University of the Witwatersrand

School of Computer Science and Applied Mathematics
Faculty of Science

Honours Research Report

GPU-Based Parallel Machine Learning Algorithms in Automated Machine Learning Frameworks

A.E. Bank

Supervisor(s): Dr. Hairong Wang, Dr. Matthew Woolway, Prof. Turgay Celik

October 2019, Johannesburg

Declaration
University of the Witwatersrand, Johannesburg
School of Computer Science and Applied Mathematics
SENATE PLAGIARISM POLICY

I, A.E. Bank, (Student number: 604124) am a student registered for COMS4059A in the year 2019.

I hereby declare the following:

- I am aware that plagiarism (the use of someone else's work without their permission and/or without acknowledging the original source) is wrong.
- I confirm that ALL the work submitted for assessment for the above course is my own unaided work except where I have explicitly indicated otherwise.
- I have followed the required conventions in referencing the thoughts and ideas of others.
- I understand that the University of the Witwatersrand may take disciplinary action against me if there is a belief that this in not my own unaided work or that I have failed to acknowledge the source of the ideas or words in my writing.

Signature:

Signed on ________________ day of ________________ , 2019 in Johannesburg.

Abstract

Recent advances in various optimisation procedures have allowed the rapid growth of the fledgling field of automated machine learning, leading to various closed and open source AutoML implementations. This is a computationally expensive exercise (exploration of the search space). We extend the ATM platform with GPU-based machine learning algorithms to improve the efficiency of using such an AutoML platform. In particular, we implement GPU versions for SVM, RF and k-NN classifiers, currently available in ATM. Additionally, we implement a method for running ATM on a GPU enabled high performance computing (HPC) cluster. The extended platform is evaluated by benchmarking test datasets against the stock system, on an HPC cluster, with total runtime being the test metric.

Acknowledgements

Thank you to my grandmother, Nati, for providing unwavering love and support throughout my academic career. I would like to thank my partner Niambh Blundell, for her love and support, without which this work would not be possible. I would also like to thank Mr. Shunmuga Pillay for organising a dedicated GPU cluster for this research. Finally, thank you to Dr. Hairong Wang for your guidance and encouragement throughout this journey.

Avi Bank

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The popularity of machine learning (ML) has led to a vast amount of approaches to learning a model. Techniques such as support-vector machines (SVMs), random forests (RFs) and k-nearest neighbours (k-NN), all requiring their own set of hyperparameters, creates a large search space of approaches. Automated machine learning (AutoML) is a method for finding the best ML algorithm and corresponding tuned hyperparameters for a given dataset, without human intervention.

This is important for both ML experts and non-experts alike. For example, a medical researcher using machine learning on their dataset may not be familiar with less popular ML techniques that may in fact be best suited for their dataset. For ML experts, AutoML shifts the focus of the researcher to the feature engineering aspect of the ML pipeline, rather than spending a large amount of time trying to find the best algorithm or hyperparameters for a given dataset. As datasets grow larger with ML's growing prominence in various economic and academic sectors, learning models will require more computational power. This is especially true for AutoML systems. Although there have been many advances in solving the search space problem, all AutoML approaches must still test a number of different methods first before deciding which best suits the given dataset. This task is computationally expensive and so any improvement in search efficiency is valued.

Research in graphics processing unit (GPU)-based versions of ML algorithms shows significant performance improvements over serial or CPU-based implementations.

The research question is therefore whether implementing GPU-based ML algorithms in an AutoML platform will improve the performance and efficiency of that platform. In particular, we will investigate if the use of these algorithms adds any significant performance to an already distributed AutoML system when using a high performance computing (HPC) cluster with GPU enables nodes.

1.1 Organisation of This Report

In this research we will attempt to integrate two existing state-of-the-art GPU ML libraries in the an AutoML framework. Additionally, we implement a simple k-NN GPU implementation. Chapter 2 will introduce these libraries and discuss important concepts for the underlying ML algorithms and their GPU implementations. We also discuss AutoML in more detail and, importantly, motivate our choice of AutoML framework in Sections 2.2 and 2.2.1 respectively. This leads to our formally stated research hypothesis in Section 3.2 of Chapter 3. In this chapter we also detail the process used to integrate the GPU algorithms in the AutoML framework. In

Chapter 4 we present our run time results and the experimental setup we used. We interpret the results and come to a conclusion about our research hypothesis. We finish the paper in Chapter 5.

Chapter 2

Background and Related Work

2.1 Introduction

In the introductory chapter, we described the concept of AutoML, it's relevance and the motivation for improving it's performance. In particular, we motivated the need for improving the speed of the AutoML process when it comes to building models.

This chapter will attempt to explain the AutoML process more in depth in Section 2.2, focusing on aspects relevant to our research. We discuss the three machine learning algorithms used in our research as well as approaches to their parallel implementations in Sections 2.3 (support vector machines), 2.4 (random forests) and 2.5 (k-nearest neighbours).

2.2 AutoML

Several approaches to implementing AutoML systems are proposed in the literature. Each system usually takes a novel approach to searching the method and hyperparameter space but does little in the way of optimising the actual ML algorithms. Most of the time, an existing library such as `scikit-learn` forms the basis for creating the AutoML pipelines. Although this has its advantages, such as being ubiquitous in the ML community, there are other implementations of these popular algorithms that are far superior to the stock library implementations. In the next section, we introduce the Auto-Tuned Models system and motivate its use in our work.

2.2.1 Auto-Tuned Models

Swearingen et al. [2017] propose an AutoML system unique amongst others in the literature: Auto-Tuned Models (ATM). ATM offers a distributed computing platform which allows for simultaneous execution of multiple methods on an HPC cluster. This significantly improves speed of computation. Although ATM is not considered in Balaji and Allen [2018] and Truong et al. [2019], both of these studies provide detailed summaries of prominent AutoML frameworks currently available. When deciding on ATM, we considered many of the same criteria as those authors. Primarily, the extensibility of and documentation for ATM influenced our decision to use it in this study. Furthermore, since ATM does not yet have any GPU implementations, our work would be a more significant contribution this framework than one which already provides a GPU library.

The Auto-Net, Auto-sklearn and Posh Auto-sklearn AutoML systems were all designed to win AutoML competitions, and while they largely succeeded in that respect, this is not as important to our work as a system that has been designed with both the end-user and researcher in mind [Mendoza et al. 2016; Feurer et al. 2015 2018].

In ATM, different models are trained simultaneously on a dataset, or multiple datasets, using distributed computing. This is achieved by using a central database called the *ModelHub*, to store the performance of models. A single AutoML process for a certain dataset, testing certain methods, with certain hyperparameters is called *datarun*.The ModelHub stores and organises results from different dataruns.

2.3 Support Vector Machines

Support vector machines (SVMs) are a popular form of classical supervised machine learning algorithms. A survey conducted by Lu et al. [2014] shows that considerable effort has been invested in the optimisation of SVMs, in both a mathematical and computational context. The computational advances usually follow and implement those made in the mathematical literature. Formally, we describe the SVM as follows: For a given training data set X, each training point x_i has a label $y_i \in \{+1, -1\}$, where a positively (negatively) labelled training point corresponds to y_i =+1 (-1). The SVM determines a hyperplane separating the positive and negative training points in the feature space such that:

- the margin of separation is maximised.
- the classification error is simultaneously minimised.

It follows from the above definition that training an SVM is equivalent to solving an optimisation problem. This optimisation problem is known as the *primal form* of SVM training.

$$\underset{\boldsymbol{w},\boldsymbol{\xi},b}{\operatorname{argmin}} \quad \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i \tag{2.1}$$

$$\text{subject to:} \qquad \begin{aligned} y_i\left(\boldsymbol{w}\cdot\boldsymbol{x}_i + b\right) &\geq 1 - \xi_i, \\ \xi_i \geq 0, \forall i &\in \{1,\ldots,n\} \end{aligned}$$

where $\boldsymbol{w}$ is the normal vector of the hyperplane, C is the regularisation constant, ξ_i is a slack variable, and b is the bias of the hyperplane. The purpose of the slack variables is to account for some training instances falling on the wrong side of the hyperplane.[Wen et al. 2018]

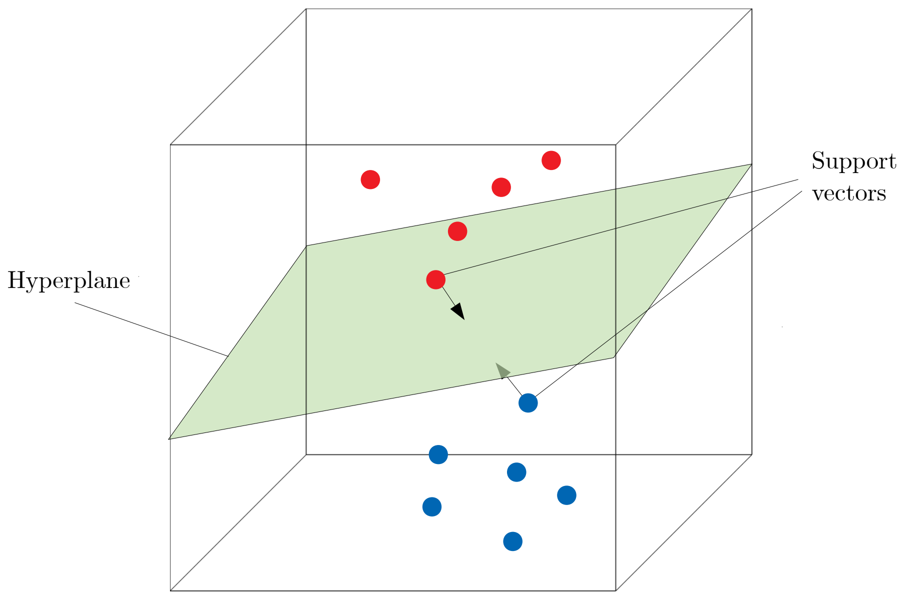


Figure 2.1: Diagram of a support-vector machine for a dataset with two classes and 3 features.

2.3.1 Parallel Approaches to SVM

Wen et al. [2018] recently developed a method for parallelising SVMs on GPUs, packaged as ThunderSVM (TSVM). In the results of this paper, TSVM is shown to significantly outperform similar methods in terms of speed. In terms of accuracy, the results show TSVM yields identical accuracy to similar methods. Both CUDA and OpenMP are used for parallelisation. We chose to use the TSVM library based on these results. We summarise the parallelisation techniques the authors used in Section 3.5.

2.4 Random Forests

The decision tree classifier is a simple and effective ML algorithm. However, decision trees suffer from overfitting as a result of high variance, particularly for deep trees.[1] The random forest technique attempts to address and improve these issues. In this section, we develop the fundamental concepts in subsections 2.4.1 and 2.4.2, culminating in a description of the random forest classifier in subsection 2.4.3.

[1]A deep tree simply refers to the height of the decision tree being large.

2.4.1 Bootstrapping

In the context of random forests, bootstrapping refers to creating multiple training datasets by choosing a uniform number of random samples with replacement, from a given training dataset. This method of generating datasets ensures none of them are (significantly) correlated. Correlation among these datasets would negate the desired effect of lowering the variance for the final model.

2.4.2 Bagging

Bagging is a method of aggregating the outputs of multiple models into a single model. Each of these models are trained on datasets generated by the bootstrap sampling procedure, described in the previous paragraph. In doing so, the final model is shown to have less variance, making it more robust than a single model. [Breiman 1994]

2.4.3 Random Forest Classifier

The random forest classifier uses a decision tree as its base model in a bagging scheme, as described in subsection 2.4.2. Additionally, random forests use a bagging scheme during construction of the individual trees, where at each split point (tree node), a random subset of features is selected. This further ensures the individual models will not become correlated. Formally, we produce a random forest for a given dataset as follows:

1. Given training data $X = \{x_I, \dots, x_n\}$, with labels $Y = \{y_1, \dots, y_n\}$:
2. Choose the number of trees in the random forest to be some $\epsilon \in \mathbb{Z}+$.
3. (*Bagging*) For $i = 1 \dots \epsilon$:
    (a) Generate a new training set from the original training set, using bootstrap sampling.
    (b) Train a decision tree using the generated training set, using feature bagging.

To classify a new data point, we run it through the trees in the random forest, created during the learning phase. The new point is classified by selecting the class that the majority of trees predicted (voting).

2.4.4 Parallel Approaches to RF

Wen et al. [2019], the same authors of ThunderSVM, recently released a preprint of their latest work which implements a CUDA-based random forest algorithm, ThunderGBM (TGBM). TGBM utilises sparse data representation to significantly reduce the memory required to store datasets. Prediction is implemented using a

GPU-based tree traversal. The experimental results of TGBM show marginal to no speed improvements over similar methods. However, the results also show that TGBM can handle some datasets that similar methods cannot. The parallelisation techniques of TGBM are summarised in Section 3.6.

2.5 k-Nearest Neighbours

The k-NN classifier takes an intuitive approach to point-based learning. For a dataset with n features, we map all points in the dataset to the n-dimensional space, $\mathbb{R}^n$. The simplest implementation of this algorithm uses the k-shortest Euclidean distances in the dataset from a query point.

$$d\left(x_i, x_j\right) \equiv \sqrt{\sum_{r=1}^{n}\left(f_r\left(x_i\right) - f_r\left(x_j\right)\right)^2} \tag{2.2}$$

In 2.2, $f_r\left(x_i\right)$ is the value of the rth feature for the x_ith data point. $d\left(x_i, x_j\right)$ is the distance (Euclidean in this case) between two points x_i and x_j. The basic k-NN algorithm will run this calculation for a query point against every other point in the data set. The distance metric need not necessarily be Euclidean. Any Minkowski metric would be suitable, as we demonstrate in our experiments, where we use the Manhattan and Euclidean variations of the Minkowski metric. By keeping the points in the training dataset constant, and querying all possible points in the testing dataset, the k-NN algorithm produces a Voronoi Diagram of training data. This is a decision surface over the complete space of the dataset [Mitchell 1997].

2.5.1 Parallel Approaches to k-NN

Garcia et al. [2010] developed an efficient and general purpose GPU implementation for k-NN. The authors intend for the code to be adapted to specific devices which has lead to the implementation ageing well alongside the rapid advances of GPU technology. The authors have thus maintained their sourcecode for the last nine years, making it a suitable starting point for our implementation. The approach parallelises the sorting and distance calculations in the k-NN algorithm. Choy [2017] slightly modified the implementation by Garcia et al. [2010] and added a wrapper for using the algorithm in Python. We base our implementation on Choy [2017] but experiment using PyCuda as opposed to pure CUDA C.

2.6 Related Work

The H2O.ai [2019] system provides a GPU library for its AutoML framework. This library, named H2O4GPU, currently provides GPU implementations for k-Means, gradient boosting machine, singular value decomposition and principal components analysis. The authors have not yet implemented support-vector machine nor k-nearest neighbours algorithms. Similar to our work, the library provides implementations with a similar API to `scikit-learn`. However in our work, when a certain `scikit-learn` method is required by ATM that does not exist in a GPU implementation, we create it using our own approach and utilise the GPU for the new method when possible. In contrast, H2O4GPU simply falls back to the standard `scikit-learn` method if it has not been implemented in the GPU algorithm.

2.7 Conclusion

In this chapter we explained the background and related research necessary for completing this project. We have described the three ML algorithms and approaches to their GPU implementations. Namely, support-vector machines, random forests and k-nearest neighbours. We also introduced the ATM AutoML system and motivated our decision to use it as a basis for testing GPU based algorithms in an AutoML framework. Although our initial impression of ATM is positive, its shortcomings are later revealed in Chapters 3 and 4 .

Chapter 3

Research Methodology

3.1 Introduction

We now present our implementations of the GPU-versions of the SVM, RF and k-NN algorithms in the ATM framework. Since ATM currently only supports classification tasks, we restrict our implementations to only provide the classification versions of the algorithms and omit the regression versions.

In Section 3.2 we formally present our research hypothesis. We then describe how we extended the distributed computing aspect of ATM to run over an HPC cluster in Section 3.3. Section 3.4 describes the general process of adding an algorithm to ATM. Following that, we discuss the GPU implementations of SVMs, random forests and k-NN in Sections 3.5, 3.6 and 3.7, as well as how we integrated them with ATM.

3.2 Research Hypothesis

Using GPU-based ML algorithms in an AutoML platform will improve the performance and efficiency of that platform and this result will scale when the approach is deployed on an HPC cluster. In particular, using GPU-based versions of SVM, RF and k-NN will improve the performance of the ATM system as compared to using serial versions of those three algorithms.

3.3 ATM on an HPC Cluster

Although ATM is built for distributed computing, it does not provide a method for running in a distributed manner on an HPC cluster. The interface provided by ATM is only able to spawn multiple threads on a single local machine. We have therefore designed a tool that allows a user to run ATM on an HPC cluster. We describe the design of this tool in this section.

3.3.1 Network File System (NFS)

The Network File System (NFS) is a way of mounting Linux directories over a network. We store the ATM ModelHub, which contains the unique identifier of each datarun, on the HPC Cluster's NFS server. Thus ATM workers initialised on different physical machines in the HPC cluster are able to access the ModelHub as if they were running on the same machine.

3.3.2 Secure Shell (SSH)

In order to launch workers simultaneously on each node in the HPC cluster, we need a way to:

1. Log in to each node using ssh.
2. Enter our environment for running ATM with our libraries.
3. Launch the atm worker.

3.3.3 Shell Script

Simultaneous node login is achieved using the xpanes tool from Yamada [2019]. We create a python script to launch the necessary commands on each node. Additionally, this script accepts optional flags which allows the entire process to run silently.

We note that although it is possible to have multiple workers on each compute node in the HPC cluster, we limit the number of workers to one. This is because each compute node contains one GPU. If multiple workers on the same machine are assigned jobs requiring the GPU, this will cause a bottleneck.

We also note that an alternative approach would be an MPI solution to start the workers simultaneously across the cluster using `mpirun`. Due to issues with our environment on our HPC cluster, this was not possible.

3.4 Creating Custom Classifiers in ATM

Since the documentation for ATM is not always clear, we outline the process used to create a new classifier. We create a new classifier in ATM as follows:

1. Create a JSON file containing a path to the classifier code. This file also defines the hyperparameters ATM will test, along with the classifiers ranges and values.

2. In the ATM methods file, we register the new method by giving it a name and linking it to the JSON file created in the previous step.

3. The classifier must provide fit(), predict() and predict_proba() methods. ATM calls these methods during a datarun, for each classifier it tests.

ATM uses the Python pickle library to save models during a datarun. Pickle is outdated and the ATM authors have already begun replacing it with JSON serialisation. The TSVM and TGBM libraries are unable to have their models saved using pickle due to the structuring of the classes. Since we only need to time the datarun we modified ATM to be able to continue with a datarun even if it cannot pickle the model. For validating our methods, we are able to access a models' metrics using a data log created during the datarun.

3.5 Implementing ThunderSVM

3.5.1 The Sequential Minimal Optimisation Algorithm

In TSVM, the primal optimisation problem for SVM described in section 2.3 is solved using the sequential minimal optimisation (SMO) algorithm. Briefly, the SMO algorithm is an iterative optimisation algorithm based on support vectors and Lagrange multipliers. It involves three main steps:

Step 1: Find 2 outliers in the training data that could best (potentially) improve the SVM.

Step 2: Improve the Lagrange multipliers of the two outliers.

Step 3: Update the optimality indicators for all training points.

3.5.2 ThunderSVM GPU Implementation of Support-Vector Machines

The authors parallelise steps 1 and 3 of the SMO algorithm as outlined in subsection 3.5.1, rationalising that step 2 is computationally inexpensive. Step 1 is parallelised by using a parallel reduction (once per outlier) during the search for two outliers. In the reduction, the entire array is loaded from the GPU global memory to the GPU shared memory. The size of the array is then repeatedly halved until it becomes a singleton. For Step 3, each optimality indicator is assigned a thread which updates it. The authors exploit the kernel calculation within the sub problems by efficiently parallelising matrix multiplication using CUDA libraries. Additionally, the amount of memory-accesses for the GPU is reduced by utilising the GPU memory buffer for storing reusable kernel values.

3.5.3 Integration with ATM

We create a new classifier in ATM called svm_gpu as described in Section 3.4. Without any modification, running ATM with our svm_gpu classifier produces an error relating to ctypes. After a considerable amount of debugging between the ATM and ThunderSVM source code, Numpy was found to be the source of the error. In particular, ThunderSVM expects standard Python data types for the input parameters. The parameters are then converted to C data types to be used in CUDA C code. Since ATM passes all parameters to the classifiers as Numpy data types, we modified the ThunderSVM source code to convert all parameters to standard Python data types.

3.6 Implementing ThunderGBM

3.6.1 ThunderGBM GPU Implementation of Random Forests

ThunderGBM parallelises three aspects of the random forest algorithm, namely:

1. GPU threads are created, with each thread computing the gradient and second order derivative of an instance, before moving onto another instance. The gradient and second derivative are used later for choosing a split point in the tree.
2. During training, the tree is constructed by finding possible split points. Depending on the density ratio [1] of the given dataset, TGBM utilises a histogram approach, which fixes the number of split points allowed. Each feature is mapped to a histogram, corresponding to the distribution of the training instances. The bins of the histograms are the second order derivative and gradient values for the training points within that bin. For each feature, partial histograms are created at the thread block level using shared memory in the GPU. The final histogram for that feature is computed from the accumulation of all partial histograms.
3. The best candidate split point is chosen based on its *gain*, where $\textit{gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right]$, where G_L and G_R are the sum of the gradients, and H_L and H_R are the sum of the second order derivatives, of all instances in the left and right nodes respectively. λ is a regularisation constant. For each split point candidate, a GPU-thread is reserved for finding the candidates' gain. The split point with the largest gain is found using a parallel reduction.

3.6.2 TGBM Integration with ATM

We create a new classifier in ATM called rf_gpu as described in Section 3.4. We encountered the same error from Section 3.5.3, which was solved in the same way. TGBM lacks a predict_proba() method required

[1] Where the density ratio is given by $\frac{\text{total features}}{\text{training instances} \times \text{dimension}}$

by ATM. However, it does provide a means of returning probabilities instead of class labels; by using the multi:softprob parameter. We add a predict_proba() method to TGBM using this parameter. While the algorithm computes all the probabilities for each class and test datapoint, it only returns the first probability value for each point. We create the predict_proba() method by reshaping the array containing all the probabilities, and returning it. Unfortunately this method was incompatible with binary classification, since we could not use both multi:softprob and the binary:logistic parameter needed for binary classification at the same time. TGBM provides both random forest and gradient boosted decision tree methods; we restrict the method to only use random forests, by fixing a parameter in our classifier's JSON file.

3.7 Implementing GPU k-NN

As mentioned in Section 2.5.1, we implement two versions of parallel k-NN. The first uses PyCUDA and the second uses standard CUDA C++ code with Python wrappers. Both use variants of the Garcia et al. [2010] implementation for the CUDA kernels. The PyCUDA implementation is our own and the C++ implementation is modified from Choy [2017]. We decided to try adapt the Choy [2017] implementation because initially it seemed much faster than our PyCUDA implementation. Furthermore, our PyCUDA implementation ran out of memory for fairly small datasets. This was somewhat expected for reasons discussed in Section 3.7.1, however we attributed added memory usage to the use of Numpy datatypes enforced by PyCUDA. We later realised that the Choy [2017] implementation does not do any calculations nor does it give any errors, when given a large dataset. This explains why our method seemed to fail, since it gave a memory error, while the other method seemed to work.

3.7.1 Parallel Distance Calculation

We create two arrays in shared memory; one for the reference points and one for the query points. Each thread then copies one element from each array in GPU global memory into the shared memory arrays. Once the two shared memory arrays are populated, each thread calculates the distance between one reference point and one query point. Once all distances have been calculated, each thread must copy its computed distance into the final output array, which resides in GPU global memory. The final output array has size $Q \times R$, where Q is the number of query points and R is the number of reference points. We can see that the output array can become very large. This is a known limitation of the k-NN method, especially on GPUs where memory is relatively scarce. In this approach, the Euclidean distance is used. However, it would be possible to implement alternative distance metrics such as the Chebyshev, Manhattan and Mahalanobis distance metrics.

3.7.2 Parallel Insertion Sort

Using the output array described in the above section, we assign each thread a list of distances between one query point and all the reference points, as shown in Figure 3.1. This algorithm is parallel in the sense that each thread performs a modified insertion sort on the list of distances for one query point. However, the insertion sort itself is serial. This should theoretically still result in speedup. The speedup should be more noticeable for a larger number of query points. The theoretical complexity of the parallel approach is $O(R^2\frac{Q}{t})$, where t is the number of threads. Comparatively, the serial complexity is $O(R^2Q)$, since we are sorting Q lists and insertion sort has complexity $O(R^2)$.

Algorithm 1: Modified insertion sort

```
Input: Input array A and integer k.
Result: List with the first k elements sorted.
max = A[0];
// Sort first k elements;
for j=2 to k do
    key = A[j];
    // Insert A[j] into sorted sequence A[1 ...j-1];
    i = j-1;
    while i > 0 and A[i] > key do
        A[i+1] = A[i];
        i = i -1 ;
    A[i+1] = key;
    if A[i+1] > max then
        max = A[i+1];
// Find elements in rest of list which belong in the first k, and insert them;
for j=k+1 to A.length do
    if A[j] < max then
        key = A[j];
        // Insert A[j] into sorted sequence A[1 ...k];
        i = j-1;
        while i > k and A[i] > key do
            A[i+1] = A[i];
            i = i -1 ;
        A[i+1] = key;
        max = A[k];
```

The sorting algorithm used by each thread is a variation of insertion sort, which takes advantage of the fact that we only need to find the k smallest distances. Pesudocode is provided in Algorithm 1. First, the algorithm performs standard insertion sort on the first k elements in the unsorted list. A variable keeps track of the maximum distance in the sorted list. Second, the algorithm goes through the list from the $k + 1$th element to the end of the list, comparing each value to the maximum distance. If an element is less than the maximum distance, then it must belong somewhere in the first k positions of the **sorted** list. It is then inserted into its correct position. If an insertion is made, then the maximum distance variable is updated to the value of the kth element. While it is not shown in the pseudocode, a list of indices corresponding to the original positions of the reference points is sorted simultaneously with the distance array.

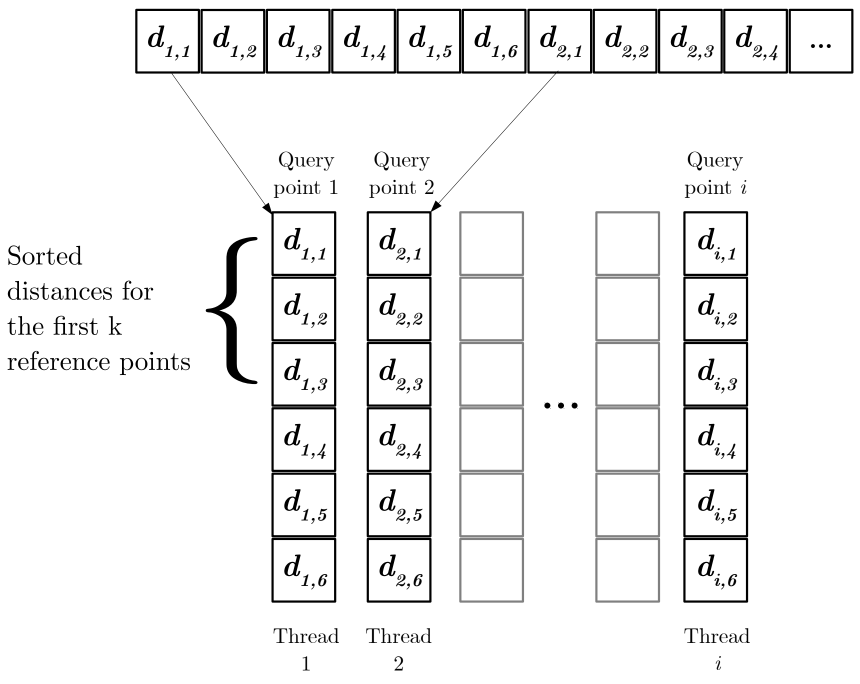

$d_{1,1}$ $d_{1,2}$ $d_{1,3}$ $d_{1,4}$ $d_{1,5}$ $d_{1,6}$ $d_{2,1}$ $d_{2,2}$ $d_{2,3}$ $d_{2,4}$ ...
Query point 1
Query point 2
Query point i
Sorted distances for the first k reference points
Thread 1
Thread 2
Thread i

Figure 3.1: Memory scheme and thread allocation for the parallel insertion sort of the distance matrix.

3.7.3 Integrating GPU k-NN in ATM

In order for the k-NN implementation to fit the format required by ATM, it must provide fit(), predict() and predict_proba() methods as mentioned in Section 3.4. Since data fitting in k-NN is essentially a formality, the fit() method only reads in and stores the training data and the value for k. This is consistent with the serial algorithm used by ATM. The predict() method takes in testing data, calculates the distances as described in Section 3.7.1, and then sorts the distances using the algorithm laid out in Section 3.7.2. We then use the indices array returned by the sorting algorithm to find the modal class label of the k nearest neighbours for each query point. Essentially, we find the majority vote of the first k points. Finally, the predict_proba() method finds the probabilities of each class for every query point. These probabilities are calculated by counting the frequency of each class label and dividing by the sum of the counts.

3.7.4 Python Wrapper Issues

Since we are using CUDA C with Python, we need some sort of wrapper to call the C code from Python. The k-NN implementation that we based ours on uses a C++ library known as Boost to accomplish this. Unfortunately, the code was written for Python 2.7, while we were using Python 3.6 for all our methods. Recompiling the k-NN code to be compatible with Python 3.6 was somewhat difficult to achieve, as the information we needed was obscure and hard to find. We eventually converted the code by changing two path values in the Makefile and Makefile.config files.

3.8 Conclusion

This chapter has provided detailed descriptions of our implementations of GPU version of algorithms used in ATM. We have discussed how the methods are parallelised and how we integrate them with ATM. We have also explained how we can use ATM on an HPC cluster, as well as the tools we utilise. In the next chapter we will present the result of our methods.

Chapter 4

Results

4.1 Introduction

In this Chapter, we first describe our experimental setup in Section 4.2.1. We then discuss alterations made to ATM in an attempt to prevent unbiased results between the serial and GPU classifiers in Section 4.2.2. The run time results comparing performance of the serial and GPU methods are given in Section 4.3. We go on to discuss these results in Section 4.4, and conclude by answering our research question in Section 4.5.

4.2 Experiments

4.2.1 Experimental Setup

We tested the methods using up to 80 nodes on an HPC cluster, with a 1 GbE network. Each node has the following specifications:

- 16 GB RAM (DDR4)
- Intel Core i7-7700 CPU @3.6 GHz
- 1 Nvidia GeForce GTX 1060, 6 GB RAM (DDR5)
- OS: Ubuntu 18.04.1

We use 3 different datasets of increasing size, to demonstrate how the performance of ATM scales for each GPU method. It is common in the AutoML literature to test multiple datasets, often from the OpenML library for benchmarking the system. In keeping with this trend, we use datasets from Bischl et al. [2017] as shown in Table 4.1.

Dataset ID	Dataset name	Number of instances	Number of features	Number of classes
882	pollution	60	15	2
40709	analcatdata_happiness	60	3	3
1542	volcanoes	1183	3	5
1459	artificial-characters	10218	7	10

Table 4.1: Datasets used for experiments.

4.2.2 Hyperparameter Options

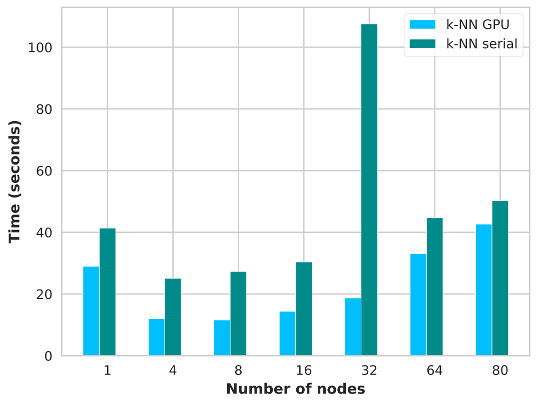
ATM chooses from a user-given list of hyperparameters and possible values. When performing a datarun, it tests the methods using combinations of these values. In order to avoid bias in the results, we restrict the serial algorithms to use only parameter values which are available to the corresponding GPU methods. We do this to ensure that both methods will have the same number of hyperparameter values, and will result in fair comparisons.

We do not give results on the accuracy of our methods, only run time. Since we are restricting the hyperparameter options, ATM may fail to find the optimal method, and therefore give less accurate results. Furthermore, the ranges of certain hyperparameters made TSVM immediately flood the GPUs cores, causing the datarun to hang indefintely.

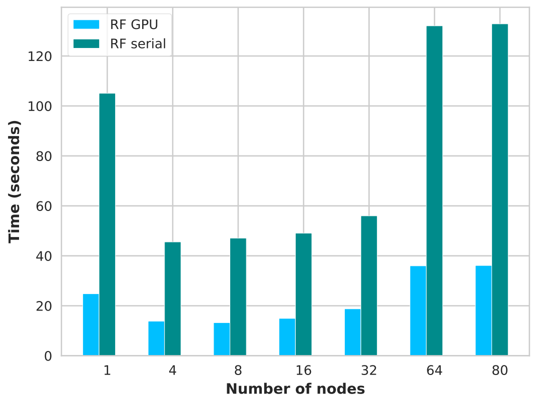
4.3 Runtime Results

Our main metric for performance in this research is run time. Due to the problems described in Section 4.4, the accuracy of the classifiers could not reliably be tested.

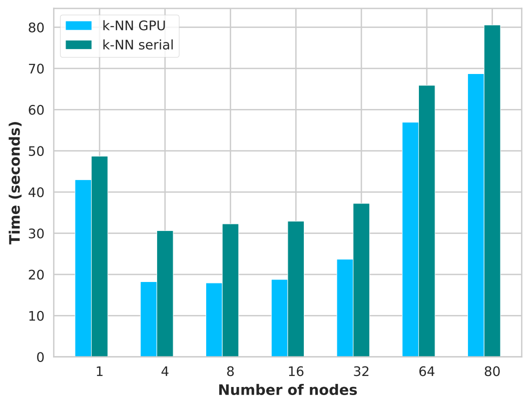
To obtain results we ran serial and GPU versions of each algorithm on the same dataset, and timed the run in seconds. We tested each algorithm and dataset combination on different numbers of HPC cluster nodes, from 1 to 80. The results for the k-NN method can be seen in Figure 4.1. We tested this method on all four datasets. Figure 4.2 shows the results for the RF method, which was only run on two of the datasets. We were unable to test the RF GPU method on the pollution dataset as it had only two classes. As explained in Section 3.6.2, our RF predict_proba() function was incompatible with binary classification problems. Finally the results for the SVM method are in Figure 4.3. We were only able to run this method on one dataset. For reasons which will be explained in the next section, we found that the run time of the SVM GPU method was very slow and prone to failure, and so it was impractical to test it on a larger dataset.



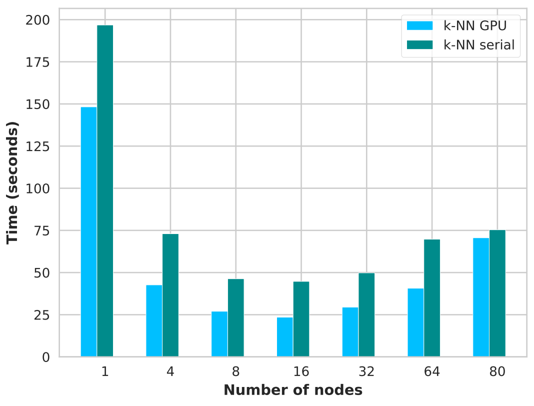
(a) Pollution dataset.



(b) Analcatdata dataset.



(c) Volcanoes dataset.



(d) Characters dataset.

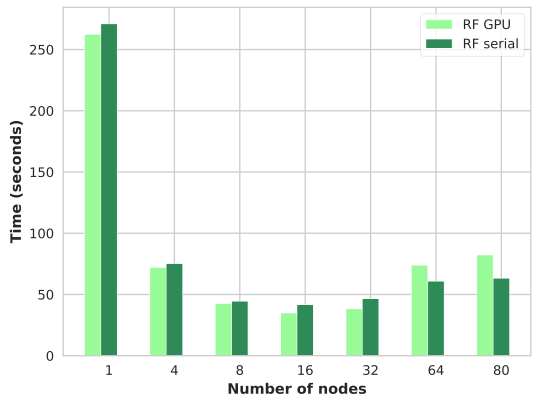
Figure 4.1: Run time for serial and GPU versions of k-NN, for all datasets and different numbers of nodes.

4.4 Interpretation of Results

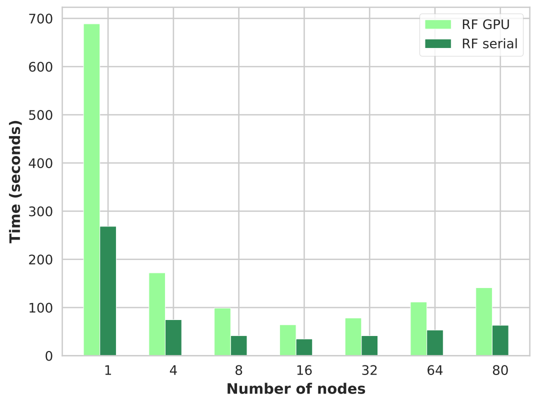
We can see from the results of the experiments that the Goldilocks zone is at 16 nodes in the cluster. This is most likely due to the 1 GbE network which becomes a bottle neck as the communication overhead increases. Specifically, when the number of workers writing to the ModelHub using NFS increases.

The time it takes xpanes to log in to 80 nodes, initialise the runtime environment and launch a worker may result in under utilisation of the cluster; all 100 classifiers may be trained by the time the 80th worker is launched. We limited the number of classifiers to 100 due to time constraints for this work.

There are several possible reasons for the bad performance of two of the GPU methods. The main usefulness of these methods is their ability to run on very large datasets. For smaller datasets, the performance gains are small or even nonexistent as the overhead is too high compared to the short run time. Another reason is



(a) Analcatdata dataset.



(b) Volcanoes dataset.

Figure 4.2: Run time for serial and GPU versions of RF, for two datasets, for different numbers of nodes.

that ThunderSVM and ThunderGBM are designed to run on sparse datasets. When they receive input in a non-sparse format, they convert it to a sparse form. This conversion results in additional overhead for the GPU methods. ATM itself cannot run on sparse matrices, so using sparse data was not an option.

For ThunderGBM specifically, we believe that the run time is roughly double what it should be. This is because we had to create a predict_proba() function, as described in Section 3.6.2, in order to integrate it with ATM. The predict_proba() method must unfortunately compute the same results as the predict() function in order to calculate the probabilities rather than the class labels. This is related to how ThunderGBM is structured and could not be changed. It means that our RF GPU method is doing a large amount of unnecessary computation compared to the serial method, and this drastically effects our results.

While testing ThunderSVM, we found that for certain hyperparameter values, the method would fail to complete in a reasonable amount of time. It was mentioned in the source code for ATM that this was a possible occurrence for SVMs, and was caused by a bad choice of hyperparameters. We tried restricting the time each worker was allowed to run, using an option provided by ATM, but this did not appear to have any effect. This could be due to extremely complex issues related to the operating systems interactions with the GPU.[1] We also learned that others had trouble replicating the results published by Wen et al. [2018] in the ThunderSVM paper [Xtra-Computing a]. We communicated the poor performance issues with the TSVM authors. They have began an investigation in attempt to resolve this issue [Xtra-Computing b].

The results for the k-NN GPU method show that using GPU-based methods can lead to significant performance increases. Our method is faster than the serial version for all numbers of nodes, achieving speedups of up to two times the speed of the serial algorithm.

[1] see https://github.com/pytorch/pytorch/issues/24081

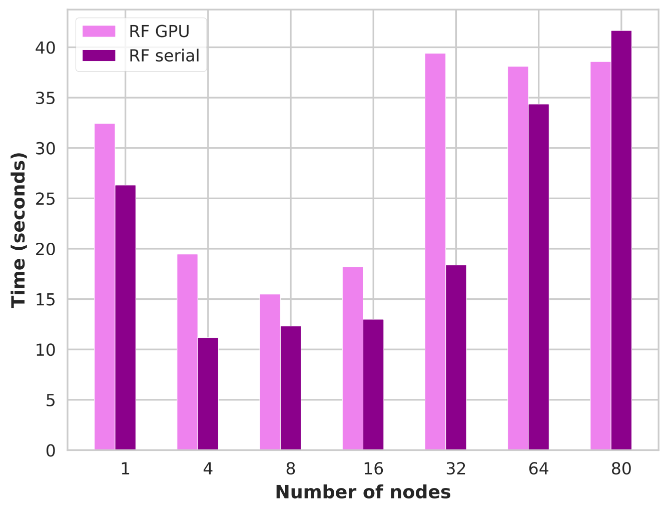


Figure 4.3: Run time for the serial and GPU versions of SVM, for the analcatdata dataset, for different numbers of nodes.

4.5 Conclusion

In this section we presented the results of our experiments which tested the research hypothesis defined in Section 3.2. These results are inconclusive; we cannot say for certain the research hypothesis is true. However, we did demonstrate a partial result with the k-NN method. TSVM and TGBM did not perform as expected. This was due to design conflicts between the AutoML framework and the GPU libraries.

Chapter 5

Conclusion

In this work, we attempted to implement three GPU based ML algorithms in the ATM framework. The k-NN GPU implementation, which we had the most control over, showed the greatest performance improvement over its serial counter part. The RF classifier sometimes did slightly better than the serial implementation but was hindered by the implementation of a method required by ATM. TSVM also had multiple issues when integrated into ATM, as well as inherent issues with performance which we brought to the attention of Wen et al. [2018]. Although it was not the main focus of this work, one of the larger contributions of this work is the method we created to allow ATM to be run on an HPC cluster.

ATM is deeply dependent on classifiers providing three scikit-learn methods. If a new classification algorithm does not already implement these methods efficiently, retrofitting them can be problematic, as was the case with TGBM. ATM also depends on classifiers adhering to data structures which allow created models to use the pickle library, even though the authors have replaced pickle with JSON in some portions of the source code. This is also problematic when trying to implement modern libraries which will most likely not support pickle.

Modern GPU machine learning algorithms use sparse datasets to improve performance. ATM only accepts dense datasets, which precludes the benefits of using the latest GPU based ML libraries.

Combining software libraries is labour intensive. As we have demonstrated, even if the components to be combined espouse a common API, as ATM, TSVM and TGBM espouse the scikit-learn interface, compatibility issues still arise. After conducting this research, we are inclined to think that an AutoML system, designed with GPU algorithms in mind from inception, would be a better way to ensure its usefulness, as computational requirements for ML problems grow larger.

Due to the time spent resolving the problems described in Chapters 4 and 3, we could not spend as much time conducting all the experiments we had aimed to. In particular, running ATM with multiple combinations of the GPU and serial classifiers on larger datasets would have taken days to run to completion. Since hardware resources and time were severely limited for this research, we recommend further tests to confirm or deny the research hypothesis. In particular, most of the latest GPU libraries are benchmarked using Nvidia P100 GPUs. These are necessary for truly reliable results and performance improvements.

5.0.1 Future work

For any future work related to GPU accelerated AutoML systems, we recommend the following areas:

- Adding support for sparse datasets in ATM.
- Adding an efficient predict_proba() method in TGBM
- Creating a CUDA-aware MPI AutoML system from scratch.
- Using pure C based systems as opposed to Python.
- Deviation from sci-kit API in favour of performance and extensibility.

Bibliography

Balaji, A. and Allen, A. (2018). Benchmarking automatic machine learning frameworks. *arXiv preprint arXiv:1808.06492*.

Bischl, B., Casalicchio, G., Feurer, M., Hutter, F., Lang, M., Mantovani, R. G., van Rijn, J. N., and Vanschoren, J. (2017). Openml benchmarking suites and the openml100. *arXiv preprint arXiv:1708.03731*.

Breiman, L. (1994). Bagging predictors. univ. california technical report no. 421.

Choy, C. (2017). modified k-NN GPU.

Feurer, M., Eggensperger, K., Falkner, S., Lindauer, M., and Hutter, F. (2018). Practical automated machine learning for the automl challenge 2018. In *International Workshop on Automatic Machine Learning at ICML*.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015). Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970.

Garcia, V., Debreuve, E., Nielsen, F., and Barlaud, M. (2010). K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*, pages 3757–3760. IEEE.

H2O.ai (2019). H2O4GPU.

Lu, Y., Zhu, Y., Han, M., He, J. S., and Zhang, Y. (2014). A survey of gpu accelerated svm. In *Proceedings of the 2014 ACM Southeast Regional Conference*, page 15. ACM.

Mendoza, H., Klein, A., Feurer, M., Springenberg, J. T., and Hutter, F. (2016). Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, pages 58–65.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

Swearingen, T., Drevo, W., Cyphers, B., Cuesta-Infante, A., Ross, A., and Veeramachaneni, K. (2017). Atm: A distributed, collaborative, scalable system for automated machine learning. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 151–162. IEEE.

Truong, A., Walters, A., Goodsitt, J., Hines, K., Bruss, B., and Farivar, R. (2019). Towards automated machine learning: Evaluation and comparison of automl approaches and tools. *arXiv preprint arXiv:1908.05557*.

Wen, Z., Shi, J., He, B., Li, Q., and Chen, J. (2019). ThunderGBM: Fast GBDTs and random forests on GPUs. *To appear in arXiv*.

Wen, Z., Shi, J., Li, Q., He, B., and Chen, J. (2018). Thundersvm: A fast svm library on gpus and cpus. *The Journal of Machine Learning Research*, 19(1):797–801.

Xtra-Computing. issue 162. `https://github.com/Xtra-Computing/thundersvm/pull/162`. Oct 2019.

Xtra-Computing. issue 172. `https://github.com/Xtra-Computing/thundersvm/issues/172`. Oct 2019.

Yamada, Y. (2019). tmux-xpanes. https://github.com/greymd/tmux-xpanes.