Parallelisation of Random Forest Algorithms through GPU Computing

Student Name: C.F. Calder

Supervisor Name: S. McGough

Submitted as part of the degree of MEng Computer Science to the
Board of Examiners in the School of Engineering and Computing Sciences, Durham University

2nd April 2017

Abstract —

Background — Machine learning techniques continue to see frequent use due to their applications in data analysis. However, the training of machine learning models can take significant amounts of time, sometimes on the order of days. GPU computing has proved to be a powerful tool for solving computationally intense problems by providing massive parallelism on commonly available hardware. With success in deep learning, it is natural to try to apply GPU computing techniques to the broader field of machine learning.

Aims — This paper aims to explore the advantages and disadvantages of the use of GPU computing for machine learning techniques, in particular Random Forest models. By outlining potential improvements in execution time, possible pitfalls, and techniques and patterns which can be exploited when developing such solutions, we gain useful insights into how GPU computing can be applied to machine learning.

Method — Random Forest models are a common and well established machine learning technique, with many opportunities for parallelism. By implementing a random forest classifier using the NVIDIA CUDA toolkit, we gain a baseline for GPU computing to compare the speed of the model against CPU-based models. Perhaps more importantly, redesigning a classifier for execution on GPUs gives insight in to how other machine learning models may also be adapted to the GPU or other parallel environments.

Results — Performance tests show that industry standard models can be out-performed by implementing random forest classifiers on the GPU, with training speeds up to 6 times faster than scikit-learn. This implementation of random forests on the GPU shows very promising scaling as more trees are added to the forest, but also identifies that growing trees with greater depths is a particular issue for GPUs, likely due to the large branching factor of the algorithm when deep trees are used.

Conclusions — GPU computing provides a viable alternative to standard CPU based implementations of random forest algorithms. Taking in to account ideas presented in this paper and by CudaTree, another GPU random forest implementation, future development could result in implementations of GPU random forests that out-perform CPU implementations regardless of the complexity of the data it is trained on.

Keywords — Random Forest, Machine Learning, CUDA, GPU, Data Analysis

I INTRODUCTION

A Machine Learning

Machine learning algorithms are a class of algorithms that can learn to discern patterns in data without having the patterns explicitly programmed as part of the algorithm. These algorithms show incredible accuracy on even complex datasets; Patel and Kalyani (2016) achieved over 97% accuracy on the MNIST Handwritten Digit Dataset using Support Vector Machines, and similar results are achieved in a variety of experiments, including on genome sequencing tasks and in autonomous driving applications.

A major distinction made between types of machine learning algorithms is whether an algorithm is supervised or unsupervised. Supervised learning involves the recognition of patterns in data where the training data is labelled. The job of the algorithm is to learn from this labelled data, and to assign a label to unlabelled data. Supervised learning is again separated in to two categories - classification, where labels are discrete, and regression, where the labels are continuous. Examples of classification include deciding which of the digits from 0-9 are shown in an image, as in the MNIST Handwritten Digits dataset. Regression problems include predicting the temperature tomorrow, based on the weather on previous days. In unsupervised learning, models attempt to detect patterns between given samples, such as finding clusters in data, or finding anomalies in a dataset (Hastie et al. 2009).

As machine learning techniques have become more viable in recent years due to an increase in the quantity of data collected, and advances in computing power that are required for such techniques, data analysis is becoming a common task. While it is possible to analyse significantly larger quantities of data than ever before, the training of some machine learning models can take significant amounts of time - on the order of hours or days - to reach the accuracy required (Riedel et al. 2015). It is therefore necessary to find new approaches to implement these algorithms, aiming to reduce execution times as much as possible.

B Decision Trees

Decision trees are machine learning models backed by a binary tree structure. Data flows in from the root node, and at each child node follows either the left or right branch according to some decision criteria associated with the node and a feature of the data. Once data reaches a leaf node, it is assigned a label according to the labels of training data that also reached that node (Hastie et al. 2009).

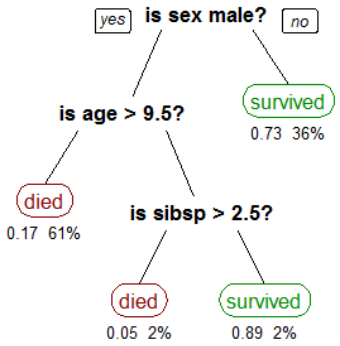

Figure 1: A Basic Decision Tree of Depth 3, trained on the Titanic Survival Dataset
Source: https://en.wikipedia.org/wiki/Decision_tree_learning

Suppose the following rows were to be classified by the decision tree outlined in Figure 1,

Feature	id	sex	age	sibsp
Value	1	male	5.5	2
	2	female	30.0	0

Passenger 1 would be classified by entering at the root node, and taking the left branch as their sex is male, taking the right branch of the following node as their age is ≤ 9.5, followed by the right branch again as their number of siblings present on the ship (sibsp) is less than 2.5. Passenger 1 will then be classified as a survivor as 89% of passengers in the training data that followed that path in the tree were survivors.

Passenger 2 would take the right branch from the root node and be classified as a survivor, as 73% of the passengers in the training data who were female survived.

C Random Forests

Random forest algorithms are a subset of supervised learning algorithms, which are typically used for classification, but can also be used for regression. Random forests are ensemble algorithms, taking many weak classifiers or regressors and combining them in to a single more powerful model. For random forests in particular, the 'weak classifiers' forming their basis are decision trees.

By constructing multiple decision trees on different subsets of either the data, features, or with some other variance, many classifications are generated according to different subsets of the data, which can be consolidated according to their confidence to result in a final classification or regression decided by the forest (Ho 1995). By utilising many subsets of the data, the model becomes less prone to over-fitting the training data, a common issue with many machine learning models (Hastie et al. 2009).

D GPU Computing

Graphical Processing Units (GPUs) were developed to render computer graphics, however in recent years General Purpose GPU (GPGPU) Computing, or just GPU Computing, has become

more popular. The paradigm has recently risen in popularity particularly in scientific computing and deep learning where it has proven to be a very effective technique for these tasks which require high-performance computing. GPU computing utilises the thousands of threads that can be run in parallel on GPU hardware to accelerate computation, and is best utilised by algorithms that can exploit this parallelism. With many simulation problems lending themselves to this kind of architecture, it has gained a large amount of popularity in the scientific research community (Neelima & Raghavendra 2010).

The advancement of GPU computing has lead to specialist hardware being built based around the paradigm, providing tens of thousands of compute cores in a single machine (NVIDIA 2016), however commodity hardware providing thousands of compute cores is available for low prices when compared to other specialised hardware, such as Field Programmable Gate Arrays (FPGAs). As affordable yet powerful technology, it is relatively easy to provide researchers with the potential to utilise tools developed around the GPU computing paradigm.

There are many ways to implement algorithms on a GPU. Originally programmers would utilise vertex and fragment shaders typically used in graphics rendering pipelines to process data, however with the advent of toolkits such as the NVIDIA CUDA Framework and OpenCL, it is possible to develop solutions using purpose-made tools (Neelima & Raghavendra 2010). Providing more accessible tools for development on these powerful systems means researchers and businesses can more easily utilise GPU computing as a tool for research and development.

E CUDA

NVIDIA's CUDA framework is a highly parallel computing architecture and API designed to run on GPUs. CUDA exposes up to thousands of compute cores which run in parallel on data which has been transferred to the device through the CUDA API.

Parallelism is handled at multiple levels in CUDA. From the standpoint of the API, the two levels of parallelism are block-level and thread-level. Each thread runs in parallel, with a group of threads forming a block. Each block runs in parallel independently of the other blocks, and the sum of all blocks forms a grid. Threads can share memory between threads in the same block, or can access the device's global memory (Cook 2013).

At the hardware level, an NVIDIA GPU has a number of streaming multiprocessors (SMs), usually between 10 and 20. Each streaming multiprocessor has many - usually around 128 - CUDA compute cores. Blocks are assigned to a single SM, whose threads divided in to groups of 32 - called a warp - which then run in parallel. The true power of CUDA comes from its ability to quickly switch context between warps, so that when one warp is waiting for data, another warp can be almost immediately swapped in to perform its calculations until it must wait for data itself. This results in significantly reducing memory latency, allowing the core to be better saturated with data and to perform at its maximal arithmetic intensity (Cook 2013).

F Purpose of this Project

This project aims to consider the question of 'Can we use GPU computing to design faster machine learning algorithms?'. It has been shown repeatedly that GPU computing is effective in scientific computing, and also in a subset of machine learning - deep learning (Liu et al. 2017). Research in to the application of GPU computing in the wider field of machine learning in general is limited, and exploring the joining of these two growing technologies could provide valuable

tools for future research.

Random forests are a common machine learning technique which are a natural fit for parallelisation (Liao et al. 2013) due to multiple independent classifiers forming the ensemble. CudaTree is the primary example of an implementation of random forests on the GPU, which was designed by Liao et al. Unfortunately, the project is no longer maintained, while GPU technology has advanced.

This paper evaluates the application of GPU computation to random forest algorithms in an attempt to further understand the implications of GPU computing in the field of machine learning outside of its typical use in deep learning.

With current GPU based machine learning implementations becoming outdated outside of deep learning, and with GPU availability increasing in workstations and clusters alike, it makes sense to revisit the application of CUDA to random forests and other machine learning algorithms in an effort to achieve faster data processing when utilising these algorithms.

The results from this project show that an random forest classifiers can perform better than the current industry standard implementations in some scenarios by utilising GPU computing to accelerate the training of the random forest model. This paper also investigates which aspects of GPU computing are best exploited when implementing these algorithms, and suggests directions for future implementations of GPU random forests to take to ensure they perform efficiently.

II RELATED WORK

A Random Forests

Random forest algorithms were first designed in 1995 by Tin Kam Ho (Ho 1995), and the idea has been well refined through research since it's inception due to their popularity as a 'good out of the box' classifier, requiring little tuning of hyper-parameters (Hastie et al. 2009). Their maturity is reinforced by their basis model, decision trees, also being mature algorithms that are well studied (Rokach & Maimon 2014).

Of particular importance to the construction of decision trees is calculating which feature and value is best to split the data set on at any given node. The problem of constructing an optimal tree is in NP-complete (Hyafil & Rivest 1976), and so research in to random forests often looks at how to compute these optimal, or close to optimal 'splits' efficiently. Comparing two potential splits is typically done using split metrics - measures of how well a given split separates the data set in to chunks which can be more easily classified. John Mingers gives a comprehensive comparison of common split selection measures in 'An Empirical Comparison of Selection Measures for Decision-Tree Induction'. Perhaps the most common of these are Gini Impurity and Information Gain, however the Extremely Random Trees (ExtraTrees) algorithm is also used when fast tree building is required at the potential loss of some accuracy (Geurts et al. 2006) by randomly selecting a subset of all potential splits to test.

The ExtraTrees algorithm can be used in conjunction with the other algorithms by utilising them as a 'Score' function to compare randomly selected splits, or it can be used to select each split randomly within the bounds of the data at each node.

B Deep Learning on GPUs

Graphics processing is based around linear algebra, and so GPUs are designed with vector and matrix operations in mind. Neural networks, whose underlying data structures can be modelled in this way (Oh & Jung 2004), are therefore a perfect fit for this hardware (Liu et al. 2017); despite using complex vertex shaders to implement their algorithms, Oh and Jung showed a 20 times speedup compared to corresponding CPU implementations of neural networks using their GPU neural network implementation. Liu et al. have shown that this large disparity in performance continues in recent research.

Google's TensorFlow framework is a recent development in deep leaning, released in 2015. It features GPU computing capabilities built on the CUDA framework, and its primary focus is as a way to model neural networks. While TensorFlow is a relatively low level framework, its python bindings provide a more easy-to-use interface than a low level C API, such as CUDA (Rampasek & Goldenberg 2016).

With the recent success of GPU computing for deep learning, a natural and important question is whether the rest of the machine learning field can learn anything from this success, and encourages us to look at how we can exploit similar techniques to develop faster machine learning models outside of deep learning by utilising GPU computing. As deep learning utilities become less difficult to implement on GPU hardware and GPUs become more integrated in to data analysis pipelines, providing effective implementations of other, more general machine learning algorithms on GPUs further expands the toolkit available to data analysts.

C Random Forests on the GPU

The idea of training random forests on GPU hardware has been visited previously, with attempts from Grahn et al. and Liao et al. in 2011 and 2013 respectively, both using the CUDA framework. Each team used very different approaches, likely due to the differences in available hardware.

Grahn et al. approached the problem by building one tree per CUDA thread. This made sense at the time; with their benchmarking hardware using a NVIDIA GeForce GT220 graphics card with 48 CUDA cores split between two streaming multiprocessors, the only other sensible approach would be to use all cores to build a single tree or risk over-complication with little to no performance gain.

Liao et al. present two approaches to tree building on the GPU. Their 'depth-first' approach utilises the full power of the GPU to calculate the characteristics of a single node, while their 'breadth-first' algorithm utilises one CUDA thread block per node to disperse the load of constructing multiple nodes across the GPU cores. Their final implementation - CudaTree - makes use of both of these tree building algorithms, building the first few layers of a tree using the depth-fist algorithm, and building lower layers using the breadth-first approach. CudaTree showed promising results, with up to 7x speed improvements over the implementation that was provided by scikit-learn in 2013. CudaTree's maintenance has since been discontinued. Revisiting training random forests on the GPU has provided insight in to how best to continue research in to machine learning and GPU technology, while utilising ideas put forward by Liao et al. that continue to be effective for GPU based random forests.

III SOLUTION

A Building Random Forests

The implementation designed during this project makes use of standard algorithms for building decision trees, however, as each algorithm has been designed for serial execution, some adaptations have been made with the GPU environment in mind. Consideration was also given to how tasks and data could be distributed to and executed on each streaming multiprocessor or thread. These details are discussed in the following section, along with their implications.

In this project, some focus was given to the ability to tune as many hyper-parameters as possible, so that it would be possible to study how each parameter influences the performance characteristics of the algorithm.

A.1 ExtraTrees

The ExtraTrees algorithm, when used in conjunction with another algorithm for scoring each split, provides a way to build trees without having to sample every possible split value for a given node. It allows split scoring using any algorithm, while significantly reducing run times in a way that Geurts et al. argue does not reduce the accuracy of the classifier. As one of the primary aims of this project is to attempt to reduce execution time, ExtraTrees is a good fit for this implementation.

The primary hyper-parameters of the ExtraTrees algorithm when used along with a scoring algorithm are the number of random features to check for a potential split in, and the number of random splits to test for each selected feature. These parameters are configurable in the implementation.

As each node is provided with multiple threads to compare splits with, this algorithm was parallelised in a way that attempted to saturate the thread pool as much as possible by spreading splits evenly across threads. As a trivial example, suppose there is a dataset with 2 features f_1 and f_2, for each of which 6 splits are considered, and that there are 4 CUDA threads dedicated to each node. The threads for a given node are distributed as follows:

Split	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$	$f_{1,6}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	$f_{2,4}$	$f_{2,5}$	$f_{2,6}$
Thread	t_1	t_2	t_3	t_4	t_1	t_2	t_3	t_4	t_1	t_2	t_3	t_4

Utilising the abundance of CUDA threads available, a tree reduction can be run in parallel to select the best split from a group of splits. This allows hundreds of splits to be compared as efficiently as possible, and means that the potential splits do not need to be copied back to the host device and back for the optimal split to be found.

A.2 Split Metrics

Information gain is generally considered a good metric for deciding upon which feature to split on (Mingers 1989). It does, however, suffer from some downsides. When a feature of a dataset can take on a large number of distinct values, this feature will have a high mutual information, as the feature is closer to uniquely identifying each sample in the dataset. Because of this, the information gain will be high when splitting on this feature, however this will tend to overfit the

data. As an example, suppose a forest is being trained on a dataset where each sample has a uniquely identifying feature. Splitting on this feature will give a high information gain due to the high mutual information in the variable, however, when unseen data is classified by the tree, its unique key is not encoded in the forest and these splits, while high scoring, will be near useless (Mitchell 1997).

Information gain also has an issue in that it makes heavy use of the log function. While on CPU implementations this may be less of an issue, on a GPU complex operations such as log are very expensive, and so it is best to avoid implementations making use of these types of operation.

Gini impurity is also considered an effective split measurement technique (Mingers 1989). As opposed to information gain, the Gini impurity measure does not make use of expensive operations such as log, and uses mostly summation and some divisions to calculate a split's score. Mingers also found that Gini impurity is a more effective measure to compare splits, and is the preferred measure in *The Elements of Statistical Learning* (Hastie et al. 2009) and by Liao et al., however this is disputed by Grahn et al. in *CudaRF: A CUDA-based Implementation of Random Forests*.

In this project, Gini impurity was used as the split metric, as evidence seems to support that it is a more dependable measure of a split's viability. Packages such as scikit-learn, against which the GPU implementation was tested, also use Gini impurity as a split comparison metric, providing more common ground between the implementations and ensuring that the focus is CPU vs GPU, and not specific implementation details. It would, however, be valuable to research if rather than adapting current split metrics for GPU computation, split metrics could be designed to exploit the GPU's parallelism in a way to outperform these metrics which are typically run serially. In this project, as the parallelism of the split selection algorithm is handled by parallelising the ExtraTrees algorithm, there was little parallelism that could be included in the Gini scoring itself.

A.3 Tree Pruning

To reduce over-fitting training data and to reduce model complexity, decision trees can use a procedure called pruning, which removes parts of the decision tree which have low classifying power (Mehta et al. 1995). In this project, pruning was not considered or implemented, as the focus was on execution speed; the dominant computationally intensive task in training decision trees is constructing the trees and calculating the splits at each node in the forest (Hastie et al. 2009) and the training times will become dominated by tree construction.

A.4 Data Structures

An important part of GPU Computing is considering how to best move data to and from the GPU, and how to create and manipulate the data structures on the device in a way that properly represents the data, while keeping memory management overhead to a minimum (Cook 2013).

A popular way to implement binary tree structures in C-like languages is to represent each node using a struct or object to represent a node, with a left and right pointer to other nodes, and with other information related to the node. As a very natural mapping of the data structure, tree-based algorithms are very straightforward to implement using this model. This has the benefit of taking up exactly as much memory is needed to represent the tree, however managing each

pointer and assigning the correct amount of memory as the tree grows quickly becomes complex when the tree becomes more sparse.

An alternative to this method is to use an array of node structs or objects with enough assigned memory to build a full binary tree, and to label some nodes as inactive. This has a few benefits over the 'struct-and-pointer' method; it provides good spacial proximity to help with caching, and it also makes the tree very predictable - it will take up $(2^h - 1) \cdot sizeof(node)$ bytes in memory for a tree of height h, and the index of a child nodes in the array is simply $left\_child\_id = 2 \cdot parent\_id + 1, right\_child\_id = 2 \cdot parent\_id + 2$, for a zero-indexed array.

In this project, the array based method was used due to its ease of use with regards to memory management. While this simplified the implementation, this design decision limits the tree depth due to its comparatively large memory footprint. As will be discussed later, this has a negative effect on tree accuracy, and future work will be needed to allow GPU random forest implementations to be used on truly large scale data.

A.5 Parallelising Tree Building

As each tree in a forest is independent of the others, multi-core computing gives a nearly free speed up by utilising multiple cores to build trees in parallel. Many popular CPU implementations of random forests make use of this, and in this project a similar approach was used for tree building. Due to the comparatively huge number of cores available in a GPU, however, it also made sense to seek other ways to exploit parallelism to further increase the speed of the implementation.

The approach used when designing the implementation was to attempt to map the problem to the hardware as closely as possible; as CUDA uses a hierarchical view of parallelism, the implementation mirrors this in its design. By approaching the problem in this way, it is possible to abstract away some of the complexities at each level when implementing functionalities at other levels.

Tree level parallelism — Each block is assigned to a streaming multiprocessor, and so mapping each tree to a block provides good parallelism immediately. Due to the way CUDA handles block loading, this is not exactly the same as utilising 20 CPU cores to build 20 trees in parallel; it is possible that warps from multiple blocks are in the SM's current pool of active warps, and the SM may therefore utilise its fast context switching to build multiple trees in parallel. If a similar procedure was to be implemented on a CPU, additionally complex threading mechanics would have to built in to the implementation, and due to the lack of near-instantaneous context switching on a CPU this would likely do nothing but add overhead to the implementation.

Data level parallelism — With many hundreds of threads being run in parallel on each streaming multiprocessor, this project explored ways to exploit this additional parallelism. The solution used implements data-level parallelism by utilising multiple threads to calculate the most optimal split value for each node, with each thread considering different features to potentially split on.

This is in contrast with the implementation of GPU random forests by Liao et al., where at the tree level each tree is built serially, and parallelism is only exhibited within each tree. This project attempts to take this parallelism a step further, utilising recent advancements in GPU computing which provide more CUDA cores and streaming multiprocessors per card.

B Testing

The implementation was tested using a variety of datasets - the Iris Dataset, MNIST Handwritted Digit Database, and the Spambase dataset. Details of the datasets are presented in Table 1 below.

Table 1: Datasets

Name	Features	Training Samples	Testing Samples	Samples	Classes
Iris	4	97	53	150	3
Spambase	57	1356	3245	4601	2
MNIST	784 (28 x 28 images)	60000	10000	70000	10

The Iris and Spambase datasets were not provided with a training set and testing set as MNIST was, and so they were divided in to approximately 70/30 train-test splits to be used for model evaluation.

By testing the implementation on a variety of datasets, it is possible to get a good idea of how the implementation performs in a variety of environments. As machine learning algorithms are used to analyse many types of data, coming in many shapes and sizes, it is important to understand how different algorithms and implementations perform under different circumstances so that data analysts have the information required to choose the most appropriate tool for the job.

In order to have a baseline against which to compare the model, each dataset was also used to benchmark the performance of scikit-learn, a popular analysis framework written in python and Cpython which is often used in data analysis pipelines. By having a representation of the industry standard, it is possible the establish how the GPU implementation would perform if it were to be used in place of scikit-learn for data analysis.

When evaluating machine learning algorithms, the main metrics to consider are the speed and accuracy of the model (Hastie et al. 2009). The training speed of a model is particularly important - in today's fast-paced world, models must be trained as quickly as possible for analysis results to be generated on time. In this project the main focus was on execution time, as commonly used random forest implementations such as scikit-learn are mature projects with a lot of optimisation with regards to accuracy through procedures such as tree pruning. While tree pruning adds some computational complexity, the time spent building random forests is predominantly used on deciding on the split criteria for each node - a problem known to be in NP-complete (Hyafil & Rivest 1976).

B.1 Testing Setup

The tests were run on a headless Linux workstation running Ubuntu 16.10 with CUDA Toolkit 8.0.44. The test machine ran on the following hardware.

Table 2: Test Hardware

Processor	Properties
CPU	i7-950 @ 3.33 GHz 6GB DDR3 RAM
GPU	NVIDIA GTX 1080 8GB GDDR5X RAM 2560 CUDA Cores @ 1607 GHz 20 Streaming Multiprocessors

To ensure rigorous testing, each test was run at least 3 times, and the results averaged.

B.2 Cross-validation

A common way to evaluate a model in data analysis is to use k-fold cross validation. A set of training data is split in to k random 'folds' and the model is trained k times, each time using a different fold as the testing data, and the remaining folds as the training data for the model. This way, it is far less likely for a model to be trained on a lucky train-test split, and anomalous results are identified as there are multiple runs of the model on different data sets (Hastie et al. 2009).

Random forests do not need to be evaluated using cross-validation, however, as each tree in the ensemble is trained on a subset of the data as it would have been in cross-validation, further fragmenting that data would do nothing to assist in evaluating the accuracy of the model.

B.3 CudaTree

The current standard implementation for random forests on the GPU is CudaTree, and so it would be useful to compare the performance of our model against this. Unfortunately, this is made difficult as the CudaTree implementation provides little availability for tuning hyper-parameters of the model, in particular, the tree depth cannot be changed. It is, however, possible specify the number of trees built by CudaTree, and so we can look at how CudaTree scales horizontally in comparison to the implementation built for this project.

In performance tests where a specific tree depth is used, the tree depths of the other models are selected such that the accuracy of the models are comparable, implying that the underlying classifiers are of similar strength and therefore have similar tree depths.

IV RESULTS

There are a number of interesting trends that appear when analysing the performance characteristics of the implementation. For shallow trees, such as those required for the Iris dataset, GPU random forests show very good horizontal scaling - increasing the number of trees does not significantly increase the amount of time taken when compared to scikit-learn or CudaTree.

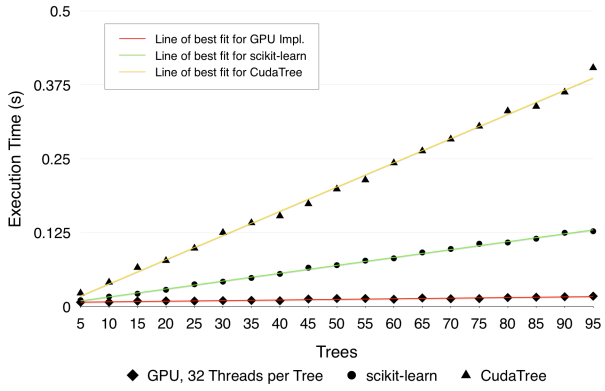

Figure 2: Execution Time (s) vs Tree Count on the Iris Dataset, Tree Depth 3

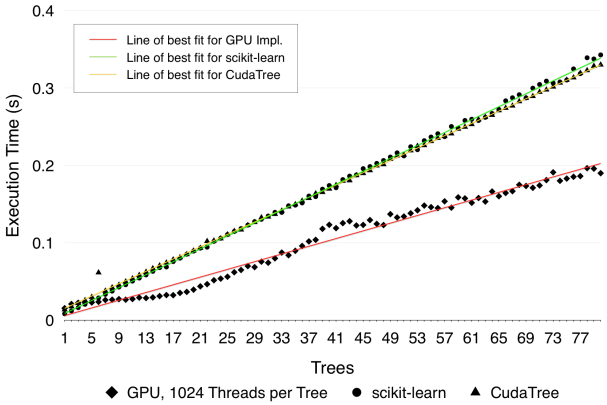

Figure 3: Execution Time (s) vs Tree Count on the Spambase Dataset

Figure 2 shows that for shallow trees and on small datasets, the implementation out-performs scikit-learn and CudaTree on a small number of trees, and out-performs both to a considerable degree at high tree counts.

For this test, the tree depth was 3, and 32 CUDA threads were used to build each tree in the CUDA implementation.

Similarly in Figure 3, where each implementation was run on a larger dataset but again with shallow trees, the new GPU implementation shows strong horizontal scaling, but in this case is slightly out-performed at low tree counts by both scikit-learn and CudaTree.

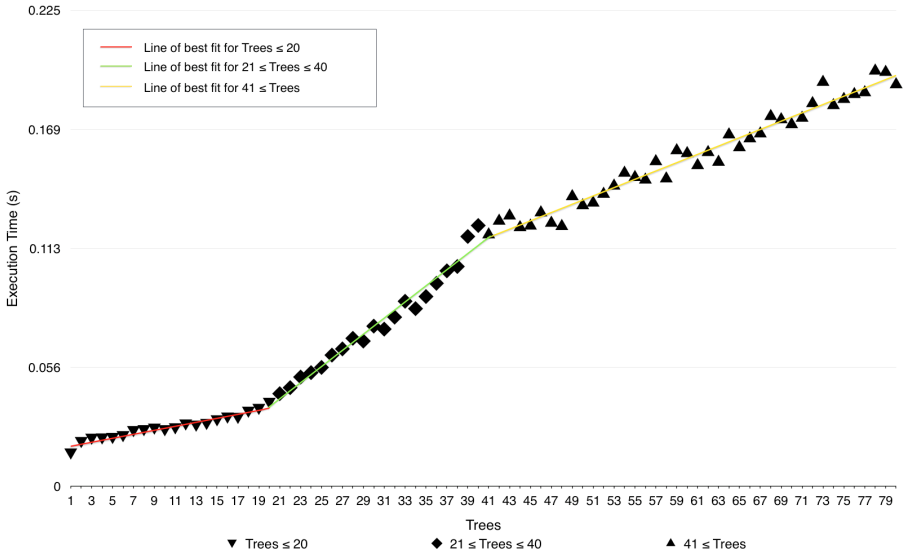

Figure 4: Execution Time (s) vs Tree Count on the Spambase Dataset

Perhaps more interesting as the dataset size increases, is that a pattern emerges in the execution time of the new GPU implementation. As seen in Figure 4, three distinct patterns emerge in the execution times as thresholds appear to be crossed, causing different runtime characteristics as the model scales.

As the tree count begins to increase, there are overheads associated with building an extra tree and so there is a steady increase in execution time. Once the tree count reaches 20, however, each of the 20 streaming multiprocessors on the GTX 1080 graphics card are saturated when constructing the forest, and so the scaling characteristics change; the card is less able to balance its workload by simply utilising another free streaming multiprocessor. At around 40 trees, another threshold is crossed and the horizontal scaling capabilities of the classifier change again, likely due to the SMs being even more heavily saturated to the point where the warp scheduling efficiency increases. The reason behind this second threshold is unclear however, as NVIDIA do not publish details of the CUDA warp scheduler.

In the Spambase tests, a tree depth of 2 was used, and 1024 CUDA threads were used per tree in the GPU implementation.

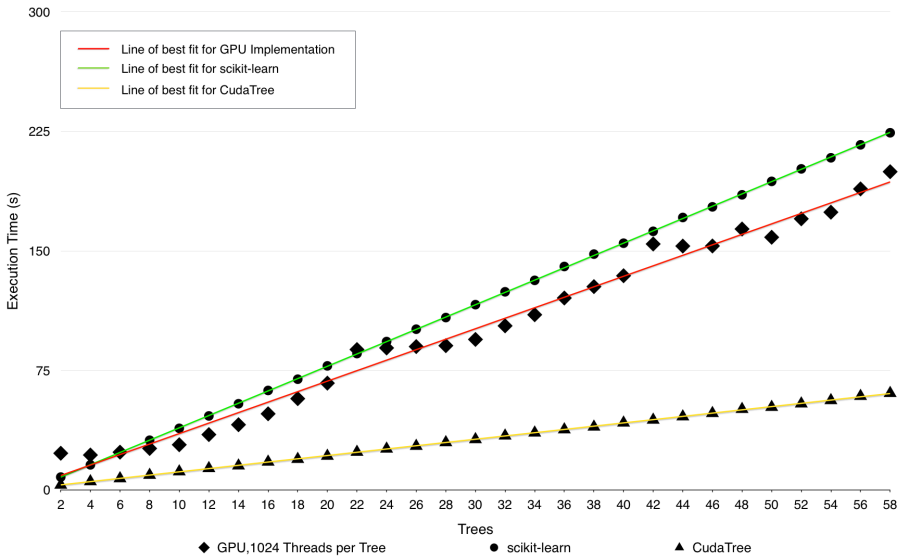

Figure 5: Execution Time (s) vs Tree Count on the MNIST Handwritten Digits Database, Tree Depth 7

In the MNIST tests, scikit-learn and this project's GPU random forest implementation used trees of depth 7, which provided reasonable accuracy. The GPU implementation used 1024 threads per tree when constructing the forest.

From Figure 5, it becomes clear that the GPU implementation begins to lose its speed when compared to scikit-learn as datasets become more complex, and the efficiency of CudaTree for deep trees puts CudaTree in to the lead in terms of execution time, and also accuracy. The project's GPU implementation continues to show superior horizontal scaling compared to scikit-learn, overtaking it in terms of speed once the tree count becomes larger than around 7 trees per forest.

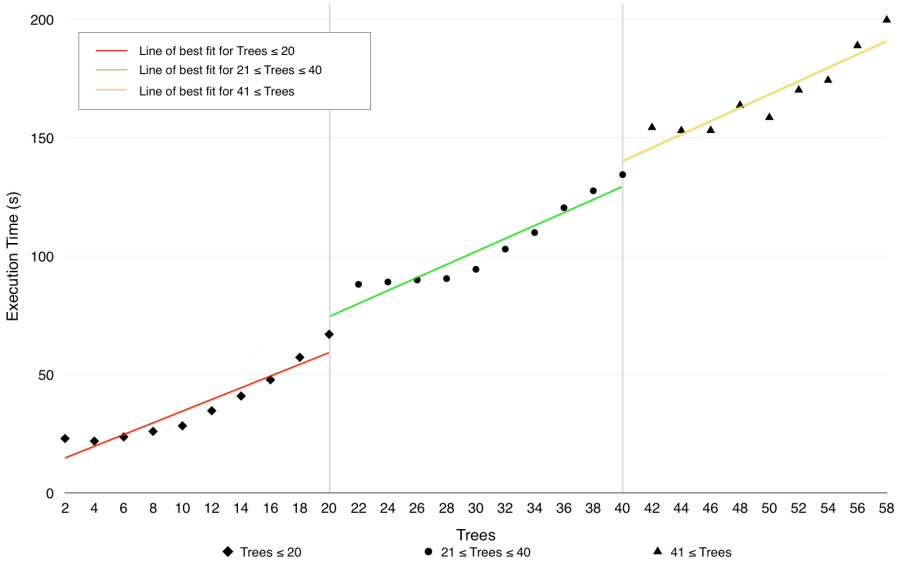

Figure 6: Execution Time (s) vs Tree Count on the MNIST Handwritten Digits Database, Tree Depth 7

Similarly to in Figure 4, training times on the MNIST Database show cyclic pattern in their scaling behaviour, as shown by Figure 6. As with the Spambase data, this pattern appears as the tree count reaches multiples of 20, however, the jump which occurs as the streaming multiprocessors are saturated is much more pronounced, as the overheads of building additional trees become more negligible compared to the build time for the trees.

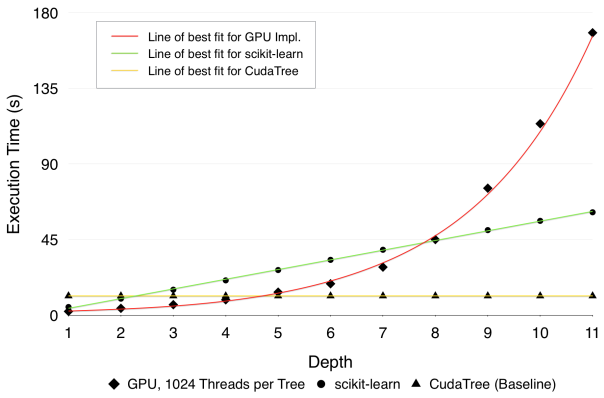

Figure 7: Execution Time (s) vs Tree Depth on the MNIST Handwritten Digits Database, 10 Trees

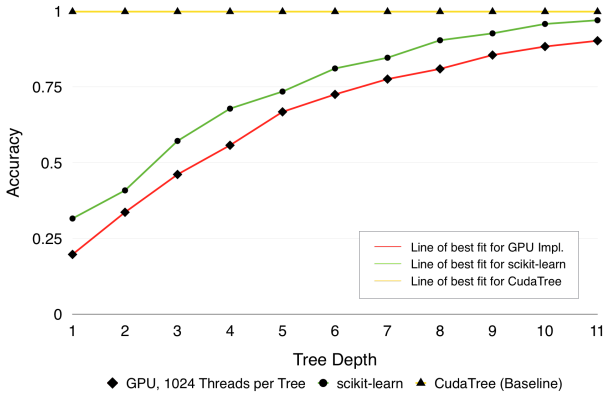

Figure 8: Classification Accuracy vs Tree Depth on the MNIST Handwritten Digits Database, 10 Trees

With regards to tree depth, this is where the implementation chosen begins to fall behind in comparison to other algorithms. As the depth of the trees increases, execution time increases exponentially, compared to scikit-learn's linear execution times. This can be seen in Figure 7.

The exponential nature of the runtimes are a direct result of exponential growth of the decision tree. No design decisions were made in an attempt to mitigate this - in future implementations, a better study in to how to avoid this issue would be valuable, and could allow GPU random

forests to be competitive with deep trees; indeed, CudaTree seems to have achieved this to a high degree, and their 'breadth-first' approach to building trees, which is utilised when building deep trees, shows a lot of success in these scenarios.

Figure 8 shows that the implementation approaches higher levels of accuracy at a similar rate to scikit-learn. It could be speculated that through pruning the trees, a higher level of accuracy could be achieved, as suggested by Mehta et al. and Hastie et al., which may help to 'close the gap' between this implementation and scikit-learn.

Table 3: Maximum Achieved Accuracies

Dataset	Implementation	Training Time (s)	Accuracy (%)
Iris	GPU	0.0072	96.67
	scikit-learn	0.011	97.98
	CudaTree	0.041	94.23
Spambase	GPU	0.13	99.41
	scikit-learn	0.20	100.00
	CudaTree	0.19	100.00
MNIST	GPU	248.69	90.22
	scikit-learn	61.76	97.24
	CudaTree	13.68	99.84

On the Spambase and Iris datasets, the GPU implementation achieved good accuracy, and did so with less training time than both scikit-learn and CudaTree. On the MNIST dataset, as discussed above, both the accuracy and execution speed of the implementation begin to fail due to the depth of the trees required on such a complex dataset, and partially due to the lack of pruning of the trees.

V EVALUATION

This project has provided insights in to how random forest algorithms can be implemented on GPUs, and how they perform in GPU computing environments. Keeping the latest developments in GPU hardware in mind, a new approach was investigated, taking a different approach to previous explorations of GPU random forests. The implementation was tested against a variety of datasets, and compared to the current industry standards for both GPU random forests and CPU-based implementations, CudaTree and scikit-learn. This provided context as to where this new GPU implementation fits in to the landscape of available packages, in particular the performance characteristics of the new implementation compared to these industry standards. The properties of how the implementation scales with regards to tree depth and tree count were also investigated, and provide insight in to how the design decisions made affect the performance of the model as it scales.

The results above show that the new GPU implementation outperforms both scikit-learn and CudaTree when the trees are shallow. This is sufficient for simple datasets, such as the Iris and Spambase dataset, which were used to test the performance characteristics of the models. For more complex datasets, however, the implementation begins to show its limitations due to the requirement for deeper trees in the forest. For these more complex datasets, CudaTree shows its strength on complex data, outperforming scikit-learn and the new GPU implementation.

A particular strength of this project's implementation is its ability for horizontal scaling - increasing the number of trees in the forest has a much lesser affect on the training times of the model compared to scikit-learn and CudaTree. This is particularly evident in the Spambase and MNIST tests, where a small number of trees takes longer to train than scikit-learn's implementation, however above around 5-7 trees, the GPU implementation begins to build the forest in less time than scikit-learn.

As the basis for this project was the random forest classifier, which has been well researched in serial environments, little work in this project focused on the development of new algorithms. The ExtraTrees and Gini Impurity algorithms provided a good, commonly used basis for the algorithm, which were adapted to a parallel environment for the purpose of studying their performance characteristics. Additional work could have been done to improve the accuracy of the classifier, such as implementing a tree-pruning algorithm which reduces over-fitting the training data, a common problem for machine learning models (Hastie et al. 2009).

The implementation was developed with a agile methodologies in mind. Development took a test-driven approach, where tests were written with expected outputs given certain inputs, and code was written to pass those tests. The project's development was also documented in a github repository, where each change and addition to the code can be seen in chronological order, and the full implementation's source code is presented in an open-source, free way.

VI CONCLUSIONS

This paper presents an implementation of random forest classifiers on the GPU platform which outperforms the industry standard CPU implementation scikit-learn when building wide, shallow forests. The investigation of the performance characteristics also show that CudaTree, a previous attempt at GPU random forests, can outperform scikit-learn on more complex data. In future implementations, it may be possible to combine ideas from both the implementation presented in this paper, and from CudaTree to create a GPU based implementation of random forests that outperforms CPU based implementations on all data. This would further push the machine learning field towards utilising GPUs for computation, as has been the case with deep learning (Liu et al. 2017).

This paper also provides a study in to how GPU environments are affected by different algorithms. The high level of branching associated with decision trees at lower levels was a major pitfall for this implementation, however Liao et al. appear to have solved this issue by providing an efficient classifier for complex datasets. The combination of tree- and data-level parallelism implemented in this project, on the other hand, produces excellent horizontal scaling properties and exploit the GPU platform well.

This research has shown that continuing to research GPU computing in the context of machine learning outside of deep learning is still valuable, and that neural networks are not the only machine learning models that can benefit from this powerful technology. Development will need to continue, and mature projects will have to be established before GPU based models can truly replace serial implementations, however this paper and others, such as Liao et al.'s *Learning Random Forests on the GPU*, show that alternatives to CPU based computation are viable, and should continue to be researched and developed in order to build the best possible machine learning models in the shortest amount of time.

References

Cook, S. (2013), *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st edn, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Geurts, P., Ernst, D. & Wehenkel, L. (2006), 'Extremely randomized trees', *Machine Learning* **63**(1), 3–42.

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

Hastie, T., Tibshirani, R. & Friedman, J. (2009), *The Elements of Statistical Learning*, 2nd edn, Springer.

Ho, T. K. (1995), Random decision forests, *in* 'Proceedings of 3rd International Conference on Document Analysis and Recognition', Vol. 1, pp. 278–282 vol.1.

Hyafil, L. & Rivest, R. L. (1976), 'Constructing optimal binary decision trees is NP-complete', *Information Processing Letters* **5**(1), 15 – 17.

Liao, Y., Rubinsteyn, A., Power, R. & Li, J. (2013), Learning random forests on the GPU. Available at http://news.cs.nyu.edu/~jinyang/pub/biglearning13.pdf (Last accessed 15th April 2017).

Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y. & Alsaadi, F. E. (2017), 'A survey of deep neural network architectures and their applications', *Neurocomputing* **234**, 11 – 26.

Mehta, M., Rissanen, J. & Agrawal, R. (1995), MDL-based decision tree pruning, *in* 'Proceedings of the First International Conference on Knowledge Discovery and Data Mining', KDD'95, AAAI Press, pp. 216–221.

Mingers, J. (1989), 'An empirical comparison of selection measures for decision-tree induction', *Machine Learning* **3**(4), 319–342.

Mitchell, T. M. (1997), *Machine Learning*, 1 edn, McGraw-Hill, Inc.

Neelima, B. & Raghavendra, P. S. (2010), Recent trends in software and hardware for GPGPU computing: A comprehensive survey, *in* '2010 5th International Conference on Industrial and Information Systems', pp. 319–324.

NVIDIA (2016), The worlds first AI supercomputer in a box, Technical report, NVIDIA. Available at https://images.nvidia.com/content/technologies/deep-learning/pdf/61681-DB2-Launch-Datasheet-Deep-Learning-Letter-WEB.pdf (Last accessed 14th April 2017).

Oh, K.-S. & Jung, K. (2004), 'GPU implementation of neural networks', *Pattern Recognition* **37**(6), 1311 – 1314.

Patel, A. & Kalyani, T. V. (2016), 'Support vector machine with inverse fringe as feature for MNIST dataset', pp. 123–126.

Rampasek, L. & Goldenberg, A. (2016), 'Tensorflow: Biologys gateway to deep learning?', *Cell Systems* **2**(1), 12 – 14.

Riedel, M., Goetz, M., Richerzhagen, M., Glock, P., Bodenstein, C., Memon, A. S. & Memon, M. S. (2015), Scalable and parallel machine learning algorithms for statistical data mining - practice & experience, *in* '2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)', pp. 204–209.

Rokach, L. & Maimon, O. (2014), *Data Mining With Decision Trees: Theory and Applications*, 2nd edn, World Scientific Publishing Co., Inc.