GPU-based State Adaptive Random Forest for Evolving Data Streams

Ocean Wu
School of Computer Science
The University of Auckland
Auckland, New Zealand
hwu344@aucklanduni.ac.nz

Yun Sing Koh
School of Computer Science
The University of Auckland
Auckland, New Zealand
ykoh@cs.auckland.ac.nz

Giovanni Russello
School of Computer Science
The University of Auckland
Auckland, New Zealand
g.russello@auckland.ac.nz

***Abstract*—Random forest is an ensemble method used to improve the performance of single tree classifiers. In evolving data streams, the classifier needs to be adaptive and work under constraints of space and time. One benefit of random forest is its ability to be executed in parallel. In our research we introduce a random forest model utilizing a hybrid of both GPU and CPU, called GPU-based State-Adaptive Random Forest (GSARF). We address the pre-existing challenges of adapting random forest for data streams, specifically in the area of continual learning. Our novel approach reuses previously seen trees in the random forest when previous concepts reappear. This allows us to retain prior knowledge and provide a more stable predictive accuracy when changes occur in the data stream. Our random forest for data streams stores three types of trees, foreground trees which are trees that are currently used in prediction, background trees which are trees that are built when we are aware of possible changes in the data streams, and candidate trees which are trees that had been highly used in the previous concepts, but are now discarded due to changes in the data stream. We store candidate trees as they may be potentially useful at a later period in a repository and can be accessed when needed. We empirically show our technique performs up to 138 times the speed compared to current CPU-based random forest benchmarks. Our approach has shown to outperform a baseline GPU-based approach in terms of cumulative accuracy performance.**

***Index Terms*—Adaptive Random Forest, GPU, Data Streams**

I. Introduction

Adaptive real-time mining of continuous data streams is necessary with the increasing volume, velocity and volatility of data collected. The need for the data stream to be processed on-the-fly with fast reaction and adaptation to changes is ever increasing. In a streaming environment, decision based on data require real-time analysis which has to be processed in parallel with low-latency. Most research focus on designing an algorithm to more efficiently adapt to changes without fully utilizing the underlying computational architecture. The focus of our research concentrates on utilizing the available GPU processors. We explore the usage of both GPU and CPU for applications of data stream mining. In terms of dealing with evolving data we explore the random forest technique for data streams. The benefit of random forest is its high accuracy and ability to adapt to new concepts within the stream. Random forest is parallelizable, thus, little effort is needed to separate the random forest algorithm into a number of parallel tasks [1].

One of our focus is on being able to store and utilize knowledge built from the past, to improve for the future. For example in the human activity recognition task such as remote patient monitoring, rehabilitation and assisting disabilities. Humans behaviours are repetitive, thus having no long term memory of our model is wasteful. For data with temporal dependence, there is the focus on using deep learning. However in this particular research we are focused on being able to explicitly detect changes within the model as well as store some pre-exiting knowledge. The general intuition of approaches of handling recurring concepts is to not lose knowledge gathered over time [2], [3]. There is a need to maintain a pool of trees and use single or ensemble of them when we detect a change in the data stream. This is known as concept drift adaptation [4], whereby the technique tries to adapt to the new concept in the data stream as soon as possible. Most recurring concept detection techniques [5], [6] are explicit whereby there is a specific drift detector used alongside a recurring model matching phases from a repository. Instead our approach adapts to implicit drifts where we capture smaller shifts that will eventually lead to a real concept drift.

We proposed a variation of the random forest approach, called GPU-based State Adaptive Random Forest (GSARF). Current random forest techniques for data streams concentrates on adapting to concept drift by preemptively building additional trees when there are early warning of when the trees are becoming less accurate. These additional trees are called background trees. A background tree is swapped into the random forest replacing a foreground tree when a drift is signaled. This indicates that the foreground tree it is replacing is no longer accurate. Each foreground tree is used to represents a current state of the stream. From a continual learning process perspective, the foreground trees that we are replacing may be valuable in the future. Thus, a simple discard and replace approach can be wasteful in terms of computational processing. Instead we store the foreground trees in a candidate tree set. This can be seen as a buffer to store trees that may still be useful in the future. By using pre-existing candidate trees we now can reduce the training effort in the future.

Random forest processing on GPU provides a massively parallel architecture with many cores and is throughput oriented, GPUs have better performance-per-watt than CPUs [1].

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

Unlike the current GPU machine learning algorithm [7], [8], our implementation also exploits the CPU memory to improve the GPU-based random forest implementation. We stores trees that may be reused in the future, which we call the CPU tree pool in the CPU memory. Since GPUs are memory bound, by utilizing the available CPUs we can manage our storage load balancing of our trees efficiently.

The contributions of our work are two-fold. Firstly, we propose an adaptation strategy that stores foreground trees that are no longer accurate whenever a concept drift is detected in a repository called CPU tree pool. When a drift warning is detected, we choose candidate trees from the pool to best match the new data stream and also start training a new background tree. When a drift is signalled, we replace the tree in our random forest with either a candidate tree or its background tree, whichever is more accurate. Candidate trees allow us to retain previous knowledge and capture concept recurrence within the data stream. Secondly, we present an implementation which uses both GPU and CPU efficiently.

The remainder of this work is organized as follows. In Section II we briefly discuss related works in the area. Section III describes our architecture for our GPU based random forest algorithm. In Section IV we present our State Adaptive Random Forest technique. In Section V the experimental results of the experiments are presented. Finally, Section VI concludes this work and poses directions for future work.

II. Related Work

Random Forest on CPU. Abdulsalam et al. [9] designed streaming random forests to tackle concept drifts, an algorithm using an entropy-based drift-detection technique was developed for evolving data streams. Gomes et al. [10] proposed a parallel implementation of Adaptive Random Forest (ARF) which has no degradation in terms of classification performance in comparison to a serial implementation, since trees and adaptive operators are independent from one another.

Random Forest on GPU. Grahn et al. [8] proposed to use a GPU thread to train one decision tree for the random forest. Thus, many decision trees can be trained in parallel. Marron et al. [1] used classification algorithms such as random forest for mining large amount of stream data using GPU.

Recurrent drift detection. Some research [5] concentrates on the re-use of previously learned classifiers for better accuracy when concept recur.

Current random forest are designed to adapt to concept drift without retaining historical knowledge in the data streams. Crucially we may be discarding useful knowledge. In our approach, we store a pool of candidate trees that retain partial historical knowledge that may be useful in the future.

III. Random Forest on GPUs

We present the random forest model designed for GPU, which is our base model. Fig. 1 depicts how trees are encoded to single arrays using breadth-first traversal. Each node in the tree array is encoded as a 32-bit unsigned integer, with its most significant bit indicating if the node is a leaf. It represents an

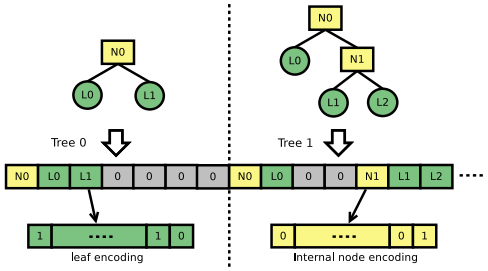


Fig. 1: Forest layout on GPU.

attribute ID for an internal node when the bit is turned off, otherwise it is an offset to other related data structures that store extra information about the leaf, such as the class and number of instance seen at the leaf, for computing information gain. The rest of the bits have different meaning for its leaf and internal nodes. In comparison to previous GPU based random forest techniques, such as GVFDT [1], which allocate memory for the full tree, our memory requirement grows exponentially to the number of the attributes.

Our GPU based random forest implementation achieves massive parallelism by launching multiple threads per tree, each traversing down the tree with a different data instance. Computation for information gain and leaf splitting is performed on all the reached leaves from the previous tree traversal, across all trees at the same time. As a result, the number of instances used for each tree traversal has a key impact on the speed. On the other hand, the more instances that are training the model in parallel, the slower the GPU based random forest is able to detect drifts.

IV. GPU-based State Adaptive Random Forest

The main novelty of our GPU-based State Adaptive Random Forest (GSARF) is the additional pool trees to allow for faster updating of the random forest when concept drift occurs. Previous research randomly train new trees in the background when a drift warning is detected and replacing the drifted trees in the foreground when a drift is detected in the forest [10]. GSARF has additional candidate trees to cope with evolving data streams. A candidate tree pool stores the trees that were previously used in an old concept that has expired but may recur in the future. This candidate tree pool is stored on the CPU instead of the GPU memory. We will explain the decision to store the trees on the CPU in the following subsections. Our state adaptive strategy is based on using a drift monitor per tree to track warnings and drifts, and match candidate trees to the state of the stream when a warning is detected.

An overview of our GSARF technique is depicted in Fig. 2. We have two main components: the first is the state matching component (Box 1), and the second is the candidate tree replacement component (Box 2). **State matching process**: when drift warnings are detected, the algorithm tries to find the closest match from candidate or background trees to the current stream. **Tree replacement process**: when actual drifts are detected, the drifted trees are either replaced by their

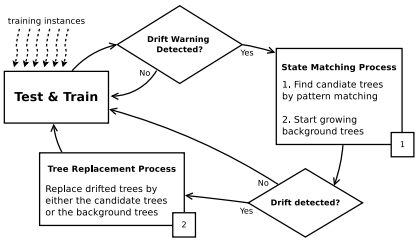


Fig. 2: Flowchart for GSARF.

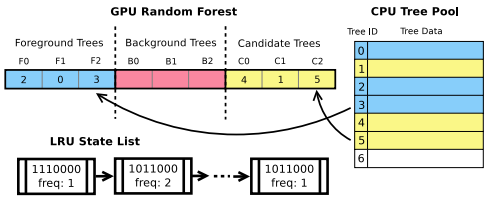


Fig. 3: Data structures

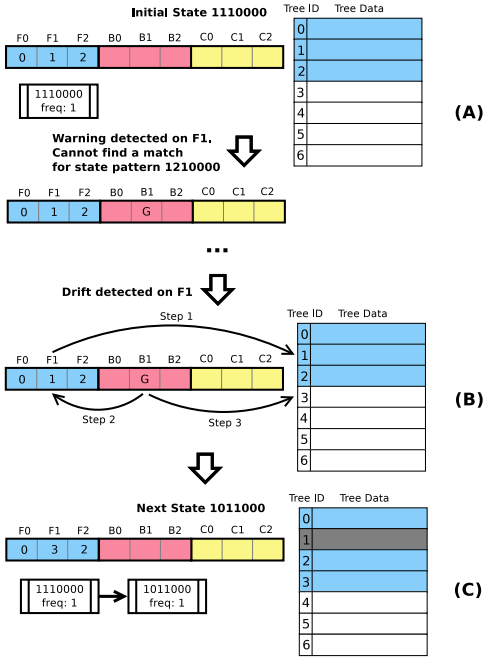


Fig. 4: An example of initial state transition.

corresponding background trees, or the best candidate trees in the forest depending on their accuracy.

Data Structures. We maintain a pool of trees that have been initially built on the GPU in the CPU, shown by the CPU tree pool in Fig.3. Each of the trees is associated with a tree ID, which is used for referencing trees in the forest to trees in the CPU tree pool. The tree IDs are also used to build states for state matching. A subset of the expired trees that were previously used is stored.

On the GPU side, we have a forest which consists of the following three groups of trees. (1) **Foreground trees** are trees that are trained and participate in voting. (2) **Background trees** are trees that are trained only when their corresponding foreground trees detect warnings. (3) **Candidate trees** are foreground trees that were used frequently but have expired. These trees may potentially be useful in the future.

We keep the state patterns in an linked list, called the Least Recently Used (LRU) State List, as shown in Fig. 3. Each list node contains the state pattern represented as a bitset, as well as the number of times this pattern has been matched. The state list has a fixed capacity. Once the number of states exceeds the list capacity, the states are evicted by the least recently used policy. The linked list ensures that the eviction time complexity is constant.

Initialization. In the initialization phase, the state adaptive random forest is initialized with three types of trees. The top of Fig. 4 represents the initial state of the trees. In the first step (Fig. 4 (A)), we built our random forest with n foreground trees, and each foreground tree has a tree ID, as well as an allocation in the CPU tree pool identified by the tree ID. Each of the foreground trees, will be assigned a drift detector, with a drift warning threshold. When a warning is detected on $F1$ and we did not find a match the second bit representing the $F1$'s tree ID is turned off. $F1$'s background tree $B1$ will start growing (Fig. 4 (B)). When a drift is detected on $F1$ and there is no candidate tree, $F1$ is firstly copied to CPU tree pool (Step 1), $B1$ gets a new ID and $B1$ replaces F1 (Step 2) and an allocation in CPU tree pool (Step 3). Finally $F1$ consist of the new background tree $B1$ which is also stored in as Tree ID 3 in the CPU Tree pool (Fig. 4 (C)).

State Matching Process. When a warning is detected, state matching finds candidate trees to add to the candidate group of the forest. This process iterates the LRU state list, and finds the *closest* state pattern that satisfies the following criteria: (1) Minimum edit distance to the current state, (2) Highest frequency when multiple states have the same minimum edit distance, and (3) Positions that matches the ID of the warning trees must be unset. Fig. 5 illustrates how a state is constructed and matched. Consider a fixed CPU tree pool size of 7, and there are 3 trees in each group of the trees in the forest, as shown in Fig. 5. The current state pattern is built according to the presence of trees in the forest's foreground group. Since the trees with IDs 0, 4 and 2 are in the foreground group of the forest, we set the 0th, 4th and 2nd positions (zero-based indexing) of the state pattern bitset, and leave the rest unset, *i.e.*, 1010100. Suppose we have detected a warning on the tree labeled $F1$ in the forest. We firstly set the 4th position of the current pattern to 2 *i.e.,* 1010200. This means any pattern with its 4th position with value of 1 should not be considered. We specifically set the value of the position as 2 instead of 0 or 1. The value 2 denotes that that specific tree is changing, and should not be considered in the next round, whereas 1 indicates that it is still useful and should be considered. We then try to find the best match for the *target* pattern 1010200 across all the states in the LRU pattern list, for example, the 3 patterns shown in Fig. 5. State 2 is not a match since it has the 4th bit set to 1. State 0 is also not a match as it had lower frequency than State 1, despite them having the same minimum edit

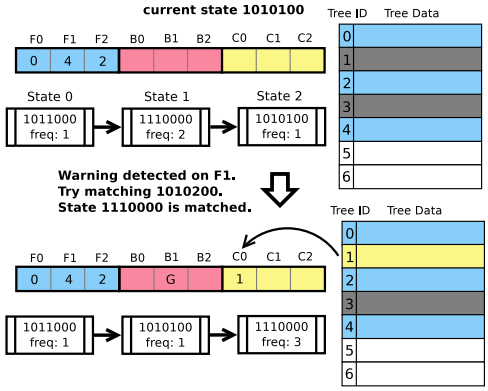


Fig. 5: Pattern matching for CPU tree in candidate group

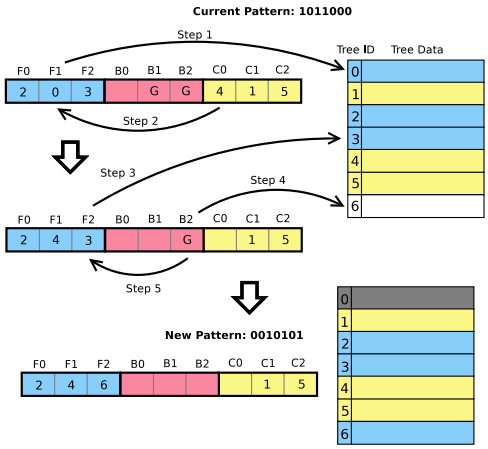


Fig. 6: Tree replacement upon drift detection on both foreground trees $F1$ and $F2$, and only candidate tree $C0$ has better performance than both of the drifted trees.

distance to the target pattern. State 1 with pattern 1110000 is selected in this case. The State 1's frequency counter is increment and the State object is removed from its position and append it to the end of the pattern list. Note that $F1$'s background tree $B1$ is also set to start growing. Next, we select candidate trees with positions of bits that are set but are unset in the current pattern. In our example, the tree with ID 1 from the CPU tree pool is copied to the next available position in the candidate group in the GPU forest. In the event the candidate group of the forest is full, we remove the LRU candidate trees in the forest. The warning detection shown in Fig. 5 gives an example for such case. In this case, only the drifted tree's corresponding background tree starts growing.

Tree Replacement Process. When a drift is detected, we first try to replace the drifted trees with candidate trees based on performance. We then try to replace the drifted trees with their background trees, if none of the candidate trees is qualified to replace the drifted trees. The individual candidate tree's confusion matrix is used to calculate Cohen's κ statistics [11] for assessing the performance of each candidate tree. It measures inter-rater agreement for the prediction of a tree against the true prediction. Candidate trees with the best κ value is compared against the drift foreground tree's κ value. The drifted trees will be replaced by the best candidate trees only if the candidate tree's κ value is greater than the drifted tree and above an ϵ threshold. The ϵ threshold is used to account for variation within the data stream. If the difference in the agreement is above the ϵ threshold, the candidate tree is assumed to be legitimate. If the candidate tree is not chosen, the drifted tree is replaced with its background tree.

If a background tree is chosen to replace the drifted tree, we compare its κ value to the κ value for the drifted tree. If their difference in κ value is below the δ threshold, we add the background tree to the next available slot in the CPU tree pool and assign it with a new tree ID. This constraint avoids trees with similar performance to existing tree in the CPU tree pool from being added to the pool. The background tree is added to the CPU tree pool at this stage, in preparation for it potentially becoming a candidate tree in the future. The current state pattern is also updated, since it represents the presence of trees in the foreground group of the forest. The updated current state pattern will then be used to update the LRU state pattern list. As an example, we use the current state of the trees (*i.e.* 1011000) shown in Fig. 6. Suppose we detect a drift on both tree $F1$ and $F2$. The candidate trees is first sorted by their kappa measurement κ to find the best candidate trees. Suppose tree $C0$ has the best κ value and is greater than that of the $F1$'s. $F1$ first copies itself back to the CPU tree pool (Step 1), since it may have grown from when it was last pulled from the CPU tree pool. $F1$ then gets replaced by $C0$ (Step 2), and the current state pattern becomes 0011100. If the drifted tree $F2$ had no candidate tree that has better performance than itself, and that its background tree $B2$ has a better κ value than it over δ. $F2$ first copies itself back to the CPU tree pool at position 3 (Step 3). $B2$ gets the next CPU tree ID 6 (Step 4), replaces $F2$ (Step 5). The current state pattern now becomes 0010101. Lastly, the background trees of the drifted trees get disabled after the tree replacement process.

Pseudocode Discussion. Our random forest tree training uses the Hoeffding tree [12] as base classifer. The drift detector used is ADWIN [4]. The function responsible for state transition is shown in Algorithm 1. The state adaptive algorithm has two parameters ϵ and δ. The ϵ is used to reduce false positive tree matching and the δ threshold is used to avoid adding trees with similar background trees that already exist the CPU tree pool. Both these parameters need to be tuned depending on the memory and runtime requirements. Setting too low of an ϵ threshold, may results in no candidate tree being reused, thus rendering the candidate group useless. On the other hand, setting ϵ threshold too high would often result in finding a matching candidate tree. This would render the entire random forest less accurate. One would have to set the ϵ threshold proportional to the variability of the fluctuation of the data stream. If the δ threshold is set too low, we would be storing additional trees in our CPU pool.

Theoretical Analysis. Similar to a single Hoeffding tree,

Algorithm 1: State Transition

Data: Forest, CPU Tree Pool, CurrentState

TargetState ← CurrentState;
foreach T_x *in Forest* **do**
  if *warning detected* **then**
    reset and start growing background tree B_x;
    TargetState[$tree\_id(T_x)$] ← 2;
  else if *drift detected* **then**
    DriftedTreeList.append(T_x);
  end
end
ClosestState ← StatePatternMatch($TargetState$);
if $ClosestState \neq \emptyset$ **then**
  for $i \in [0,$ *StateLength*$)$ **do**
    if *CurrentState*$[i] = 0$ *and ClosestState*$[i] = 1$ *and* T_i *is not in the Forest* **then**
      pull T_i from CPU Tree Pool and add it to the candidate tree section in the forest;
    end
  end
end
Sort candidate trees by kappa;
NextState ← CurrentState ;
foreach D_i *in DriftTreeList* **do**
  Copy D_i back to the CPU Tree Pool;
  SwapTree ← $null$;
  C ← candidate tree with the highest kappa;
  F_i ← foreground tree at position i;
  if *Kappa(C) - Kappa*$(F_i) \geq \epsilon$ **then**
    SwapTree ← C
  else
    SwapTree ← B_i;
    if *Kappa*(B_i) − *Kappa*$(D_i) \geq \delta$ **then**
      assign a new tree_id for B_i;
      add B_i to CPU tree pool;
    end
  end
  Replace D_i with SwapTree in the forest ;
  Disable B_i in the forest ;
  NextState[$tree\_id(SwapTree)$] ← 1;
  NextState[$tree\_id(D_i)$] ← 0;
end
LRUStateList.Add(NextState);
CurrentState ← NextState;

the complexity of memory required, given the maximum features per split m, the number of classes c, the number of leaves l, and the maximum number of possible values per feature v, is $\mathcal{O}(lmcv)$ [12]. Given T as the total number of trees and l_{max} as the maximum number of leaves for all trees, a plain random forest algorithm, without warning/drift detection, requires $\mathcal{O}(Tl_{max}mcv)$.

When we use background trees with drift detection, the space allocated for each tree is $\mathcal{O}(T((Mlog(W/M) + l_{max}mcv))$. This is similar to ARF [10], whereby ADWIN requires $\mathcal{O}(Mlog(W/M))$ [11], such that M is the number of buckets, while W is maximum error rate window size. In the worst case, like ARF, we allocate $2n$ trees for both foreground and background trees concurrently. Similarly, the number of candidate trees stored on the GPU is normally greater than n. However, this number is normally dependent on allowable CPU pool storage τ and is inversely proportional to the concept recurrence rate in the data stream, r, *i.e.* $\tau \cdot n \cdot \frac{1}{r}$. In terms of execution time, if it costs t_T seconds to build a tree, and t_A to access and match a recurrent tree, whereby $t_A < t_T$, then the gain in time is $t_T - t_A = t_G$. Given the recurrence rate τ, the execution time gain is $t_G \cdot \tau$.

V. Experimental Evaluation

We evaluate the performance of our GSARF algorithm by classification performance, speed, and memory usage. The classification performance is based on prequential evaluation. In our implementation, the state transition algorithm can be turned off such that foreground trees in the forest can only be replaced by its background tree upon drift detection, which is the GPU Adaptive Random Forest (GARF) algorithm. We use this as a baseline algorithm.

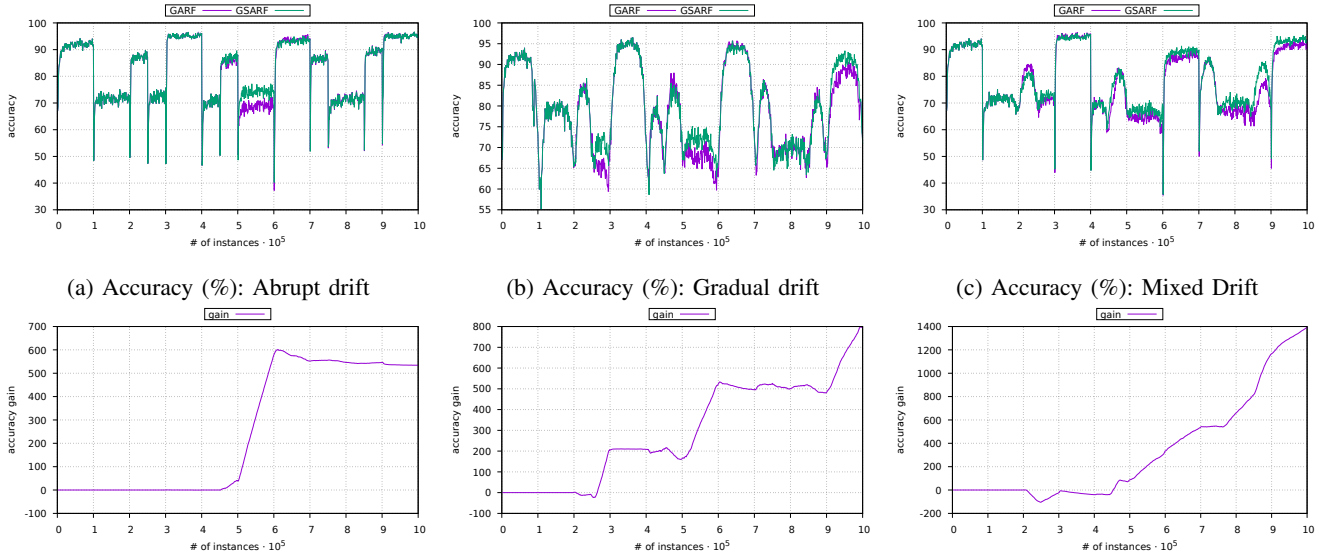
The experimentation is performed on the following machine and architecture: NVIDIA Tesla V100 16G Passive GPU, Dell PowerEdge R740 with 40 CPUs, 125.42 GiB (Swap 976.00 MiB) and Ubuntu 18.04 with AMD64 4.15.0-46-generic. Our code, synthetic dataset generators and test scripts are available here[1] for reproducible results. In our experiments, the set of foreground and candidate trees are set to 100 unless otherwise specified. We use two synthetic and two real datasets (KDD99 Cup [13] and Covertype datasets [14]) on our experiments. The synthetic data sets include recurrent abrupt, gradual, and mixed drifts data stream, while the real data sets have been thoroughly used in the literature to assess the classification performance of data stream classifiers. We use common dataset generators including Agrawal and LED.

Accuracy Evaluation. Data streams may contain new concepts or old concepts may reoccur, indicated by changes in feature space and decision rules losing relevance over time. Under such circumstance one should take into account computational aspects such as processing time, recovery of the model after the concept change, and memory usage. Fast updating of a learning model and recovery of old models may often be more reasonable comparing to rebuilding a completely new model. We are interested in tracking its characteristics over the course of stream progression [15].

In these sets of experiments, we generate four types of data streams: recurrent abrupt drift streams (drift interval of 1 instance), recurrent gradual drift stream (drift interval of 25K), mixed drift streams consisting of both abrupt and gradual concept drifts, with each containing 2 or 3 different concepts. Both Figs. 7 (a-c) and 8 (a-c) are examples of results from Agrawal and LED generator. In the figures the purple line represents accuracy for GARF, and the green line represents the accuracy for GSARF. We noticed that with our technique, the accuracy recovered faster when there was a drift as opposed to GARF. This is to demonstrate the results of our techniques. The cumulative accuracy gain is shown in Figs. 7 (d-f) and 8 (d-f). The cumulative accuracy gain is:

$$\text{accuracy gain} = \sum((\text{accuracy(GSARF)} - \text{accuracy(GARF)})$$

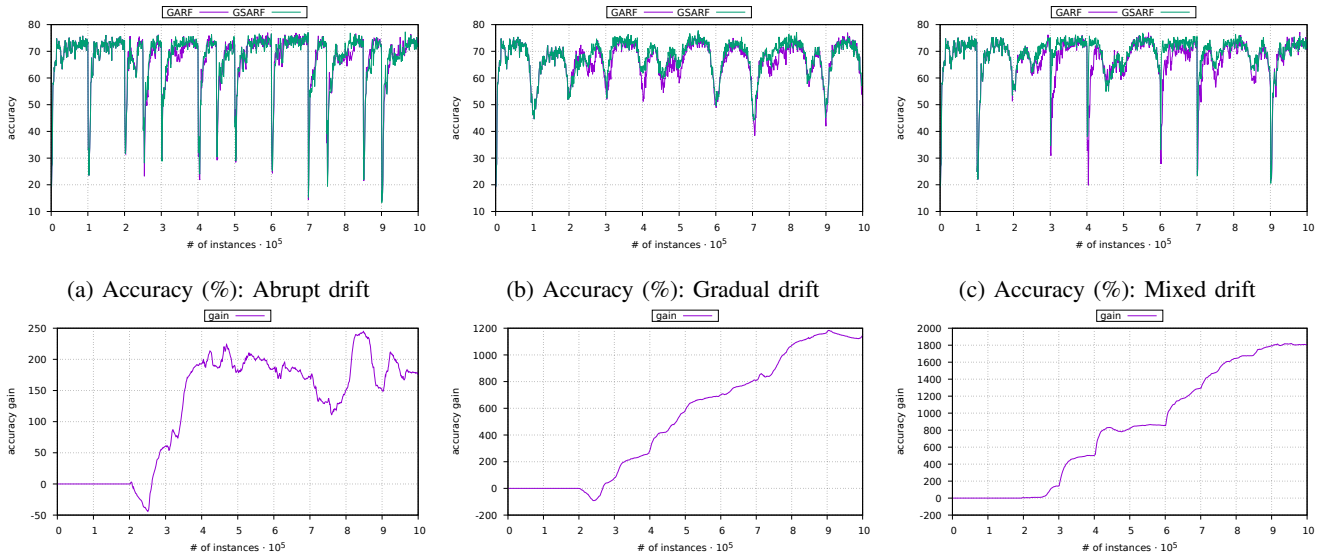
[1]https://github.com/ingako/gsarf



(a) Accuracy (%): Abrupt drift (b) Accuracy (%): Gradual drift (c) Accuracy (%): Mixed Drift

(d) Cumulative Accuracy Gain: Abrupt drift (e) Cumulative Accuracy Gain: Gradual drift (f) Cumulative Accuracy Gain: Mixed drift

Fig. 7: Example of Performance Accuracy on Agrawal datasets



(a) Accuracy (%): Abrupt drift (b) Accuracy (%): Gradual drift (c) Accuracy (%): Mixed drift

(d) Cumulative Accuracy Gain: Abrupt drift (e) Cumulative Accuracy Gain: Gradual drift (f) Cumulative Accuracy Gain: Mixed drift

Fig. 8: Example of Performance Accuracy (LED datasets)

This metric is inspired by measures used in lifelong learning research and provides a more accurate depiction of the recovery period after drift occurs. Intuitively the positive gain shows the advantage obtained during recovery period just after a drift. As compared to previous metrics, such as average accuracy, previous metrics may be bias to the long stable stream period, when both the baseline and our approach perform similarly.

We repeated all experiments ten times with varying seeds for every synthetic dataset configuration. Table I summarized the accuracy mean and standard deviation. We noticed that there is no discernible difference between GARF and GSARF, as effects are dampened over the entirety of the stream, whereby the long periods of stream stability in between drifts allows both GSARF and GARF to provide similar accuracy levels. The recovery of accuracy when a drift appears is shown using the cumulative gain plots in Figs. 7 and 8. The #Concepts represents the number of recurring concepts captured in the system. As both KDD99 and Covertype are real world datasets we do not have details on their drift intervals.

Runtime and Memory Evaluations. In these experiments, we used the *NVProf*, a CUDA applications profiling tool, for measuring the runtime of each the CUDA memcpy and kernel

TABLE I: Recurrent Concept Drifts Evaluation.

Dataset	Drift Type	#Concepts	Accuracy (%)			Runtime	
			ARF (CPU)	GARF (GPU)	GSARF (hybrid)	ARF (CPU)	**GSARF (hybrid)**
Agrawal	Abrupt	2	87.04 ± 0.12	81.84 ± 0.38	82.19 ± 0.76	18m17s	2m44s
	Gradual	2	86.59 ± 0.10	84.20 ± 0.13	84.48 ± 0.18	31m44s	4m31s
	Abrupt	3	89.14 ± 0.13	82.21 ± 0.38	82.76 ± 0.43	31m00s	2m42s
	Gradual	3	82.82 ± 0.14	79.36 ± 0.30	80.30 ± 0.27	38m16s	3m40s
	Mixed	3	84.56 ± 0.13	79.48 ± 0.65	79.83 ± 0.58	37m55s	3m10s
LED	Abrupt	2	71.21 ± 0.22	66.58 ± 0.36	71.44 ± 0.57	4m30s	1m31s
	Gradual	2	64.86 ± 0.12	66.53 ± 0.24	68.17 ± 0.43	6m20s	1m46s
	Abrupt	3	71.53 ± 0.16	67.40 ± 0.52	67.66 ± 0.39	4m44s	0m57s
	Gradual	3	67.16 ± 0.24	66.48 ± 0.47	67.32 ± 0.20	5m34s	1m36s
	Mixed	3	69.49 ± 0.30	67.35 ± 0.33	68.83 ± 0.37	4m49s	1m20s
KDD99	unknown	unknown	99.87	99.85	99.85	11m16s	7m4s
Covertype	unknown	unknown	72.48	77.82	77.82	7m07s	1m00s
Agrawal*	Gradual*	3	82.40	73.21	77.52	32h22m37s	14m32s

NOTE: Both GARF and GSARF have tree depth limited. We omitted the speed evaluation of GARF, since GARF is simply GSARF having the state-adaptive algorithm turned off. Agrawal* contains a high number of data instances (3 million) for assessing performance in the long run.

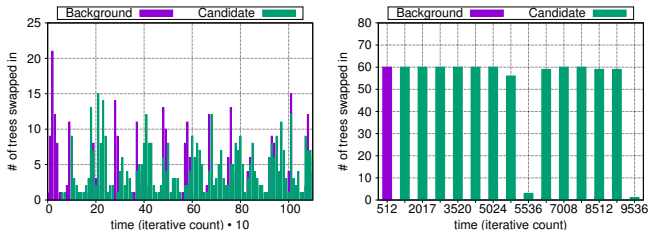
TABLE II: Number of drifts and runtime evaluation: Agrawal

Drift Type	# Drifts	ARF (CPU)	GARF (GPU)	**GSARF (hybrid)**
Abrupt	12	31m00s	2m38s	2m42s
	27	23m27s	3m02s	2m50s
	40	18m47s	3m28s	3m36s
Gradual	12	38m16s	4m08s	3m40s
	27	30m28s	4m07s	4m35s
	40	34m21s	4m26s	5m01s

TABLE III: Number of trees and CPU memory usage

Dataset	Drift Type	# Concepts	# Trees	Memory (GB)
Agrawal	Abrupt	2	667 ± 197	1.80 ± 0.53
	Gradual	2	1068 ± 70	2.89 ± 0.19
	Abrupt	3	1124 ± 61	3.05 ± 0.17
	Gradual	3	1090 ± 36	2.95 ± 0.10
	Mixed	3	923 ± 28	2.50 ± 0.08
LED	Abrupt	2	142 ± 37	0.38 ± 0.10
	Gradual	2	357 ± 18	0.95 ± 0.05
	Abrupt	3	751 ± 11	2.04 ± 0.03
	Gradual	3	458 ± 20	1.24 ± 0.05
	Mixed	3	389 ± 29	1.05 ± 0.08
KDD99	unknown	unknown	120	0.33
Covertype	unknown	unknown	100	0.27
Agrawal*	Gradual*	3	2260	6.12

calls. The runtime was affected by the number of trees grown, the number of instances being trained in parallel, and the number of drift points in the dataset. Table I compared the runtime between ARF (CPU) and GSARF. We omitted the runtime results of GARF, since both GARF and GSARF have the same bottleneck on data transfers, as GARF is simply GSARF with the state-adaptive algorithm turned off. As a result, the difference in runtime between GSARF and GARF is negligible. In our implementation, whenever a warning or drift is detected on any number of trees, the full state of the GPU random forest are copied from GPU to CPU's side of memory for simplicity. We also conducted a runtime analysis of number of drifts for the Agrawal dataset as shown in Table II.



(a) Gradual drift (b) Abrupt drift

Fig. 9: Candidate vs Background Tree Swap (Average Case)

Results for the number of trees stored on the CPU side and a memory estimation as shown in Table III. In our experiments we have fixed the tree depth to 11 since the size of the tree state grows exponentially to the depth. It takes approximately 2.71 MB to store the full state of a single tree. The tree can be brought back to the GPU and continue growing.

Tree Reuse Rate Evaluation. We evaluate the performance of candidate tree reuse rates across data streams with different characteristics. The proportion of candidate to background tree swaps from both a gradual and abrupt dataset are shown in Fig. 9. The experiments demonstrated the algorithm swapped in candidate trees as the expected.

Parameter Sensitivity Evaluation. There are three parameters that may have an effect on the system specifically the ϵ threshold, δ threshold, the minimum edit distance threshold. We varied the ϵ threshold from 0.01 to 0.80. This threshold is calculated from the tree's prediction and the true label for

the data instance. Table IV shows an increasing ϵ threshold had a relatively consistent decreasing effect on the reuse rate of abrupt drifts. This gives us an indication that in abrupt datasets, the foreground trees are significantly dependant on the candidate trees to be swapped into the ensemble.

TABLE IV: Sensitivity Analysis ϵ Threshold

Datasets	ϵ Threshold	Accuracy Gain	Reuse Rate
LED (Abrupt)	0.01	4,956.46	0.90
	0.05	4,849.98	0.89
	0.20	4,810.29	0.89
	0.60	4,129.89	0.79
LED (Gradual)	0.01	1,637.79	0.63
	0.05	1,525.81	0.58
	0.20	887.71	0.32
	0.60	23.46	0.18

TABLE V: Sensitivity Analysis δ Threshold

Datasets	δ Threshold	Accuracy Gain	# Trees
LED (Abrupt)	0.00	4,962.37	136.2 ± 31.8
	0.05	4,805.28	144.7 ± 38.7
	0.15	4,857.62	144.3 ± 38.1
	0.20	4,868.42	144.2 ± 37.9
LED (Gradual)	0.00	1,367.78	362.3 ± 11.8
	0.05	1,687.28	371.1 ± 23.3
	0.15	1,803.62	362.2 ± 21.2
	0.20	1,669.98	336.8 ± 25.2

TABLE VI: Minimum Edit Distance Threshold

Datasets	Min. Edit Distance	Accuracy Gain	Reuse Rate
LED (Abrupt)	50	22.18	0.02
	100	22.18	0.02
	110	165.13	0.03
	120	4,849.98	0.89
LED (Gradual)	50	570.25	0.18
	100	1,764.38	0.60
	110	1,689.00	0.62
	120	1,637.79	0.63

Table V showed a consistent number of potential candidate trees appended to the CPU. Table VI shows that reducing the minimum edit distance threshold from 120 resulted in an abrupt reduction in the accuracy gain and reuse rate. This indicates that GSARF finds pattern even when given a larger edit distance threshold.

VI. Conclusion

By exploiting the parallelism, provided by GPU, and more available CPU memory, we designed an algorithm that memorizes seen concepts and effectively adapted them when the same concepts reappear. We have shown that we can get a substantial speed up in comparison to CPU based random forest technique. Our results show that the state adaptive algorithm effectively maintains its accuracy when the data streams contains recurrent concepts. When compared against random forests without memory of discarded foreground trees, our technique is shown to achieve a gain in accuracy.

In the future, we may fine tune the number of trees allowed in the random forest. This will make the size of the random forest adjustable to the distribution of the stream. In terms of design requirements, the swapping scheme between GPU and CPU pools can potentially be devised to reduce the affect of data transfers.

Acknowledgment

This project is funded by the Vice-Chancellors Strategic Development Fund, The University of Auckland, New Zealand. We also like to thank Tom Hinton for assisting and reviewing the paper.

References

[1] 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*, ser. ECAI'14, 2014, pp. 615–620.

[2] A. Haque, L. Khan, M. Baron, B. Thuraisingham, and C. Aggarwal, "Efficient handling of concept drift and concept evolution over stream data," in *IEEE ICDE*. IEEE, 2016, pp. 481–492.

[3] Y. Spenrath and M. Hassani, "Ensemble-based prediction of business processes bottlenecks with recurrent concept drifts," in *EDBT/ICDT Workshops*, 2019.

[4] A. Bifet and R. Gavaldà, "Learning from time-changing data with adaptive windowing," in *In SIAM International Conference on Data Mining*, 2007.

[5] Y. S. Koh, D. T. J. Huang, C. Pearce, and G. Dobbie, "Volatility drift prediction for transactional data streams," in *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018*, 2018, pp. 1091–1096.

[6] K. Chen, Y. S. Koh, and P. Riddle, "Proactive drift detection: Predicting concept drifts in data streams using probabilistic networks," in *2016 International Joint Conference on Neural Networks, IJCNN 2016, Vancouver, BC, Canada, July 24-29, 2016*, 2016, pp. 780–787.

[7] Y. Chen, L. Zhou, N. Bouguila, B. Zhong, F. Wu, Z. Lei, J. Du, and H. Li, "Semi-convex hull tree: Fast nearest neighbor queries for large scale data on gpus," in *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018*, 2018, pp. 911–916.

[8] H. Grahn, N. Lavesson, M. H. Lapajne, and D. Slat, "CudaRF: A CUDA-based implementation of random forests," in *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, Dec 2011, pp. 95–101.

[9] H. Abdulsalam, D. B. Skillicorn, and P. Martin, "Classification using streaming random forests," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 1, pp. 22–36, Jan 2011.

[10] 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*, vol. 106, no. 9, pp. 1469–1495, Oct 2017.

[11] A. Bifet, J. Read, I. Žliobaitė, B. Pfahringer, and G. Holmes, "Pitfalls in benchmarking data stream classification and how to avoid them," in *Machine Learning and Knowledge Discovery in Databases*, H. Blockeel, K. Kersting, S. Nijssen, and F. Železný, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 465–479.

[12] 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*, 2000, pp. 71–80.

[13] S. Hettich and S. D. Bay, "The UCI KDD archive," http://kdd.ics.uci.edu, Irvine, CA: University of California, Department of Information and Computer Science, 1999.

[14] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: massive online analysis," *J. Mach. Learn. Res.*, vol. 11, pp. 1601–1604, 2010.

[15] B. Krawczyk, L. L. Minku, J. Gama, J. Stefanowski, and M. Woźniak, "Ensemble learning for data stream analysis: A survey," *Information Fusion*, vol. 37, pp. 132 – 156, 2017.