The Journal of Supercomputing (2022) 78:10480–10500
https://doi.org/10.1007/s11227-021-04290-6

Parallel construction of Random Forest on GPU

Kennedy Senagi[1] · Nicolas Jouandeau[2]

Accepted: 21 December 2021 / Published online: 24 January 2022

Abstract
There is tremendous growth of data generated from different industries, i.e., health, agriculture, engineering, etc. Consequently, there is demand for more processing power. Compared to computer processing units, general-purpose graphics processing units (GPUs) are rapidly emerging as a promising solution to achieving high performance and energy efficiency in various computing domains. Multiple forms of parallelism and complexity in memory access have posed a challenge in developing Random Forest (RF) GPU-based algorithm. RF is a popular and robust machine learning algorithm. In this paper, coarse-grained and dynamic parallelism approaches on GPU are integrated into RF(dpRFGPU). Experiment results of dpRFGPU are compared with sequential execution of RF(seqRFCPU) and parallelised RF trees on GPU(parRFGPU). Results show an improved average speedup from 1.62 to 3.57 of parRFGPU and dpRFGPU, respectively. Acceleration is also evident when RF is configured with an average of 32 number of trees and above in both dpRFGPU and parRFGPU on low-dimensional datasets. Nonetheless, larger datasets save significant time compared to smaller datasets on GPU (dpRFGPU saves more time compared to parRFGPU). dpRFGPU approach significantly accelerated RF trees on GPU. This approach significantly optimized RF trees parallelization on GPU by reducing its training time.

Keywords Machine learning · Random Forest · GPU · Dynamic parallelism · Coarse-grained

✉ Kennedy Senagi
ksenagi@icipe.org

Nicolas Jouandeau
n@up8.edu

[1] International Centre of Insect Physiology and Ecology, Nairobi, Kenya

[2] LIASD, Université Paris8, Paris, France

1 Introduction

To-date, computing power has grown rapidly to a point where Moore's Law of doubling transistor counts every 2 years, coined in the 1960's, is no longer practical. Computer systems that have multicore CPU and many-core GPU that have brought enormous computing power to laptops and clustered computers. GPU has been proven to offer unprecedented computing power compared to CPU. It is therefore inherent that software engineers should come-up with better and more efficient ways of utilizing readily available huge computing power offered by GPU to accelerate software applications. Ideally, with the capabilities of GPU, performance of software should be better. However, software engineers fail to harness full potential of GPU due to several reasons including complexity in parallelism and memory [1, 2].

RF is a popular and robust machine learning algorithm which is applied in solving a wide array of research problems including agricultural, bioinformatics, etc. [3, 4]. RF is an ensemble classifier. Logically, RF builds independent diverging trees that have minimal data dependencies[1]. Moreover, in decision-making, the majority class in each tree is computed independently; the modal class in all decision trees is computed after all trees have been built and majority class found [5]. This makes it an ideal algorithm for GPU parallelization.

Parallelizing RF on GPU has been a challenge due to various bottlenecks including memory access design, parallelism, synchronization between thread[2] and warp[3] divergence. Consequently, this has lead to poor utilization of GPU computing resources resulting to poor computation time among other computing problems [2, 6]. This research presents coarse-grained and dynamic parallelism approaches to parallelizing RF trees on GPU on low-dimensional datasets. Experiment results show that these approaches significantly accelerated parallelization of trees in RF on GPU. Ultimately, this can significantly reduce time of training RF algorithm on GPU.

2 Related works

Wang et al. [7] adopted three scheduling decisions: prioritized execution of the child thread blocks, bound them to the stream multiprocessors (SMXs) occupied by their parent thread blocks and maintained workload balance across compute units. Experiments showed an average of 27% performance improvement over the baseline round robin thread block scheduler, commonly used on modern GPUs [7].

Rich and Alexandru [8] did an empirical evaluation of supervised learning on high-dimension data. Performance evaluated was done using accuracy (ACC), Area

[1] Data independence in RF is facilitated by bagging. In bagging, each tree is built from an independent subset of data. Each subset of data is generated by randomly sampling data from the original dataset with replacement [5].

[2] This is caused by irregular execution paths of threads in a warp.

[3] This is caused by irregular execution paths of warps in a block.

under the ROC Curve (AUC), and squared error (RMS). They also studied the effects of increasing dimensionality on SVM (LaSVM kernel and RBF kernels using stochastic gradient descent), ANN, Logistics Regression and Nave Bayes (NB) algorithms. AUC performance metrics showed that RF was a clear winner, followed by k-NN.

Similarly, Manuel et al. [9] evaluated 179 classifiers fetched from 17 families, namely discriminant analysis, multiple adaptive regression splines, Bayesian, rule-based classifier, boosting, ANN, bagging [10] stacking, RF and other ensembles, SVM, decision trees, generalized linear models, logistic and multinomial regression, k-NN, partial least squares and principal component regression and other methods. Parallel RF, a version of RF, turned out to be the best classifier when implemented in R and accessed without caret [9].

RF and gradient boosting machine (GBM) predictive algorithms were used to describe and predict Foodborne pathogens in poultry farming environments. Nawar and Mouazen [11] tested GBM, RF and ANN on soil information datasets. Their results showed that RF and ANN prediction models gave almost similar results but better than GBM on spatial soil information [11].

The accurate prediction of coal temperature played a vital role in preventing and controlling spontaneous combustion of coal in coal mines. RF and SVM were introduced and compared in predicting coal spontaneous combustion based on the in situ monitoring data. Results showed that RF was more robust and less sensitive to its parameters [12].

Vouzis and Sahinidis [4] created multiple kernels that executed different phrases of Basic Local Alignment Search Tool for Protein (BLASTP) sequence. A fine-grained mapping approach was adopted by using warps of threads to accelerate one sequence alignment to leverage the abundant parallelism offered by GPU.

Zhang et al. [6] proposed cuBLASTP that improved performance of BLASTP on GPU. cuBLASTP addressed the irregular execution paths caused by divergence and irregular memory access. They integrated decoupling and binning-sorting-filtering. These approaches optimized cuBLASTP on GPU [6].

Wen et al. [13] came up with run-length encoding compression and thread/block workload dynamic allocation, and reusing intermediate training results for efficient gradient computation. Gradient Boosting Decision Trees (GBDTs) executed faster on GPU. Experiments show that GBDTs executed 10–20 times faster than sequential version of XGBoost and had a speedup of 1.5–2.0 over a 40 threaded XGBoost on 20 CPU cores [13].

Search queries in a B+ tree are data-parallel, therefore, mapping them onto Accelerated Processing Unit (APU) (CPU+GPU) was inefficient. The coarse-grained approach led to better utilization of the available memory bandwidth in APU. Their parallelism in tree search executed multiple searches in parallel to optimize the Single Instruction Multiple Data (SIMD). This approach was 4.9x faster on the best case and 2.5x on average compared to a six-threaded, hand-tuned, Streaming SIMD Extension (SSE) optimized and CPU implementations [14].

Chen et al. [15] presented a hybrid Parallel Random Forest (PRF) algorithm for big data on Apache Spark platform. It combined data-parallel and task-parallel optimization. It was trained and tested on different datasets in UCI and other projects.

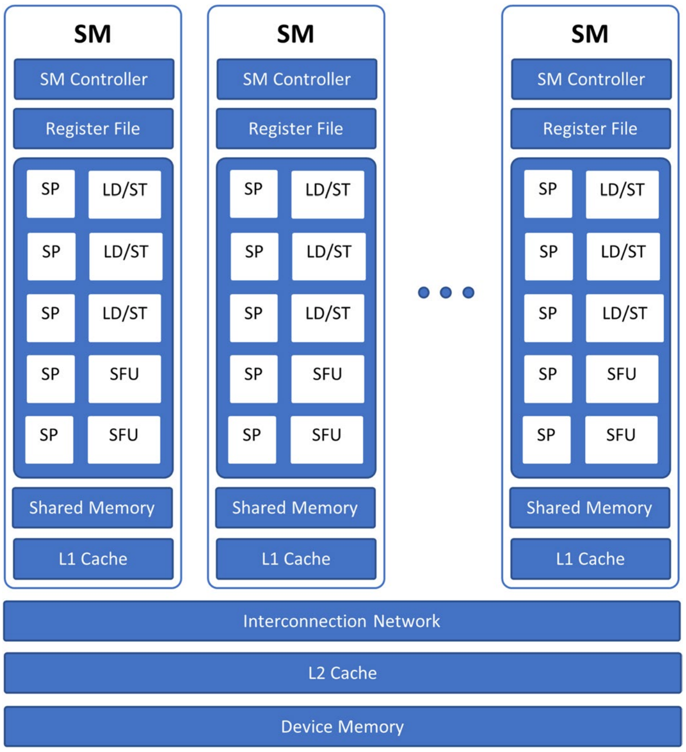

Fig. 1 GPU architecture of streaming multiprocessors

Experiment results showed that PRF algorithm had a better accuracy and scalability compared to algorithms implemented by Spark MLlib [15].

Generally, RF is a robust and widely used ML algorithm. GPU is a highly threaded platform that software engineers can parallelize algorithms to achieve better time of execution. However, while programming on GPU, many algorithms face several challenges including divergence/irregular memory access and execution path [6, 13]. Parallelizing RF algorithm of GPU faces the similar challenges [15, 16]. This research integrates dynamic parallelism and coarse-grained approaches to improve time of execution of RF on GPU.

3 The graphical processing unit

The GPU, also known as graphics accelerator card, is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate creation of images in a frame buffer intended for output on a display. Figures 1 and 2 show simple representations of GPU architecture and threads accessing different memories. GPU contains numerous (e.g., thousands) cores that are grouped into streaming multiprocessors (SMs). In the Compute Unified Device Architecture (CUDA) capable GPU, threads are grouped into blocks which are also called thread blocks. A number of SMs form one block; this number varies from one CUDA GPU generation to another. Each SM has a number of Stream-Processors (SPs) that share control logic bundled in the Special Function Units (SFU), Load(LD)/Store(ST) units and cache (L1). At any given timestamp, an SM executes instructions of one thread block. Accessing GPU's global memory is an expensive computation, therefore, accessing GPU global memory should be avoided as much as possible. Moreover, irregular accesses to global memory are even more expensive due to the small number of cache lines.

GPU threads are organized in blocks, and blocks are organized in grids. The number of blocks per grid is limited by streaming multiprocessors. The number of threads in a block varies, e.g., 512 or 1024. CUDA uses Single Instruction Multiple Thread (SIMT) architecture to manage and execute threads. It groups threads into a unit called warp. A warp is the smallest work unit of GPU. In the warp, all threads execute the same instruction in parallel. Threads in the same block share the same memory called shared memory while all threads in a grid share the same memory called global memory. Therefore, to optimally utilize GPU resources, algorithms need to have a good parallelism and memory access patterns. Moreover, synchronization overhead between threads and warp divergence should be reduced to further optimize parallelism on GPU. Better memory usage can be improved by reducing irregular memory access in global memory. Also, excessive or unorganized use of shared memory can cause bank conflicts resulting to re-execution of memory instructions which can degrade GPU performance. Generally, GPU is built for throughput rather than latency. Throughput measures the amount of work done per a given period of time, while latency is the time taken from the start to the end a process [2].

Today, GPUs are used for non-graphics applications. The platform is often referred to as general-purpose graphics processing units, GPGPU. The GPGPU is rapidly emerging as a promising solution to achieve high performance and energy efficiency in various computing domains ranging from embedded to big data computing [1, 17].

Compared to CPUs, GPUs are tolerant to latency and have high throughput. GPUs have a better handling of data parallelism, while CPUs have a better handling of task parallelism, CPUs have multi-threaded cores, while GPUs have single-instruction multiple threaded[4] (SIMT) cores and GPU support more threads than CPU. These make GPUs very powerful computers compared to CPUs [1, 13, 18].

In practice, parallelizing algorithms on GPU offer better performance than CPU. However, GPU performance has been limited due to various reasons including thread divergence, unsuitable access pattern in memory and load imbalances. These make it difficult to find performance bottlenecks on GPU architectures. It is

[4] Single-instruction multiple threaded is execution where a set of instructions are dispatched in a multi-threaded manner.

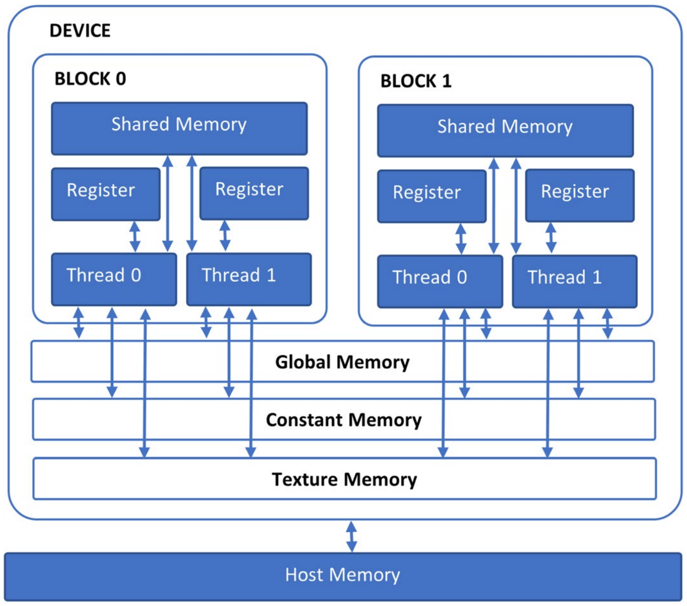

Fig. 2 Threads organized in blocks accessing different memories in the GPU

therefore important to use good performance evaluation tools in order to find performance bottlenecks and further guide in performance optimization.

4 The Random Forest algorithm and it's implementation

4.1 Random Forest description

RF is an ensemble classifier introduced by Leo Breiman [5]. Ensembles use the divide-and-conquer approach when learning from a dataset. Growing ensembles of trees (by generating random vectors for each tree) and letting them vote for the most popular class has resulted to significant improvements in classification accuracy.

Algorithm 1 outlines an overview of RF. The principle idea behind RF is to build many decision trees from the same dataset $\{X, Y\}$ using bootstrapping and randomly sampled variables to create trees with variations (randomization), where $X \in (x_1, x_2, x_n)$ and Y is the label. The dataset (X, Y) is then split into training (X_t, Y_t) and test (X_s, Y_s) datatsets. Bootstrapping generates new datasets $B_1, B_2, \dots B_n$ by randomly sampling examples from the training (X_t, Y_t) data uniformly with replacement. These bootstraps are then used for constructing decision trees (DT) which make-up a forest (RF), i.e., $(RF \in DT_1, DT_2, \dots, DT_N)$ where each $(DT_i \in B_i)$. Each tree is constructed by the principle of divide-and-conquer. That is, in each tree, starting at the root node, each attribute is recursively split. An attribute is picked (randomly or having the highest entropy) and split. The best split value in an attribute is evaluated using a splitting criterion, e.g., information gain or Gini index. The splitting criteria evaluate how good an attribute can be separated into homogeneous classes (considering the target variables). When building a tree, splitting a node is done recursively till when the leaf node is arrived at. The leaf node is a terminal node and its information cannot be split further. The leaf node gives the class attribute that is most common occurring. Each tree is grown to the largest extent possible. Binary tests are associated with each internal node. Test data (X_s, Y_s) are passed from the root to the leaves. Each of the leaves nodes contains a predictor label $\hat{y}$. In classification and regression trees (CART), we assume $y = f(x)$ for some unknown function f, the goal of learning is to estimate the function f given a labeled training set, and to make predictions using $\hat{y} = \hat{f}(x)$. In RF, the predictor label $\hat{y}$ at the leaf is either the average value of Y observations in the training set associated to that leaf (regression) or the modal/majority value of Y (classification). This forms the basis of accuracy performance metrics evaluation. During bootstrapping, the randomness minimizes correlation while maintaining strength. The main principle behind RF is to build a group of weak learners to form a strong learner [5]. RF is a popular and robust machine learning algorithm and has been used to solve diverse prediction problems [3, 4].

Algorithm 1 Random Forest [3]

```
Require: $\{X,Y\} \in \{(x_1,y_1),...,(x_n,y_n)\}, cols \in (x_1,...,x_n), rows \in (r_1,...,r_n)$
 1: $cols \leftarrow \phi$                                  ▷ Expected number of columns in each bag
 2: $rows \leftarrow \beta$                                 ▷ Expected number of rows in each bag
 3: $split \leftarrow \lambda$                              ▷ Ratio of train to test sets
 4: $NoTs \leftarrow \theta$                                ▷ Expected number of trees to be built
 5: $\{X_t,Y_t\},\{X_s,Y_s\} \leftarrow$ SPLITDATASET($\{X,Y\}, split$)   ▷ Split to training $\{X_t,Y_t\}$ and test $\{X_s,Y_s\}$ sets
 6: procedure RANDOMFOREST($\{X_t,Y_t\},\{X_s,Y_s\}, NoTs, rows, cols$)
 7:     $Bags \leftarrow$ BAGGING($\{X_t,Y_t\}, NoTs, rows, cols$)
 8:     $accuracy\_avg \leftarrow 0$
 9:     for each $bag$ in $Bags$ do
10:         $tree \leftarrow$ CONSTRUCTTREE($bag$)                ▷ Each tree constructed from each bag
11:         $accuracy \leftarrow$ TESTING($tree, \{X_s,Y_s\}$)
12:         $accuracy\_avg \leftarrow accuracy\_avg + accuracy$
13:     $accuracy\_avg \leftarrow accuracy\_avg / NoTs$
```

4.2 Random Forest implementation

RF programming on GPU was done in CUDA C [19]. Let us describe the modules developed to realize RF.

4.2.1 Bagging

Bagging is also known as bootstrap. The bagging module generated b bags/bootstraps from the original dataset. In each bag, x records were randomly generated with replacement from the original database. Nonetheless, in each bag, y features were randomly sampled. Note that, in bagging, a user can configure number of bags (b), number of random records in each bag (x) and number of random features (y). Bagging brings about data independence which is necessary for constructing independent diverging trees. This reduces thread synchronization issues that reduce efficiency.

4.2.2 Trees construction: feature selection, splitting and pruning

The most popular algorithms for evaluating a split criteria are ID3 and its successor C4.5. Compared to ID3 algorithm, C4.5 has a better handling of pruning, missing variables and works well with both discrete and continuous data [20]. This research implemented ID3 in trees construction because it was simple.

Entropy of labels, $E(y)$, is calculated using Eq. 1, where c is the number of unique labels in the feature, i.e., classes. If variables in the feature are completely homogeneous/pure, the entropy is zero. And if they are equally divided, the entropy is one. Considering Eq. 1, entropy was calculated for each feature $E(y, x)$ using Eq. 2, where $P(c)$ is probability of a class in the labels and $E(c)$ is the entropy of the class(es) in the labels. Information Gain, $\text{Gain}(y, x)$, for each feature is calculated using Eq. 3. $\text{Gain}(y, x)$ values for all features were then arranged in descending order. From the array, a feature was then picked for splitting one by one. The first to be split at the root node was the most impure, i.e., heterogeneous. Technically, each bag discussed in previous subsection was used to construct a tree.

$$E(y) = \sum_{i=0}^{c-1}(-p_i \log_2 p_i) \tag{1}$$

$$E(y,x) = P(c)E(c) \tag{2}$$

$$\text{Gain}(y,x) = E(y) - E(y,x) \tag{3}$$

Gini index split criteria were recommended by Breiman [5] in splitting features. Gini index measures the impurity levels of information. The attribute that provides the largest reduction in impurity is chosen to split the node. A split point/value was picked randomly and its Gini index was calculated using Eq. 4. Average Gini index was based on the left(x_L) and right (x_R) splits using Eq. 5. The best split was calculated using the Non-Deterministic Algorithm explained in [3]. In this case, 10% of random split points were sampled and thresholds of $\text{Gini}(y, x) == 0$ were evaluated to stop further splitting. After splitting, x_L and x_R were sent to the left and right nodes, respectively. If $\text{GI}(x_\mathrm{L}) == 0$ or $\text{GI}(x_\mathrm{L}) == 0$, i.e., pure class, the respective node was marked as a leaf otherwise the node was split. A node was not split further if it was marked as a leaf or if maximum depth of splitting a feature or tree was reached. If a tree reached its maximum depth with a node not marked as leaf, i.e., not a pure node, the class with the highest frequency was selected as the final class label, and the node was marked *semi-leaf*.

$$\mathrm{GI}(x) = 1 - \sum_{i=0}^{i-1} (p_i^2) \tag{4}$$

$$\mathrm{Gini(Average)} = p_{x_\mathrm{L}} \times GI(x_\mathrm{L}) + p_{x_\mathrm{R}} \times GI(x_\mathrm{R}) \tag{5}$$

Furthermore, post-pruning measures were integrated. In post-pruning, as a tree grows, out-of-bag (OOB) generalization error is evaluated and tracked before splitting a node. If OOB increased in two sequential splits, further splitting was stopped and the node was evaluated as a semi-leaf. The same stopping criteria were used when a node got to the maximum tree depth.

4.2.3 Testing

In our implementation, a node was marked as with a leaf or semi-leaf and had a label $\hat{y}$. Constructed trees were evaluated using accuracy. To calculate accuracy, unseen data were run from the root to the leaves and semi-leaves of all trees. Each record of the unseen data had a label y. Each leaf and semi-leaf of each trees was evaluated with, $\hat{y} = y$, and accuracy is calculated using Eq. 6, where n is the number of nodes marked leaf and semi-leaf. All records (of unseen data) accuracies were averaged to give each tree's overall accuracy. Majority vote of all the trees was taken as RF overall final accuracy. Moreover, the time taken to train and test RF, i.e., time of execution, is evaluated using Eq. 7 and recorded.

$$\mathrm{acc}(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} 1(\hat{y}_i = y_i) \tag{6}$$

$$\text{Time of Execution} = \text{Time(End)} - \text{Time(Start)} \tag{7}$$

5 The granularity concept on GPU

Granularity is the ratio of computation to communication in a parallel program. A grain is the smallest amount of work to computer. Tasks with fewer computations are referred to as fine-grained tasks, while tasks with many computations are referred to as coarse-grained tasks. If a task is too fine, scheduling and mapping overhead are large and consume a significant amount of total execution time. Therefore, decomposition procedures should be employed to find a good balance between number of tasks and their granularity [21].

In a program, parallel execution can be performed at different levels: instruction, statement, loop and/or function. These levels result to different task granularities. For instance, a fine grained tasks can result when a small number of instructions or statements are consolidated, a coarse-grained task comes-about when functions are executed to form a significant amount of computations, while a medium-grained tasks are typical at the loop level since each loop iteration consists of several statements [21].

Optimization of an algorithm is highly dependent on how the algorithm and the application environment are structured [14, 22]. Essentially, thread blocks implement coarse-grained scalable parallelism, while lightweight threads (within a block) implement fine-grained parallelism. This characteristic makes it easy to implement independent data parallelism in algorithms [23].

6 The dynamic parallelism concept on GPU

Dynamic Parallelism [24] is a new feature provided in the latest architectures of GPU cards that have compute capability[5] 3.5 and higher, such as the Kepler and Maxwell architectures. Dynamic parallelism allows CUDA kernel (parent) to create new kernels (children) within the device(GPU). The parent kernel can create children kernels, and the children can create other grandchildren kernels, and so on asynchronously. Nonetheless, dynamic parallelism allows synchronization between parent and child kernels, i.e, a child CUDA kernel can be called from a parent kernel and then synchronizes on completion of that child kernel. Unlike recursion, dynamic parallelism has the ability to create new work for itself in the device. This reduces work spent in transferring data and execution control between the device and the host. It is important to note that dynamic parallelism launches cannot target other GPU device in a multi-GPU system. Dynamic parallelism programming can be done in platforms such as OpenCL or CUDA [7, 25].

This research prototyped experiments in CUDA. We implemented RF in C language and parallelization with CUDA on GPU and tested our algorithms on GeForce GTX 1080 and profiled our programs using nvprof on GPU. Performance was evaluated using speedup [26–28].

7 Custom Random Forest implementation for GPU

Implementing RF for GPU architecture required a different programming approach considering GPU has device, host, and kernels to be configured. Moreover, GPU has different architecture specification, i.e., different numbers of threads per block, different number of blocks per grid, grids and memory capabilities. This study used GeForce GTX 1080, whose dimension was 1024 x 1024 x 64, with a maximum threads per block of 1024, total global memory of ≈ 8.5 GB, ≈ 50 KB of shared memory per block and ≈ 65 KB per register.

Inspired by [27, 29, 30], this research implemented variants of RF executions from scratch, i.e., sequential execution on CPU, parallel execution on GPU and dynamic parallelism combined with coarse grained execution on GPU. The implementations are explained as follows:

[5] Compute capability determines the general specifications and available features of a GPU.

1. Sequential execution of RF on CPU (seqRFCPU) - RF trees were implemented to run sequentially on CPU considering principles outlined by [5]. The sequential RF algorithm implementation is explained in this section and is also outlined in Algorithm 1.
2. Parallelizing RF trees on GPU (parRFGPU) - RF tree was developed on GPU considering principles outlined by [5]. However, kernels were launched to construct RF trees in parallel on GPU, as outlined in Algorithm 2 and explained in this section. Time of execution results were recorded and used in calculating speedup using Eq. 8. Note that, parallelizing RF on CPU was not implemented since this research was solving RF GPU parallelization challenges.
3. Dynamic parallelism in RF (dpRFGPU) - This was our core contribution. RF tree was constructed based on [5] principles. In Sects. 6 and 5, this research noted that coarse-grained and dynamic parallelism are promising approach in parallelizing algorithms. We did not come across any literature covering coarse-grained dynamic parallelism of RF trees on GPU. To integrate these concept in RF, device-side nested kernels on GPU were launched to build trees dynamically, simultaneously and independently. Coarse-grained approach was incorporated in bagging by increasing work done by each thread and each thread built each tree independently. Details are explained in this section and in Algorithm 2. Time of execution results were recorded and used to calculate speedup using Eq. 8.

Algorithm 2 presents RF learning algorithm [5] in a summarized way; where *NoTs* is the expected number of trees, NoRs is the number of rows in each bag and NoCs is the number of columns in each bag. The initial dataset DS $\in \{(x_1, y_1), \ldots, (x_n, y_n)\}$ is divided in a training dataset $\mathrm{DS}_{\mathrm{train}}$ and a testing dataset $\mathrm{DS}_{\mathrm{test}}$, defined by the ratio λ of train to test sets.

Algorithm 3 presents the Random Forest decision given by taking the majority vote of each tree decision. The *acc* value computed is the accuracy value regarding to all of the samples of $\mathrm{DS}_{\mathrm{test}}$. The core of the Random Forest decision process, defined for a single sample x, is only between lines 3 and 7.

In both parRFGPU (Algorithm 4) and dpRFGPU (Algorithm 5), dataset was loaded in global memory and linearized to 1D array. In parRFGPU, a kernel was launched to transfer the linearized data and other configuration parameter to the device. In the device, the linearized data were reorganized back to 2D array, dataset split to training and test sets, bagging functions called and, thereafter, each tree built simultaneously and independently. Each thread built a tree. However, in dpRFGPU, after linearizing data to 1D in host and the parent kernel was launched to initialize dynamic parallelism. Similarly, the parent kernel transferred the linearized data and other configuration parameter to the device. However, in the device, dataset was split to training and test sets then children kernels launched to create new work and build threads of trees dynamically, simultaneously and independently. Note that, only one parent kernel was launched in the host. To further optimize the parent kernel in dpRFGPU, a coarse-grained approach was integrated. In the coarse-grained approach, amount of work of each thread was increased by ensuring that each thread (in each block) built its own tree independently with its own data (generated by bagging within the children kernels). In implementations of Algorithm 4 and 5, synchronization barriers were not introduced, therefore, trees were constructed and tested to completion, in parallel, without waiting for each other.

RF recorded significant accuracies between 2 and 512 NoTs [3, 31, 32]. Considering this, this study built RF up-to 1024 NoTs, meaning a maximum of 1024 threads were required. This research invoked one thread per block, to build trees independently and simultaneously. This could also reduce challenges of warp divergence, memory overload or leakages in the blocks, its registers or cache. Moreover, to reduce frequency of accessing global memory, we fetched the linearized data once from global memory, converted it to 2D and passed it the bagging function. Through bagging, a subset of the dataset was used to build a tree, i.e., data parallelism. Bags were loaded from the global memory since they could not fit in shared memory especially for huge datasets. Data parallelism was essential in parallelizing algorithms on GPU, i.e., it avoided memory and thread synchronization overheads which are catastrophic in parallelism [25].

Algorithm 2 Random Forest learning

```
Require: $DS_{train}, NoTs, NoRs, NoCs$
1: $Bags \leftarrow$ BAGGING($DS_{train}, NoTs, NoRs, NoCs$)
2: $\mathscr{C} \leftarrow \emptyset$
3: for each $bag$ in $Bags$ do
4:     $tree \leftarrow$ CONSTRUCTTREE($bag$)          ▷ Each tree constructed from each bag
5:     $\mathscr{C} \leftarrow \mathscr{C} + tree$          ▷ Each tree is added in the forest
6: return $\mathscr{C}$
```

Algorithm 3 Random Forest decision

```
Require: $DS_{test}, \mathscr{C}$
1: $acc \leftarrow 0$
2: for each $(x, y)$ in $DS_{test}$ do
3:     $\mathscr{D} \leftarrow \emptyset$
4:     for each $tree$ in $\mathscr{C}$ do
5:         $y_i \leftarrow$ DECISION($x, tree$)          ▷ Each tree gives a sub-decision
6:         $\mathscr{D} \leftarrow \mathscr{D} + y_i$          ▷ Each sub-decision is collected in a set
7:     $\hat{y} \leftarrow$ MAJORITY($\mathscr{D}$)          ▷ The final decision is the most choosen sub-decision
8:     If $\hat{y} = y$ then $acc \leftarrow acc + 1$
9: $acc \leftarrow acc/$SIZEOF($DS_{test}$)
```

Algorithm 4 Random Forest with Trees Parallelized on GPU

```
1: $\mathscr{T} \leftarrow 2, 4, 8, 32, 64, 128, 512, 1024$
2: $DS \leftarrow$ LoadToGlobalMemory()
3: $ds \leftarrow$ LinearizeDataset($DS$)
4: $rows \leftarrow x$                                  ▷ Expected number of rows in each bag
5: $cols \leftarrow y$                                  ▷ Expected number of columns in each bag
6: $split \leftarrow z$                                 ▷ Ratio of train to test sets
7: for each $\theta$ in $\mathscr{T}$ do
8:     $nblocks \leftarrow \theta$                      ▷ Equivalent to number of trees
9:     $nthreads \leftarrow 1$
10:    procedure Kernel_RF<<< $nblocks, nthreads$ >>>($ds, rows, cols, split$)
11:        $ds2D \leftarrow$ ConvertTo2D($ds, rows, cols$)
12:        ($ds\_train, ds\_test$) $\leftarrow$ SplitDataset($ds2D, split$)
13:        $bag \leftarrow$ Bagging($ds\_train$)
14:        $tree \leftarrow$ ConstructTree($bag$)
15:        $tree\_accuracy \leftarrow$ Testing($tree, ds\_test$)
```

Algorithm 5 Random Forest Coarse-Grain and Dynamic Parallelism of Trees on GPU

```
1: $\mathscr{T} \leftarrow 2, 4, 8, 32, 64, 128, 512, 1024$
2: $DS \leftarrow$ LoadToGlobalMemory()
3: $ds \leftarrow$ LinearizeDataset($DS$)
4: $rows \leftarrow x$                                  ▷ Expected number of rows in each bag
5: $cols \leftarrow y$                                  ▷ Expected number of columns in each bag
6: $split \leftarrow z$                                 ▷ Ratio of train to test sets
7: for each $\theta$ in $\mathscr{T}$ do
8:     $nblocks \leftarrow \theta$                      ▷ Equivalent to number of trees
9:     $nthreads \leftarrow 1$
10:    procedure Parent_Kernel_RF<<< $nblocks, nthreads$ >>>($ds, rows, cols, split$)
11:        $ds2D \leftarrow$ ConvertTo2D($ds, rows, cols$)
12:        ($ds\_train, ds\_test$) $\leftarrow$ SplitDataset($ds2D, split$)
13:        procedure Children_Kernels_RF<<< $nblocks, nthreads$ >>>($ds\_train$)
14:            $bag \leftarrow$ Bagging($ds\_train$)
15:            $tree \leftarrow$ ConstructTree($bag$)
16:            $tree\_accuracy \leftarrow$ Testing($tree, ds\_test$)
```

8 Experiment set-up

Section 7 discussed various implementations of RF: CPU sequential execution (seqRFCPU), GPU parallelization (parRFGPU) and GPU coarse-grained dynamic parallelism (dpRFGPU). This research noted that high-dimensional datasets were cumbersome to configure and execute in dynamic parallelism probably due to the high number of asynchronously children threads generated which created numerous nested deep decision trees in RF. Therefore, implementations were tested on four

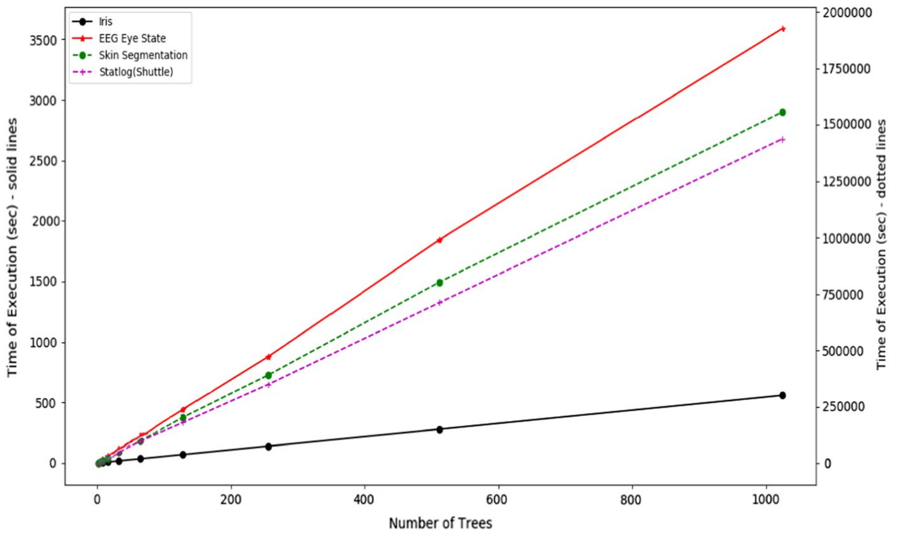

Fig. 3 Time of execution (sec) against number of trees across four datasets - sequential execution of RF on CPU

low-dimensional datasets[6] collected from UCI Machine Learning [33] namely EEG Eye state (14 attributes and 14980 records), Skin Segmentation (4 attributes and 245057 records), Iris (4 attributes and 150 records) and Statlog(Shuttle) (9 attributes and 58000 records). Experiments were run on CPU (Intel(R) Xeon(R) CPU E5-46100 @ 2.40GHz) and GPU (GeForce GTX 1080). On GPU, experiments were profiled using nvprof [34]. Experiments were prototyped in CUDA. RF was implemented in C language.

Performance of algorithms on GPU was evaluated using speedup, described in Eq. 8, where t_{seq} is time recorded in sequential execution and t_{par} is time recorded in parallel execution [25].

$$\text{speedup} = \frac{t_{\mathrm{seq}}}{t_{\mathrm{par}}} \tag{8}$$

9 Results and analysis

Figure 3 shows a line graph of time of execution against NoTs across four datasets when RF trees were executed sequential on CPU. Time of execution of RF when training all datasets apart from Iris increased steadily. Time of execution of

[6] The selected datasets had a variety of characteristics (e.g., dimensionality and number of records) that could reduce biasness in the experiments. These datasets also worked well with the program prototypes this research developed for experiments.

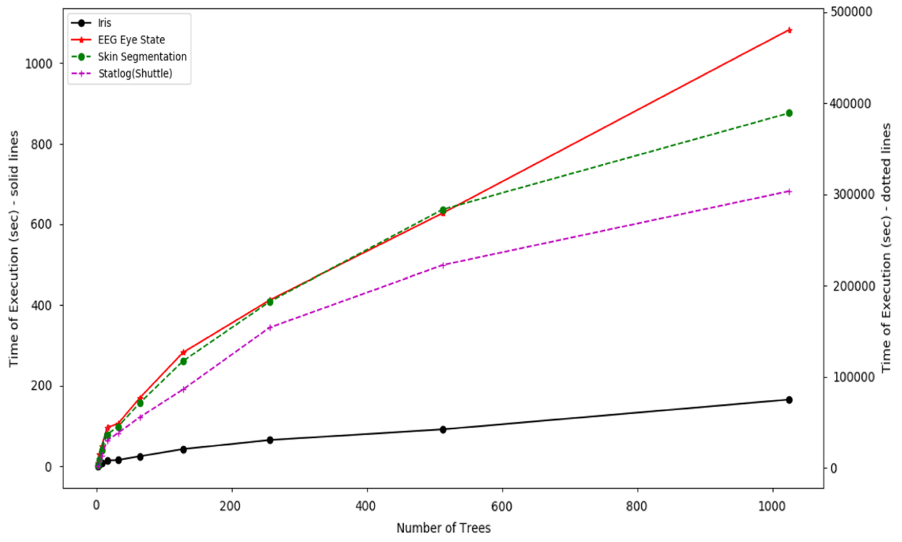

Fig. 4 Time of execution(sec) against number of trees across four datasets - parallelized RF on GPU

Iris increased slowly. RF required more time to train Statlog (Shuttle) dataset followed by Skin Segmentation, EEG Eye State then Iris. This could be due the size of the dataset; Statlog (Shuttle) was the largest and Iris the smallest. Larger datasets required more computation time. This resulted to change in time of execution to be more significant compared to smaller datasets. It was also observed that, increase in NoTs when training RF increased time of execution linearly.Training more trees required more CPU time leading to execution time growing linearly, perhaps, there was also little interference from Inter-Process Communication (IPC) within the processes.

Figures 4 and 5 contain line graphs of time of execution(sec) against NoTs across of four datasets of parRFGPU and dpRFGPU, respectively. Generally, the lines rise slowly. These results inform us that dpRFGPU had a shorter time of execution compared to parRFGPU at all NoTs instances. In dpRFGPU, with the coarse-grained RF, dynamic parallelism created new work and build trees dynamically and simultaneously. Dynamic parallelism could have also reduced processor's idling time to process more work. These made dpRFGPU execute faster than parRFGPU. Moreover, time of execution rise slowly in both parRFGPU and dpRFGPU. Normally, it is expected that time of execution should reduce with the same magnitude as the processors are increased linearly, for instance, if the processors are doubled, time of execution to be reduced by half. But this was not the case because some time was consumed in IPC. At fewer NoTs, time of execution rose sharply then slowed down. This could be due to smaller NoTs require lesser IPC while larger NoTs require more IPC.

Speedup, calculated using Eq. 8, was used to evaluate acceleration of parRFGPU and dpRFGPU configured with different NoTs. The speedup values against NoTs for parRFGPU and dpRFGPU were plotted in Figs. 6 and 7, respectively. In

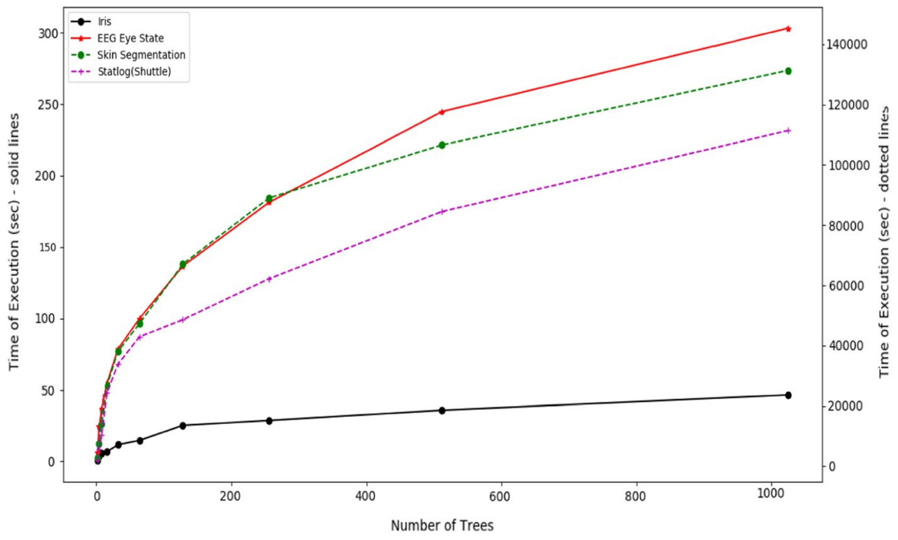

Fig. 5 Time of execution (sec) against number of trees across four datasets - dynamic parallelism of RF on GPU

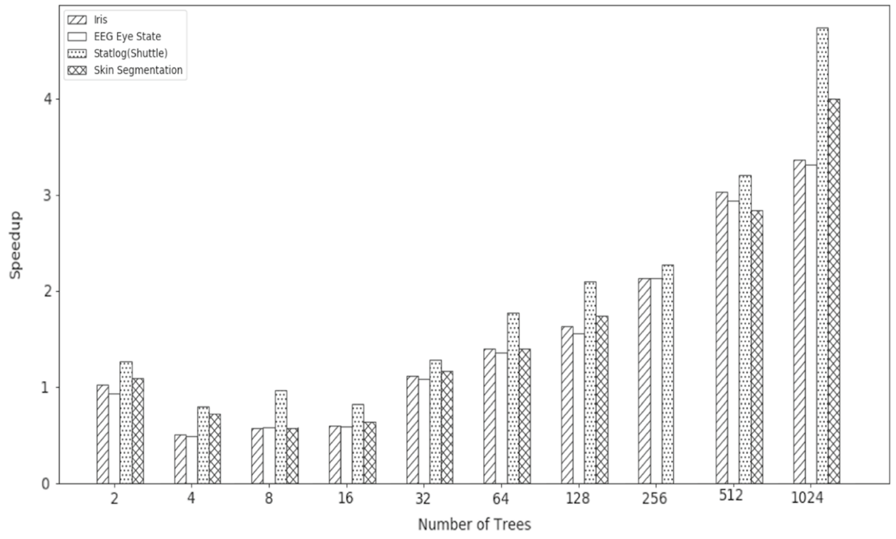

Fig. 6 Speedup against number of trees across four dataset - Parallelized RF on GPU

both bar graphs, for all the datasets, speedup went down slightly then rose moderately. Basically, speedup informs us if a problem is accelerated or not. Acceleration to a problem is realized if, speedup > 1. Generally, in both parRFGPU and dpRFGPU shown in Figs. 6 and 7, respectively, it was evident that acceleration was realized when RF was configured with 32 NoTs and above. We would expect

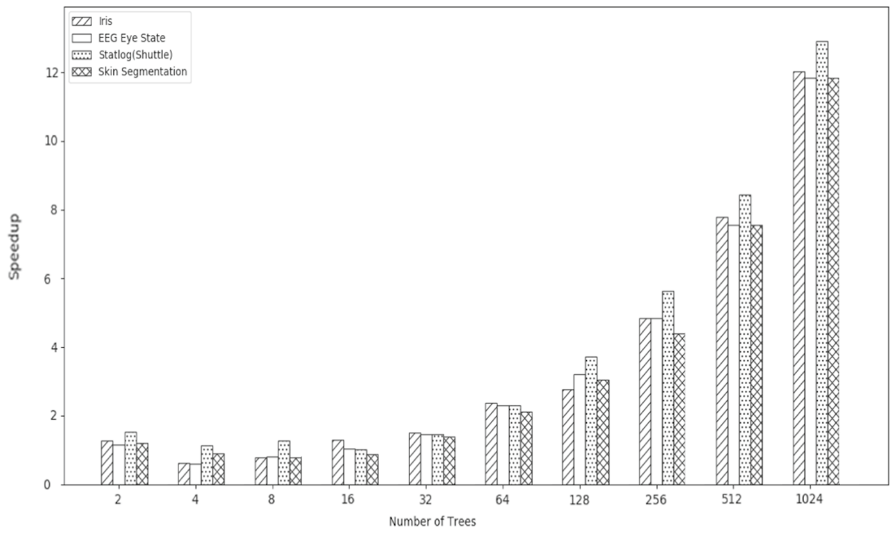

Fig. 7 Speedup against number of trees across four dataset - Dynamic Parallelism of RF on GPU

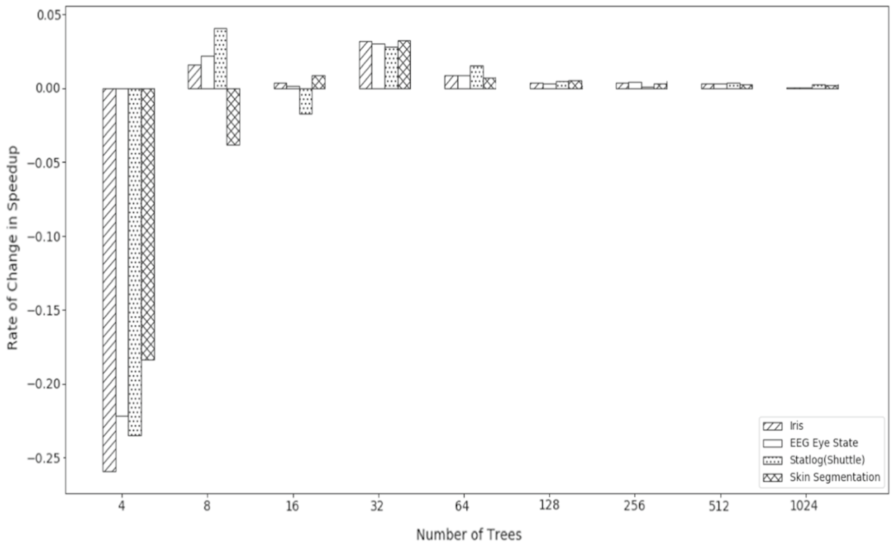

Fig. 8 Rate of change in speedup across four datasets - parallelized RF on GPU

parRFGPU and dpRFGPU to accelerate RF when tuned with the different NoTs, but this was not the case below 32 NoTs. When GPU parallelizations were profiled, when a kernel was launched, time was wasted due to moving data between the host and device. This could be the reason behind parRFGPU and dpRFGPU not accelerating below 32 NoTs threshold. However, above the threshold, besides

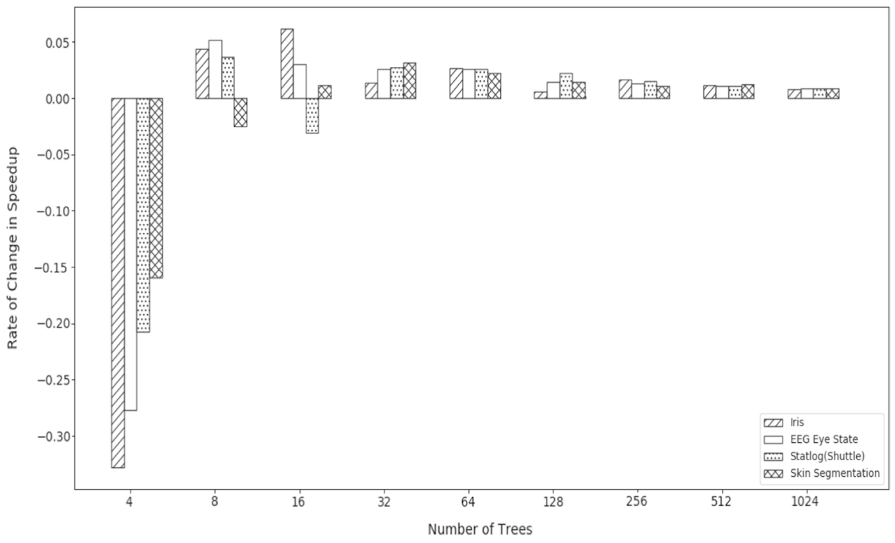

Fig. 9 Rate of change in speedup across four datasets - dynamic parallelism of RF on GPU

time wasted in sending data between the host and device, dpRFGPU and parRFGPU harnessed GPU computing power and performed better.

Moreover, Figs. 8 and 9 illustrate the rate of change of speedup against different NoTs for parRFGPU and dpRFGPU. Generally, for both parRFGPU and dpRFGPU, at the beginning, rate of change of speedup was at the negative. Then, there was a significant change to the positive, rose a little bit, then went down moderately. At 4 to 8 NoTs, there was a significant rate of change in speedup. At the 16 and 64 NoTs, both parRFGPU and dpRFGPU had maximum rates of change in speedup, thereafter, rates of change in speedup slowed down. The significant change of rate of speedup to the positive could signify GPU's range of breaking from time wasted when data were transferred between the host to the device. Thereafter, the slowing down of change in acceleration could be due to increase in IPC within the threads when more number of trees are parallelized. The rate of change in acceleration slowed down meaning increase in NoTs did not improve the speed of accelerating RF trees when parallelized on GPU. However, in parRFGPU, as seen in Fig. 8, a negative change in speedup is observed at 16 NoTs in Iris, EEG Eye State and Statlog (Shuttle) datasets. While in dpRFGPU as shown in Fig. 9, there was a negative change in acceleration at 16 NoTs in Statlog (Shuttle) dataset. The negative change in accelerations came at a time when the GPU is about to experience speedup, as we earlier observed in Figs. 6 and 7, in both parRFGPU and dpRFGPU, respectively. Probably the negative rate of speedup signified breakpoint to acceleration on GPU.

Table 1 contains average speedups for parRFGPU and dpRFGPU. From our research prototypes and outcomes, a user training RF using parRFGPU and dpRFGPU can experience approximate 1.62 and 3.57 accelerations, respectively.

Table 2 shows time a user could have saved when training different datasets at worse case scenario (1024 number of trees), compared to sequential execution.

Table 1 Average speedup results

Dataset	Average speedup	
	parRFGPU	dpRFGPU
Iris	1.54	3.53
EEG eye state	1.50	3.48
Statlog (Shuttle)	1.92	3.94
Skin	1.63	3.41
Averages	1.62	3.57

Table 2 Worse case time evaluations of parRFGPU and dpRFGPU compared to sequential execution on CPU at 1024 number of trees

Dataset	μ Time (hours) of execution			Time (hours) improved compared to seqRFCPU	
	seqRFCPU	parRFGPU	dpRFGPU	parRFGPU	dpRFGPU
Iris	0.155	0.046	0.013	0.109	0.142
EEG eye state	0.997	0.301	0.084	0.697	0.913
Statlog (Shuttle)	399.336	84.289	30.934	315.047	368.402
Skin	432.196	108.085	36.486	324.111	395.710

Table 3 Accuracy, $F1$ and precision scores of RF on different datasets

Dataset	Accuracy	$F1$ score	Precision
Iris	0.9777	0.9780	0.9803
EEG eye state	0.9321	0.9319	0.9329
Statlog (Shuttle)	0.9994	0.9994	0.9994
Skin	0.9995	0.9995	0.9995

Generally, dpRFGPU saved more time than parRFGPU. Dynamic parallelism and course grained approaches utilized GPU kernels better making dpRFGPU perform better than parRFGPU. Moreover, parRFGPU and dpRFGPU saved significant time on relatively larger datasets compared to smaller datasets. Meaning parRFGPU and dpRFGPU approaches were better for larger datasets that require more throughput and latency.

This research evaluated the accuracy, $F1$ and precision scores of RF on the different datasets and tabulated the results in Table 3. RF was able to learn, adopt and make accurate predictions from the datasets. Acceleration of tasks was the main reason for parallel execution of algorithms. Therefore, running an algorithm in parallel and sequentially would give the same accuracy, $F1$ and precision score keeping other factors constant [35]. This research adopted the same RF code but had different variants of RF executions GPU. Consequently, in this research, we expected no variations in accuracy, $F1$ and precision scores.

10 Conclusion

General-purpose graphics processing units are rapidly emerging as a promising solution to achieving high performance and energy efficiency in various computing domains. However, there are key bottlenecks in developing GPU-based algorithms (e.g., Random Forest) ranging from multiple forms of parallelism to complexity in memory access. This paper integrated coarse-grained and dynamic parallelism to parallelize RF trees on GPU (dpRFGPU). Experiment results showed that acceleration was evident when both dpRFGPU and parRFGPU were configured with more that 32 number of trees on low-dimensional datasets. dpRFGPU had a better acceleration than parRFGPU. These approaches accelerated RF and ultimately reduced time users take in training RF on GPU. Dynamic parallelism can be applied in parallelizing algorithms that are characterized by minimal diverging computations and data independence. Further studies need to be carried on memory analysis of dynamic parallelisms and applications on larger dimension datasets.

References

1. Kirk DB, Hwu WW (2010) Programming massive parallel processors. Elsevier Inc., eBook ISBN: 9780123814739
2. Zheng R, Hu Q, Jin H (2018) GPUPerfML: a performance analytical model based on decision tree for GPU architectures. In: The Proceedings of the 20th International Conference on High Performance Computing and Communications, IEEE. https://doi.org/10.1109/HPCC/SmartCity/DSS.2018.00110
3. Senagi K, Jouandeau N (2018) Confidence in Random Forest for performance optimization. In: Bramer M, Petridis M (eds) Artificial intelligence. XXXV SGAI 2018. Lecture notes in computer science, vol 11311. Springer, Cham. https://doi.org/10.1007/978-3-030-04191-5_31
4. Vouzis PD, Sahinidis NV (2011) GPU-BLAST: using graphics processors to accelerate protein sequence alignment. J Bioinf (Oxford England) 27(2):182–188. https://doi.org/10.1093/2Fbioinformatics/2Fbtq644
5. Breiman L (2001) Random Forests. Mach Learn 45(1):5–32. https://doi.org/10.1023/A:1010933404324
6. Zhang J, Wang H, Feng W (2017) cuBLASTP: fine-grained parallelization of protein sequence search on CPU+GPU". In: The Proceedings of IEE/ACM Transactions on Computational Biology and Bioinformatics, vol.14(4). https://doi.org/10.1109/TCBB.2015.2489662
7. Wang J, Rubin N, Sidelnik A, Yalamanchili S (2016) LaPerm: locality aware scheduler for dynamic parallelism on GPUs. In: The Proceeding of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), vol. 44(3), pp 583–595, IEEE. https://doi.org/10.1109/ISCA.2016.57
8. Rich C, Alexandru NM (2006) An empirical comparison of supervised learning algorithms. In: ICML '06 Proceedings of the 23rd International Conference on Machine learning, pp 161–168, ACM. https://doi.org/10.1145/1143844.1143865
9. Manuel FD, Eva C, Senen B (2014) Do we need hundreds of classifiers to solve real world classification problems? J Mach Learn Res 15:3133–3181
10. Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140
11. Nawar S, Mouazen AM (2017) Comparison between Random Forests, artificial neural networks and gradient boosted machines methods of on-line vis-NIR spectroscopy measurements of soil total nitrogen and total carbon. Sensors. https://doi.org/10.3390/s17102428
12. Lie C, Deng J, Cao K, Xiao Y, Ma L, Wang W, Ma T, Shu C (2018) A comparison of Random Forest and support vector machine approaches to predict coal spontaneous combustion in gob. ScienceDirect 239:297–311. https://doi.org/10.1016/j.fuel.2018.11.006
13. Wen Z, He B, Ramamohanarao K, Lu S, Shi J (2018) Efficient gradient boosted decision tree training on GPUs". In: The Proceedings of International Parallel and Distributed Processing Symposium, IEEE. https://doi.org/10.1109/IPDPS.2018.00033

14. Daga M, Nutter M (2012) Exploiting Coarse-grained parallelism in B+ tree Searches on an APU. In: The Proceedings of the SC Companion: High Performance Computing, Networking Storage and Analysis, USA, IEEE. https://doi.org/10.1109/SC.Companion.2012.40
15. Chen J, Li K, Tang Z, Bilal K, Yu S, Weng C, Li K (2017) A parallel Random Forest algorithm for big data in a spark cloud computing environment. IEEE Tran Parallel Distrib Syst 28(4):919–933. https://doi.org/10.1109/TPDS.2016.2603511
16. Genuer R, Poggi J, Tuleau-Malot C, Villa-Vialaneix N (2017) Random Forests for big data. Big Data Res 9:28–46. https://doi.org/10.1016/j.bdr.2017.07.003
17. Lo WT, Chang YS, Sheu RK, Chiu CC, Yuan SM (2014) CUDT: a CUDA based decision tree algorithm. Sci World J. https://doi.org/10.1155/2014/745640
18. Hughes C, Hughes T (2008) Professional multicore programming: design and implementation for C++ developers. Wiley Publishing, Inc,
19. NVIDIA Corporation. CUDA Toolkit. [Online]. https://developer.nvidia.com/cuda-toolkit. Date Accessed[April 2019]
20. Quinlan JR (1994) C4.5 programs for machine learning. Mach Learn 16:235–240
21. Rauber T, Rünger G (2010) Parallel programming for multicore and cluster systems. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-04818-0
22. LeBard DN, Levine BG, Mertmann P, Barr SA, Jusufi A, Sanders S, Klein ML, Panagiotopoulos AZ (2012) Self-assembly of coarse-grained ionic surfactants accelerated by graphics processing units. J Soft Matter. https://doi.org/10.1039/c1sm06787g
23. Nickolls J, Dally WJ (2010) The GPU computing Era. IEEE Micro. https://doi.org/10.1109/MM.2010.41
24. NVIDIA. [Online]. Available https://docs.nvidia.com/cuda/index.html. [Accessed: April 2019]
25. Barlas G (2015) Multicore and GPU programming an integrated approach. Elsevier Inc
26. Luo GH, Huang SK, Chang YS, Yuan SM (2013) A parallel bees algorithm implementation on GPU. Elsevier. https://doi.org/10.1016/j.sysarc.2013.09.007
27. Nasridinov A, Lee Y, Park YH (2013) Decision tree construction on GPU: ubiquitous parallel computing approach. Springer. https://doi.org/10.1007/s00607-013-0343-z
28. Lettich F, Lucchese C, Maria Nardini F, Orlando S, Perego R, Tonellotto N, Venturini R (2018) Parallel traversal of large ensembles of decision trees. IEEE. https://doi.org/10.1109/TPDS.2018.2860982
29. You Y, Zhang Z, Hsieh CJ, Demmel J, Keutzer K (2019) Fast deep neural network training on distributed systems and cloud TPUs. IEEE. https://doi.org/10.1109/TPDS.2019.2913833
30. Mahale K, Kanaskar S, Kapadnis P, Desale M, Walunj SM (2015) Acceleration of game tree search using GPGPU. In: The Proceedings of the International Conference on Green Computing and Internet of Things (ICGCIoT), IEEE. https://doi.org/10.1109/ICGCIoT.2015.7380525
31. Senagi K, Jouandeau N (2018) A non-deterministic strategy for searching optimal number of trees hyperparameter in Random Forest. In: Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS), IEEE. https://doi.org/10.15439/2018F202
32. Oshiro TP, Perez SJ, Baranauskas A (2012) How many trees in a Random Forest?. In: Proceedings of the International Workshop on Machine Learning and Data Mining in Pattern Recognition, Springer, Berlin, Heidelberg, pp 154–168, 2012. https://doi.org/10.1007/978-3-642-31537-413
33. Dua D, Taniskidou KE (2017) UCI machine learning repository. [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science
34. NVIDIA Corporation: Profiler user's guide. [online]. https://docs.nvidia.com/cuda/profiler-users-guide/#nvprof-overview. [Date Accessed: April 2019]
35. Senagi K, Jouandeau N and Kamoni P (2017) Using parallel Random Forest classifier in predicting land suitability for crop production. Journal of Agricultural Informatics 8(3), 23–32