelized further by GPU efficiently.

In the chapter 4, I explained the experiment settings, and the results of the experiments. LED Dataset and Covertype dataset are used in the experiments as other stream learning studies. I re-implemented a C++ version of ARF because the original implementation in MOA [6] is written in Java, and the overhead comes from Java was anticipated. The C++ ARF is 2.1 times and 5 times faster speed than the Java ARF for LED dataset and Covertype dataset respectively. The relationship between the accuracy and the max tree depth was measured. This showed that the dynamic tree node memory allocation allows GPU ARF to be applied to more complicated problem such

Covertype dataset in this thesis. GPU ARF shows better scalability for the number of trees than CPU ARF with multi cores. The performance with different batch sizes is also experimented to find the suite configurations for LED dataset and Covertype dataset. The experiments with some batch sizes showed unexpected accuracy regressions. It requires further investigation for this issues. However, overall GPU ARF achieved the considerable speed up against CPU ARF written in C++, while keeping the accuracy at the same level. GPU ARF is 3 to 7 times faster than the CPU ARF written in C++, and is 15 to 18 times faster than the original CPU ARF written in Java.

In the chapter 5, I discussed with the experiment results, and future works. With a high-end CPU that has many cores/threads, there is not large advantage of using GPU for ARF. On the other hand, GPU ARF is suitable for machines with a CPU that doesn't have many cores/threads. In an embedding system, it is also important to utilize all available resources on the hardware. Therefore, GPU ARF is suitable. There are problems that are not covered by this thesis. The performance of GPU ARF probably have different characteristics with datasets with numerical attributes or regression problems.

Overall, this thesis achieved the objectives set in the chapter 1.

1. Speed up ARF algorithm with GPU

   GPU ARF is 3 to 7 times faster than ARF with multiple CPU cores optimized in C++.

2. Resolve the max tree depth limitation that GPU RF has.

   The limitation was resolved by the dynamic memory allocation strategy with memory pools. While the max tree depth limit was around 15 with the static pre-memory allocation, GPU ARF didn't need to set any limitation on tree depth.

3. Keep the accuracy

   The speed up was achieved without decreasing accuracy dramatically.

Bibliography

[1] AMD EPYC™ 9654P, https://www.amd.com/en/products/cpu/amd-epyc-9654p.

[2] Intel® Xeon® Platinum 8380 Processor (60M Cache, 2.30 GHz), https://www.intel.com/content/www/us/en/products/sku/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz/specifications.html.

[3] Kaggle, https://www.kaggle.com.

[4] Nvidia a100 tensor core gpu architecture, https://images.nvidia.com/aem-dam/en-zz/solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[5] A. Bifet and R. Gavaldà. Adaptive learning from evolving data streams. In N. M. Adams, C. Robardet, A. Siebes, and J.-F. Boulicaut, editors, *Advances in Intelligent Data Analysis VIII*, pages 249–260, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[6] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. Moa: Massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, aug 2010.

[7] J. A. Blackard and D. J. Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, 24(3):131–151, Dec. 1999.

[8] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[9] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[10] G. De Francisci Morales and A. Bifet. Samoa: Scalable advanced massive online analysis. *J. Mach. Learn. Res.*, 16(1):149–153, jan 2015.

[11] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, page 71–80, New York, NY, USA, 2000. Association for Computing Machinery.

[12] D. Dua and C. Graff. UCI machine learning repository, 2017.

[13] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001.

[14] H. M. Gomes, A. Bifet, J. Read, J. P. Barddal, F. Enembreck, B. Pfharinger, G. Holmes, and T. Abdessalem. Adaptive random forests for evolving data stream classification. *Machine Learning*, 106(9-10):1469–1495, jun 2017.

[15] V. Jagannath. Random forest diagram, under CC BY-SA 4.0, via Wikimedia Commons.

[16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3149–3157, Red Hook, NY, USA, 2017. Curran Associates Inc.

[17] C. Manapragada, G. I. Webb, and M. Salehi. Extremely fast decision tree. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 1953–1962, New York, NY, USA, 2018. Association for Computing Machinery.

[18] D. Marron, A. Bifet, and G. D. F. Morales. Random forests of very fast decision trees on gpu for mining evolving big data streams. In *Proceedings of the Twenty-First European Conference on Artificial Intelligence*, ECAI'14, page 615–620, NLD, 2014. IOS Press.

[19] V. Mehta. Accelerating random forests up to 45x using cuml, Aug 2022.

[20] T. Sharp. Implementing decision trees and forests on a gpu. In D. Forsyth, P. Torr, and A. Zisserman, editors, *Computer Vision – ECCV 2008*, pages 595–608, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[21] O. Wu, Y. S. Koh, and G. Russello. GPU-based state adaptive random forest for evolving data streams. In *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, jul 2020.

Acknowledgements

I would like to express my deepest appreciation to my professor for his patientful and crucial feedback. I am also indebted to my defense committees, who provided candid advice at mid-term defense and final defense. Additionally, this endeavor would not have been possible without our company and my managers, who provided expense assistance.

I am also graceful to the lab members, who studied together from Tokyo. Having mates who have similar motivations in similar situations helped me continue this course. Thanks should also go to the lab members in Ishikawa. They supported server setups for me to access the machines for this study.

Last but not least, I would like to thank my family especially my spouse and children. I could not keep this process without her support , encouraging me to continue this course and taking care of our children while I was studying. I believe that keep studying myself is the best way to tell the importance of studying to my children.