GPU Implementation of Grid Search based Feature Selection

Using Machine Learning to Predict Hydrocarbons using High Dimensional Datasets

by

T. A. Ament

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Friday February 21, 2020 at 10:30 AM.

Student number:	4080300	
Project duration:	March 15, 2019 – February 21, 2020	
Thesis committee:	Prof. dr. ir. H. X. Lin,	TU Delft, Thesis advisor
	Dr. ir. A. van Genderen,	TU Delft
	Dr. ir. C. B. M. te Stroet MBA,	Biodentify, Daily advisor

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This report is the master's thesis of the Computer Engineering master at the Delft University of Technology. From my childhood I've always been interested in the graphics processing unit, or GPU. When I was younger it was "that thing that plays games" and finally, when I first built a PC to play games on, discovered the wide range of available GPU's. During my bachelor, I came into contact with research using GPU's to accelerate the analysis of data, which peaked my interest, this interest was one of the main reasons I chose the Computer Engineering master and finally this research topic for my thesis.

Without the support of my friends and family this thesis would not have existed. First of all, my parents, who have always supported me with unconditional love. Secondly, my girlfriend, for being there with love and her father for providing feedback. Thirdly, my housemates and the coffee machine, for providing an endless supply of coffee when I needed it the most. And lastly my colleagues and my committee members, who have provided advice and answered questions throughout the process. Thank you all for your support.

T. A. Ament
Delft, February 2020

Abstract

To optimize the exploitation of oil and gas reservoirs both on- and offshore, Biodentfiy has developed a method to predict prospectivity of hydrocarbons before drilling. This method uses microbiological DNA analysis of shallow soil or seabed samples to detect vertical upward microseepage from hydrocarbon accumulations, which change the composition of microbes at the surface. Microbiological DNA analysis of shallow soil or seabed samples results in a high-dimensional dataset, which is interpreted using machine learning. Using the machine learning method Elastic Net, features (microbes) are selected from an existing DNA database to classify new shallow soil or seabed samples. Multiple models, each with a different combination of externally set parameters (called hyperparameters), are trained to improve accuracy, essentially creating a grid of models. The aim of this thesis is to investigate if it is possible to accelerate feature selection on high-dimensional datasets by implementing a parallel design on a GPU to train this grid of models, and to investigate the performance of this GPU implementation. Inspired by an implementation called Shotgun, which is able to improve performance by exploiting parallelism across features when training a single model on a CPU, an implementation, named GPU Shotgun (GPU-SG) was devised, which could exploit parallelism across samples, features, and multiple models in the grid (of combinations of hyperparameters). Depending on the size of the grid and the hardware, using GPU-SG, a speedup of between 0.2 and 5.26 can be reached for sparse datasets (a datasets with lots of 0 values) when compared to standard CPU implementations. When considering dense datasets (a dataset with few 0 values), using GPU-SG, a speedup of between 0.5 and 10 can be achieved. The amount of memory available to store a dataset is lower for GPU's than for a CPU, and currently the design is limited by this, because the design does not allow a dataset that is larger than the memory available. GPU-SG can be used to design improved implementations, which reduce the time when the GPU or CPU is idle to improve performance.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

This chapter provides an introduction to using machine learning to predict prospectivity of hydrocarbons. Furthermore it briefly explains the background biological- and geological science on which this technology is based. This prediction of hydrocarbons is done with classification of the data using machine learning. Due to high dimensional dataset and low amount of samples, features have to be selected first, which is the focus of this thesis. This feature selection requires a large amount of computational power (or time), which is a problem that many researchers try to tackle. A good candidate method to decrease the time required to select features is an implementation of a parallel algorithm on the Graphics Processing Unit (GPU). This method will be researched in this thesis and makes the objective to speed up this process by using the GPU.

This chapter is structured as follows: Firstly, background information on the thesis is given, introducing the company Biodentify and the biological- and geological science on which this thesis is based. Secondly, the focus of the thesis, feature selection, and the problem statement, concerning training time is presented. Thirdly, the research question and objectives are formulated. Finally, the structure of the thesis is presented.

1.1. Background Information

This thesis is carried out at Biodentify, which is a startup, located in Delft, the Netherlands and has a business development and support office, located in Houston, USA. It was founded in December 2014 as a spin out from Dutch R&D group TNO. The company has developed a method to predict prospectivity of hydrocarbons (to find likely locations to drill for oil and gas) before drilling, through the use of "microbiological DNA analysis of shallow soil or seabed samples" and artificial intelligence [1]. Microbiological DNA analysis is the technique of matching the DNA found in a sample against microbes (also called DNA fingerprints) in order to detect and count them.

The goal of Biodentify is to optimize the exploitation of oil and gas reservoirs both on- and offshore, by analysing drilling sites with a combination of DNA analysis and artificial intelligence to identify hydrocarbon sweet spots. To achieve this goal, Biodentify uses its own workflow and technologies [2]. This workflow consists of six parts (also seen in fig. 1.1):

1. Sample collection: collect shallow soil samples in the field or shallow seabed samples offshore.
2. Lab analysis: extract DNA from the shallow soil samples.
3. Bio-marker Identification: match the extracted DNA to microbial DNA fingerprints.
4. Existing DNA Database: select a training set from the Biodentify database with similar DNA fingerprints. The training set has known production sites attached to soil samples.
5. Use machine learning to create a model to classify the unknown soil samples to determine prospectivity.

6. Using the classification, map and analyse the prospectivity of the drilling site.

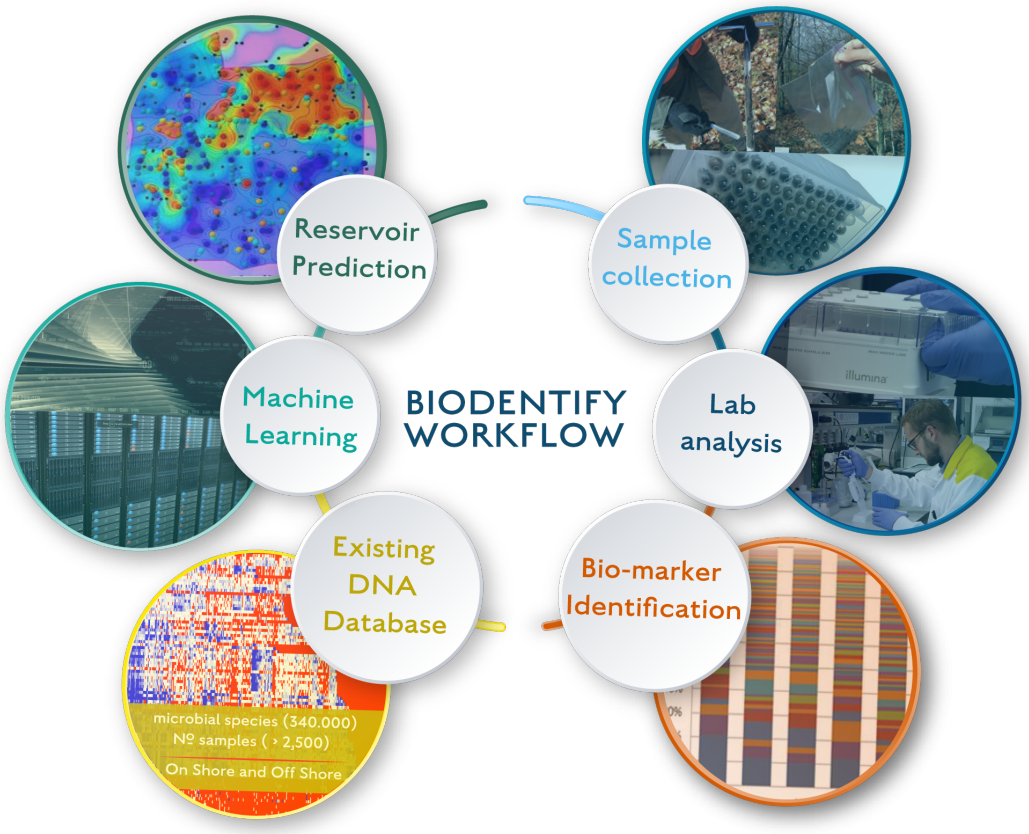

Figure 1.1: Biodentify workflow

The technology has at its core that vertical upward microseepage (the leakage of very small amounts of liquids through porous materials) from hydrocarbon accumulations in the subsurface changes the composition of microbes at the surface [17]. These changes can be revealed by an analysis of the DNA fingerprints of shallow soil samples. However, since there are over 300.000 different microbes present in training sets, only by using machine learning can the changes linked to microseepage be seperated from other changes, random variations, or noise.

1.2. Thesis Focus

This thesis focuses on one part of the workflow of Biodentify, the classification through machine learning. Classification is the process of determining, from the DNA fingerprints, which ground samples contain oil or gas microseepage and which have no microseepage. This classification is done in three steps:

1. Select a training set with a similar DNA fingerprint to the new ground samples.
2. Select from that training set those microbes (features) that are correlated with the presence or absence of microseepage.
3. Use the selected features to classify the new test set.

The focus is on step two of the classification, which is called feature selection. **Feature selection** is particularly useful when the training set has many more features (p) than it has samples (n). Such a dataset is also known as a (super)high dimensional dataset. Biodentify has such a dataset, with over 300.000 features and less than 10.000 samples ($p >> n$).

1.2.1. Problem Statement

To accurately select those features that are correlated with the presence or absence of microseepage, two choices have to be made:

- An estimator, which is the type of model to. For example: one of the many estimators that use the Support Vector Machine (SVM) method, or one of the estimators that use linear regression such as Lasso and Elastic Net (EN).
- The hyperparameter(s) for the estimator. A hyperparameter is a parameter that can't be inferred from the data by the model and is set externally. For example: the regularization parameter (C) and the parameter of the Gaussian Kernel[1] (γ) for SVM, the L1-norm penalty (α) for Lasso, and the penalty strength (α) and the ratio between the L1- and L2-norm penalty ($L1_{ratio}$) for EN.

The choice for the best estimator is made by examining the dataset, looking at the amount of features and samples, the sparsity (amount of non-zero values) and the goal of the model (e.g. classification, regression or feature selection).

Hyperparameters control the amount of regularization on the trained model. This is usually required to avoid fitting on noise in the data, resulting in an overfitted model. In order to create a predictive model, opposed to a fitted model, there is a need for regularization. To find the best combination of hyperparameters, one of the most common strategies is to create an n-dimensional grid with on every axis a number of choices for a hyperparameter. Then train a model for every point in the grid, and then search for the trained model that scores the best. This procedure is referred to as a *Grid Search*. However, for large datasets, this quickly requires a large amount of computing power, which in turn results in long training times and/or high costs. Therefore, finding a way to decrease training time, while still reaching the desired accuracy, is highly beneficial for Biodentify. A good candidate for decreasing training time, is parallelization on a GPU opposed to the current implementation, which is parallelization on a central processing unit (CPU). Designing a GPU program can be done using the CUDA platform, which is a parallel computing platform created by Nvidia.

1.3. Research Question

The goal of this thesis is to investigate if it is possible to accelerate feature selection on high-dimensional datasets by implementing a parallel design on a GPU. Specifically, the Elastic Net algorithm, using a grid search to find the best combination of hyperparameters, will be implemented on a GPU using CUDA. This will answer the following question:

> *What is required to design a GPU program, that does parallel feature selection on high-dimensional datasets to predict prospectivity of hydrocarbons, with the Elastic Net algorithm, using a grid search to find the best combination of hyperparameters, and what is the performance compared to a CPU implementation?*

1.3.1. Objectives

In order to answer this question, the following objectives have been set:

- Review existing parallel machine learning algorithms, including, but not limited to, Elastic Net, on GPU and CPU;
- Design and implement a parallel Grid Search on the GPU featuring the Elastic Net algorithm that parallelizes over the features;
- Benchmark the parallel Grid Search implementation against the current CPU implementation, for different datasets.

[1] Not to be confused with CUDA Kernels, which is what "kernel" refers to in this thesis.

1.4. Outline of Thesis

To present the matter of this thesis in a clear and constructive format, it is constructed as follows:

Chapter 2 gives a detailed overview of the feature selection procedure, which includes the choice of estimator for the feature selection. Model cross-validation is explained and grid search is presented to find the optimal choice of hyperparameters.

Related literature concerning the current state of research on the topic of machine learning, and parallel feature selection on both the CPU and GPU, is reviewed in Chapter 3.

The current implementation of the feature selection algorithm on the CPU is given, the parallelization possibilities are discussed, the steps taken to implement this algorithm on the GPU are given in Chapter 4.

The results of the CPU, Cyclical GPU, and Shotgun GPU implementations on both a random dense dataset and the sparse Biodentify dataset are given in Chapter 5.

The GPU implementations are discussed in Chapter 6. This discussion includes the performance and the limitations of the GPU implementations. A discussion on the economic impact of the choice between a CPU and a GPU implementation is included as well.

The conclusion of this thesis is presented in Chapter 7

Finally, in Chapter 8 recommendations for future work are discussed.

2

Feature Selection

This chapter explains the purpose of feature selection for high-dimensional datasets and describes the theory behind feature selection. The chapter is divided into the following four sections:

1. The purpose of feature selection for high-dimensional datasets is explained and the motivation for using Elastic Net to train the model is presented.
2. The mathematical background of Elastic Net is explained.
3. Cross-Validation is introduced to check if the trained model is stable.
4. The requirement for an exhaustive Grid Search to determine the hyperparameters is explained.

2.1. Purpose of Feature Selection

As mentioned in section 1.2, feature selection is required in datasets where the amount of features is many times larger than the amount of samples ($p >> n$). When attempting to train a classification model on such a dataset without doing feature selection, three problems arise: the first problem is that (generally) training time increases exponentially with feature size; the second problem is that the model tends to overfit due to redundant data/noise; the third problem is that accuracy is reduced due to misleading data.

Feature selection is generally done by using a supervised model, these models use the classification of the samples to select those features that are relevant to the solution. Unsupervised feature selection is called dimensionality reduction, where the models learn to inherent structure using only unclassified data. These models, however, tend to be less accurate than the supervised models. This is especially true when dealing with a dataset where the amount of features greatly outnumbers the number of samples (300.000 features and 2.000 samples). Therefore it is likely that one of the estimators in the category "regression" in fig. 2.1 will probably be the best. When considering the choice of supervised learning, given that the dataset has less than 10K (10.000) samples with over 100k (100.000) features and few features should be important, the estimators Lasso or Elastic Net are good candidates for feature selection.

The goal of feature selection is to find the sparse set of weights (or coefficients, $w_1, \ldots, w_p$), which can produce a correct classification ($y = -1$ or $y = 1$ in dual classification) from a sample (containing features $x_1, \ldots, x_p$, where x is normalized and thus in $[0, 1]$).

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_p x_p \tag{2.1}$$

Elastic net attempts to find the optimal solution for the weights ($w_1, \ldots, w_p$) using an extended form of the Ordinary Least Squares (OLS) problem, which is explained in the next section.

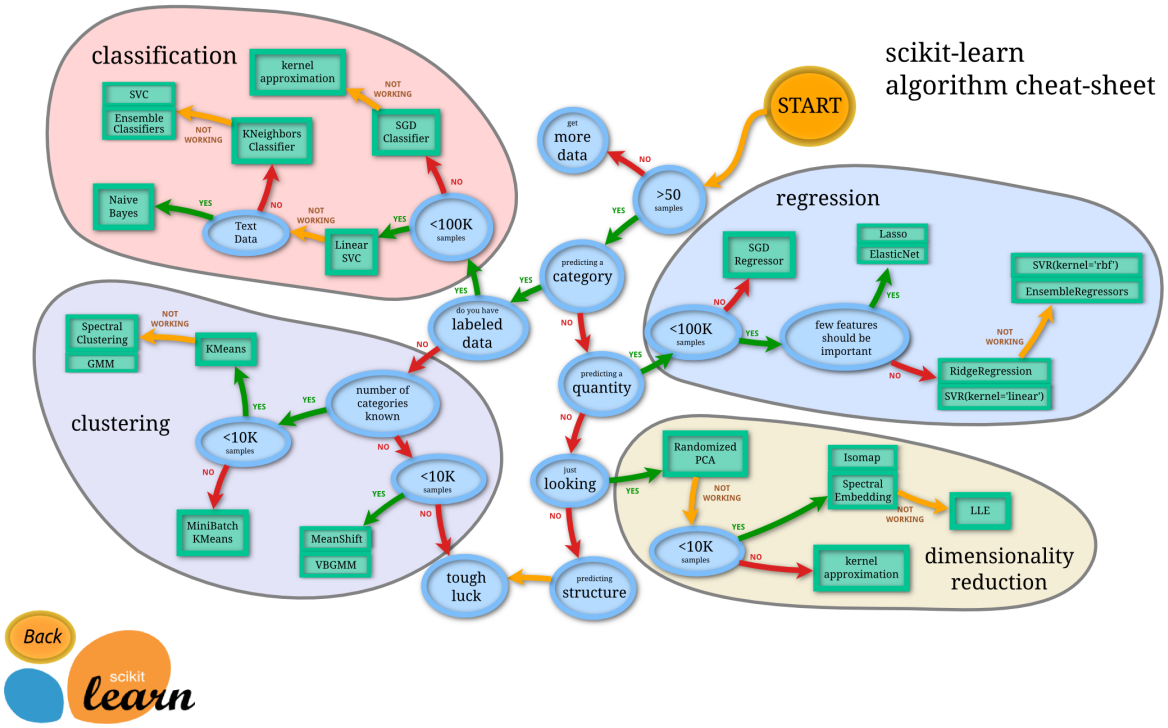

Figure 2.1: Choosing the right estimator `https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html`

2.2. Optimization of Elastic Net

In order to find the correct weights the model is trained using a dataset containing multiple samples. The dataset is represented by the matrix $\mathbf{X}$, which is a $n \times p$ matrix, with $n = n_{samples}$ and $p = n_{features}$. The dataset has a corresponding solution vector y of length n. The weights are defined as a vector $\mathbf{w}$ of length p. eq. (2.1) then becomes:

$$\mathbf{y} = \mathbf{X}\mathbf{w} \tag{2.2}$$

However, $\mathbf{w}$ only has a unique solution in very specific circumstances and thus the search for the most optimal $\mathbf{w}$ becomes the goal. Various objective functions have been formulated to solving eq. (2.2) through optimization. Several popular objective functions will now be discussed.

2.2.1. Ordinary Least Squares

The purpose of OLS is the search of the optimal solution w, such that the error ($||\mathbf{X}\mathbf{w} - \mathbf{y}||$) is minimal. This results in the following objective function :

$$\hat{\mathbf{w}}_{OLS} = \min_{\mathbf{w}} ||\mathbf{X}\,\mathbf{w} - \mathbf{y}||_2^2 \tag{2.3}$$

Which can be written as

$$(\mathbf{X}^T\mathbf{X})\hat{\mathbf{w}}_{OLS} = \mathbf{X}^T\mathbf{y} \tag{2.4}$$

where $\mathbf{X}^T\mathbf{X}$ is known as the Gramian matrix (or Gram matrix) and $\hat{\mathbf{w}}$ is the optimal solution. $\hat{\mathbf{w}}$ is thus expressed as:

$$\hat{\mathbf{w}}_{OLS} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{2.5}$$

For OLS to create a stable model, the features should be linear independent. If there are features that are linear dependent, the minimisation will be highly influenced by observational errors in the data $\mathbf{X}$, due to errors creating a large variance. This is called multicollinearity. Another problem for OLS is an over-defined model, which is a model with more features than samples [12]. Since the dataset of Biodentify is such a dataset, OLS will be a bad choice for an estimator.

Ridge and Lasso attempts to overcome these problems.

2.2.2. Ridge

To overcome the multicollinearity problem, it's possible to add a penalty to OLS by using L2 regularization. Ridge ([10]) adds a parameter ℓ_2, of the identity matrix ($\mathbf{I}$) to the Gram matrix. This changes eq. (2.5) to:

$$\hat{\mathbf{w}}_{Ridge} = (\mathbf{X}^T\mathbf{X} - \ell_2\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{2.6}$$

This adds a term to eq. (2.4), and we get the Ridge model:

$$\hat{\mathbf{w}}_{Ridge} = \min_{\mathbf{w}} ||\mathbf{X}\,\mathbf{w} - \mathbf{y}||_2^2 + \ell_2||\mathbf{w}||_2^2 \tag{2.7}$$

ℓ_2 can be used to adjust the penalty, with $\ell_2 = 0$ resulting in eq. (2.3) and $\ell_2 = \infty$ results in $||\mathbf{w}|| = 0$. As such, a correct choice of ℓ_2 is somewhere between 0 and ∞. However, the choice of ℓ_2 is not easy, and it is the first hyperparameter introduced. How the choice for a hyperparameter is made will be explained later, in section 2.4.

The addition of the L2 regularization penalty results in a more stable model, which is more robust to random noise, solves the problem of multicollinearity, and attempts to overcome the problems of an over-defined model. The main disadvantage of Ridge however, is that the trained model ends up with no coefficients that are zero, and thus is not actually selecting features, but rather selecting all of them.

2.2.3. Lasso

Using Lasso (eq. (2.9)) results in very few non-zero coefficients, resulting in a sparse model, by adding a penalty with the use of L1 regularization [20]. If two features are heavily correlated, Lasso picks one of them arbitrarily, in contrast to Ridge, which would have picked both of them with a lower coefficient. This results in very few features that are picked, but it is less stable than Ridge.

To create a sparse model Lasso adds a penalty to OLS by adding a penalty to the L1-norm. It adds a parameter ℓ_1, which controls the L1 regularization, resulting in the following objective function:

$$\hat{\mathbf{w}}_{Lasso} = \min_{\mathbf{w}} ||\mathbf{X}\,\mathbf{w} - \mathbf{y}||_2^2 + \ell_1||\mathbf{w}||_1 \tag{2.8}$$

This results in many more features set to 0, contrary to Ridge. As features with weight 0 can be excluded from the model, Lasso does do feature selection. However, Lasso is not good in selecting features that are correlated, since it arbitrarily sets one of them to 0 and is discarded from the model. This could result in a less predictive model when doing classification, since the feature could have been useful for classifying new samples.

As with Ridge, ℓ_1 is a hyperparameter, and this will be discussed in section 2.4.

$$\min_{\mathbf{w}} \frac{1}{2n}||\mathbf{X}\,\mathbf{w} - \mathbf{y}||_2^2 + \ell_1||\mathbf{w}||_1 \tag{2.9}$$

2.2.4. Elastic Net

Elastic Net [22] combines both the stability of Ridge and the sparsity of Lasso, by using both the L1 and L2 regularization. Combining both eq. (2.6) and eq. (2.8) results in the following objective function:

$$\hat{\mathbf{w}}_{EN} = \min_{\mathbf{w}} ||\mathbf{X}\,\mathbf{w} - \mathbf{y}||_2^2 + \ell_1||\mathbf{w}||_1 + \ell_2||\mathbf{w}||_2^2 \tag{2.10}$$

Equation (2.10) can be rewritten by combining ℓ_1 and ℓ_2 into two new parameters: the L1-ratio (ℓ), and the regularization factor (α). The L1-ratio is defined as $\ell = \frac{\ell_1}{\ell_1+\ell_2}$, which can be used to select between Lasso ($\ell = 1$) and Ridge ($\ell = 0$), or any combination of Lasso and Ridge ($0 < \ell < 1$). The regularization factor is defined as $\alpha = \ell_1 + \ell_2$. Substituting α and ℓ for ℓ_1 and ℓ_2 gives the following objective function:

$$\hat{\mathbf{w}}_{EN} = \min_{\mathbf{w}} ||\mathbf{X}\,\mathbf{w} - \mathbf{y}||_2^2 + \alpha \cdot \ell ||w||_1 + \alpha(1-\ell)||w||_2^2 \tag{2.11}$$

This results in a sparse model, but is better at selecting correlated features than Lasso. There are now two hyperparameters, α and ℓ, which will be discussed in section 2.4.

Coordinate Descent

The objective function eq. (2.10) is solved using a Coordinate Descent based on [9]. Coordinate descent has as a basis that the minimization of an objective function with multiple features such as eq. (2.10) can be achieved by updating the features one at a time, iterating over time. Given the feature vector $\mathbf{w} = (w_1, w_2, \ldots, w_p)$ it starts with an initial vector $\mathbf{w}^{(0)} = (w_1^{(0)}, w_2^{(0)}, \ldots, w_p^{(0)})$ and then updates each feature:

$$\begin{aligned}
\mathbf{w}_1^{(1)} &\in \min_{w_1} f(w_1, w_2^{(0)}, \ldots, w_p^{(0)}) \\
\mathbf{w}_2^{(1)} &\in \min_{w_2} f(w_1^{(1)}, w_2, \ldots, w_p^{(0)}) \\
\mathbf{w}_p^{(1)} &\in \min_{w_p} f(w_1^{(1)}, w_2^{(1)}, \ldots, w_p) \\
\mathbf{w}_1^{(2)} &\in \min_{w_1} f(w_1, w_2^{(1)}, \ldots, w_p^{(1)}) \\
\mathbf{w}_2^{(2)} &\in \min_{w_2} f(w_1^{(2)}, w_2, \ldots, w_p^{(1)}) \\
\mathbf{w}_p^{(2)} &\in \min_{w_p} f(w_1^{(2)}, w_2^{(2)}, \ldots, w_p) \\
\mathbf{w}_1^{(k)} &\in \min_{w_1} f(w_1, w_2^{(k-1)}, \ldots, w_p^{(k-1)}) \\
\mathbf{w}_2^{(k)} &\in \min_{w_2} f(w_1^{(k)}, w_2, \ldots, w_p^{(k-1)}) \\
\mathbf{w}_p^{(k)} &\in \min_{w_p} f(w_1^{(k)}, w_2^{(k)}, \ldots, w_p)
\end{aligned}$$

This feature update is done until convergence is reached or a user-defined maximum value of k is reached. The function that is done each update for Elastic Net uses a residual ($\mathbf{r} = \mathbf{y} - \mathbf{X}\mathbf{w}$) to keep track of the objective, which is updated after each feature is updated to reduce computation. The weight of a feature (w_i) is then updated every iteration using the following function:

$$f(w_i) = \texttt{sign}(\mathbf{r} \bullet \mathbf{x}_i) \frac{\max(|\mathbf{r} \bullet \mathbf{x}_i| - \ell_1, 0)}{||\mathbf{x}_i||_2^2 + \ell_2} \tag{2.12}$$

Since $||\mathbf{x}_i||_2^2$ does not change, it can be calculated before the coordinate descent is started.

The algorithm boils down to:

Data: $\mathbf{y}, \mathbf{X}, \mathbf{w}, \ell_1, \ell_2$
Result: $\mathbf{w}$
$\mathbf{r} = \mathbf{y} - \mathbf{X}\mathbf{w}$;
while not converged or reached max iterations **do**
  $\delta w_{max} = 0$;
  $w_{max} = 0$;
  for $i = 0$ **to** p **do**
    $w_{old} = w_i$ $\mathbf{r} = \mathbf{r} + w_i \mathbf{x}_i$;
    $t = \mathbf{r} \bullet \mathbf{x}_i$;
    $w_i = \texttt{sign}(t) \frac{\max(|t| - \ell_1, 0)}{||\mathbf{x}_i||_2^2 + \ell_2}$;
    $\mathbf{r} = \mathbf{r} - w_i \mathbf{x}_i$;
    $\delta w_{max} = \texttt{max}(\delta w_{max}, |w_i - w_{old}|)$;
    $w_{max} = \texttt{max}(w_{max}, |w_i|)$;
  end
  `Convergence check`
end

Algorithm 1: Coordinate Descent for Elastic Net

In algorithm 1 the current feature is indicated by i, thus $\mathbf{x}_i$ is the column in $\mathbf{X}$ which corresponds with the feature, and w_i is the weight of the feature in $\mathbf{w}$. The absolute maximum change in weights and the absolute maximum weight during a loop over all the features is stored in δw_{max} and w_{max} respectively, since they are required to check for convergence. The while loop stops when a user-defined maximum amount of iterations is reached.

Convergence Check

The convergence check is based on [11] and taken directly from the Sklearn [13] CPU implementation without further modifications and thus is not explained in-depth. The convergence check on algorithm 1 is done using the duality gap. ($||\mathbf{x}||_\infty$ gives the maximum absolute value in vector $\mathbf{x}$) The algorithm is as follows:

Data: $\mathbf{y}, \mathbf{X}, \mathbf{w}, \epsilon, \mathbf{r}, \ell_1, \ell_2, \delta w_{max}, w_{max}$
Result: Converged y/n
while not converged **do**
  if $w_{max} == 0$ **OR** $\delta w_{max} / w_{max} < \epsilon$ **then**
    dual = $||\mathbf{X}^T\mathbf{r} - \ell_2\mathbf{w}||_\infty$;
    if dual > ℓ_1 **then**
      $\rho = \frac{\ell_1}{\text{dual}}$;
      gap = $\frac{1}{2} \cdot ||\mathbf{r}||_2^2 \cdot (1 + \rho^2)$;
    else
      $\rho = 1$;
      gap = $||\mathbf{r}||_2^2$;
    end
    gap = gap + $\ell_1 \cdot ||\mathbf{w}||_1 - \rho \cdot \mathbf{r} \bullet \mathbf{y} + \frac{1}{2} \cdot \ell_2 \cdot ||\mathbf{w}||_2^2 \cdot (1 + \rho^2)$;
  end
  if gap < $\epsilon \cdot ||\mathbf{y}||_2^2$ **then**
    break (converged);
  end
end

Algorithm 2: Duality Gap for Elastic Net

In algorithm 2 ϵ is a user-defined tolerance value, which as default value has $1e^{-4}$.

2.3. Model Cross-Validation

The data set is split into i random train-test sets (generally in the range of 70/30-80/20). This is done for two reasons: the first reason is to create multiple trained models with a random variation in the train sets, to try and filter out noise in the data. The second reason is to check the validity of each model with the use of cross-validation. Generally the data set has to be split in at least 100 different sets to ensure the presence of a sample in at least one train set.

2.3.1. Shuffle Split Cross-Validation

Cross-validation is required to determine the stability of a model. Without cross-validation, the model might be under- or overfitting the data. Underfitting results in a model that scores poorly on both the training data and the test data. Overfitting results in a model with a very high accuracy for the train set, but with a very low accuracy when applying this model on the test set. One of the most common causes is that noise or random fluctuations is regarded as real data by the model, instead of being discarded by the model. To check if the model is stable, the train set is split up in i splits. Every model will then have a different training set, and can be scored against its own test set to be checked for stability.

Currently a stratified shuffle split is used. This type of split preserves the percentage of samples for each classification. Samples are allowed to be left out any number of time (including never left out). The splitting of an example dataset is visualised in fig. 2.2.

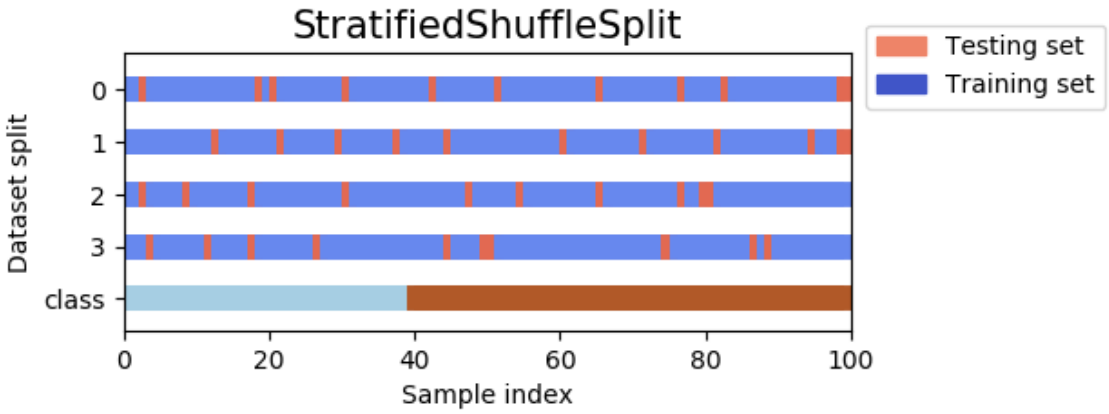

Figure 2.2: Visualisation of the Stratified Shuffle Split. Class represents the visualisation of the dataset when considering the classification label

2.3.2. Final Validation on Left Out Test Set

After the model is fully trained, one final validation is performed to check if the model is truly predictive. Using the weights from the trained model, the solution for a test set that is not used before by the model is predicted and the error between the predicted solution and the known solution is a very strong indication of the performance of the model.

2.4. Hyperparameters

Many estimators (including Elastic Net) have a number of user-defined variables that are not trained, called hyperparameters. Various aspects of a machine learning algorithm are configured by these hyperparameters and the hyperparameters can have a huge impact on the resulting model and subsequently its performance. Besides the mentioned hyperparameters for Ridge (ℓ_2), Lasso (ℓ_1), and Elastic Net (α and ℓ, or ℓ_1 and ℓ_2), there are hyperparameters present in other estimators as well. Support Vector Machine has two hyperparameters C and γ. Random Forest Trees has a number of hyperparameters, which set the amount of trees, the depth, the maximum number of features considered, and more. Neural Networks has 4 primary hyperparameters, the learning rate, the batch size, the momentum, and the weight decay.

Choosing the hyperparameters that result in a well-performing model is no trivial task. To find the optimal combination of hyperparameters, there are a number of strategies:

1. Brute force, this strategy generates a grid composed of every possible combination of hyperparameters, learns a model for each point in the grid and then picks the model that performs best. The problem with this strategy is that the grid quickly grows in size when there are more than two hyperparameters.

2. Randomly search the grid, this strategy uses the same grid as strategy 1, but instead of learning a model for each point in the entire grid, however it does so only for a random subset of the grid. This is done when the grid is too large to search in its entirety.

3. Combine strategy 1 and 2, the result of the random search (strategy 2) is used to built a new smaller grid around the solution. This new (smaller) grid which can then be brute forced using strategy 1.

4. Gradient Descent over the grid, this strategy attempts to find the most optimal combination of hyperparameters by finding the minimum using a gradient descent. This strategy is used when the grid is exceptionally large. However, this strategy is less and less used as both strategy 1 and 2 can be done in parallel by distributing it across a cluster of computers.

Using any of these strategies will result in another problem that has to be addressed as well. This problem is that comparing the learned models from the grid and picking the best one, will likely result in an over-fitted

model, since the learned models are not tested against unseen data. To prevent this, the models have to be cross-validated, which will be explained in section 2.4.2.

2.4.1. Grid Search

Since Elastic Net only has two hyperparameters (α and ℓ, or ℓ_1 and ℓ_2), strategy 1, using a brute force grid search, will be used to select the best hyperparameters. Every point in the grid thus solves eq. (2.10), for different ℓ_1 and ℓ_2. With h_1 = number of ℓ_1, and h_2 = number of ℓ_2, the gridsize becomes:

$$gridsize = h_1 * h_2 \tag{2.13}$$

2.4.2. Predictive Model Cross-Validation

To create a predictive model, instead of a fitting model, another cross-validation is required when training a model for a combination of hyperparameters. The training set obtained after the stratified shuffle split is split again into different training and test sets. In this case a stratified k fold is used, which divides the samples in k groups (folds) and the model is trained for every combination of $k-1$ folds. The fold left out for each model is used as a test set, to be used when scoring the grid. When creating the folds, the StratifiedKFold tries to preserve the percentage of samples for each class. The splitting of data is visualised in fig. 2.3.

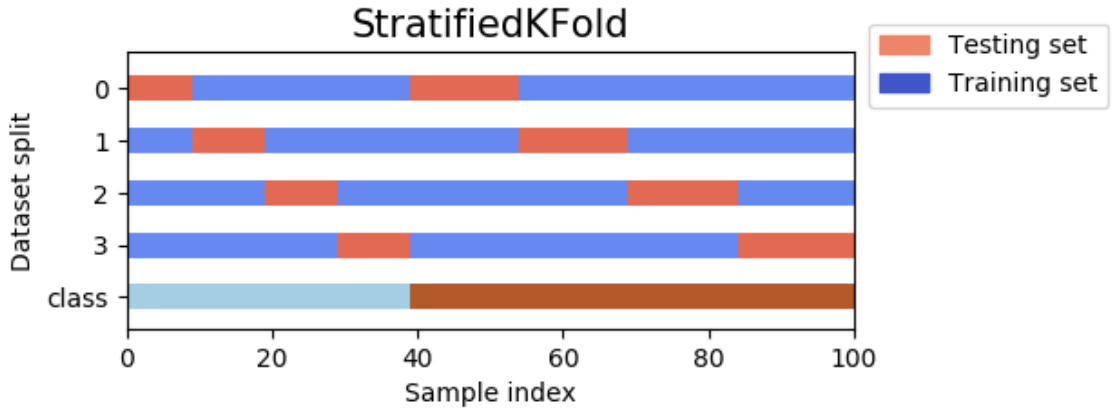

Figure 2.3: Visualisation of the Stratified k-fold. Class represents the visualisation of the dataset when considering the classification label

The final grid is then formed from all the possible combinations of the specified hyperparameters and the training sets obtained from the stratified k fold. The gridsize from eq. (2.13) is thus extended to the following size:

$$gridsize = h_1 * h_2 * k \tag{2.14}$$

Where h_1 and h_2 are the number of hyperparameters and k is the number of folds. A trained model is then estimated for every point int the grid using eq. (2.10), for different $\mathbf{X}$, ℓ_1, and ℓ_2. There are then k learned models for every combination of ℓ_1 and ℓ_2.

2.4.3. Choice of Hyperparameter

To determine the best choice of hyperparameters, each trained model is cross-validated against k cross-validation folds. This results in k scores for every combination of ℓ_1 and ℓ_2, which are averaged, resulting in a single score for each combination of ℓ_1 and ℓ_2. The best score indicates the best choice of hyperparameter for the specific training set. Using this score, the model is trained again on the entire training set, resulting in feature selection. Recall that this procedure is done i times, since the dataset is split i times as described in section 2.3.

3

Literature Review

In the previous chapter, the theory behind the feature selection procedure was presented. As stated in chapter 1, the research question was:

> *What is required to design a GPU program, that does parallel feature selection on high-dimensional datasets to predict prospectivity of hydrocarbons, with the Elastic Net algorithm, using a grid search to find the best combination of of hyperparameters, and what is the achieved speedup compared to a CPU implementation?*

To help answer the research question, this literature review explores the current research on parallel machine learning algorithms on the CPU and GPU. This review comprises the following sections:

1. Firstly, the literature on the range of Machine Learning algorithms that are implemented on the GPU is reviewed. This is done to construct an overview of the available research on general GPU implementations concerning Machine Learning, and which Machine Learning algorithms receive the most attention for implementation on the GPU.

2. Secondly research on parallel Elastic Net on the CPU is reviewed. Since the Elastic Net minimalisation (eq. (2.11)) is solved with a Coordinate Descent Method (CDM), this review contains research on parallel CDM's (PCDM) on the CPU as well. Knowledge gained from available research on this topic can be used for EN-GPU.

3. Thirdly a review of the research on implementations of PCDM on the GPU (PCDM-GPU) and EN-GPU implementations is given.

4. Fourthly, (the absence of) available research on EN-GPU and PCDM-GPU in a grid search setting is reviewed. This is required to either declare a gap in the literature, or otherwise find an opportunity for improvement.

5. Finally, the literature review is summarized and a gap in the literature is declared.

3.1. Machine Learning on the GPU

There is plenty research on the topic of utilizing the power of the GPU to reduce the training time of various Machine Learning algorithms. Most of the research focuses on providing a toolbox to develop machine learning models using tensors (N-dimensional matrices) at its core, which generally refers to the Neural Networks model. Besides literature on Neural Networks concerning GPU's, there is literature on implementations of the Support Vector Machine (SVM) and Random Forest Trees on the GPU.

The literature concerning tensor-based Neural Networks GPU implementations is mostly about toolboxes, which have as their goal to ease the development for scientists when training a Neural Network on large data sets (referred to as big data). These environments use the GPU to reduce the training time significantly when working on such large datasets. MXNet [7], Torch7 [8], and Tensorflow [3] are examples of such environments. The speedup attained by these implementations vary widely as a result of the varying problem size (for varying data size). When the problem size is small, the GPU is outperformed by CPU implementations (as expected), but for larger problem sizes, the GPU quickly outperforms the CPU. In the case of Torch7, the speedup is 2-3x when looking at their largest problem sizes and comparing their GPU implementation (runnin on NVIDIA GTX 460) vs their OpenMP implementation (running on Intel i7 950, 4 cores), as can be seen in fig. 3.1. The reason that GPU's are extensively used for Neural Networks, is due to the use of tensors to represent data in Neural Networks, and the many matrix-matrix multiplications performed on these tensors.

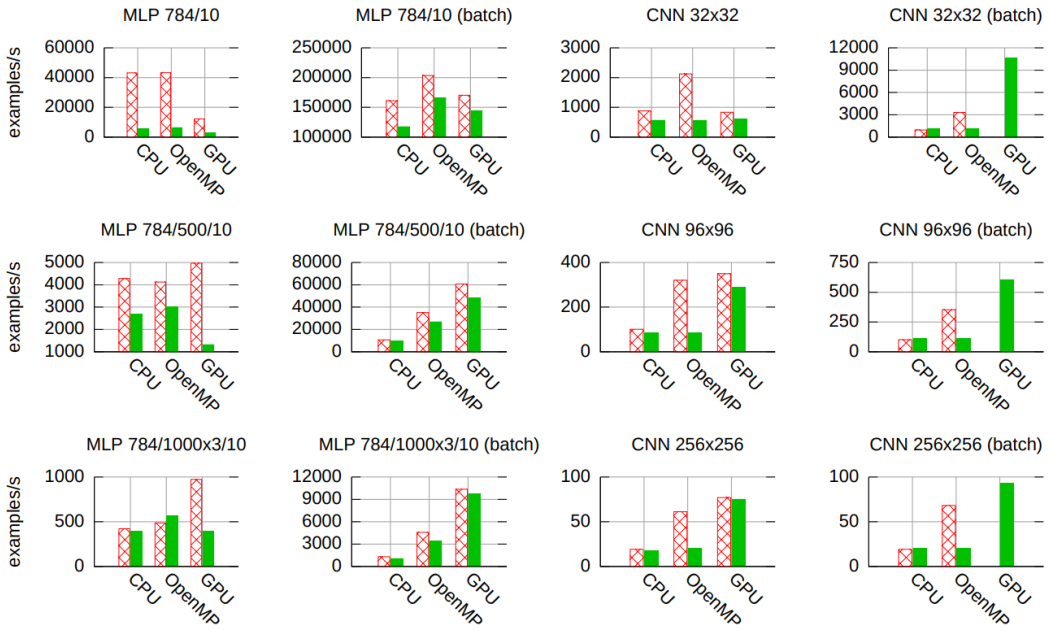

Figure 3.1: Neural Networks (GPU): Benchmarks of Torch7 (red stripes) versus Theano (solid green), while training various neural networks architectures with SGD. Considered are a single CPU core, OpenMP with 4 cores and GPU alternatives. Performance is given in number of examples processed by second (higher is better). "batch" means 60 examples were fed at the time when training with SGD. Batch convolutions for Torch7 are not using CUDA (yet). *Figure 2 in [8].*

Besides Neural Networks, Support Vector Machines is a well studied topic for implementation on the GPU. [4] proposes to use the GPU to perform the computation of the kernel matrix eq. (3.1):

$$\Phi(\mathbf{x},\mathbf{y}) = \exp^{-\gamma||\mathbf{x}-\mathbf{y}||_2^2} \tag{3.1}$$

Where $\mathbf{x} = \mathbf{x}_i, \mathbf{y} = \mathbf{x}_j$ are two training vectors (two samples) from a certain dataset $\mathbf{X}$. $||\mathbf{x}-\mathbf{y}||^2$ can be written as:

$$||\mathbf{x}_i - \mathbf{x}_j||_2^2 = \mathbf{x}_i \bullet \mathbf{x}_i + \mathbf{x}_j \bullet \mathbf{x}_j - 2\mathbf{x}_i \bullet \mathbf{x}_j \tag{3.2}$$

However, when $\mathbf{Z} = \mathbf{X}\mathbf{X}^T$, eq. (3.2) can be rewritten to:

$$||\mathbf{x}_i - \mathbf{x}_j||_2^2 = \mathbf{Z}_{ii} + \mathbf{Z}_{jj} - 2\mathbf{Z}_{ij} \tag{3.3}$$

Equation (3.3) results in a lookup table in the form of matrix for every possible i and j, which is generated using operations that can easily be performed on a GPU. After the calculation of this matrix lookup table, it is moved back to the host (CPU) and the rest of the calculations for the SVM are performed on the CPU. As can be seen in fig. 3.2 and fig. 3.3, for large problem sizes the speedup rises to 70x for the largest problem size.

Random Forest Trees have gotten a lot of attention as well. [19] proposes *"to transform the forest's data structure from a list of binary trees to a 2D texture array"* (a CUDA memory allocation mechanism) and perform

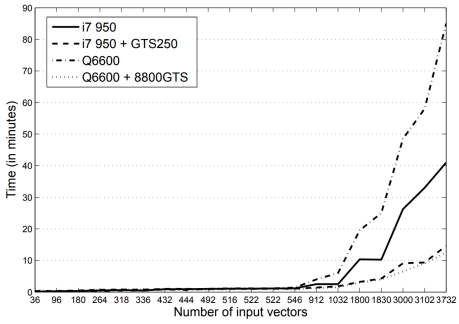

Figure 3.2: SVM (GPU): Processing times for input vectors of size 500 *Figure 2 in [4]*

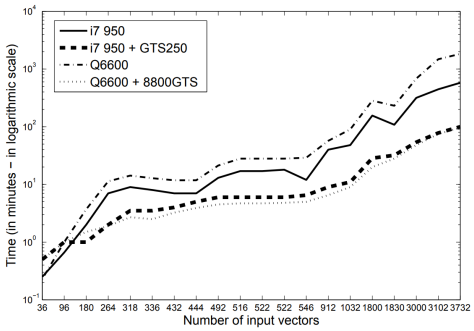

Figure 3.3: SVM (GPU): Processing times for input vectors of size 6000 *Figure 3 in [4]*

Resolution (pixels)	Output Mode	Trees	Classes	CPU (ms)	GPU (ms)	Speed-up (×)
320 × 240	`TreeLeaf`	1	N/A	70.5	0.75	94
320 × 240	`ForestLeaves`	4	N/A	288	3.0	96.1
320 × 240	`Distribution`	8	4	619	5.69	109
320 × 240	`ArgMax`	8	23	828	6.85	121
640 × 480	`TreeLeaf`	1	N/A	288	2.94	97.8
640 × 480	`ForestLeaves`	4	N/A	1145	12.1	95.0
640 × 480	`Distribution`	8	4	2495	23.1	108
640 × 480	`ArgMax`	8	23	3331	25.9	129

Figure 3.4: Random Forest (GPU): Timings for evaluating a forest of decision trees. The GPU implementation evaluates the forest in about 1% of the time required by the CPU implementation. *Figure 10 in [19].*

the Boolean test on the GPU. This specific implementation runs entirely on the GPU and reached a speedup of 100 for object recognition compared to the CPU, as can be seen in fig. 3.4

There are two central themes in the literature concerning Machine Learning on GPU:

- A large problem size. Generally the implementations on the GPU for Neural Networks, SVM, and Random Forest Trees all require large data sets as training sets to be beneficial. With the current trend of "Everything is Big Data", the data sets are getting larger and larger. This is one of the main reasons that the amount of literature on implementing machine learning algorithms on the GPU is increasing the last decade.
- The algorithms all extensively use scalar, vector, vector-vector, matrix-vector, and/or matrix-matrix operations. This property makes implementation on a GPU interesting, since when doing these operations (on a large data set), a GPU outperforms a CPU.

3.2. Parallel Elastic Net and Parallel Coordinate Descent Methods on CPU

To explore opportunities for parallel Elastic Net, it is required to look at the research that has been done in the parallelization of the Elastic Net model on the CPU, since the characteristics that are exploited to compute Elastic Net in parallel on the CPU, can almost certainly be exploited on the GPU. Since Sklearn solves the eq. (2.11) with a Coordinate Descent Method (CDM), besides looking for specific implementations of the Elastic Net, parallel implementations of general CDM's should be investigated as well.

There is only one paper that discusses a parallel implementation on the CPU specific to the Elastic Net model, however since that paper implements this directly on the GPU as well, it will be discussed in section 3.3.

There are several promising papers on PCDM's across features, to increase the speedup. Although the papers only discuss PCDM for Lasso, it can be interesting for Elastic Net as well. [18] proposes to randomly select a feature to be updated each iteration, instead of going trough the features cyclical. Selecting the features at random has been implemented in the Sklearn (CPU) version. The method of choosing the features at random is referred to as *Shooting* in later papers. [6] improves upon *Shooting* by proposing *Shotgun*: to divide the feature into random blocks of features, and update all features in one block in parallel across multiple CPU cores. Implementing Shotgun for Elastic Net changes algorithm 1 to the following algorithm:

Data: $\mathbf{y}, \mathbf{X}, \mathbf{w}, \ell_1, \ell_2, p_{parallel}$
Result: $\mathbf{w}$
$\mathbf{r} = \mathbf{y} - \mathbf{X}\mathbf{w}$;
while not converged or reached max iterations **do**
  $\delta w_{max} = 0$;
  $w_{max} = 0$;
  for $k = 0$ **to** $p/p_{parallel}$ **do**
    In parallel *on* $p_{parallel}$ *processors* **do**
      Choose $i \in \{1, \ldots, p\}$ uniformly at random;
      $w_{old} = w_i$ $\mathbf{r} = \mathbf{r} + w_i \mathbf{x}_i$;
      $t = \mathbf{r} \bullet \mathbf{x}_i$;
      $w_i = \texttt{sign}(t) \frac{\max(|t| - \ell_1, 0)}{||\mathbf{x}_i||_2^2 + \ell_2}$;
      $\delta w_{max} = \texttt{max}(\delta w_{max}, |w_i - w_{old}|)$;
      $w_{max} = \texttt{max}(w_{max}, |w_i|)$;
    end
    $\mathbf{r} = \mathbf{y} - \mathbf{X}\mathbf{w}$;
  end
  `Convergence check`
end

Algorithm 3: Shotgun Coordinate Descent for Elastic Net

In algorithm 3 $p_{parallel}$ is a problem and hardware dependent setting which is equal to the amount of cores, or the maximum amount of features to be updated in parallel without causing the fitting process to diverge from the solution.

[15] takes *Shotgun* even further and updates all the blocks in parallel as well on a distributed (CPU) system. The speedup of these *Shotgun* and distributed *Shotgun* implementations are dependent on the amount of separability of the features. Higher separability allows for more speedup. As such the speedup differs greatly per dataset. The speedup ranged from 1 (no speedup) to over 10 for different datasets.

3.3. Elastic Net on the GPU

There appears to be no direct implementation of Elastic Net on the GPU. When reviewing literature specifically for implementations of the Elastic Net, [21] proposes to reduce Elastic Net to an SVM, eq. (2.11) is rewritten to:

$$\min_{a_i \geq 0} ||\hat{\mathbf{Z}}\boldsymbol{\alpha}||_2^2 + \lambda_2 \sum_{i=1}^{2p} \alpha_i^2 \quad s.t. \quad \sum_{i=1}^{2p} \alpha_i = |\boldsymbol{\alpha}^*|_1 \tag{3.4}$$

Where $\hat{\mathbf{Z}} = (\hat{y}_1 \hat{\mathbf{x}}_1, \ldots, \hat{y}_m \hat{\mathbf{x}}_m)$ and $\mathbf{w} = \boldsymbol{\alpha}^* / |\boldsymbol{\alpha}^*|_1$. For the precise reformulation, see page 3211 & 3212 in [21].

Using eq. (3.4), both a CPU and a GPU version is implemented. Both perform in most cases significantly faster than other (CPU) implementations of Elastic Net, as can be seen in fig. 3.5. However, this Elastic Net to SVM implementation is compared against CPU versions of solving Elastic Net. Therefore it is not clear how it would compare against a pure Elastic Net GPU implementation. It does not take a grid search into account

as well.

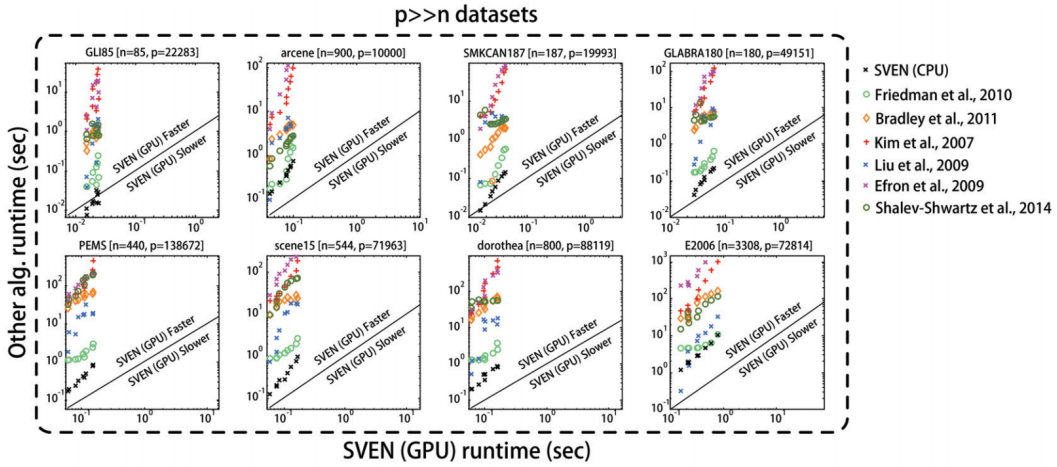

Figure 3.5: Elastic Net to SVM (GPU): Training time comparison of various algorithms in $p >> n$ scenarios. Each marker compares an algorithm with SVEN (GPU) on one (out of eight) datasets and one parameter setting. The X,Y-axes denote the running time of SVEN (GPU) and that particular algorithm on the same problem, respectively. All markers are above the diagonal line (except SVEN (CPU) for GLI-85), indicating that SVEN (GPU) is faster than all baselines in all cases. *Figure 2 in [21]*

One paper ([14]) was found on the implementation of a PCDM on the GPU, however the achieved speedup reported in this paper was so unbelievable (over 20 million(!) compared to the benchmark), in combination with the relative low amount of references, that this paper was only glanced upon.

3.4. EN Grid Search on the GPU

[5] claims to have implemented an Elastic Net based Grid Search on the GPU, however it is not possible to reproduce the steps taken, since the steps are not explained. To the best of my knowledge, no other papers exist on this topic.

3.5. Research Gap

There has been much research on implementing machine learning algorithms on the GPU, however this research focuses mainly on Neural Networks, SVM's, and Random Forest Trees. This has two reasons:

- Machine Learning generally involves creating a model based on a very large data set, and GPU's generally work better on large problem sizes.
- The mathematical algorithms for Neural Networks, SVM's, and Random Forest Trees all extensively use scalar, vector, vector-vector, matrix-vector, and/or matrix-matrix operations. These kind of operations are (among others) when a GPU outperforms a CPU (when done on a large data set).

The literature concerning parallel Elastic Net and PCDM on the CPU is promising. The algorithms *Shooting* and *Shotgun* could benefit from a GPU implementation as well, and should therefore be explored.

The amount of literature on implementing Elastic Net on the GPU is limited. The most promising ([21]) rewrites Elastic Net to an SVM and implements this SVM on the GPU. There are no promising papers on the implementation of PCDM on the GPU.

For the combination of a Grid Search and Elastic Net on the GPU, one paper was found. From [5], however, it is not possible to reproduce the steps implementing their solution on the GPU.

To conclude, there appears to be a gap in the literature concerning parallel Elastic Net within a grid search on the GPU. As such, the next chapter will describe a parallel GPU design for the Elastic Net algorithm in combination with a grid search.

4

Implementation

The focus of this chapter is to describe the original implementation and the benchmarks identifying the bottleneck(s), and the required changes to the original program, resulting in the implementation on the GPU. This is to answer part of the research question, namely:

> *What is required to design a GPU program, that does parallel feature selection on high-dimensional datasets to predict prospectivity of hydrocarbons, with the Elastic Net algorithm, using a grid search to find the best combination of of hyperparameters, . . . ?*

4.1. Experimental Setup

4.1.1. Hardware

For the comparison experiment, the following two computing platforms were used:

1. A laptop, with a 4-core I5-8300H CPU running at 2.3-4.0 GHz with 8 GB RAM, and a GTX 1050 TI with 4 GB VRAM and 6 Streaming Multiprocessors (SM) capable of computing 1981 GFLOPS (Giga Floating point Operations Per Second) single precision, as a GPU.
2. A compute server, with a 14-core I9-7940X CPU running at 3.1-4.3 GHz with 128 GB RAM, and 6 RTX 2080 TI's with 11 GB VRAM and 64 SM's capable of computing 11750 GFLOPS single precision, as GPU's.

4.1.2. Software

The software for the GPU implementations is written in C++ and compiled using the Nvidia CUDA Compiler (NVCC). The choice to write the GPU implementation in CUDA is due to the compatibility with the GPU's, which are manufactured by Nvidia, and because there is plenty of documentation available for CUDA. Another reason is the availability of the CUDA Basic Linear Algebra Subroutines (CUBLAS) library, which is an implementation of the BLAS library on CUDA. CUBLAS contains almost all vector-vector, matrix-vector, and matrix-matrix manipulations and is highly optimized for the execution of such operations on a GPU. The CUDA workflow is illustrated in 4.1.

The software is compiled to a shared library, so that the program can be easily integrated in the existing python code from Biodentify and Sklearn.

4.2. Elastic Net on the CPU

Before discussing the implementation of EN-GPU, Elastic Net on the CPU is explained. This is done to highlight where in the program changes should be made. As mentioned in section 2.1, the choice for the feature

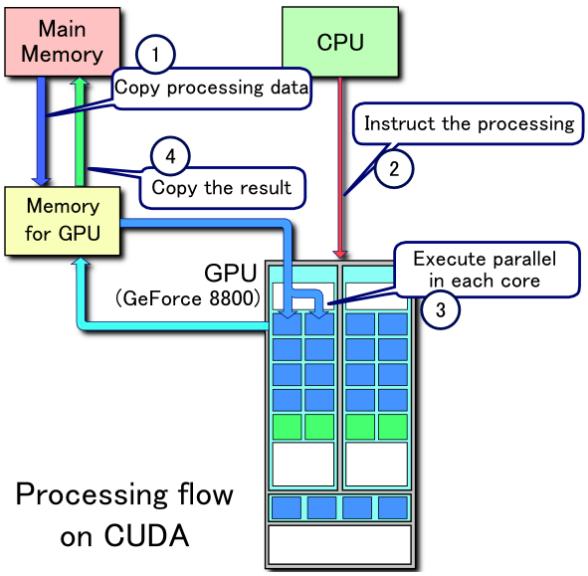

Figure 4.1: Example of CUDA processing flow, figure 2 from [16]

selection algorithm is the Elastic Net. The implementation on the CPU uses Sklearn and broadly consists of the following 5 steps:

1. Preprocessing. Arguments that can be set by the user are declared at the start of the program. After this the data is loaded and put in the required format, which requires scaling and setting it sparse.
2. Data Splitting: Splits the data into i random train-test sets, using a stratified shuffle split.
3. Grid Search: Create the grid to search on, using the user input parameters for different α's, different ℓ_1-ratio's and number of folds.
   (a) Fit and Score: Fits on the train folds using the EN algorithm. The result of this fit is then scored on the corresponding validation fold.
4. Feature Selection: Using the scores from the previous step, the best combination of hyperparameters is chosen. The training data from step 2 is fitted with EN using this combination of hyperparameters and this fit results in the feature selection.
5. Model Validation: The learned model, using the features from step 5, is validated against the test set from step 2.

There are three loops in these steps, which can be seen in fig. 4.2:

1. The outer loop: Step 2 creates i data sets. Step 3, 4, and 5 are repeated for every set.
2. The grid loop: Step 3 creates a grid. The program iterates over the grid and thus step 3a is repeated for every point in the grid. This can be done in parallel on CPU cores.
3. The inner loop: As discussed previously, Elastic Net is solved using a CDM, which is an algorithm that iterates over the features until convergence is reached.

4.2.1. Preprocessing

A number of arguments can be set for the program, however only the essential ones are named:

- The amount of runs or splits (i) for the stratified shuffle split,

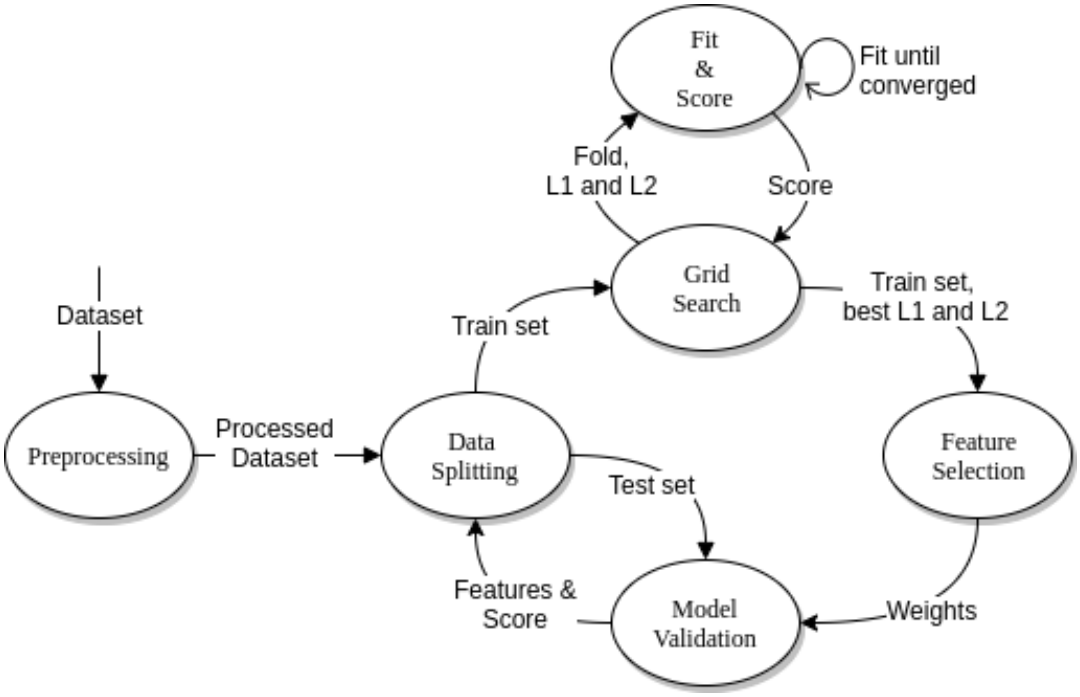

Figure 4.2: Current C implementation. Fit and score is done on P processors in parallel.

- The selected hyperparameters (ℓ_1 and ℓ_2)to run the grid search on.
- The number of folds (n_{folds}) to be used for the crossvalidation in the grid search.
- The number of jobs (n_{jobs}), how many CPU cores should be used.

After the retrieval of the arguments, the data is loaded and then scaled. This is done to normalize the data. Before continuing to the next step of the program, the data is put into a sparse column format. This is done for two reasons: to reduce the memory footprint and to speed up the EN fit.

4.2.2. Data Splitting

The outer loop iterates over the different $\mathbf{X}_{train}$ sets created

The current implementation has as inputs:

- The dataset, consisting of matrix $\mathbf{X}$ and corresponding solution vector $\mathbf{y}$
- The amount of splits i

It then splits the data using a stratified shuffle split into i random train-test sets and iterates over the sets. After the data is split, one such train-set data set is fed to the next part of the program, grid creation.

The following is passed to the part Grid Creation:

- Train set: $\mathbf{X}_{train}$ and $\mathbf{y}_{train}$

From Grid Creation the following is received:

- The selected features, in vector $\mathbf{w}$

4.2.3. Grid Search

The grid is created from the selected hyperparameters (ℓ_1 and ℓ_2) and the k cross-validation sets, which number is equal to the number of folds (n_{folds}). It can be seen as a 3d grid with on the axes ℓ_1, ℓ_2, and k respectively. The hyperparameters ℓ_1, ℓ_2 and k (equal to n_{folds}) are user-defined as mentioned in the preprocessing step. After the grid is created, each point is queued and distributed using a job manager. When a thread is done with computing Fit and Score for a point in the grid, the next point in the queue will be computed. This is a form of MIMD parallelism, where nothing in memory is shared between threads.

After the entire grid has been scored using Fit and Score, the combination of ℓ_1 and ℓ_2 that scores best on average across k is chosen as the most optimal combination and is passed on to feature selection.

Fit and Score

The data in every grid point is fit with the use of the Elastic Net algorithm and then scored. Fitting is done using algorithm 1 and it is checked for convergence according to algorithm 2. After creating a fitted model, it is scored against its cross-validation fold using the default ROC AUC score, which is the Receiver Operating Characteristic, Area Under the Curve. This is one of many scoring methods to determine the accuracy of a model, the explanation and choice of scoring method is beyond the scope of this thesis.

4.2.4. Feature Selection

Using the most optimal combination of hyperparameters from the grid search, one more Fit is done using the Elastic Net algorithm, but this time on the train set as determined by Data Splitting. This model results in a sparse set of weights ($\mathbf{w}$), which is passed to Model Validation.

4.2.5. Model Validation

Using the weights from the model trained in Feature Selection, the model is validated against the test set from Data Splitting. This validation results in a score, which is then stored along with the features.

4.2.6. Bottleneck

Profiling on the original CPU implementation showed that > 99% of the runtime is spend during the fitting of the model.

4.2.7. Sparse Sklearn

Sklearn has an implementation used for sparse datasets. The sparse implementation skips all zero values, thus increasing computation efficiency. The sparse implementation will be used when the data set is sparse.

4.3. Elastic Net on the GPU

Since almost all computation takes place during the fitting of the model, the focus is on designing a parallel method for the Elastic Net fitting algorithm (algorithm 1) while using the Grid Search to increase the data parallelism. Three different implementations are designed:

- Cyclical Elastic Net, which cycles through the features, exploiting parallelism in sample size and number of hyperparameters.
- Shooting Elastic Net, which randomly picks features, exploiting parallelism in sample size and number of hyperparameters.
- Shotgun Elastic Net, which randomly picks multiple features, exploiting parallelism in sample size, number of hyperparameters, and number of features.

First the cyclic design is presented, and then the changes to accommodate the other designs are presented.

4.3.1. Cyclical Elastic Net

A number of design choices are made when implementing the cyclical Elastic Net on the GPU.

1. Since memory on the GPU is limited, there should be only one dataset in global memory at any time where the kernels read from. This means that only one cross-validation training set is used for training at a time.
2. Memory reads and writes across kernels should be coalesced
3. Kernels will make redundant computations, since knowledge of whether a fit is converged or not, is not known by the kernels.
4. The convergence check is not critical, as it is only done at the end of a fit. It will be left for the CPU to check for convergence.

Data Transfer

To perform the EN algorithm on the GPU, the train data has to be transferred to the GPU. As such, for all the arrays and matrices memory has to be allocated. The largest memory allocation is the train set **X**, other memory requirements scale approximately $n * gridsize$, resulting in a low memory footprint.

After memory is allocated, **X**, **y**, **w** (if warmstarting), ℓ_1, and ℓ_2 are transferred to the GPU from the host.

Initial Values

Before starting the EN fit algorithm, the norms of the columns of **X** are calculated, as well as the initial values for the residuals **R**, which is now a matrix of size $n \times gridsize$. The weights are stored in matrix **W**, with size $p \times gridsize$

Loop over features

The program loops over the features almost the same way as the CPU version (algorithm 1). A cuBLAS kernel performs the residual addition for all samples and fits in parallel. This is done using a matrix-matrix multiplication:

$$\mathbf{R} = \mathbf{X}_i \mathbf{W}_i + \mathbf{R}, \tag{4.1}$$

where $\mathbf{X}_i$ is a $1 \times n$ matrix and $\mathbf{W}_i$ is a $gridsize \times 1$ matrix. This updates all values for **R** for all samples and fits simultaneously.

A second cuBLAS kernel performs the calculation for the dot product for all fits in parallel. This is done using a matrix-vector multiplication:

$$\mathbf{t} = \mathbf{R}^T \mathbf{x}_i, \tag{4.2}$$

where $\mathbf{x}_i$ is the column vector of **X** for feature i. This results in vector `tmp`, a buffer containing all the dot products between the residuals and the samples for one feature, for every fit.

The weight is then updated according to eq. (2.12) using a custom kernel, with a number of threads equal to the number of fits and each thread is updating the feature for a fit, which performs the following update:

$$\mathbf{w}_{thread} = \texttt{sign}(\mathbf{t}_{thread}) \frac{\max(|\mathbf{t}_{thread}| - \ell_1, 0)}{||\mathbf{x}_i||_2^2 + \ell_2}, \tag{4.3}$$

where $||\mathbf{x}_i||_2^2$ has already been precalculated as the norm of the columns of **X**. The custom kernel then updates the maximum absolute change in weights (δw_{max}) and the absolute maximum weight (w_{max}) for each fit, required for the convergence check and then updates the weight vector.

A third cuBLAS kernel then performs the residual substraction, which is essentialy the same as eq. (4.1):

$$\mathbf{R} = \mathbf{X}_i \mathbf{W}_i - \mathbf{R}, \tag{4.4}$$

These kernels are performed untill all features have had their weights updated.

Convergence Check

The convergence check remains almost completely the same as in algorithm 2 and will be executed on the CPU, with the addition of copying values stored on the GPU to the host. In order to check for convergence, the δw_{max} and w_{max} values for each fit are copied from the device to the host. When a certain fit passes the initial *IF* statement as in algorithm 2, the vectors **w** and **r** from matrices **W** and **R** which contain the values for that fit are copied to the host along. This ensures that only values that are required for the convergence check are copied back when they are eligible for a convergence check. The rest of the convergence continues as described in algorithm 2. If a fit has converged, the corresponding vectors **w** and **r** for that fit are set to 0. This is done to avoid redundant computations and cuBLAS has a slight performance gain when multiplying with 0 values.

After convergence for all fits have been reached, the GPU processing ends and the program returns to python where the models can be scored using Sklearn.

4.3.2. Shooting Elastic Net

Designing the Shooting implementation is quite trivial. When taking the cyclical EN, only one minor change has to be made. In eq. (4.1), eq. (4.2), eq. (4.3), and eq. (4.4) i is cyclical from 0 to p, where in shooting i is picked randomly using the *rand* function the library *stdlib*. The required change from Cyclical to Shooting can be seen in listing 4.1 and listing 4.2 respectively.

Listing 4.1: Cyclical Elastic Net

```
...
// Cycles through the features
for(i = 0; i < n_features; i++)
{
    ...
    feature = i;
    ...
}
...
```

Listing 4.2: Shooting Elastic Net

```
...
// Updates n_features features,
// picked at random
for(i = 0; i < n_features; i++)
{
    ...
    feature = rand() % n_features;
    ...
}
...
```

4.3.3. Shotgun Elastic Net

Designing Shotgun Elastic Net is not as trivial as Shooting, the design constriction of memory coalescence plays a big role here. In order to have a speedy kernel, the randomly chosen features should first be copied to a buffer, to have them sequential in memory. This results in an extra memory copy, combined with the increased computation of recomputing the complete residual R after every iteration, enough features should be updated in parallel to be computationally efficient. The amount of features to be updated in parallel is defined as $p_{parallel}$

Recall algorithm 1 and algorithm 2, to extend these algorithms to update multiple features in the same iteration on the GPU, the following changes have been made:

1. $\mathbf{r} = \mathbf{y} - \mathbf{Xw}$ is calculated completely every iteration.
2. A number of random unique features ($p_{parallel}$) have to be selected, which are chosen using the *rand* function on the CPU and then copied to the GPU in a vector **ind** containing the indices of the randomly

selected features with length $p_{parallel}$. The choice has been made to select them on the CPU, since creating a unique vector is faster on the CPU than the GPU. The data from $\mathbf{X}$ which correspond with the chosen features are stored in matrix $\mathbf{X}_{ind}$.

3. $\mathbf{r} = \mathbf{r} + w_i \mathbf{x}_i$ has to be done for multiple features (multiple different i)

4. $t = \mathbf{r} \bullet \mathbf{x}_i$ has to be calculated for multiple features.

5. $f(w_i)$ has to be computed for multiple features.

6. $\mathbf{r} = \mathbf{r} - w_i \mathbf{x}_i$ no longer has to be computed, since the residual is recalculated at the start of every iteration.

Step 2 results in a vector **ind** containing the indices of the randomly selected features with length o. Step 3 and 4 can be rewritten, as substituting $\mathbf{r}$ from step 3 in step 4 results in:

$$\begin{aligned} t &= (\mathbf{r} + w_i \mathbf{x}_i) \bullet \mathbf{x}_i \\ t &= \mathbf{r} \bullet \mathbf{x}_i + w_i \mathbf{x}_i \bullet \mathbf{x}_i \\ t &= \mathbf{r} \bullet \mathbf{x}_i + w_i ||x_i||_2^2 \end{aligned}$$

This is good for the case of multiple features, since $w_i ||x_i||_2^2$ is simply a multiplication with the norm of the columns of X, which can be precomputed and can be easily extended to multiple features. The choice is made to calculate $\mathbf{r} \bullet \mathbf{x}_i$ separately and to incorporate the computation of $w_i ||x_i||_2^2$ in the custom kernel already created for Cyclical Elastic Net. Extending $t = \mathbf{r} \bullet \mathbf{x}_i$ to multiple features results in the following vector-matrix multiplication:

$$\mathbf{t} = \mathbf{r}\mathbf{X}_{ind} \tag{4.5}$$

Where $\mathbf{r}$ is a horizontal vector and $\mathbf{X}_{ind}$ is a $n \times p_{parallel}$ matrix, which is the subset of matrix X containing only the randomly selected features. The resulting vector $\mathbf{t}$ of length $p_{parallel}$ contains values used in eq. (2.12). Now extending eq. (4.5) to allow for multiple fits results in the following matrix-matrix multiplication:

$$\mathbf{T} = \mathbf{R}^T \mathbf{X}_{ind} \tag{4.6}$$

Where $\mathbf{R}^T$ is a matrix of size $gridsize \times n$. The resulting matrix $\mathbf{T}$ is a matrix of size $gridsize \times p_{parallel}$. This matrix is computed using the standard cuBLAS matrix-matrix multiplication.

Now recall eq. (4.3), to allow multiple features, it is changed to a custom kernel that uses $gridsize \times o$ threads ($\beta = fit$ and $\gamma = feature$ from **ind**, is used to differ between the two dimensions used as threads), changing it to:

$$\mathbf{W}_{\beta,\gamma} = \texttt{sign}(\mathbf{T}_{\beta,\gamma} + \mathbf{w}_{\beta,\gamma} ||x_\beta||_2^2) \frac{\max(|\mathbf{T}_{\beta,\gamma}| - \ell_1, 0)}{||\mathbf{x}_\beta||_2^2 + \ell_2}, \tag{4.7}$$

The new weights are immediately updated in their corresponding place in matrix $\mathbf{W}$ containing all the weights. Care was given to update the maximum absolute change in weights (δw_{max}) and the absolute maximum weight (w_{max}) for each fit atomically, to ensure that the values are correct.

However, upon bench marking this method using NVIDIA Visual Profiler, it quickly became clear that updating the complete residual each iteration was infeasible. The time spent computing the residual was over 90% of the time spent on the GPU. To overcome this problem, a new method was devised. Computing the complete residual was replaced by subtracting the difference in the weights of those weights that were chosen in **ind**. Instead of calculating $\mathbf{R} = \mathbf{y} - \mathbf{X}\mathbf{w}$ at the start of each iteration, the residual is updated as follows:

$$\mathbf{R} = \mathbf{R} - \mathbf{X}_{ind} \Delta \mathbf{w}_{ind} \tag{4.8}$$

This improved performance greatly.

4.3.4. Multithreading

After all the performance improvements it was discovered that, for increasingly larger gridsizes, the convergence check became a bottleneck. This is mainly due to the fact that the convergence check is being handled by a single thread (or core) on the CPU, so when multiple fits are eligible for a convergence check concurrently, they are processed sequentially. To overcome this bottleneck, instead of handling the convergence check on a single thread, a thread is created for each fit that is eligible for a convergence check. This will result in negative performance on smaller gridsizes, since there is some overhead involved when there are very few concurrent convergence checks. However, for larger gridsizes this will improve performance.

The multithreading is implemented as such:

1. When a fit is eligible for convergence checking, the corresponding r and w for that fit is copied back to the host asynchronously. This means that the host can continue checking for other eligible fits while the memory transfer is underway.
2. After all fits have been checked for eligibility for convergence checks, for those fits that are eligible, check per fit if the memory transfer is completed and then spawn the thread for that fit.
3. Convergence check is now done as normal, but in parallel whenever possible.
4. Wait until all convergence checks are done, and continue to the next iteration (or exit when all fits have converged).

5

Results

This chapter reports the results of the different implementations. First the implementations are benchmarked against eachother on several different randomly generated dense datasets with varying feature-, sample-, and gridsize.

Secondly, the GPU shotgun (GPU-SG) implementation is benchmarked against the shooting CPU implementation from Sklearn on the sparse Biodentify dataset for an increasing gridsize.

Finally, the effect of multithreading is investigated.

5.1. Dense Dataset

To benchmark the implementations on a dense dataset, 3 different parameters are varied:

- The number of features, starting at 1.000 and doubling to reach 32.000.
- The sample size, starting at 1.000 and doubling to reach 128.000.
- The number of duplicate combinations of hyperparameters in the grid, starting at 10 and doubling to reach 5120, and therefore increasing the size of the grid.

The number of cross-validation sets is fixed at two and the number of jobs at eight (equal to the amount of virtual cores). The number of parallel feature updates for shotgun is fixed at two. The dataset was generated using *make_classification* from Sklearn, with the number of informative and redundant features set to 100 along with the settings mentioned above. After generating the dataset, it is split in a train/validation set of 80/20 before entering the datasplitting part of the program. Cyclical and shooting have the exact same iteration time and will be combined in this section. Note that one iteration of cyclical, shooting, and CPU is a loop over the features that is equal to the amount of features, while one iteration of shotgun is a loop over the features equal to the amount of features divided by the number of features done in parallel. For these generated datasets, a maximum of two features could be computed in parallel. If more features were computed in parallel, convergence was no longer guaranteed.

5.1.1. Varying Number of Features

Figure 5.1 shows that the CPU greatly outperforms the GPU. The computation time scales linearly with the number of features for both the CPU and GPU, which is to be expected, since the cyclical and shooting implementations do not exploit data parallelism across features. Only one feature from matrix $\mathbf{X}$ is computed on at any given time on the GPU. Adding more features will only linearly increase the amount of computations since the residual has to be updated between each feature update, blocking parallel computation across

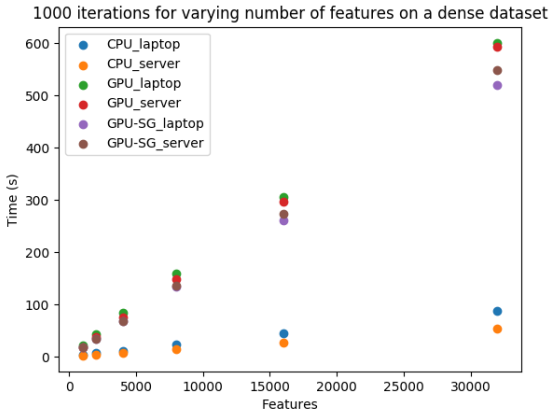

Figure 5.1: Time to compute 1000 iterations for dense dataset, for doubling number of features with samples size set at 1000 and gridsize set at 10.

features. Interestingly, for two features in parallel, shotgun has a better performance than cyclical and shooting. For increasing features on this dataset, the GPU will never outperform the CPU in hardware setup 1 and hardware setup 2.

5.1.2. Varying Sample Size

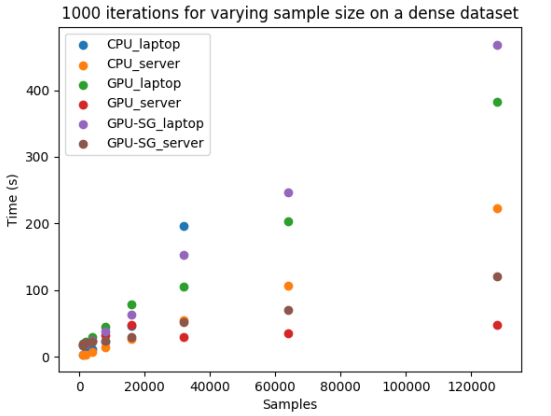

Figure 5.2: Time to compute 1000 iterations for dense dataset, for doubling sample size with number of features set at 1000 and gridsize set at 10. Computation for sample sizes larger than 32.000 were not possible for the CPU due to a lack of memory.

Figure 5.2 shows that the computation time for the CPU is linear up to 32.000 samples, at which point the RAM memory was almost completely full Therefore the computation time for 1000 iterations on 32.000 samples is significantly longer than linear. Larger sample sizes could not be tested due to a lack of RAM memory. The computation time for the GPU increases with larger sample sizes, however it does not increase linearly with respect to the amount of samples, but rather sublinear. This is to be expected, since all GPU implementations exploit data parallelism across samples. Extra samples will allow for additional parallel computations, which works especially well on the GPU since extra samples add an extra element to vector $\mathbf{X}_i$ and an extra row to matrix $\mathbf{R}$, which are used in the kernels that compute eq. (4.1), eq. (4.2), and eq. (4.4). For shotgun, a row is added to the matrices $\mathbf{X}_{ind}$ and $\mathbf{R}$, which are used in the kernels to compute eq. (4.6) and eq. (4.8). For two features in parallel, shotgun has a worse performance compared to cyclical and shooting for both hardware setups. The GPU outperforms the CPU at 32.000 samples and more for both hardware setups.

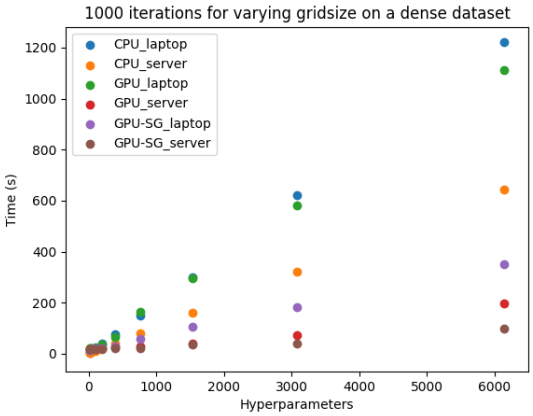

Figure 5.3: Time to compute 1000 iterations for dense dataset, for doubling size of the grid with features and samples size fixed at 1000

5.1.3. Varying Gridsize

Figure 5.3 shows that the computation time for the CPU is again linear, as expected. Since the implementations for the GPU exploits data parallelism across the grid, from a gridsize of 1600 and more, the GPU outperforms the CPU. Similar to the varying samples, extra combinations of hyperparameters add an extra element to the vectors $\mathbf{W}_i$ and $\mathbf{t}$, and an extra column to matrix $\mathbf{R}$, which are used in the kernels to compute eq. (4.1), eq. (4.2), eq. (4.3), and eq. (4.4). For shotgun, an extra column is added to matrices $\mathbf{W}$, $\mathbf{T}$, and $\mathbf{R}$, which are used in the kernels to compute eq. (4.6), eq. (4.7), and eq. (4.8). The GPU-SG implementation outperforms the cyclical and shooting GPU and CPU implementations for both hardware setups.

5.2. Sparse Data Set

This section covers the results on the Biodentify sparse dataset using the GPU-SG implementation. The dataset that is used is a sparse dataset, containing 1650 samples with 82410 features, and it has a sparsity of 0.975. Of those samples, 1526 are used for training and 124 are used for testing. Since a cross-validation of two is used, this means that every cross-validation training set contains 763 samples.

Since the dataset is sparse, the sparse implementation of Sklearn is used. *The sparse implementation of Sklearn on this dataset results in a speedup of roughly five compared to the dense implementation of Sklearn.* The cyclical and shooting GPU implementations are no longer investigated, since it was clear that using the results of fig. 5.1, using 82410 features and a sample size of 1650, the cyclical and shooting GPU implementations would not outperform the CPU when it is not possible to exploit parallelism across features. The convergence times are compared to the sparse Shooting CPU implementation of Sklearn, since Shotgun is random.

The tests on the sparse dataset show the following:

Firstly, the influence of the amount of parallel feature updates done in parallel on the performance is investigated to see if GPU-SG can outperform the sparse CPU implementation. The amount of parallel feature updates is doubled until convergence of the model is no longer guaranteed, this is required for the tests that follow this one in this chapter.

Secondly, the influence of increasingly larger grid with a fixed number of iterations is investigated. Varying sample size and feature size are not investigated on this dataset.

Thirdly, the total convergence time required to reach convergence is investigated for a varying gridsize.

5.2.1. Parallel Feature Updates

The convergence time is measured for fixed and duplicated α and $\ell_{1-ratio}$, an increasing number of features to be updated in parallel, and a fixed random seed for Shotgun. The random seed is fixed to ensure comparability between different runs. These tests are done for two reasons:

- To investigate the performance with varying number of feature updates done in parallel.
- To find the maximum amount of features to be updated in parallel while guaranteeing convergence.

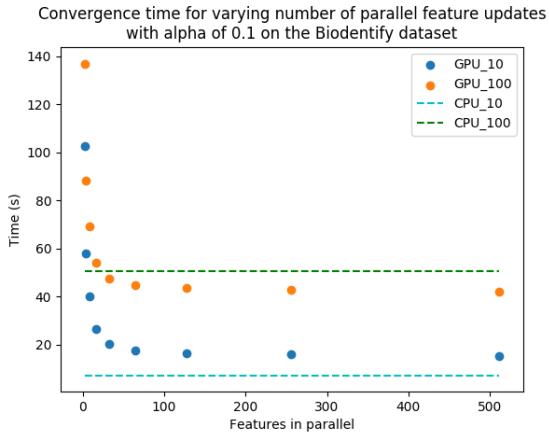

Figure 5.4: Convergence time on the Biodentify dataset for increasing number of features done in parallel for both a grid of 10 and a grid of 100 for the shotgun implementation compared to the sparse CPU implementation. α and $\ell_{1-ratio}$ are fixed at 0.1 and 0.5 respectively. With more than 512 parallel features, convergence is no longer guaranteed.

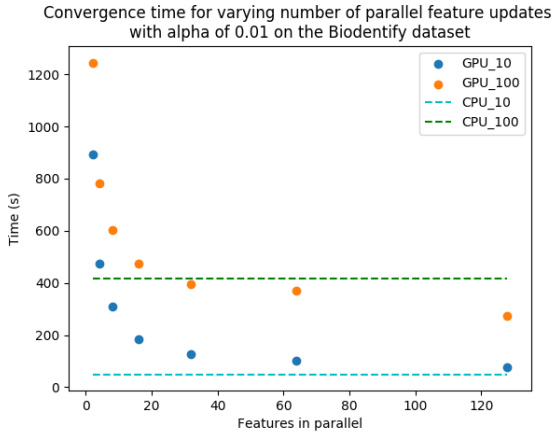

Figure 5.5: Convergence time on the Biodentify dataset for increasing number of features done in parallel for both a grid of 10 and a grid of 100 for the shotgun implementation compared to the sparse CPU implementation. α and $\ell_{1-ratio}$ are fixed at 0.01 and 0.5 respectively. With more than 128 parallel features, convergence is no longer guaranteed.

In fig. 5.4 and fig. 5.5 it is clear that the convergence time decreases with increasing number of features to be updated in parallel. As already shown in the previous section, the performance of Shotgun increases when the convergence time grows larger, either due to an increasingly larger grid or due to an increasing number of iterations needed to reach convergence. It should be noted that the maximum amount of features to be updated in parallel is not only dependent on the dataset, but on the the choice of hyperparameters as well since the convergence is no longer guaranteed when increasing the amount of parallel feature updates beyond 512 for $\alpha = 0.1$ and beyond 128 for $\alpha = 0.1$ as shown in fig. 5.4 and fig. 5.5.

5.2.2. Varying Gridsize

With the amount of parallel feature updates to guarantee convergence found, the influence of varying the size of the grid can be investigated. Figure 5.6 shows that the CPU implementation on hardware setup 1

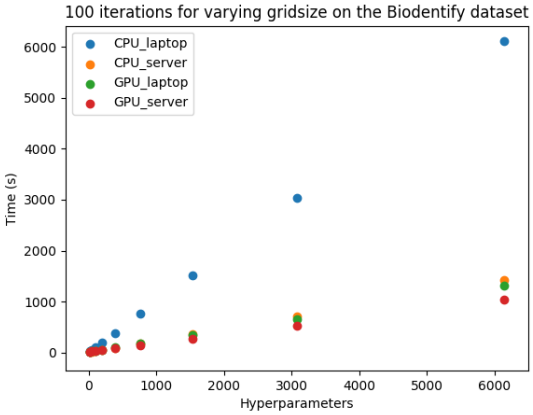

Figure 5.6: Time to compute 100 iterations for the biodentify dataset, for doubling gridsize with amount of parallel feature updates set to 128. This compares the shotgun GPU implementation to the sparse CPU implementation

is outperformed by all the other implementations. To make the differences between the other three tests clearer, a new figure (5.7) without the CPU implementation on hardware setup 1 is created. Figure 5.7 shows that the GPU implementation on both hardware setups performs better than the CPU implementation on the server. It should be noted that due to the low amount of iterations, the time spent in convergence check is comparatively high.

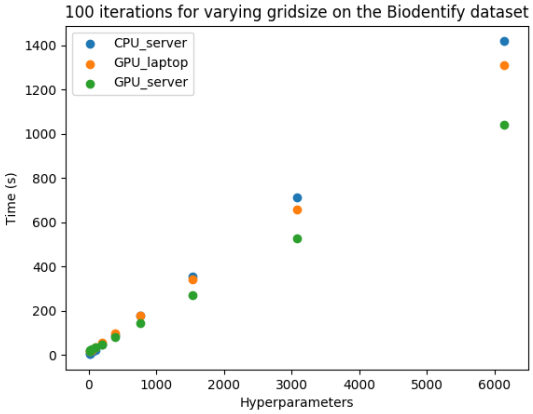

Figure 5.7: Time to compute 100 iterations for the biodentify dataset, for doubling gridsize with amount of parallel feature updates set to 128. This compares the shotgun GPU implementation to the sparse CPU implementation. CPU laptop implementation omitted.

5.2.3. Gridsearch Convergence Time

The convergence times of the GPU-SG and Shooting CPU implementations are investigated for 3 different gridsizes, for both hardware setups. The results are all plotted in boxplots with the boxes at 25-75 en the whiskers at 5-95. The CPU tests are all done 10 times. The GPU-SG tests are done 50 times for fig. 5.8 and fig. 5.9, and 25 times for fig. 5.10. The following settings were used:

- $\ell_{1-ratio} = [0.2, 0.5, 0.9]$.
- $\alpha = \texttt{numpy.logspace}(-4, -2, \texttt{num} = ?, \texttt{base} = 10)$. For the number of α's see the figures.

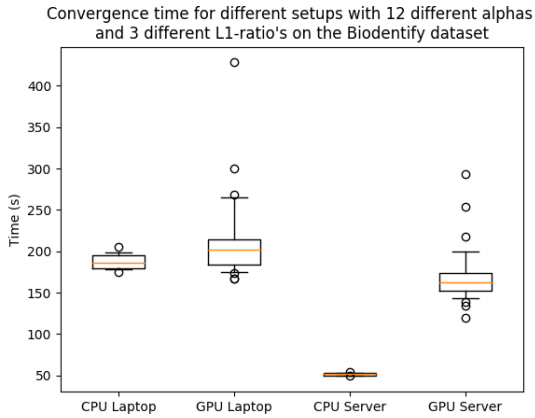

Figure 5.8: Convergence times on the Biodentify dataset for the shotgun implementation compared to the sparse shooting CPU implementation for both hardware setups for 12 different α's.

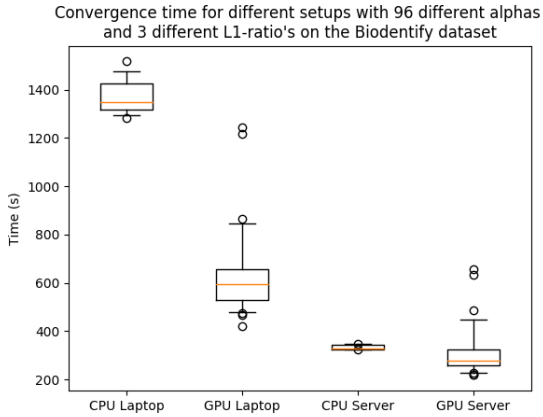

Figure 5.9: Convergence times on the Biodentify dataset for the shotgun implementation compared to the sparse shooting CPU implementation for both hardware setups for 96 different α's.

- 128 features are updated in parallel.

Figure 5.8, fig. 5.9, and fig. 5.10 show the following:

- For a small gridsearch, the GPU-SG does not outperform the CPU implementation for both hardware setups.
- For larger gridsearch, the GPU-SG will outperform the CPU implementation for both hardware setups.
- The spread in convergence times is larger for the GPU-SG than for the CPU implementation.
- The speedup is less than would be expected when looking at fig. 5.3.

5.3. Multithreading

This section covers the findings when profiling the program with NVIDIA Visual Profiler to identify the new bottleneck, introduced due to a combination of scaling up the problem and performance gains in the fitting computations.

The previous tests for the sparse Biodentify dataset are then done again with multithreading implemented, to show the gain in performance on larger gridsizes.

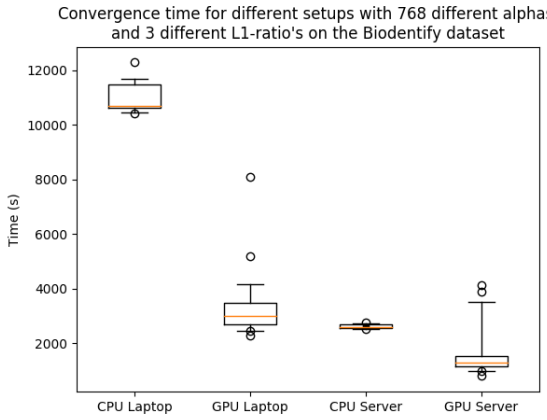

Figure 5.10: Convergence times on the Biodentify dataset for the shotgun implementation compared to the sparse shooting CPU implementation for both hardware setups for 768 different α's.

5.3.1. Profiling with NVIDIA Visual Profiler

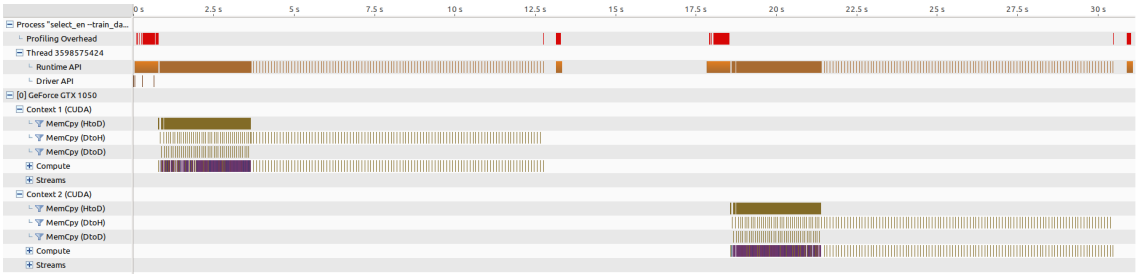

Figure 5.11: NVIDIA Visual Profiler analysis on a grid of 100, converging after roughly 40 iterations

Figure 5.11 shows the main reason for limited performance gain at larger grids. The actual training of the model happens (roughly) from 0.5s to 3.5s for the first cross-validation training set, and from 18.5s to 21.5s for the second cross-validation training set. The convergence checks for the 100 different models for the first cross-validation training set runs from 3.5s to 12.5s, and from 21.5s to 30.5s for the second cross-validation training set. Thus for this scenario, convergence checks take up roughly $\frac{3}{4}$ of the time spend on training the models.

5.3.2. Sparse Dataset Multithreading

First the test depicted in fig. 5.6 and fig. 5.7 is done again for multithreading. However, the CPU implementations are omitted from this test. The results can be seen in fig. 5.12, from which it is clear that in these cases, multithreading contributes to a speedup of 2 for larger gridsizes. However, in these cases, all fits have a convergence check simultaneously, and thus the performance gain on real problems will have varying results based on the amount of simultaneous convergence checks.

Due to thesis time constraints, only 10 runs per setup for the multithreading variant have been done. As expected, for the smaller gridsize, the performance has decreased slightly for the multithreading variant as you can see in fig. 5.13. For the larger gridsizes the performance has increased as can be seen in fig. 5.14 and fig. 5.15 for hardware setup 1. For hardware setup 2, it appears that for 96 different α's the performance is similar for with and without multithreading. For 768 different α's, the performance gain for hardware setup 2 is greater than for hardware setup 1, when multithreading is included.

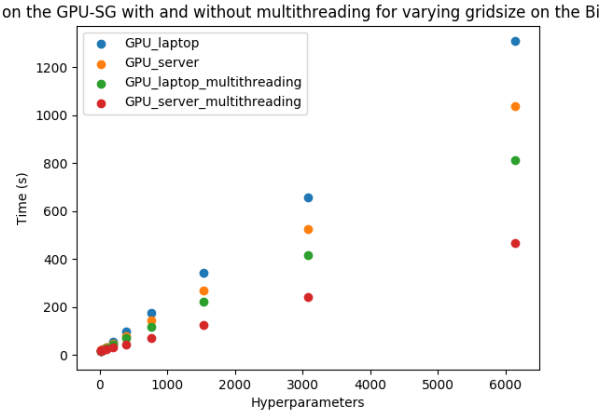

Figure 5.12: Time to compute 100 iterations for the biodentify dataset, for doubling gridsize with amount of parallel feature updates set to 128. This compares the GPU-SG implementation for both with and without multithreading. CPU implementations omitted.

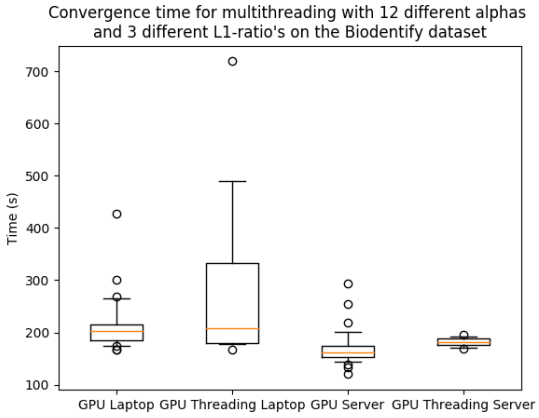

Figure 5.13: Convergence times on the Biodentify dataset for the multithreaded shotgun implementation compared to the shotgun implementation for both hardware setups for 12 different α's.

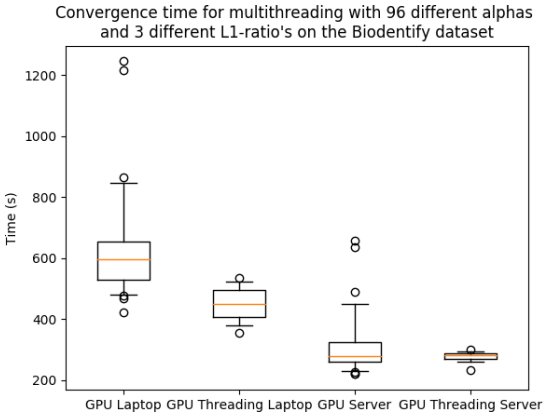

Figure 5.14: Convergence times on the Biodentify dataset for the multithreaded shotgun implementation compared to the shotgun implementation for both hardware setups for 96 different α's.

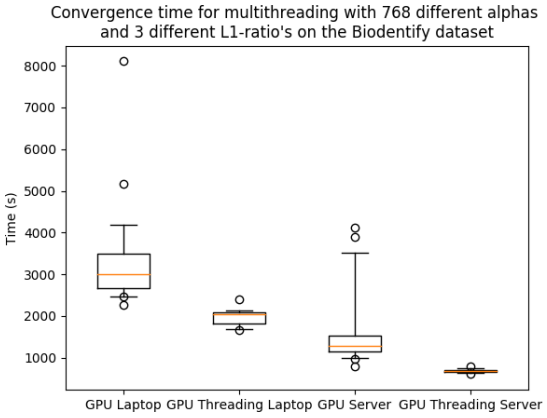

Figure 5.15: Convergence times on the Biodentify dataset for the multithreaded shotgun implementation compared to the shotgun implementation for both hardware setups for 768 different α's.

6

Discussion

This chapter discusses the results shown in the previous chapter, and the limitations of the implementation itself. Firstly the results on the computation times for different problems are discussed. Secondly the limitations of the GPU implementation are discussed. Thirdly the economic impact of the results are discussed, since the price and availability of hardware is a factor for a company in addition to the time factor.

6.1. Performance

As expected, for all problems the computation time on the CPU increases linearly with the problem sizes. When looking at a large number of features, the performance of the cyclical and shooting GPU implementations is extremely low. The performance of the shotgun GPU implementation is highly dependent on the amount of features that can be updated in parallel. However, for an increasingly larger number of samples and/or for an increasingly larger grid, all the GPU implementations outperformed the CPU implementations. Especially the performance on dense datasets is good when the amount of samples or the grid is large, and a speedup of more than 10 can easily be reached.

Comparing the performance differences between the GPU-SG implementations for both hardware setups shows that with a small grid, the GPU is underutilized, since in fig. 5.8 the GPU implementation in both hardware setups perform nearly identical, while the GPU in the server is more powerful than the GPU in the laptop. As you can see in fig. 5.9 and fig. 5.10 the GPU in the server performs better than the GPU in the laptop.

An important point of discussion is the comparison of performance between the Shotgun GPU implementation and the Shooting CPU implementation. This is a false comparison in itself, however it is the best comparison available within the scope of this project. A fairer comparison would be to the compare Shotgun GPU implementation to the Shotgun CPU implementation in a gridsearch setting, however the Shotgun CPU implementation only has an implementation for Lasso, and not Elastic Net. To compare the two would require expanding the current Shotgun CPU implementations to allow the use of Elastic Net, which would be time consuming and not within the scope of this project. However, since the Shotgun CPU implementation is aimed to utilize the power of multiple cores to solve a single fit and the Shooting CPU implementation already uses multiple cores when it is used in a gridsearch setting, the performance gain of using Shotgun CPU within a gridsearch setting would be limited when the grid is large enough to occupy all the available CPU cores.

One point of performance gain that has not been mentioned yet in this thesis, due to it not having scientific value, is the discovery that the original program created by Biodentify centered their dataset before training the model. The centering of the data essentially destroyed the sparsity of the dataset, which after centering became a completely dense dataset. Removing this centering had no significant impact on the accuracy, and it resulted in a huge performance gain while using the CPU implementations. *The result of this change was a speedup of roughly 25 compared to the original program.* All implementations are performed on datasets that

are not centered.

6.2. Limitations

There are several limitations on the current GPU implementation, which would require additional developing to overcome.

The first would be a train set that is too large to fit in the memory of the GPU. Currently it is not possible to use a fold that is larger than the memory of the GPU (more than 4GB for hardware setup 1, and more than 11GB for hardware setup 2). When attempting to run the program using such a train set, the program will terminate with a memory allocation error.

The second limitation is that the program in its current state is not able to use multiple GPU's concurrently.

The last limitation is the choice of the amount of features that are to run in parallel. Currently, the user has to set this parameter by trial and error, restarting the program with a lower amount of parallel features. This means that all previous progress is lost.

6.3. Economic Impact

Not only performance is important when looking at the choice between CPU and GPU implementations, the price of hardware is a factor as well. The choice between a CPU and GPU implementation basically comes down to adding an extra computer to a cluster for the CPU implementation, or adding an extra GPU to your already existing computer. As such, the cost for a computer are taken for the complete setup minus the price for the GPU and the cost for a GPU is the price of the GPU. The rough assumption is made that doubling the amount of computers will result in a speedup by 2 by neglecting overhead, since essentially you can run the same program twice. This means that for twice the amount of money, you get twice the amount of performance, so for doubling the amount of computers $\texttt{speedup} * \texttt{cost} = 1$. So if for adding a GPU, $\texttt{speedup} * \texttt{cost} > 1$, it would be more cost effective to add a GPU, and if $\texttt{speedup} * \texttt{cost} < 1$, it would be more cost effective to buy an extra computer.

For hardware setup 1, the cost of the laptop (Lenovo IdeaPad 330 15ICH 81FK003XMH) is 729 euro and the price of the GPU (GTX 1050 Ti) is roughly 140 euro ($\texttt{price}_{gpu} = 140 euro$), which results in a cost for the laptop without the GPU of 589 euro ($\texttt{price}_{cpu} = 589$). Using the median performances from the results can be extended to Table 6.1, with $\texttt{speedup} * \texttt{cost} = \frac{t_{cpu} * \texttt{price}_{cpu}}{t_{gpu} * (\texttt{price}_{gpu} + \texttt{price}_{cpu})}$.

For hardware setup 2, the purchase prices are taken. The prise of the setup without the GPU's is $\texttt{price}_{cpu} = 5158$ and the price for the GPU is $\texttt{price}_{gpu} = 1074$.

The speedup-cost results are shown in table 6.1 and table 6.2.

Table 6.1: Speedup and speedup*cost of different search spaces on the Biodentify dataset for hardware setup 1. $\ell_{1-ratio} = [0.2, 0.5, 0.9]$ and $\alpha = \texttt{numpy.logspace}(-4, -2, \texttt{num} = ?, \texttt{base} = 10)$. For the number of α's see the table. The convergence times are the medians for the tests using multithreading shown in fig. 5.13, fig. 5.14, and fig. 5.15

Number of α	**CPU Time (s)**	**GPU Time (s)**	**Speedup**	**Speedup*cost**
12	186	208	0.89	0.72
96	1349	449	3.00	2.43
768	10704	2033	5.26	4.25

Table 6.2: Speedup and speedup*cost of different search spaces on the Biodentify dataset for hardware setup 2. $\ell_{1-ratio} = [0.2, 0.5, 0.9]$ and $\alpha =$ `numpy.logspace`$(-4, -2, \texttt{num} =?, \texttt{base} = 10)$. For the number of α's see the table. The convergence times are the medians for the tests using multithreading shown in fig. 5.13, fig. 5.14, and fig. 5.15

Number of α	**CPU Time (s)**	**GPU Time (s)**	**Speedup**	**Speedup*cost**
12	52	181	0.28	0.24
96	329	280	1.18	0.98
768	2583	683	3.78	3.13

7

Conclusion

The objective of this thesis is to design a grid search based feature selection for a GPU to predict hydrocarbons using high dimensional datasets by answering the research question:

> *What is required to design a GPU program, that does parallel feature selection on high-dimensional datasets to predict prospectivity of hydrocarbons, with the Elastic Net algorithm, using a grid search to find the best combination of hyperparameters, and what is the performance compared to a CPU implementation?*

This allows the use of larger datasets and larger grids, without increasing the time required to train a model, resulting in more accurate predictions for hydrocarbons.

To answer part of the research question a literature review on machine learning concerning GPU's was conducted, and it concluded that there has been much research on implementing machine learning algorithms on the GPU. However, this research focuses mainly on Neural Networks, SVM's, and Random Forest Trees. The research on GPU implementations of Elastic Net is limited, and the literature that is available does not focus on Elastic Net in a gridsearch setting. [6] proposed the algorithm *Shotgun*, which uses parallel feature updates to improve objective times for Lasso using a coordinate descent algorithm on a CPU. There appears to be a gap in literature and therefore it was researched how the idea behind *Shotgun* could be used to implement Elastic Net on a GPU within a grid search setting.

A design was conceived that combines the *Shotgun* method [6] for parallelism across features with the parallelism of a grid search. This design uses the ease of python to allow applications in other subjects, along with the power of cuBLAS and a custom CUDA kernel to improve the performance for training thousands of models with different combinations of hyperparameters. The performance of the Shotgun GPU implementation has been shown to be better than the current scikit-learn implementation if a sufficiently large search space has to be exhausted or the amount of samples of the dataset is sufficiently large. fig. 5.3 shows that Shotgun GPU scales extremely well with an extremely large search space compared to the CPU implementation.

Therefore, when exhausting a large grid of combinations of hyperparameters, using the Shotgun GPU implementation will be faster or more cost-effective.

8

Future Work

The design that is presented in this thesis is a first step towards using Elastic Net in a grid search for feature selection on a GPU.

However, this design leaves room for improvement:

Firstly, an interesting direction for future work is to design a hybrid CPU-GPU implementation, which uses the time when the CPU is idle to perform part of the grid search on the CPU. When the most computational intensive grid points are calculated on the CPU, the whole program would benefit from this gain in speed.

Secondly, another direction of future work is building the design completely in C++, this allows for the use of dynamic parallelism, or child-kernels, which is not possible when using the program in combination with python due to software incompatibilities. Child-kernels is the process of executing a CUDA kernel within a CUDA kernel. There are some possibilities to use dynamic parallelism in the custom kernel, which could provide a speedup.

Thirdly, as a direction for future work; there are some possibilities for the CPU and GPU to work asynchronously. For example, the GPU could continue to the next iteration, while the CPU performs the convergence check. This decreases stalls and will speed up the program.

Fourthly, a sparse GPU implementation would overcome the problem of a sparse dataset, which is too large for the memory of a GPU when expressed in a dense format. Using a sparse format to represent a sparse dataset reduces memory requirements, which would allow larger sparse datasets to be computed on the GPU as well. This requires additional changes to the program to deal with the sparse format, and it could lead to an improvement in performance as well. The improvement in speedup is not so large for the GPU as for the CPU implementation already used by Sklearn, due to the different architectures.

Lastly, simultaneous training of models on different training sets is an interesting direction as well. This can be done on the same GPU, if the GPU memory is large enough to contain multiple folds and if there are enough resources available. It could also be used to train on multiple GPU's, each handling models for a different fold.

Bibliography

[1] About Biodentify. URL `https://biodentify.ai/about-biodentify/`. Accessed: 2019-08-02.

[2] Technology & Process. URL `https://biodentify.ai/technology-process/`. Accessed: 2019-08-02.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[4] Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. Gpu acceleration for support vector machines. In *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*, pages 17–55, 2011.

[5] Matteo Barbieri, Samuele Fiorini, Federico Tomasi, and Annalisa Barla. Palladio: a parallel framework for robust variable selection in high-dimensional data. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 19–26. IEEE, 2016.

[6] Joseph K Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379*, 2011.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[8] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number CONF, 2011.

[9] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010.

[10] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[11] Seung-jean Kim, Kwangmoo Koh, Michael Lustig, Stephen Boyd, and Dimitry Gorinevsky. An interior-point method for large-scale ℓ1-regularized logistic regression. In *Journal of Machine learning research*. Citeseer, 2007.

[12] D Montgomery, EA Peck, and GG Vining. Multicollinearity. *Introduction to linear regression analysis*, pages 299–300, 1982.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[14] Peter Richtárik and Martin Takáč. Efficient serial and parallel coordinate descent methods for huge-scale truss topology design. In *Operations Research Proceedings 2011*, pages 27–32. Springer, 2012.

[15] Peter Richtárik and Martin Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 156(1-2):433–484, 2016.

[16] Mr Saha, Mr Darji, Narendra Patel, and Darshak Thakore. Implementation of image enhancement algorithms and recursive ray tracing using cuda. *Procedia Computer Science*, 79:516–524, 12 2016. doi: 10.1016/j.procs.2016.03.066.

[17] Donald F Saunders, K Ray Burson, and C Keith Thompson. Model for hydrocarbon microseepage and related near-surface alterations. *AAPG bulletin*, 83(1):170–185, 1999.

[18] Shai Shalev-Shwartz and Ambuj Tewari. Stochastic methods for l1-regularized loss minimization. *Journal of Machine Learning Research*, 12(Jun):1865–1892, 2011.

[19] Toby Sharp. Implementing decision trees and forests on a gpu. In *European conference on computer vision*, pages 595–608. Springer, 2008.

[20] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[21] Quan Zhou, Wenlin Chen, Shiji Song, Jacob R Gardner, Kilian Q Weinberger, and Yixin Chen. A reduction of the elastic net to support vector machines with an application to gpu computing. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[22] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the royal statistical society: series B (statistical methodology)*, 67(2):301–320, 2005.