Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Επιτάχυνση μηχανικής μάθησης αλγορίθμων SVM σε πλατφόρμες αναδιατασσόμενης λογικής FPGA

Διπλωματικη Εργασια

του

ΧΑΡΑΛΑΜΠΟΥ Π. ΚΑΡΔΑΡΗ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων VLSI
Αθήνα, Σεπτέμβριος 2020

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων VLSI

Επιτάχυνση μηχανικής μάθησης αλγορίθμων SVM σε πλατφόρμες αναδιατασσόμενης λογικής FPGA

Διπλωματικη Εργασια

του

ΧΑΡΑΛΑΜΠΟΥ Π. ΚΑΡΔΑΡΗ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 10η Σεπτεμβρίου 2020.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Διονύσης Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2020

(Υπογραφή)

.....

Χαραλαμπος Καρδαρης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων VLSI

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Δημήτριο Σούντρη για την εμπιστοσύνη προς εμένα και την ευκαιρία που μου έδωσε να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα.

Επίσης ευχαριστώ ιδιαίτερα τον μεταδιδακτορικό ερευνητή κ. Χριστόφορο Κάχρη για την καθοδήγηση σε όλα τα στάδια εκπόνησης της διπλωματικής εργασίας.

Τέλος θα ήθελα να ευχαριστήσω την οικογένειά μου, για όλη την υποστήριξη και αγάπη τους σε μένα και κυρίως για την υπομονή που έδειξαν κατά τη διάρκεια των σπουδών μου.

Περίληψη

Αντικείμενο της διπλωματικής εργασίας είναι η επιτάχυνση της διαδικασίας μηχανικής μάθησης αλγορίθμων SVM σε πλατφόρμες αναδιατασσόμενης λογικής FPGA, μέσω της Σύνθεσης Υψηλού Επιπέδου HLS.

Η μηχανική μάθηση ορίζεται ως μελέτη υπολογιστικών αλγορίθμων, οι οποίοι έχουν τη δυνατότητα να μαθαίνουν από την επεξεργασία των δεδομένων, δηλαδή να βελτιώνουν την απόδοση τους όσον αφορά ένα πρόβλημα, αφού αποκτήσουν γνώση επί των δεδομένων.

Οι Μηχανές Διανυσμάτων Υποστήριξης είναι μια ομάδα αλγορίθμων που, μεταξύ άλλων, επιλύουν προβλήματα κατηγοριοποίησης και γραμμικής παλινδρόμησης. Μεταξύ των πλεονεκτημάτων τους είναι η υψηλή απόδοση που αποδίδουν και η μικρή ανάγκη τους για παραμετροποίηση. Μία από τις ποιο δημοφιλείς υλοποιήσεις της παραπάνω ομάδας αλγορίθμων προσφέρει η βιβλιοθήκη LIBSVM, η οποία αποτελεί και τη βάση μελέτης για αυτή τη διπλωματική εργασία.

Η επιτάχυνση του αλγορίθμου επιτυγχάνεται μέσω των εργαλείων που παρέχει η Σύνθεση Υψηλού Επιπέδου (ΣΥΕ). Η ΣΥΕ είναι μια αυτόματη διαδικασία, η οποία δέχεται οδηγίες σε μορφή κώδικα υψηλού επιπέδου που περιγράφουν μια αλγοριθμική συμπεριφορά και τις ερμηνεύει με σκοπό την παραγωγή υλικού σε πλατφόρμες αναδιατασσόμενης λογικής.

Ο σκοπός της διπλωματικής εργασίας δεν είναι η βελτίωση της υλοποίησης του αλγορίθμου SVM της βιβλιοθήκης LIBSVM, ούτε και ο σχεδιασμός εξειδικευμένων κομματιών υλικού προς αυτό το σκοπό. Αντίθετα είναι η επέκταση και βελτίωση των δυνατοτήτων της βιβλιοθήκης, με κριτήριο την ταχύτητα των διαδικασιών μάθησης, κάνοντας μια εξερεύνηση των δυνατοτήτων που μας προσφέρει η ΣΥΕ.

Λέξεις Κλειδιά

Μηχανές Διανυσμάτων Υποστήριξης, LIBSVM, Πλατφόρμες Αναδιατασσόμενης Λογικής, Επιτάχυνση, Παράλληλη Εκτέλεση, Κατηγοριοποίηση, Γραμμική Παλινδρόμηση, Σύνθεση Υψηλού Επιπέδου

Abstract

The purpose of this diploma thesis is to develop and implement a solution in order to accelerate the machine learning training process of the Support Vector Machines algorithm on Field Programmable Gate Arrays, utilizing High Level Synthesis techniques.

A machine learning algorithm is an algorithm that is able to learn from data, i.e improve its accuracy and performance regarding the execution of a given task, after having processed some relevant information.

Support Vector Machines (or Support Vector Networks) are supervised learning models with associated learning algorithms that analyze data used for classification, regression analysis and other learning problems. Among their advantages are their high performance and their low need for tuning. One the most popular implementations of an SVM algorithm is offered by the LIBSVM library, which is the base of this diploma thesis.

The acceleration of the algorithm is achieved by utilizing the tools the High Level Synthesis offers. HLS is an automated design process that interprets an algorithmic description of a desired behavior in a high-level language and creates digital hardware, commonly for FPGAs that implements that behavior.

The goal of the diploma thesis is not the improvement of the implementation of the SVM algorithm by the LIBSVM library, nor the design of specific hardware modules to be used by the algorithm. The goal is the expansion and improvement of the capabilities of the library, in regard to the actual speed of the training process, by exploring the capabilities that HLS offers.

Keywords

Support Vector Machines, LIBSVM, Field Programmable Gate Arrays, Acceleration, Parallel Computation, Classification, Linear Regression, High Level Synthesis

Contents

List of Figures

List of Tables

Chapter 1

Εισαγωγή - Introduction

1.1 Εισαγωγή στα ελληνικά

Τα τελευταία χρόνια έχει γνωρίσει μεγάλη άνθηση ο τομέας της Μηχανικής Μάθησης. Ο τομέας αυτός κατέχει κεντρικό ρόλο στην γενικότερη προσπάθεια επεξεργασίας του τεράστιου όγκου δεδομένων, που παράγονται καθημερινά, και χρησιμεύουν στην παραγωγή γνώσης. Στόχος είναι η βελτίωση των δυνατοτήτων των μηχανών που χρησιμοποιούμε, μια βελτίωση τις φέρνει πιο κοντά σε αυτό που εννοούμε με τον όρο 'νοημοσύνη'. Η Μηχανική Μάθηση αφορά τη μελέτη αλγορίθμων που βελτιώνονται αυτόματα μέσω της εμπειρίας.

Όπως και σε όλους τους επιστημονικούς τομείς, ο επιστήμονες επιδιώκουν να βελτιόσουν τις πρακτικές τους. Στην περίπτωση της Μηχανικής Μάθησης αυτό μεταφράζεται σε:

- ανάπτυξη καλύτερων αλγορίθμων
- βελτίωση των υπαρχόντων αλγορίθμων, όσον αφορά την ταχύτητα εκτέλεσης τους, της ακρίβεια των αποτελεσμάτων τους, την ενεργειακή κατανάλωση από την εκτέλεση τους, κ.α.

Η βελτίωση των υπολογιστικών συστημάτων είναι συνεχής τις τελευταίες δεκαετίες. Ξεκινώντας από μονοπύρηνα συστήματα, έχουμε πλέον τη δυνατότητα εκτέλεσης παράλληλου κώδικα σε κάρτες γραφικών και πολυπύρηνα συστήματα, τα οποία διαθέτουν ρολόγια χρονισμού με συχνότητες 3 GHz ή και παραπάνω. Παρά την αύξηση της απόδοσης αυτών των συστημάτων, ένα μειονέκτημα τους είναι και η ταυτόχρονη αύξηση της ενεργειακής τους κατανάλωσης.

Τη λύση σε αυτό το πρόβλημα έχουν έρθει να δώσουν οι πλατφόρμες αναδιατασσόμενης λογικής FPGA. Λόγω της δυνατότητας παραγωγής υλικού εξειδικευμένου για ένα συγκεκριμένο πρόβλημα, έχουμε μείωση της πολυπλοκότητας των κυκλωμάτων και συνεπώς μειωμένη ενεργειακή κατανάλωση. Ταυτόχρονα η απόδοση του συστήματος παραμένει σε υψηλά επίπεδα. Τα τελευταία χρόνια, το δύσκολο έργο της σχεδίασης και προγραμματισμού του FPGA με σκοπό να εκτελέσει ένα αλγόριθμο έχει γίνει εξαιρατικά πιο απλό με τη χρήση της Σύνθεσης Υψηλού Επιπέδου (ΣΥΕ - HLS). Η ΣΥΕ δημιουργεί μια ενδιάμεση ζώνη μεταξύ του υλικού και του κώδικα υψηλού επιπέδου, επιτρέποντας στους προγραμματιστές να εκτελέσουν

τα προγράμματά τους στο FPGA, κάνοντας ελάχιστες αλλαγές.

1.1.1 Αντικείμενο της διπλωματικής

Σύντομη περιγραφή του προβλήματος

Ο αλγόριθμος SVM αρχικά προτάθηκε ως μια μέθοςδος για την επίλυση προβλημάτων κατηγοριοποίησης δύο κλάσεων [2]. Έχουν προταθεί διάφορες τροποποιήσεις του, με σκοπό την επίλυση προβλημάτων κατηγοριοποίησης πολλών κλάσεων, γραμμικής παλινδρόμησης και άλλων προβλημάτων μάθησης.

Η βιβλιοθήκη LIBSVM υποστηρίζει τις παρακάτω τροποποιήσεις:

- C-Support Vector Κατηγοριοποίηση
- ν-Support Vector Κατηγοριοποίηση
- Εκτίμηση Κατανομής (SVM μιας κλάσης)
- ϵ-Support Vector Παλινδρόμηση (ϵ-SVR)
- ν-Support Vector Παλινδρόμηση (ν-SVR)

Τα παραπάνω είναι όλα προβλήματα τετραγωνικής ελαχιστοποίησης.
Για παράδειγμα το C-SVC ορίζεται ως εξής.
Έστω διανύσματα $\boldsymbol{x}_i \in \mathbb{R}^n, i = 1, ..., l$, σε δύο κλάσεις, και ένα διάνυσμα δείκτης $\boldsymbol{y} \in \mathbb{R}^l$ τέτοιο ώστε $y_i \in \{1, -1\}$, το C-SVC το λύνει το παρακάτω πρόβλημα ελαχιστοποίησης.

$$\begin{aligned} \min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi_i \\ \text{υπό τις συνθήκες} \quad & y_i(\boldsymbol{w}^T\phi(\boldsymbol{x}_i) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, i = 1, \ldots, l \end{aligned} \tag{1.1}$$

όπου το $\phi(\boldsymbol{x}_i)$ μεταφέρει το $\boldsymbol{x}_i$ σε ένα χώρο ανώτερης διάστασης και $C > 0$ είναι η παράμετρος κανονικοποίησης. Λόγω του πιθανού μεγάλου αριθμού διαστάσεων του διανύσματος $\boldsymbol{w}$, συνήθως επιλύουμε το παρακάτων δυϊκό πρόβλημα.

$$\begin{aligned} \min_{\boldsymbol{a}} \quad & \frac{1}{2}\boldsymbol{a}^T Q\boldsymbol{a} - \boldsymbol{e}^T\boldsymbol{a} \\ \text{υπό τις συνθήκες} \quad & \boldsymbol{y}^T\boldsymbol{a} = 0, \\ & 0 \leq a_i \leq C, \quad i = 1, \ldots, l \end{aligned} \tag{1.2}$$

όπου $e = [1, \ldots, l]^T$ είναι το διάνυσμα με όλες τις διαστάσεις ίσες με 1, Q είναι ένας $l \times l$ θετικά ημιορισμένος πίνακας , $Q_{ij} \equiv y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ και $K(\boldsymbol{x}_i, \boldsymbol{x}_j) \equiv \phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_j)$ είναι η συνάρτηση πυρήνα. Αφού λύσουμε το πρόβλημα (1.2), το βέλτιστο w ικανοποιεί

$$\boldsymbol{w} = \sum_{i=1}^{l} y_i a_i \phi(\boldsymbol{x}_i) \tag{1.3}$$

και η συνάρτηση απόφασης είναι

$$sgn(\boldsymbol{w}^T\phi(\boldsymbol{x}) + b) = sgn\left(\sum_{i=1}^{l} y_i a_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b\right)$$

Η βασική δυσκολία στην επίλυση αυτών των προβλημάτων είναι ότι ο πίνακας Q πιθανώς είναι πολύ μεγάλος για να αποθηκευτεί στη μνήμη του υπολογιστικού συστήματος. Για να αντιμετωπίσει αυτό το πρόβλημα, η βιβλιοθήκη LIBSVM εφαρμόζει μια μέθοδο παραγοντοποίησης η οποία ονομάζεται Sequential Minimal Optimization (SMO), η οποία ζητάει την επίλυση ενός απλούστερου προβλήματος δύο μεταβλητών σε κάθε βήμα της επαναληπτικής διαδικασίας.

Συνεισφορές

Το πιο υπολογιστικά απαιτητικό κομμάτι του αλγορίθμου είναι ο υπολογισμός μιας γραμμής του πίνακα Q. Αυτή η διπλωματική προτείνει μια μέθοδο επιτάχυνσης του σε FPGA. Καταφέρνουμε να εκτελέσουμε το παραπάνω κομμάτι έως και 14 φορές πιο γρήγορα σε σχέση με μια πολυνηματική εκτέλεση. Το αποτέλεσμα αυτό είναι σημαντικό γιατί εκτός του κέρδους σε χρόνο έχουμε και αποδέσμευση των πόρων του συστήματος κατα τη διάρκεια αυτών των απαιτητικών υπλογισμών.

1.1.2 Οργάνωση αυτού του τόμου

Το **Κεφάλαιο 2** αποτελεί μια εκτεταμένη περίληψη της διπλωματικής εργασίας και είναι γραμμένο στα ελληνικά. Το κεφάλαιο αυτό ακολουθεί την δομή του υπόλοιπου τόμου. Η πλήρης και αναλυτική παρουσίαση, ιδιαίτερα των θεωρητικών κομματιών, είναι γραμμένη στα αγγλικά και έχει τη δομή που ακολουθεί.

Το **Κεφάλαιο 3** περιέχει το θεωρητικό υπόβαθρο που χρειάζεται ώστε να κατανοήσουμε τον αλγόριθμο που επιχειρείται να επιταχυνθεί.

Στο **Κεφάλαιο 4** παρουσιάζονται κάποιες γενικές πληροφορίες σχετικά τα FPGA και στο **Κεφάλαιο 5** κάνουμε μια εισαγωγή στην έννοια της Σύνθεσης Υψηλού Επιπέδου.

Το **Κεφάλαιο 6** περιέχει πληροφορίες σχετικά με το hardware που χρησιμοποιούμε σε αυτή τη διπλωματική εργασία, την κάρτα επιτάχυνσης Xilinx Alveo U200.

Τα **Κεφάλαια 7 και 8** παρουσιάζουν το κύριο μέρος της δουλειάς μας. Το **Κεφάλαιο 7** ασχολείται με τη σχεδίαση του επιταχυντή και το **Κεφάλαιο 8** παρουσιάζει τα αποτελέσματα που μετρήθηκαν.

Στο **Κεφάλαιο 9** γίνεται αναφορά σε σχετικές εργασίες επιτάχυνσης του αλγορίθμου SVM σε FPGA.

Στο **Κεφάλαιο 10** κάνουμε ένα τελικό σχόλιο για την προσπάθεια αυτής της διπλωματικής εργασίας και αναφέρουμε κάποια σημεία που μπορεί να υπάρξει μελλοντική βελτίωση των αποτελεσμάτων.

1.2 Introduction in english

We are living in times where high-level technology is available around the world. This makes extremely trivial the production and consumption of data by the whole of human population. This mass production of data creates huge amounts of information, that needs to be processed, in order to produce useful knowledge and create useful services. Consumers and businesses alike want this volume of data to be utilized in their favor, by means of improving their lives and achieving success for their enterprises, accordingly.

Its natural that this huge amount of information cannot be processed by man alone, while at the same time the processing of the data from computing systems is quite complex, if the goal is not only the interpretation fo the data, but the in-depth comprehension in order to produces useful knowledge and create prediction models.

During the last years, the scientific field of machine learning is blossoming. This field has acquired a central roel in the attempt to process data and produce knowledge, with the goal of improving the capabilities of the computing systems we use. An improvent that brings the machines closer to what we would refer to as "intelligence". Machine learning (ML) is the study of computer algorithms that improve automatically through experience.

The researchers are constantly trying to optimize the results that the machine learning algorithms produce. The optimization efforts in the whole field are based, as is usually the case, in two pillars.

- development of better algorithms
- improvement of the currenly established ones, according to metrics like the speed, energy consumption and accuracy of the produced results

For many years, starting with the use of single-core CPUs in computers, the developed code was written with sequential execution in mind. Over the years, the research and development of faster CPU cores provided the ability to execute billions of instructions per second, utilizing hardware with clock rates of 3 GHz or even boosted to more than 4 GHz. The next step was multi-core CPUs and GPUs. The new capabilities of the systems, enabled the code to be parallelized, further optimizing the performance of the algorithms. However, the use of such advanced hardware, in order to achieve better performance, leads to high power consumption and dissipation, which in turn imposes a limit on the final achieved performance.

Field Programmable Gate Arrays (FPGAs), attempt to balance the field, between computational performance and power consumption. The ability to produce hardware tailored for a specific algorithm removes a big part of the design complexity of the hardware regarding general purpose algorithm execution, which results in lower power consumption without losing in performance and often gaining in that regard, as well. In recent years, the daunting task of programming the FPGA to execute a specific task has become significantly easier with the introduction of High Level Synthesis(HLS). HLS creates and abstraction layer between the hardware and the software, permitting software developers to execute their programs on FPGAs, applying only minor changes to their code.

1.2.1 Subject of the diploma thesis

Short description of the problem

The SVM algorithm was introduced as a method to solve two-class classification problems [2]. Different formulations of the initial algorithm have been proposed in order to perform multi-class classification, regression analysis and other learning tasks.

The LIBSVM library supports a number of these formulations:

- C-Support Vector Classification
- ν-Support Vector Classification
- Distribution Estimation (One-class SVM)
- ϵ-Support Vector Regression (ϵ-SVR)
- ν-Support Vector Regression (ν-SVR)

Each of the above is a quadratic minimization problem.
For example C-SVC is defined as follows.
Given training vectors $\boldsymbol{x}_i \in \mathbb{R}^n, i = 1, ..., l$, in two classes, and an indicator vector $\boldsymbol{y} \in \mathbb{R}^l$ such that $y_i \in \{1, -1\}$, C-SVC solves the following primal optimization problem.

$$\begin{aligned} \min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi_i \\ \text{subject to} \quad & y_i(\boldsymbol{w}^T\phi(\boldsymbol{x}_i) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, i = 1, \ldots, l \end{aligned} \tag{1.4}$$

where $\phi(\boldsymbol{x}_i)$ maps $\boldsymbol{x}_i$ into a higher-dimensional space and $C > 0$ is the regularization parameter. Due to the possible high dimensionality of the vector variable $\boldsymbol{w}$, usually we solve the following dual problem.

$$\begin{aligned} \min_{\boldsymbol{a}} \quad & \frac{1}{2}\boldsymbol{a}^T Q\boldsymbol{a} - \boldsymbol{e}^T\boldsymbol{a} \\ \text{subject to} \quad & \boldsymbol{y}^T\boldsymbol{a} = 0, \\ & 0 \leq a_i \leq C, \quad i = 1, \ldots, l \end{aligned} \tag{1.5}$$

where $e = [1, \ldots, l]^T$ is the vector of all ones, Q is an $l \times l$ positive semi-definite matrix, $Q_{ij} \equiv y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and $K(\boldsymbol{x}_i, \boldsymbol{x}_j) \equiv \phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_j)$ is the kernel function. After problem (1.5) is solved, using the primal-dual relationship, the optimal w satisfies

$$\boldsymbol{w} = \sum_{i=1}^{l} y_i a_i \phi(\boldsymbol{x}_i) \tag{1.6}$$

and the decision function is

$$sgn(\boldsymbol{w}^T\phi(\boldsymbol{x}) + b) = sgn\left(\sum_{i=1}^{l} y_i a_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b\right)$$

The definitions of the other SVM formulations can be found in [10].

The main difficulty of solving such problems is that Q may be too large to be stored. To address that, the LIBSVM library implements a decomposition method called Sequential Minimal Optimization (SMO), which requires the solution of a simple two-variable problem for each iteration.

Contributions

The most computationally intensive part of the above algorithm, and according to the software design of the library, is the computation of one row of the above matrix Q. The diploma thesis proposes a method of accelerating this computation on FPGA. We achieve execution times up to 14 times smaller that an execution on a multithreaded system. This result is quite important as, apart from the direct consequence of reduces execution time, we free the computational resources of our system, since we transfer the most time consuming part of the algorithm to the programmable logic.

1.2.2 Organization of this volume

Chapter 2 is an extended summary of the diploma thesis written in greek. This chapter follows the structure of the rest of the book presented in the following lines.

Chapter 3 contains the theoretical background needed in order to understand the algorithm being accelerated.

In **Chapter 4** we provide some general information about Field Programmable Gate Arrays and in **Chapter 5** we make a presentation of the basic elements of High Level Synthesis.

Chapter 6 has information about the specific hardware we are utilizing in this diploma thesis, the Xilinx U200 accelerator card.

Chapters 7 and 8 contain the the information related to our work for this diploma thesis. **Chapter 7** dives into the design of the accelerator, while **Chapter 8** presents the acceleration results.

In **Chapter 9** we make a brief reference to work related to ours. There is a presentation of a handful of projects working on accelerating the SVM algorithm on FPGAs.

In **Chapter 10** we make a brief comment on our work, as a conclusion, while also listing our ideas for future improvements.

Chapter 2

Εκτεταμένη Περίληψη

2.1 Θεωρητικό Υπόβαθρο

2.1.1 Μηχανική Μάθηση

Η Μηχανική Μάθηση ασχολείται με τη μελέτη αλγορίθμων που βελτιώνονται αυτόματα με την εμπειρία. Ένας αλγόριθμος μηχανικής μάθησης μπορεί να μαθαίνει από τα δεδομένα που επεξεργάζεται [1]. Η Μηχανική Μάθηση μας βοηθάει να επιλύσουμε προβλήματα τα οποία είναι πολύ δύσκολο να επιλυθούν με συμβατικούς αλγορίθμους.

Τέτοια προβλήματα είναι η **κατηγοριοποίηση**, η **γραμμική παλινδρόμηση**, η **αναγνώριση ανωμαλιών**, η **σύνθεση και δειγματοληψία**, κ.α.

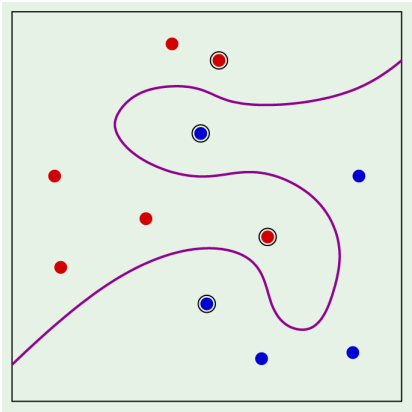
Για να αξιολογήσουμε τις ικανότητες ενός αλγορίθμου μηχανικής μάθησης χρειάζεται ένα ποσοτικό μέτρο της απόδοσης του. Συνήθως, αυτό το μέτρο εξαρτάται από το πρόβλημα. Σε προβλήματα όπως η κατηγοριοποίηση σύνηθες είναι να μετράμε την ακρίβεια του μοντέλου, δηλαδή το ποσοστό των παραδειγμάτων για τα οποία το μοντέλο παράγει τη σωστή πρόβλεψη.

Οι αλγόριθμοι μηχανικής μάθησης μπορούν να χωριστούν σε δύο ευρείες κατηγορίες, την **επιβλεπόμενη** και την **μη επιβλεπόμενη** μάθηση. Οι περισσότεροι αλγόριθμοι επεξεργάζονται ένα σύνολο δεδομένων. Ένα σύνολο δεδομένων είναι μια συλλογή από δείγματα, που με τη σειρά τους είναι μια συλλογή από χαρακτηριστικά. Οι αλγόριθμοι μη επιβλεπόμενης μάθησης ασχολούνται με σύνολα δεδομένων με αρκετά χαρακτηριστικά, και ακολούθως μαθαίνουν χρήσιμες ιδιότητες αυτών. Οι αλγόριθμοι επιβλεπόμενης μάθησης ασχολούνται με ένα σύνολο δεδομένων με χαρακτηριστικά, αλλά ταυτόχρονα κάθε δείγμα έχει και μια **ετικέτα**, δηλαδή πληροφορία για την κλάση στην οποία βρίσκεται.

2.1.2 Μηχανές Διανυσμάτων Υποστήριξης

Οι Μηχανές Διανυσμάτων Υποστήριξης (Support Vector Machines - SVM) είναι μοντέλα επιβλεπόμενης μάθησης που επεξεργάζονται δεδομένα με σκοπό την κατηγοριοποίηση και την ανάλυση παλινδρόμησης.

Κατηγοριοποίηση δεδομένων είναι η διαδικασία μέσω της οποίας, δοθέντων σημείων κάθε ένα εκ των οποίων ανήκει σε μία από δύο κλάσεις, μπορούμε να αποφασίσουμε σε ποια κλάση



Σχήμα 2.1

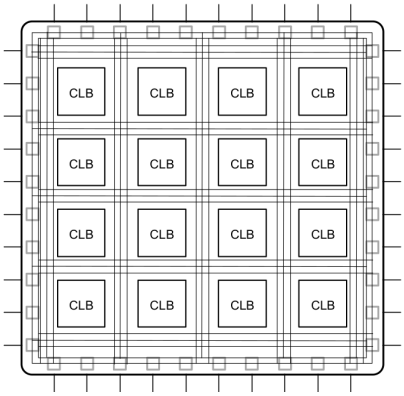
θα ανήκει ένα **νέο σημείο**. Στην περίπτωση των SVMs, ένα σημείο απεικονίζεται ως ένα διάνυσμα p διαστάσεων (μία λίστα p αριθμών), και θέλουμε να μάθουμε αν μπορούμε να χωρίσουμε αυτά τα σημεία με ένα υπερεπίπεδο $p-1$ διαστάσεων. Αυτό λέγεται γραμμική κατηγοριοποίηση.

Μερικά σύνολα δεδομένων αποτελούνται από σημεία/διανύσματα που δεν είναι γραμμικά διαχωρίσιμα σε ένα χώρο $\mathcal{X}$. Μια λύση σε αυτό το πρόβλημα είναι αν αντιστοιχούμε αυτά τα σημεία σε ένα άλλο χώρο $\mathcal{Z}$ μέσω ενός μετασχηματισμού και να προσπαθήσουμε να ελέγξουμε αν σε αυτό το νέο χώρο είναι γραμμικά διαχωρίσιμα. Σε κάθε περίπτωση η λύση του προβλήματος αυτού εμπεριέχει τον υπολογισμό των εσωτερικών γινομένων μεταξύ των διανυσμάτων στο χώρο που εργαζόμαστε. Το σχήμα 2.1 απεικονίζει ένα σύνολο σημείων που δεν είναι γραμμικά διαχωρίσιμα στον αρχικό χώρο, αλλά είναι πιθανότατα γραμμικά διαχωρίσιμα αφού αντιστοιχιστούν σε ένα άλλο χώρο.

Αυτή η τεχνική έχει όμως ένα βασικό μειονέκτημα. Η διαδικασία μετασχηματισμού των σημείων σε ένα άλλο χώρο και μετά ο υπολογισμός των αντίστοιχων εσωτερικών γινομένων, που χρειάζονται για την επίλυση του προβλήματος, έχει συχνά αυξημένο κόστος. Υπάρχει τρόπος όμως να μην το πληρώσουμε. Το **trick του πυρήνα** είναι μια μέθοδος που πετυχαίνει ακριβώς αυτό [7]. Το μόνο που χρειάζεται είναι να ορίσουμε μια συνάρτηση $K(\boldsymbol{x}, \boldsymbol{x'})$ και να αποδείξουμε ότι αυτή υπολογίζει εσωτερικό γινόμενο σε κάποιο χώρο. Από τη στιγμή που θα γίνει αυτό δεν χρειάζεται να αντιστοιχίζουμε τα σημεία στο νέο χώρο, αλλά απλώς να χρησιμοποιούμε αυτή τη συνάρτηση στους τύπους επίλυσης του προβλήματος.

Μερικοί από του πιο δημοφιλείς SVM πυρήνες είναι:

- Radial Basis Function (RBF): $K(\boldsymbol{x}, \boldsymbol{x'}) = e^{-\gamma||\boldsymbol{x}-\boldsymbol{x'}||^2}$
- Polynomial: $K(\boldsymbol{x}, \boldsymbol{x'}) = (c + a \cdot \boldsymbol{x^T x'})^d$
- Hyperbolic Tangent kernel: $K(\boldsymbol{x}, \boldsymbol{x'}) = tanh(c + a \cdot \boldsymbol{x^T x'})$



Σχήμα 2.2: Απλή FPGA αρχιτεκτονική

2.2 Πλατφόρμες Αναδιατασσόμενης Λογικής FPGA

Το FPGA είναι ένας τύπος ολοκληρωμένου κυκλώματος που μπορεί να προγραμματιστεί για διαφορετικούς αλγορίθμους μετά την κατασκευή του. Οι σύγχρονες συσκευές FPGA αποτελούνται από έως και δύο εκατομμύρια λογικά κελιά που μπορούν να ρυθμιστούν ώστε να υλοποιούν διαφορετικούς αλγορίθμους λογισμικού. Παρόλο που η παραδοσιακή ροή σχεδιασμού στα FPGA είναι πιο κοντά σε αυτήν ενός κοινού ολοκληρωμένου κυκλώματος από ότι σε αυτήν ενός επεξεργαστή, ένα FPGA παρέχει σημαντικά πλεονεκτήματα κόστους σε σύγκριση με μια προσπάθεια σχεδιασμού ενός ολοκληρωμένου κυκλώματος και προσφέρει το ίδιο επίπεδο απόδοσης στις περισσότερες περιπτώσεις. Ένα άλλο πλεονέκτημα των FPGA, σε σύγκριση με τα ολοκληρωμένα κυκλώματα, είναι η ικανότητά τους να προγραμματίζονται δυναμικά. Αυτή η διαδικασία, η οποία είναι ίδια με τη φόρτωση ενός προγράμματος σε έναν επεξεργαστή, μπορεί να επηρεάσει μέρος ή όλους τους διαθέσιμους πόρους στο FPGA [19].

2.2.1 Αρχιτεκτονική

Κάθε τσιπ FPGA αποτελείται από έναν πεπερασμένο αριθμό προκαθορισμένων πόρων με προγραμματιζόμενες διασυνδέσεις για την υλοποίηση ενός αναδιαμορφώσιμου ψηφιακού κυκλώματος και μπλοκ Ε/Ε που επιτρέπουν στο κύκλωμα να έχει πρόσβαση στον έξω κόσμο. Η βασική δομή ενός FPGA αποτελείται από τα ακόλουθα στοιχεία: Look-up Tables (LUTs), Flip-Flops (FFs), DSPs, καλώδια, μπλοκ αποθήκευσης και μπλοκ Ε/Ε.
Το σχήμα 2.2 δείχνει πως αυτά τα στοιχεία συνδυάζονται στη απλή FPGA αρχιτεκτονική.

2.2.2 Παραλληλισμός στα FPGA

Η ευρεία χρήση των FPGA στη σύγχρονη εποχή στηρίζεται, μεταξύ άλλων, στην ευκολία σχεδιασμού με στόχο την εκτέλεση παράλληλου κώδικα. Αυτό γίνεται πιο κατανοητό αν συγκρίνουμε την διαδικασία εκτέλεσης εντολών σε ένα επεξεργαστή και ένα FPGA.

Ένα κομμάτι κώδικα για εκτέλεση σε έναν επεξεργαστή χρειάζεται να μεταγλωττιστεί σε εντολές χαμηλού επιπέδου. Οι εντολές αυτές είναι στενά συνδεδεμένες με την αρχιτεκτονική

του επεξεργαστή και συχνά η απόδοση του αλγορίθμου εξαρτάται από αυτήν. Αυτό δημιουργεί πρόσθετες απαιτήσεις στον προγραμματιστή, ο οποίος είναι χρήσιμο να αξιοποιήσει τις επιπλέον δυνατότητες μιας συγκεκριμένης αρχιτεκτονικής (π.χ caching). Η όλη διαδικασία βελτιστοποίησης της απόδοσης αυξάνει σε πολυπλοκότητα.

Από την άλλη πλευρά, τα FPGA είναι σε θέση να υλοποιούν οποιαδήποτε λογική ή αριθμητική συνάρτηση, χωρίς να περιορίζονται σε ζητήματα όπως οι μοιραζόμενη μνήμη ή η μοιραζόμενη μονάδα αριθμητικής λογικής (ALU) σε ένα επεξεργαστή. Στην πράξη, η υλοποίηση ενός αλγορίθμου σε FPGA ορίζει ανεξάρτητες ομάδες από LUTs για κάθε διαφορετικό υπολογισμό. Εκτός αυτού, διαφέρει και στην πρόσβαση στη μνήμη σε σχέση με έναν επεξεργαστή. Τα στοιχεία της μνήμης είναι κατανεμημένα πολύ κοντά στην υπόλοιπη κυκλωματική λογική, με συνέπεια οι χρόνοι πρόσβασης να είναι πολύ μικρότεροι.

2.2.3 Εφαρμογές

Αρχικά, τα FPGA χρησιμοποιούνταν στις τηλεπικοινωνίες και τα δίκτυα. Με τα χρόνια, η χρήση τους επεκτάθηκε και σε άλλες εφαρμογές της βιομηχανίας (βλ. αυτοκίνητα). Στις μέρες μας, η χρησιμοποίηση του σε κέντρα δεδομένων είναι αυξημένη. Οι σύγχρονες δυνατότητες τους ευνοούν τη χρήση τους για επιτάχυνση απαιτητικών αλγορίθμων, όπως είναι οι αλγόριθμοι αναζήτησης και οι αλγόριθμοι μηχανικής μάθησης.

2.3 Σύνθεση Υψηλού Επιπέδου (HLS)

Παλαιότερα, η χρησιμοποίηση των FPGA είχε αρκετές προκλήσεις, μιας ήταν απαραίτητη η κατανόηση σε βάθος της ψηφιακής σχεδίασης υλικού. Η λύση σε αυτό το πρόβλημα ήρθε με την χρήση της Σύνθεσης Υψηλού Επιπέδου (ΣΥΕ). Η ΣΥΕ είναι μια αυτοματοποιημένη διαδικασία σχεδιασμού που ερμηνεύει μια αλγοριθμική περιγραφή και δημιουργεί υλικό που να την υλοποιεί. Αυτό ευνοεί και τους μηχανικούς υλικού, οι οποίοι μπορούν να εργαστούν σε υψηλότερο επίπεδο, χωρίς να χάνουν σε ποιότητα, αλλά και τους προγραμματιστές, οι οποίοι μπορούν να επιταχύνουν τις εφαρμογές τους σε FPGA.

2.3.1 Στάδια

Η λειτουργία της ΣΥΕ μπορεί να χωριστεί σε τρία στάδια:

- Scheduling Καθορίζει ποιες διαδικασίες θα εκτελεστούν σε κάθε κύκλο ρολογιού βάσει μεταβλητών όπως η συχνότητα του ρολογιού, ο χρόνος εκτέλεσης κάθε διαδικασίας και οι ντιρεκτίβες που ορίζονται από το χρήστη/προγραμματιστή.
- Binding Καθορίζει ποιος πόρος υλικού θα υλοποιήσει μια διαδικασία. Αυτό το στάδιο εξαρτάται άμεσα από το μοντέλο της συσκευής που χρησιμοποιούμε.
- Control Logic Extraction Εξάγει τη λογική ελέγχου για τη δημιουργία μια μηχανής πεπερασμένων καταστάσεων, η οποία θέτει την αλληλουχία των διαδικασιών στο σχέδιο ΡΤΛ.

2.3.2 Τεχνικές

Η σύνθεση υψηλού επιπέδου εφαρμόζει κάποιες τεχνικές που βελτιώνουν την απόδοση των αλγορίθμων εκμεταλλευόμενες την παράλληλη σχεδίαση των FPGA. Οι πιο σημαντικές είναι το **pipelining** και το **dataflow**.

Pipelining

Το Pipelining επιτρέπει στον σχεδιαστή να αποφεύγει τις εξαρτήσεις δεδομένων και να αυξάνει το επίπεδο παραλληλισμού σε μια εφαρμογή αλγορίθμου σε υλικό [19].

Θα εξηγήσουμε πως λειτουργεί το Pipelining με ένα παράδειγμα. Ας υποθέσουμε ότι έχουμε ένα σύνολο από n δεδομένα (π.χ. ένας πίνακας) και θέλουμε να εκτελέσουμε κάποιες εργασία σε κάθε στοιχείο αυτού του πίνακα σε βρόχο. Αυτές οι εργασίες μπορεί να είναι αριθμητικές πράξεις ή μεταφορά δεδομένων. Επίσης ας υποθέσουμε, ότι κάθε εργασία διαρκεί 1 κύκλο ρολογιού και υπάρχουν 5 από αυτές τις εργασίες στο βρόχο. Ο σχετικός όρος σε αυτήν την περίπτωση είναι ότι ο βρόχος έχει καθυστέρηση (Loop Latency - LL) ίση με 5. Κάθε επανάληψη του βρόχου χρειάζεται 5 κύκλους για να ολοκληρωθεί και ως αποτέλεσμα ολόκληρος ο βρόχος θα χρειαζόταν $5 \cdot n$ κύκλους για να ολοκληρωθεί (βλέπε σχήμα 5.1). Σε κάθε κύκλο ρολογιού μια εργασία ολοκληρώνεται από τη σχετική μονάδα υλικού. Το πρόβλημα είναι ότι ανά πάσα στιγμή μόνο μία μονάδα είναι ενεργή

Το Pipelining έρχεται να λύσει αυτό το πρόβλημα. Επιτρέπει στην επόμενη επανάληψη να ξεκινήσει το συντομότερο δυνατό, μόλις ελευθερωθεί η πρώτη μονάδα υλικού από την προηγούμενη επανάληψη. Ο αριθμός των κύκλων ρολογιού που αυτό είναι εφικτό ονομάζεται Διάστημα Έναρξης (Initiation Interval - II). Τότε ο συνολικός χρόνος εκτέλεσης θα ήταν $5 + (n - 1)$.

Γενικότερα, μπορούμε να ορίσουμε τους παρακάτω τύπους για τον χρόνο ολοκλήρωσης του βρόχου:

$$\text{Χωρίς Pipelining: } n \cdot LL$$

$$\text{Με Pipelining: } LL + II \cdot (n - 1)$$

Από τα παραπάνω είναι κατανοητό ότι είναι επιθυμητό να μειώνεται το II όσον το δυνατόν περισσότερο. Το ιδανικό είναι να πετύχουμε II ίσο με 1, μιας και τότε έχουμε την υψηλότερη χρησιμοποίηση των μονάδων υλικού σε κάθε κύκλο ρολογιού.

Αν βέβαια ο χρόνος είναι το μοναδικό κριτήριο βελτίωσης τότε μπορούμε να εκτελέσουμε όλες τις επαναλήψεις του βρόχου ταυτόχρονα (αν το επιτρέπουν οι εξαρτήσεις δεδομένων). Αυτό λέγεται **unrolling** του βρόχου και αυξάνει πολύ τη χρησιμοποίηση των πόρων, αφού χρειάζονται διαφορετικές μονάδες υλικού για κάθε επανάληψη.

Dataflow

Το dataflow είναι μια άλλη τεχνική, με παρόμοια φιλοσοφία με το pipelining. Στόχος του dataflow είναι να υλοποιήσει παραλληλισμό σε πιο τραχύ επίπεδο, μεταξύ των συναρτήσεων ενός προγράμματος. Αυτό σημαίνει ότι επιδιώκει την παράλληλη εκτέλεση των συναρτήσεων.

Διακρίνουμε δύο κατηγορίες για το τελευταίο. Αν οι συναρτήσεις είναι ανεξάρτητες μεταξύ τους, τότε η ΣΥΕ δημιουργεί τις κατάλληλες μονάδες υλικού για να εκτελούνται παράλληλα. Σε αντίθετη περίπτωση, δηλαδή αν μια συνάρτηση έχει ως είσοδο δεδομένα που παράγει μια άλλη ως έξοδο, τότε η ΣΥΕ πρέπει να υλοποιήσει και τη λογική επικοινωνίας μεταξύ των μονάδων υλικού που υλοποιούν κάθε συνάρτηση. Αυτή η σχέση ορίζει το σενάριο Παραγωγού-Καταναλωτή.

Αυτή η περίπτωση δέχεται δύο διαφορετικούς τρόπους προσέγγισης. Ο πρώτος είναι όταν η συνάρτηση καταναλωτής περιμένει να παραχθεί το σύνολο των δεδομένων από τη συνάρτηση παραγωγό πριν ξεκινήσει να εκτελείται. Τότε η μόνη παραλληλία που μπορεί να υλοποιηθεί είναι, έχοντας πολλές διαδοχικές κλήσεις των ίδιων συναρτήσεων, να ορίσουμε παράλληλη εκτέλεση της συνάρτησης καταναλωτή που επεξεργάζεται ένα σύνολο δεδομένων με τη συνάρτηση παραγωγό που παράγει το επόμενο σύνολο δεδομένων. Ο δεύτερο τρόπος είναι όταν η συνάρτηση καταναλωτής αρχίζει την εκτέλεση της μόλις αποκτήσει πρόσβαση σε μέρος των δεδομένων που έχει παράξει η συνάρτηση καταναλωτής. Και σε αυτή την περίπτωση οι δύο συναρτήσεις εκτελούνται ταυτόχρονα, όμως η διαφορά είναι ότι είναι υλοποιήσιμη ακόμα και αν κάθε συνάρτηση καλείται μία μόνο φορά.

2.3.3 Vivado HLS

Το εργαλείο Vivado HLS εφαρμόζει τη ΣΥΕ για συσκευές FPGA της Xilinx. Δίνει τη δυνατότητα στον προγραμματιστή να κάνει σύνθεση σε ένα πρόγραμμα γραμμένο σε γλώσσα C/C++ και να παράξει την κατάλληλη κυκλωματική λογική στο FPGA για την εκτέλεση του. Εκτός από τις αυτοματοποιημένες διαδικασίες που το εργαλείο εφαρμόζει, ο προγραμματιστής έχει τη δυνατότητα να δώσει συγκεκριμένες οδηγίες με τη μορφή ντιρεκτίβων (directives) και pragmas.

Τα directives και τα pragmas είναι δύο όψεις του ίδιου νομίσματος. Η διαφορά είναι ότι τα directives ορίζονται σε ξεχωριστό configuration αρχείο, ενώ τα pragmas εισάγονται στον κώδικα.

Ακολουθεί μια αναφορά στα directives που χρησιμοποιούμε στην υλοποίηση υλικού του αλγορίθμου SVM σε αυτή τη διπλωματική εργασία.

array-partition

Χωρίζει έναν πίνακα σε μικρότερους πίνακες ή ατομικά στοιχεία. Αυτό:

- Δημιουργεί κυκλωματική λογική πολλές μικρές μνήμες, αντί μιας μεγάλης.
- Αυξάνει τον αριθμό των σημείων Ε/Ε της μνήμης.
- Πιθανώς αυξάνεις την απόδοσης της σχεδίασης
- Χρειάζεται περισσότερη μνήμη και μονάδες υλικού.

dataflow

Σύμφωνα και με αυτά που αναφέραμε στην προηγούμενη ενότητα, το συγκεκριμένο directive υλοποιεί παραλληλοποίηση σε επίπεδο συναρτήσεων, δίνοντας τη δυνατότητα σε μια συνάρτηση καταναλωτή να αρχίσει την εκτέλεση της μόλις αποκτήσει πρόσβαση σε ένα μέρος των δεδομένων που παράγει μια συνάρτηση παραγωγός.

interface

Καθορίζει τα πρωτόκολλα επικοινωνίας των σημείων εισόδου και εξόδου μεταξύ των συναρτήσεων και των διαφόρων τύπων μνήμης.

pipeline

Όπως, έχουμε ήδη αναφέρει, το pipelining εφαρμόζεται πάνω σε βρόχους, ώστε να γίνεται καλύτερη αξιοποίηση των πόρων και να μειώνεται ο χρόνος εκτέλεσης.

resource

Καθορίζει τον πόρο υλικού που θα υλοποιήσει μια μεταβλητή (πίνακας, αριθμητική πράξη, παράμετρος συνάρτησης).

stream

Συνήθως οι πίνακες (arrays) υλοποιούνται σαν RAM. Αν, όμως, τα δεδομένα του πίνακα παράγονται ή καταναλώνονται σε σειρά, τότε ένας πιο αποδοτικός μηχανισμός επικοινωνίας είναι να χρησιμοποιήσουμε ροή των δεδομένων, όπου χρησιμοποιούμε FIFOs αντί για RAMs.

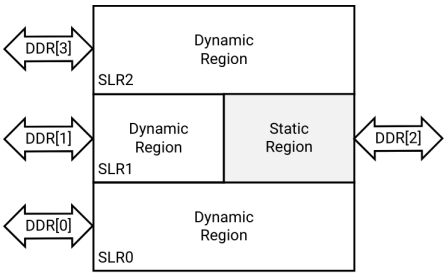
unroll

Σύμφωνα με αυτά που αναφέραμε και προηγουμένως, δημιουργεί πολλά αντίγραφα των μονάδων υλικού που υλοποιούν ένα βρόχο.

2.4 Υλικό

Οι κάρτες επιτάχυνσης Xilinx Alveo™ είναι κάρτες με υποστήριξη PCI Express® που στόχο έχουν την επιτάχυνση υπολογιστικά απαιτητικών εφαρμογών όπως η μηχανική μάθηση, η ανάλυση δεδομένων και η επεξεργασία βίντεο [20]. Σε αυτή τη διπλωματική κάνουμε χρήση της κάρτας Alveo U200.

Πάνω σε μια συσκευή της Xilinx, μια πλατφόρμα έχει μια δυναμική και μια στατική περιοχή. Η στατική περιοχή παρέχει τη βασική υποδομή, ώστε η κάρτα να επικοινωνεί με τον host. Η δυναμική περιοχή είναι ο χώρος όπου τοποθετούνται οι πυρήνες επιτάχυνσης, ώστε να εκτελεστούν. Όλη η αλγοριθμική πολυπλοκότητα προκύπτει από τον χειρισμό των πόρων αυτής της περιοχής.



Σχήμα 2.3: Διάταξη U200

Resource	SLR0	SLR1	SLR2
LUTs	355K	160K	355K
Registers	723K	331K	723K
BRAMs	638	326	638
URAMs	320	160	320
DSPs	2265	1317	2265

Πίνακας 2.1: Πόροι U200

Το σχήμα 2.3 μας δίνει μια εικόνα της διάταξης της κάρτας U200. Μπορούμε να εντοπίσουμε τις 4 μνήμες DDR που διαθέτει, καθώς και τις τρεις περιοχές SLR (Super Logic Region) στη δυναμική της περιοχή.

Ο πίνακας 2.1 παρουσιάζει τους διαθέσιμους πόρους της κάρτας ανά SLR.

Στη συνέχεια παρουσιάζω κάποια από τα χαρακτηριστικά της κάρτας και το πως επιτρέπουν την επιτάχυνση των αλγορίθμων στο FPGA.

- 4 DDR banks: Η παρουσία 4 ξεχωριστών μνημών DDR επιτρέπει την παράλληλη εκτέλεση 4 εντολών R/W. Αυτό δημιουργεί ακόμα μεγαλύτερη επιτάχυνση του αλγορίθμου, αφού μπορούμε να χωρίσουμε τον υπολογισμό σε 4 κομμάτια, τα οποία μπορούν να εκτελεστούν παράλληλα και ανεξάρτητα το ένα από το άλλο.
- 512-bit για μεταφορά δεδομένων μεταξύ μνήμης και πυρήνα. Αυτό το χαρακτηριστικό επιτρέπει την μεταφορά περισσότερων δεδομένων σε ένα κύκλο ρολογιού. Για να το εκμεταλλευτούμε χρησιμοποιούμε τύπους δεδομένων όπως `ap_uint<512>`.
- Μεταφορά δεδομένων σε ριπή (burst): Η πρώτη αίτηση για read ή write στη μνήμη είναι ακριβή, αλλά οι επόμενες δεν είναι, αν τα δεδομένα είναι συνεχόμενα και οι μεταφορές γίνονται σε ριπή.

Αναφορά στα παραπάνω χαρακτηριστικά γίνεται εκ νέου όταν παρουσιάζω την σχεδίαση του επιταχυντή στην επόμενη ενότητα.

Πίνακας 2.2: Profiling της συνάρτησης svm-train

functions \ datasets	a9a	skin	ijcnn1	w8a	Avg.
dot	79.03	37.41	63.71	72.9	63.26%
kernel_rbf	8.18	21.63	11.66	12.11	13.4%
get_Q	4.51	20.95	10.29	5,65	10.35%
select_working_set	5.31	14.45	8.31	5.08	8.29%

2.5 Υλοποίηση Επιταχυντή

2.5.1 Profiling και Επιλογή Συνάρτησης Υλικού

Η διαδικασία μέσω της οποία μπορούμε να βρούμε το πιο υπολογιστικά απαιτητικό κομμάτι ενός αλγορίθμου ονομάζεται **profiling**. Για να το κάνουμε αυτό χρησιμοποιήσαμε το εργαλείο `gprof`.

Profiling

Η υλοποίηση μας βασίζεται στην έκδοση 3.24[1] της βιβλιοθήκης LIBSVM. Κάνοντας χρήση του `gprof` παίρνουμε μια αναφορά όσον αφορά τους χρόνους εκτέλεσης για διαφορετικά σύνολα δεδομένων. Τα αποτελέσματα φαίνονται στον Πίνακα 2.2.

Συνάρτηση Υλικού

Η συνάρτηση που χρειάζεται τον περισσότερο χρόνο σε όλες τις περιπτώσεις είναι η `dot`. Αυτή η συνάρτηση παίρνει ως ορίσματα 2 δείκτες σε δεδομένα των δειγμάτων προπόνησης και επιστρέφει το εσωτερικό τους γινόμενο.

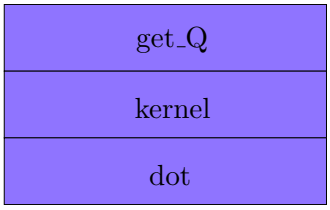
Αρχικά, επιχείρησα να δημιουργήσω έναν FPGA πυρήνα που να κάνει αυτόν τον υπολογισμό του εσωτερικού γινομένου. Τα αποτελέσματα δεν ήταν τα επιθυμητά, καθώς ο χρόνος εκτέλεσης αυτού του υπολογισμού ήταν πολύ μικρότερος από τον χρόνο που χρειάζεται για να αρχίσει να εκτελείται ο πυρήνας. Έτσι είχαμε άσκοπες καθυστερήσεις.

Εξετάζοντας ξανά των κώδικα και τις αναφορές από το profiling, κατάλαβα ότι θα έπρεπε να ασχοληθώ με κάποια συνάρτηση ανώτερου επιπέδου από τη στοίβα κλήσης (βλ. Σχήμα 2.4).

Η συνάρτηση `kernel` είναι overloaded και εξαρτάται από τις αρχικές παραμέτρους του προγράμματος. Παίρνει το εσωτερικό γινόμενο δύο διανυσμάτων και υπολογίζει τον SVM πυρήνα. Οι διαθέσιμες επιλογές είναι: `RBF, tanh, linear and polynomial`.

Το γεγονός ότι όλες οι κλήσεις της συνάρτησης `dot` έρχονται από τη συνάρτηση `get_Q` σημαίνει ότι αυτή η συνάρτηση είναι καλύτερη για επιτάχυνση στο υλικό. Το κομμάτι κώδικα που μας ενδιαφέρει είναι το παρακάτω.

[1] https://github.com/cjlin1/libsvm/releases/tag/v324



Σχήμα 2.4: Στοίβα κλήσης συναρτήσεων

```
for(j = start; j < len; j++) {
data[j] = (float)(y[i]*y[j]*
    (this->*kernel_function)(i,j));
}
```

Η λειτουργία που επιτελεί αυτό το κομμάτι κώδικα είναι ο υπολογισμός της γραμμής i του πίνακα Q, που παρουσιάσαμε στην Εξίσωση 1.2 της υποενότητας 1.1.1. Ο βρόχος ξεκινάει από τον δείκτη `start`, διότι οι προηγούμενες τιμές είναι αποθηκευμένες στην cache που υλοποιεί ο αλγόριθμος.

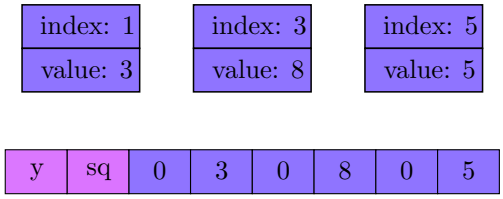
Η πρώτη απόπειρα δεν ήταν να υλοποιήσουμε ολόκληρο το βρόχο στο FPGA, αλλά απλώς να κάνουμε όλους τους υπολογισμούς των εσωτερικών γινομένων. Ο υπολογισμός των SVM πυρήνων γινόταν στον host κώδικα. Αφού καταφέραμε να έχουμε αποτελέσματα για αυτή την υλοποίηση, ακολούθως κάναμε το τελευταίο βήμα και συμπεριλάβαμε και τον υπολογισμό των SVM πυρήνων στο FPGA. Αυτό έδωσε και τα καλύτερα αποτελέσματα φυσικά.

2.5.2 Σχεδίαση του Επιταχυντή

Ο στόχος μας από την αρχή ήταν να αλλάξουμε τον αρχικό κώδικα όσο το δυνατόν λιγότερο. Προσπαθήσαμε να επιταχύνουμε τον αρχικό αλγόριθμο χρησιμοποιώντας τα διαθέσιμα εργαλεία και όχι να κάνουμε τροποποιήσεις που θα μπορούσαν να δώσουν σημαντικές επιταχύνσεις, αλλά θα άλλαζαν τα βασικά δομικά μέρη του. Τα αποτελέσματα της υλοποίησης για FPGA θα έπρεπε είναι ακριβώς τα ίδια με τα αποτελέσματα του αρχικού λογισμικού. Καταφέραμε να υλοποιήσουμε δύο εκδόσεις του κώδικα πυρήνα που κάνουν ακριβώς αυτό. Η πρώτη έκδοση αποθηκεύει τα δεδομένα στη μνήμη του FPGA σε μορφή `double`, ακριβώς όπως και ο αρχικός κώδικας και η δεύτερη έκδοση τα αποθηκεύει σε μορφή `float`.

Η πρώτη έκδοση παράγει πανομοιότυπα αποτελέσματα με το αρχικό λογισμικό, ενώ η δεύτερη έκδοση ανταλλάσσει κάποια ακρίβεια με ταχύτητα (περισσότερα για τις λεπτομέρειες σχεδίασης μπορείτε να βρείτε στην υποενότητα της σχεδίασης του FPGA πυρήνα και για τα αποτελέσματα της επιτάχυνσης στην Ενότητα 2.6). Εξετάσαμε και άλλες εκδόσεις για να εκμεταλλευτούμε περαιτέρω αυτό το tradeoff, αλλά η απώλεια στην ακρίβεια θεωρήθηκε αυξημένη, και θα μπορούσαμε να την αποφύγουμε μόνο με μια πλήρη αναδιάρθρωση του αρχικού κώδικα.

Η υλοποίηση μας κάνει μόνο μία βασική αλλαγή στον αρχικό πηγαίο κώδικα (μαζί με κάποιους άλλες απαραίτητες προσθήκες για να διευκολυνθεί αυτή η αλλαγή), η οποία είναι η αντικατάσταση του κώδικα του βρόχου που αναφέραμε προηγουμένως, με μια κλήση στη



Σχήμα 2.5: Μετατροπή συνδεδεμένης λίστας σε array - δείγμα με 6 χαρακτηριστικά

συνάρτηση `callRowKernel` και προσθέτει 3 κλήσεις συναρτήσεων που απαιτούνται για τη μετάφραση της λογικής αρχικού προγράμματος για εκτέλεση στο FPGA.

Host Κώδικας

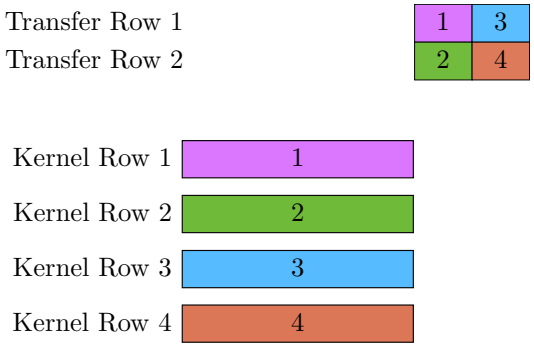
Το πρώτο πράγμα που έπρεπε να ελέγξουμε είναι ο τρόπος αποθήκευσης των διανυσμάτων στον αρχικό κώδικα. Οι συγγραφείς ρίχνουν περισσότερο βάρος στα αραιά δεδομένα. Αυτό οδηγεί στη χρήση συνδεδεμένων λιστών για την αποθήκευση των δεδομένων, με κάθε στοιχείο να αποθηκεύει τον δείκτη και την τιμή των μη μηδενικών διαστάσεων ενός διανύσματος. Η ανάγκη να έχουμε καθορισμένο αριθμό επαναλήψεων στον `Kernel` κώδικα μας έκανε να εγκαταλείψουμε αυτόν τον τρόπο και αποθηκεύουμε τα δεδομένα στην `DMA` μνήμη της κάρτας χρησιμοποιώντας `arrays`, συμπληρώνοντας με μηδενικά όταν χρειάζεται.

Επιπλέον, για να καλύψουμε την περίπτωση χρήσης των SVM αλγορίθμων με ετικέτες κατηγοριοποίησης και την περίπτωση επιλογής του πυρήνα RBF, προσθέσαμε 2 στοιχεία στην αρχή των arrays. Το πρώτο αποθηκεύει το y_i και το δεύτερο το άθροισμα των τετραγώνων $\sum_{i=0}^{dimensions} x_i^2$ του διανύσματος (βλ. Σχήμα 2.5).

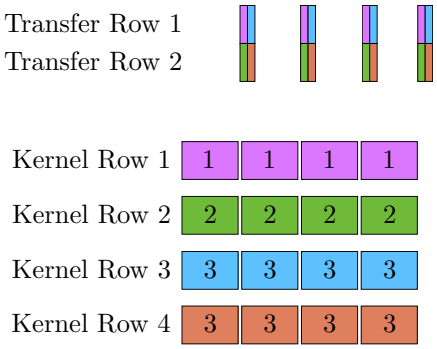
Ένα από τα σημαντικότερα πλεονεκτήματα της χρήσης της Alveo™ U200 είναι το γεγονός ότι είναι διαθέτει 4 μνήμες DDR. Αυτό επιτρέπει να πραγματοποιούνται `R/W` operations από 4 διαφορετικούς πυρήνες ταυτόχρονα. Για αυτό χωρίσαμε τα δεδομένα των διανυσμάτων σε 4 `arrays` για αποθήκευση στη μνήμη.

Ένα άλλο βασικό πλεονέκτημα είναι το εύρος των `512-bit` για μεταφορές δεδομένων μεταξύ του FPGA και της μνήμης της κάρτας. Αυτό επιτρέπει τη μεταφορά 512 bits σε ένα κύκλο ρολογιού. Στον αλγόριθμο μας αυτό μεταφράζεται σε μεταφορά 8 `doubles` ανά κύκλο ρολογιού στην πρώτη έκδοση ή 16 `floats` στη δεύτερη. Για να αξιοποιήσουμε αυτό το χαρακτηριστικό επεκτείνουμε τις διαστάσεις των διανυσμάτων μέχρι το επόμενο πολλαπλάσιο του 8 (ή του 16 αντίστοιχα), συμπληρώνοντας με μηδενικά όπου χρειάζεται. Έτσι κάθε μεταφορά φέρνει δεδομένα που αντιστοιχούν σε ένα και μόνο διάνυσμα.

Χωρίς να μπούμε σε λεπτομέρειες της σχεδίασης του κώδικα πυρήνα, χρειάζεται σε αυτό το σημείο, να αναφέρουμε ποια είναι η συμπεριφορά του, ώστε να παρουσιάσουμε τον τρόπο κλήσης του από τον host κώδικα. Κάθε ένας από τους 4 πυρήνες (kernels) αντιστοιχίζεται σε μια από τις 4 μνήμες της κάρτας. Κάθε πυρήνας παίρνει ως είσοδο, τον δείκτη i του διανύσματος που αντιστοιχεί στην σειρά i του πίνακα Q, μια παράμετρο `start` (αντιστοιχεί στο πρώτο διάνυσμα που θα χρησιμοποιήσουμε για υπολογισμούς μαζί με το αρχικό) και μια παράμετρο `products` (αντιστοιχεί στον συνολικό αριθμό των υπολογισμών). Επιλέγοντας



Σχήμα 2.6: Αρχική προσέγγιση για παράλληλη εκτέλεση πυρήνων



Σχήμα 2.7: Τελική προσέγγιση για παράλληλη εκτέλεση πυρήνων

κατάλληλα αυτές τις παραμέτρους για τους 4 πυρήνες μπορούμε να χωρίσουμε τους αρχικούς `len - start` υπολογισμούς σε περισσότερα κομμάτια τα οποία εκτελούνται παράλληλα.

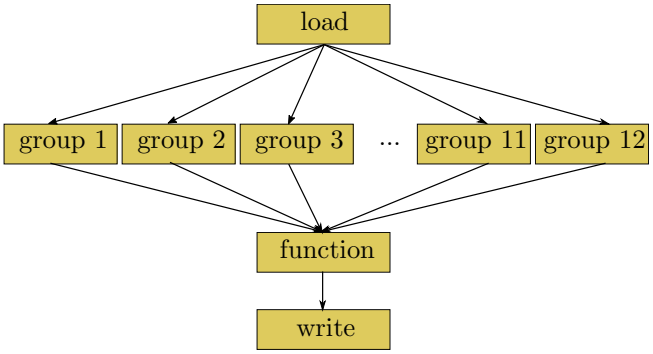
Αρχικά, χωρίσαμε τους υπολογισμούς σε 4 κομμάτια, εάν για κάθε πυρήνα. Το πρόβλημα ήταν ότι για μεγάλα σύνολα δεδομένων ο χρόνος μεταφοράς τον αποτελεσμάτων από τη μνήμη του FPGA πίσω στον host ήταν πολύ μεγάλος και έτσι δεν είχαμε την επιτάχυνση που θέλαμε (βλ. Σχήμα 2.6).

Το αντιμετωπίσαμε αυτό, χωρίζοντας τα 4 κομμάτια σε επιμέρους μικρότερα. Έτσι η μεταφορά δεδομένων συμβαίνει, καθώς εκτελείται ο πυρήνας με το επόμενο σετ (βλ. Σχήμα 2.7). Ουσιαστικά, πρόκειται για pipeline στις κλήσεις του κώδικα πυρήνα.

Κώδικας Πυρήνα

Κάθε πυρήνας αρχίζει διαβάζοντας τα δεδομένα από τη μνήμη του FPGA, κάνει τους απαραίτητους υπολογισμούς και επιστρέφει τα αποτελέσματα στη μνήμη. Από εκεί μεταφέρονται πίσω στον host.

Οι διαδικασίες ανάγνωσης και εγγραφής στη μνήμη, και έχοντας υπόψιν ότι μόνο μια DMA μεταφορά μπορεί να συμβαίνει ανά πάσα στιγμή, θέτει ένα κατώτερο όριο για το χρόνο εκτέλεσης. Όπως έχουμε ήδη αναφέρει το `data bus` μπορεί να μεταφέρει 8 `doubles` ή 16 `floats` ανά κύκλο ρολογιού. Τελικά, η καθυστέρηση της υλοποίησης μας είναι περίπου $n * d + \frac{n}{16}$ κύκλοι, όπου n είναι ο αριθμός των διανυσμάτων για τα οποία υπολογίζουμε τους SVM πυρήνες και d είναι ο αριθμός των διαστάσεων διαιρεμένος με 8 (ή 16).



Σχήμα 2.8: Dataflow πυρήνα

Για να καταφέρουμε αυτό το αποτέλεσμα, όπου ο συνολικός χρόνος εκτέλεσης εξαρτάται μόνο από τον ελάχιστο χρόνο για τη μεταφορά των δεδομένων στο FPGA από τη μνήμη, κάναμε χρήση των `pipeline`, `dataflow` and `unroll` directives. Ένα άλλο χαρακτηριστικό της κάρτας που βοήθησε ήταν οι μεταφορές δεδομένων σε ριπή μεταξύ της DMA μνήμης και του FPGA.

Έχουμε ήδη αναφέρει τι προσφέρει χρήση αυτών των ντιρεκτίβων. Όμως είναι σημαντικό να περιγράψουμε τη δομή του κώδικα πυρήνα μας όσον αφορά το dataflow. Το Σχήμα 2.8 μας δίνει μια οπτική αναπαράσταση των συναρτήσεων/μονάδων που είναι ορισμένες σε κάθε πυρήνα.

Η συνάρτηση `load` διαβάζει δεδομένα από τη μνήμη της κάρτας. Γνωρίζουμε τις διστάσεις κάθε διανύσματος, συνεπώς μπορούμε να ομαδοποιήσουμε τις αναγνώσεις. Σε κάθε κύκλο ρολογιού οι 8 (ή 16) τιμές που μεταφέρονται πολλαπλασιάζονται με τις αντίστοιχες τιμές του διανύσματος 'βάσης' (το διάνυσμα που αντιστοιχεί στην σειρά του πίνακα Q που υπολογίζουμε). Για να γίνουν αυτές οι πράξεις σε ένα κύκλο, κάνουμε χρήση του `unroll directive`.

Στη συνέχεια πραγματοποιούμε πρόσθεση αυτών των 8 (ή 16) τιμών σε δενδρική διάταξη (3 ή 4 επίπεδα αντίστοιχα) και το άθροισμα μεταφέρεται ως stream σε μια εκ των 12 μονάδες της συνάρτησης group. Αυτή η συνάρτηση προσθέτει τα αθροίσματα που αντιστοιχούν στο ίδιο διάνυσμα και υπολογίζει το εσωτερικό γινόμενο. Αυτό μεταφέρεται ξανά με streaming στη συνάρτηση με όνομα function, η οποία υπολογίζει τον πυρήνα SVM. Από εκεί τα αποτελέσματα μεταφέρονται στη συνάρτηση write, η οποία τα ομαδοποιεί ανά 16 (αφού είναι σε μορφή `float`) και γράφει πίσω στη μνήμη της κάρτας.

Το ενδιαφέρον κομμάτι της σχεδίασης του dataflow μοντέλου στον πυρήνα μας είναι η παρουσία των 12 μονάδων/αντιγράφων της συνάρτησης `group`. Στην προσπάθεια μας σχεδιάσουμε έναν πυρήνα ικανό να επεξεργαστεί διαφορετικά σύνολα δεδομένων με μεγάλο εύρος ως προς τον αριθμό των χαρακτηριστικών έπρεπε να απορρίψουμε την ιδέα για πρόσθεση όλων των διαστάσεων κάθε διανύσματος σε δενδρική διάταξη. Και αυτό γιατί το εργαλείο Vivado HLS δεν μπορούσε να κάνει τη βέλτιστη σύνθεση χωρίς πριν να ξέρει τον συνολικό αριθμό των διαστάσεων.

Έτσι η μόνη επιλογή ήταν η πρόσθεση σε αλληλουχία όλων των τιμών. Το πρόβλημα που υπεισέρχεται σε αυτή την περίπτωση είναι ότι μια πρόσθεση χρειάζεται ένα ικανό αριθμό

Πίνακας 2.3: Περιορισμοί στη χρήση πόρων ανά πυρήνα

Πόρος	**SLR0**	**SLR1**	**SLR2**	**Όριο/Πυρήνα**
LUTs	355K	160K	355K	177.5K
Registers	723K	331K	723K	361.5K
BRAMs	638	326	638	319
URAMs	320	160	320	160
DSPs	2265	1317	2265	1132

κύκλων ρολογιού για να πραγματοποιηθεί, ενώ εμείς είμαστε σε θέση να τροφοδοτούμε τη συνάρτηση με μία νέα τιμή ανά κύκλο. Αυτό σημαίνει ότι η πρόσθεση εισάγει ένα bottleneck που έπρεπε να αντιμετωπίσουμε. Με τη χρήση των 12 μονάδων μπορούμε να στείλουμε τα δεδομένα που αντιστοιχούν σε νέο διάνυσμα σε καινούρια μονάδα, ώστε να μην έχουμε καθυστερήσεις του pipeline. Η επιλογή του αριθμού 12 δεν είναι τυχαία. Βρήκαμε ότι μια πρόσθεση `double` τιμών διαρκούσε 12 κύκλους. Συνεπώς, η ύπαρξη 12 μονάδων επιτρέπει την ανάθεση τιμών σε διαφορετική μονάδα, έως ότου τελειώσει ο υπολογισμός της 1ης μονάδας. Για να το εξηγήσουμε αυτό λίγο καλύτερα ας δούμε ένα παράδειγμα. Η συνάρτηση `load` παράγει μία νέα τιμή ανά κύκλο. Έστω ότι 5 τιμές αντιστοιχούν σε ένα διάνυσμα (αρχικές διαστάσεις: $5 \cdot 8 = 40$ ή $5 \cdot 16 = 80$ για τις δύο εκδόσεις). Τότε όλη η διαδικασία πρόσθεσης χρειάζεται 60 κύκλους για να ολοκληρωθεί. Σε αυτό το διάστημα όμως έχουν παραχθεί συνολικά 60 τιμές. Για να μην έχουμε καθυστερήσεις τις χωρίζουμε ανά 5 και τις αναθέτουμε σε διαφορετικό αντίγραφο της ίδιας συνάρτησης `group`.

Από εκεί και πέρα τα εσωτερικά γινόμενα μεταφέρονται στην συνάρτηση `function` όπου γίνεται ο υπολογισμός των SVM πυρήνων.

Στο παράρτημα διατίθεται ο κώδικας πυρήνα όπου φαίνεται όλη αυτή η σχεδίαση του μοντέλου για τον πυρήνα μας.

2.5.3 Χρησιμοποίηση Πόρων

Τα FPGA έχουν κάποιους περιορισμούς όσον αφορά τους πόρους που μπορούμε να χρησιμοποιήσουμε. Εκτός από τον συνολικό αριθμό των πόρων που είναι διαθέσιμοι, υπάρχουν και κάποιες προτάσεις ώστε η απόδοση των πυρήνων να είναι βέλτιστη. Μία από αυτές είναι ότι κάθε πυρήνας θα πρέπει να μπορεί να τοποθετηθεί εξ ολοκλήρου σε μία περιοχή SLR. Έχοντας υπόψιν την διαθεσιμότητα πόρων ανά SLR (βλ. Πίνακα 2.1) και τη χρήση 4 πυρήνων στην υλοποίηση μας, έχουμε τους περιορισμούς του Πίνακα 2.3.

Η παράμετρος του design μας που επηρεάζει τον αριθμό των πόρων είναι ο μέγιστος αριθμός των διαστάσεων ανά διάνυσμα που υποστηρίζουν οι πυρήνες. Στην περίπτωση της double έκδοσης αυτός ο αριθμός είναι ίσος με 8000, ενώ στην float είναι 65000. Οι Πίνακες 2.4 και 2.5 δείχνουν τη χρησιμοποίηση των πόρων στις δύο αυτές περιπτώσεις.

Όπως βλέπουμε ο πόρος που θέτει αυτά τα όρια είναι οι BRAM. Οι BRAM είναι μνήμες αποθήκευσης και για αυτό η αναπαράσταση float έχει μεγαλύτερα περιθώρια.

Πίνακας 2.4: Πόροι ανά πυρήνα - έκδοση double

Πόρος	**Σε χρήση**	**Διαθέσιμοι**	**Χρησιμοποίηση**
BRAM_18K	278	319	87.1%
DSP48E	426	1132	37.6%
FF	93777	361686	26%
LUT	62764	177415	35.4%
URAM	8	160	5%

Πίνακας 2.5: Πόροι ανά πυρήνα - έκδοση float

Πόρος	**Σε χρήση**	**Διαθέσιμοι**	**Χρησιμοποίηση**
BRAM_18K	314	319	98.4%
DSP48E	391	1132	34.5%
FF	93777	361686	23.1%
LUT	62764	177415	31.8%
URAM	8	160	5%

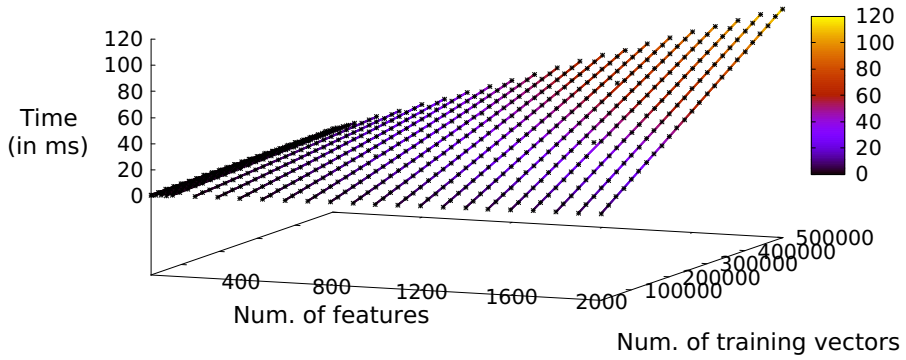
2.6 Αποτελέσματα

Σε αυτή την ενότητα θα παρουσιάσουμε τις επιταχύνσεις της υλοποίησης μας σε σύγκριση με την αρχική έκδοση. Οι επιταχύνσεις στο FPGA είναι σε σύγκριση με πολυνηματική εκτέλεση σε μια CPU που αξιοποιεί τους 4 διαθέσιμους πυρήνες της. Η CPU που χρησιμοποιήσαμε είναι η AMD Ryzen™ 2200G, η οποία έχει συχνότητα 3.5 GHz. Εξετάζουμε πως το μέγεθος του συνόλου δεδομένων και ο αριθμός των χαρακτηριστικών των δειγμάτων επηρεάζει την επιτάχυνση.

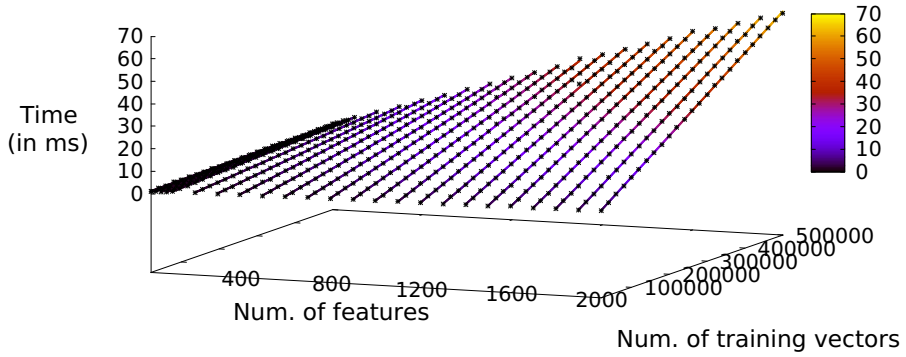
Προς τούτο, κάναμε μια εξερεύνηση των χρόνων εκτέλεσης σε διάταξη πλέγματος, δοκιμάζοντας διαφορετικά μεγέθη συνόλων και διαφορετικούς αριθμούς χαρακτηριστικών. Για να είναι είναι όσο το δυνατόν πιο αντικειμενικές οι μετρήσεις, δημιουργήσαμε custom datasets. Το αρχικό ήταν το dataset Epsilon[2]. Το συγκεκριμένο σύνολο έχει 400000 δείγματα με 2000 χαρακτηριστικά το καθένα. Αυτό που κάναμε ήταν να πάρουμε ένα υποσύνολο του με 20000 δείγματα και από αυτό να κατασκευάσουμε ξεχωριστά datasets με διαφορετικό αριθμό δειγμάτων (20000, 40000, ..., 500000) και αριθμών χαρακτηριστικών (5,10, 25, 50, 75, 100, 200, ..., 2000). Οι χρόνοι μέτρησης αφορούν τον υπολογισμό μια σειράς του πίνακα Q στο FPGA και την CPU.

Τα σχήματα 2.9, 2.10 και 2.11 παρουσιάζουν τους χρόνους εκτέλεσης όπως μετρήθηκαν για τα διαφορετικά σύνολα με χρήση των 3 διαφορετικών υλοποιήσεων (double FPGA, float FPGA, multithreaded CPU) Τα μαύρα σημεία είναι απεικονίζουν τις πραγματικές τιμές, ενώ οι γραμμές είναι προσαρμοσμένες πάνω σε αυτά. Παρατηρούμε τη γραμμική σχέση όσον αφορά τους χρόνους εκτέλεσης σε όλες τις περιπτώσεις.

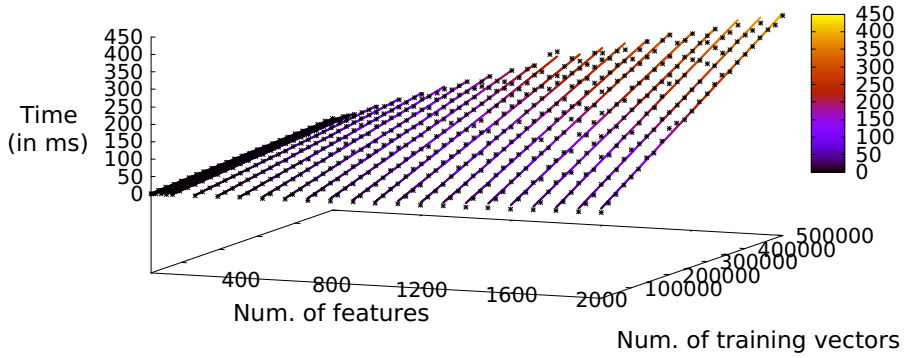
[2]https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.html#epsilon



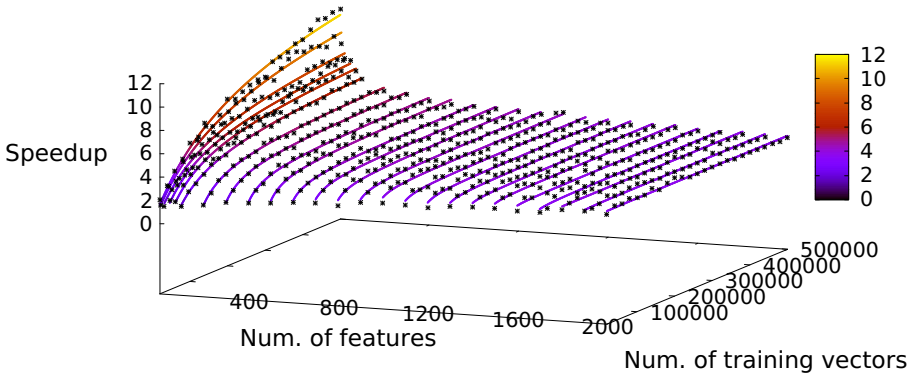
Σχήμα 2.9: Double έκδοση: Πως ο αριθμός των δειγμάτων επηρεάζει το χρόνο εκτέλεσης



Σχήμα 2.10: Float έκδοση: Πως ο αριθμός των δειγμάτων επηρεάζει το χρόνο εκτέλεσης



Σχήμα 2.11: Πολλαπλά νήματα: Πως ο αριθμός των δειγμάτων επηρεάζει το χρόνο εκτέλεσης



Σχήμα 2.12: Double έκδοση: Πως ο αριθμός των δειγμάτων επηρεάζει την επιτάχυνση

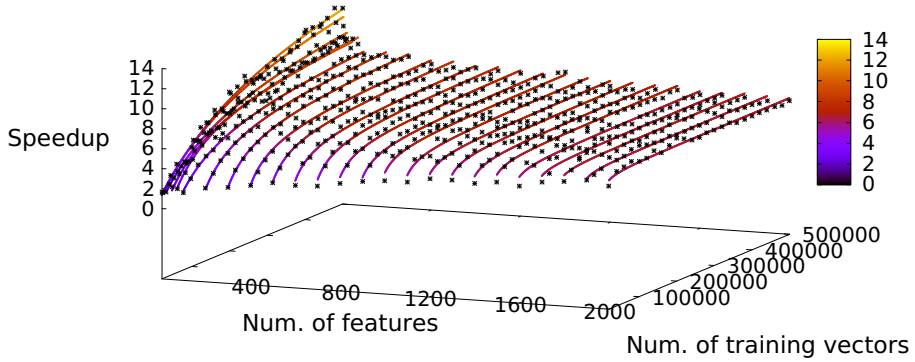
Αυτή η σχέση δημιουργεί μια ενδιαφέρουσα κατάσταση όσον αφορά την επιτάχυνση μεταξύ της εκτέλεσης σε FPGA και CPU. Ας υποθέσουμε ότι η γραμμή του χρόνου εκτέλεσης στο FPGA για κάποιο συγκεκριμένο αριθμό χαρακτηριστικών είναι $f1(x) = ax + b$ και η γραμμή της πολυνηματική εκτέλεσης στην CPU είναι $f2(x) = cx + d$, όπου x είναι η μεταβλητή που εκφράζει το σύνολο των δειγμάτων. Σε αυτή την περίπτωση μπορούμε να θεωρήσουμε ότι οι σταθερές b και d εκφράζουν το βασικό overhead της εκτέλεσης σε κάθε πλατφόρμα. Η συνάρτηση της επιτάχυνσης θα ήταν τότε $f(x) = \frac{cx+d}{ax+b}$. Αυτή η συνάρτηση (αν δεν είναι σταθερή) είναι 'υπερβολή' και έχει κάποια ιδιαίτερα χαρακτηριστικά:

- $\lim_{x\to-\infty} f(x) = \lim_{x\to+\infty} f(x) = \frac{c}{a}$
- $\lim_{x\to-\frac{b}{a}^-} f(x) = \begin{cases} +\infty, & f1(-\frac{b}{a}) < 0 \\ -\infty, & f1(-\frac{b}{a}) > 0 \end{cases}$
- $\lim_{x\to-\frac{b}{a}^+} f(x) = \begin{cases} -\infty, & f1(-\frac{b}{a}) < 0 \\ +\infty, & f1(-\frac{b}{a}) > 0 \end{cases}$

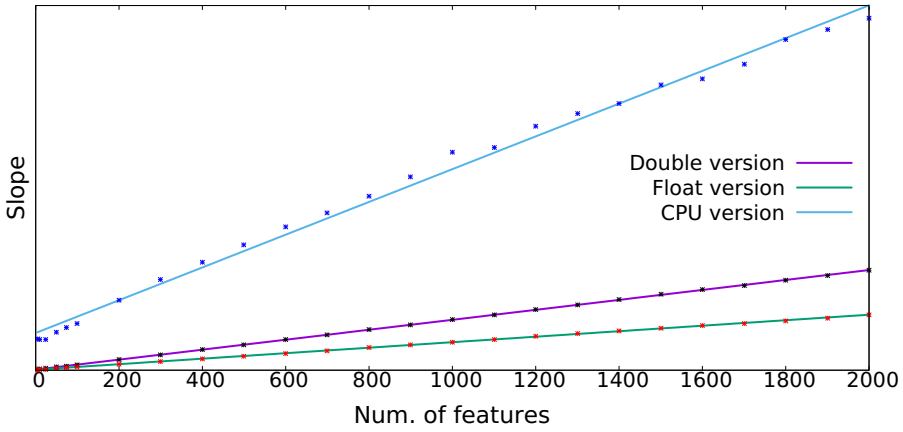
Στην περίπτωση μας, μιας και ασχολούμαστε μόνο με θετικές τιμές και η φύση του προβλήματος μας τελικά επιβάλλει $a, b, c, d > 0$, τα παραπάνω χαρακτηριστικά καθορίζουν τη μορφή της καμπύλης. Αν $f1(-\frac{b}{a}) < 0$, τότε οι τιμές $f(x)$ είναι μικρότερες και αυξάνονται καθώς το x αυξάνεται. Αν $f1(-\frac{b}{a}) > 0$, τότε οι τιμές $f(x)$ είναι μεγαλύτερες και μειώνονται καθώς το x αυξάνεται.

Τα σχήματα 2.12 και 2.13 δείχνουν τις επιταχύνσεις των υλοποιήσεων σε FPGA σε σχέση με την CPU. Παρατηρούμε πως συγκλίνουν οι τιμές, καθώς ο αριθμός των δειγμάτων αυξάνεται, καθώς και το ότι η συναρτήσεις είναι αύξουσες.

Επίσης, μπορούμε να δούμε ότι η μέγιστη επιτάχυνση διαφέρει για διαφορετικούς αριθμούς χαρακτηριστικών. Στο σχήμα 2.14 παρουσιάζουμε τις γραφικές παραστάσεις που ορίζονται από την κλίση των fitted γραμμών των χρόνων εκτέλεσης (βλ. σχήματα 2.9, 2.10, 2.11. Όπως και οι συναρτήσεις της επιτάχυνσης είναι πηλίκα γραμμικών συναρτήσεων, έτσι και η



Σχήμα 2.13: Float έκδοση: Πως ο αριθμός των δειγμάτων επηρεάζει την επιτάχυνση



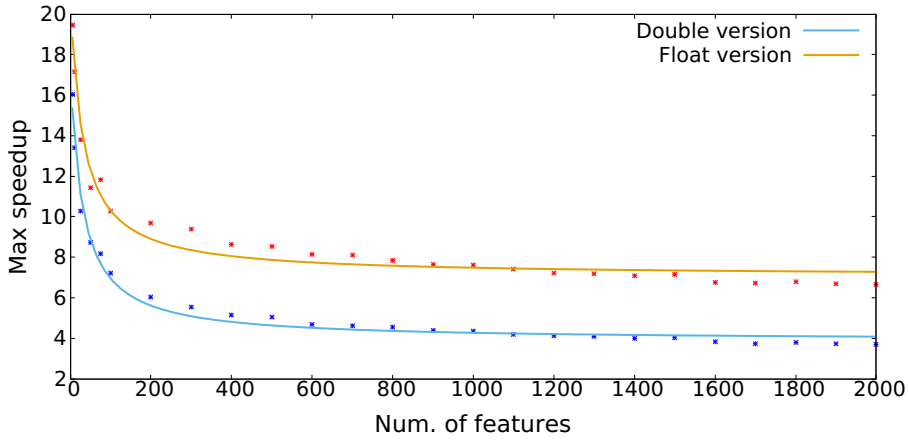
Σχήμα 2.14: Κλίση των γραφικών παραστάσεων χρόνου/αριθμού χαρακτηριστικών

συνάρτηση της μέγιστης επιτάχυνσης, είναι πηλίκο αυτών των γραφικών παραστάσεων (βλ. σχήμα 2.15). Αυτή η σχέση και πάλι καθορίζει τη μορφή της καμπύλης (είναι 'υπερβολή') και μας πληροφορεί ότι για μικρό αριθμών χαρακτηριστικών η μέγιστη επιτάχυνση, καθώς αυξάνει το πλήθος των δειγμάτων, είναι αρκετά μεγαλύτερη. Στην γενική περίπτωση, και ικανά μεγάλο αριθμό δειγμάτων και χαρακτηριστικών, η επιτάχυνση φτάνει σε όρια που είναι περίπου 3.5x και 7x για την double και την float έκδοση αντίστοιχα.

2.7 Συμπεράσματα

Σε αυτή τη διπλωματική εργασία, παρουσιάζονται τα αποτελέσματα της προσπάθειας επιτάχυνσης της βιβλιοθήκης LIBSVM.

Η προσπάθεια μας δεν επικεντρώθηκε στην βελτίωση του αρχικού αλγορίθμου, κάνοντας τροποποιήσεις σε αυτόν σε θεωρητικό επίπεδο, αλλά χρησιμοποιώντας τη Σύνθεση Υψηλού Επιπέδου ώστε να γίνει εκμετάλλευση των δυνατοτήτων που προσφέρει η κάρτα επιτάχυνσης (Xilinx® Alveo™ U200 Data Center). Στόχος ήταν να γίνει παραλληλοποίηση του κώδικα πυρήνα στο FPGA σε σημείο που το μόνο bottleneck να είναι η μεταφορές δεδομένων από και



Σχήμα 2.15: Μέγιστη Επιτάχυνση ανάλογα με τον αριθμό των χαρακτηριστικών

προς τη μνήμη της κάρτας και το FPGA. Οι μετρήσεις δείχνουν ότι επιτυγχάνεται επιτάχυνση έως 14x στην καλύτερη περίπτωση σε σχέση με μια πολυνηματική εκτέλεση του αλγορίθμου στον επεξεργαστή AMD Ryzen™ 3 2200G.

Chapter 3

Theoretical background

3.1 Machine Learning

Definition

Machine learning (ML) is the study of computer algorithms that improve automatically through experience. In other words, a machine learning algorithm is an algorithm that is able to learn from data [1]. But what do we mean by learning? Mitchell [4] provides a succinct definition:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T, as measured by P, improves with experience E.

One can imagine a wide variety of experiences E, tasks T, and performance measures P.

Machine learning enables us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. In this relatively formal definition of the word "task," the process of learning itself is not the task. Learning is our means of attaining the ability to perform the task.

Some of the most common machine learning tasks include:

- **Classification**: In this type of task, the computer program is asked to specify which of k categories some input belongs to.
- **Regression**: In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe the information into discrete textual form.
- **Anomaly Detection**: In this type of task, the computer program sifts through a set of events or objects and flags some of them as being unusual or atypical.
- **Synthesis and Sampling**: In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data.

To evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. Usually this performance measure P is specific to the task

T being carried out by the system. For tasks such as classification we often measure the accuracy of the model. Accuracy is just the proportion of examples for which the model produces the correct output.

Machine Learning algorithms can be broadly categorized as **unsupervised** or **supervised**, based on the kind of experience they are allowed to have and process. Most machine learning algorithms simply experience a dataset. A dataset is a collection of examples, which are in turn collections of features. Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset. Supervised learning algorithms experience a dataset containing features, but each example is also associated with a **label** or **target**.

3.2 Support Vector Machines

In machine learning, support-vector machines are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis [6].

Classification of data is the process by which, given a collection of data points each belonging to one of two classes, we are able to decide which class a **new data** point will be in. In the case of support-vector machines, a data point is viewed as a p-dimensional vector (a list of p numbers), and we want to know whether we can separate such points with a $(p-1)$-dimensional hyperplane. This is called a linear classifier.

Simplest case: linearly-separable data - linear transform

Problem 1. *We are given a set of n data points of the form $(\boldsymbol{x_i}, y_i)$, $y_i \in \{-1, 1\}$ denoting the class of vector $\boldsymbol{x_i}$. We want to find the "maximum-margin hyperplane" that divides the group of points $\boldsymbol{x_i}$ for which $y_i = 1$ from the group of points for which $y_i = -1$, which is defined so that the distance between the hyperplane and the nearest point $\boldsymbol{x_i}$ from either group is maximized.*

Solution

Any hyperplane can be written as the set of points $\boldsymbol{x_i}$ satisfying $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$. Let $\boldsymbol{x_n}$ be the nearest data point to the hyperplane. We first need to normalize $\boldsymbol{w}$, so that $|\boldsymbol{w} \cdot \boldsymbol{x_n} + b = 1|$

Let $\boldsymbol{w}$ be a vector perpendicular to the plane and $\boldsymbol{x'}$, $\boldsymbol{x''}$ be two data points of the plane (see figure 3.1).

Then,

$$\begin{cases} \boldsymbol{w^T} \cdot \boldsymbol{x'} + b = 0 \\ \boldsymbol{w^T} \cdot \boldsymbol{x''} + b = 0 \end{cases} \implies \boldsymbol{w^T} \cdot (\boldsymbol{x'} - \boldsymbol{x''}) = 0$$

Take any point $\boldsymbol{x}$ on the plane. Let $\widehat{\boldsymbol{w}}$ be a vector parallel to $\boldsymbol{w}$ with $|\widehat{\boldsymbol{w}} = 1|$, so $\widehat{\boldsymbol{w}} = \frac{\boldsymbol{w}}{||\boldsymbol{w}||}$ (see Figure 3.2). We consider the projection of $\boldsymbol{x_n} - \boldsymbol{x}$ on $\boldsymbol{w}$.

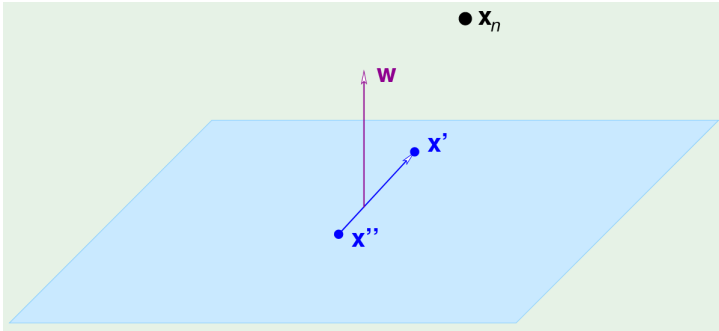


Figure 3.1

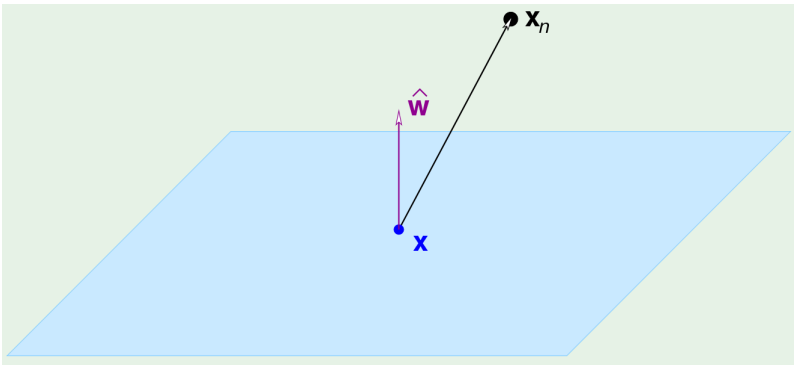


Figure 3.2

Then,

$$distance = |\widehat{\boldsymbol{w}}^{\boldsymbol{T}} \cdot (\boldsymbol{x_n} - \boldsymbol{x})| = \frac{1}{||\boldsymbol{w}||} \cdot |\boldsymbol{w}^{\boldsymbol{T}} \cdot \boldsymbol{x_n} - \boldsymbol{w}^{\boldsymbol{T}} \cdot \boldsymbol{x}| = \frac{1}{||\boldsymbol{w}||} \cdot |\boldsymbol{w}^{\boldsymbol{T}} \cdot \boldsymbol{x_n} + b - \boldsymbol{w}^{\boldsymbol{T}} \cdot \boldsymbol{x} - b| \implies$$

$$\implies distance = \frac{1}{||\boldsymbol{w}||}$$

Based on the above, in order to solve Problem 1 we need to solve

Problem 2. *Maximize* $\frac{1}{||\boldsymbol{w}||}$, *subject to* $\min_n |\boldsymbol{w}^{\boldsymbol{T}} \cdot \boldsymbol{x_n} + b| = 1$
(note: $|\boldsymbol{w}^{\boldsymbol{T}} \cdot \boldsymbol{x_n} + b| = y_n(\boldsymbol{w}^{\boldsymbol{T}} \cdot \boldsymbol{x_n} + b)$*)*

which is equivalent to

Problem 3. *Minimize* $\frac{1}{2}\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w}$, *subject to* $y_n \cdot (\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x_n} + b) \geq 1$, *for* $n = 1, 2, \ldots, N$

The Lagrange formulation of the above problem is

Problem 4. *Minimize* $\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{a}) = \frac{1}{2}\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} - \sum_{n=1}^{N} a_n(y_n(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x_n} + b) - 1)$,
w.r.t $\boldsymbol{w}$ *and* b *and maximize w.r.t each* $a_n \geq 0$

To solve Problem 4 we set

$$\nabla_{\boldsymbol{w}}\mathcal{L} = \boldsymbol{w} - \sum_{n=1}^{N} a_n y_n \boldsymbol{x_n} = \boldsymbol{0}$$

and

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{n=1}^{N} a_n y_n = 0$$

Substituting in the Lagrangian we get

$$\mathcal{L}(\boldsymbol{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} y_n y_m a_n a_m \boldsymbol{x_n^T x_m}$$

and the problem to solve becomes

Problem 5. *Maximize* $\mathcal{L}(\boldsymbol{a})$*, w.r.t* $\boldsymbol{a}$ *subject to* $a_n \geq 0$ *for* $n = 1, 2, \ldots, N$ *and* $\sum_{n=1}^{N} a_n y_n = 0$

or better suited for a quadratic programming solver

Problem 6. *Minimize* $-\mathcal{L}(\boldsymbol{a})$*, w.r.t* $\boldsymbol{a}$ *subject to* $a_n \geq 0$ *for* $n = 1, 2, \ldots, N$ *and* $\sum_{n=1}^{N} a_n y_n = 0$

The solution of this problem is a vector $\boldsymbol{a} = a_1, a_2, \ldots, a_N \implies \boldsymbol{w} = \sum_{n=1}^{N} a_n y_n \boldsymbol{x_n}$

The Karush-Kuhn-Tucker conditions of the Lagrangian problem are:

$$a_n(y_n(\boldsymbol{w^T x_n} + b) - 1) = 0, \quad n = 1, 2, \ldots, N$$

The above means that

$$a_n \geq 0 \implies \boldsymbol{x_n} \text{ is a } \textbf{support vector}$$

The decision function is then

$$f(x) = sgn(\boldsymbol{w^T x} + b) = sgn(\sum_{n=1}^{N} a_n y_n \boldsymbol{x_n x} + b)$$

Non-linear transform

Some datasets contain data points that are not linearly separable in space $\mathcal{X}$. A solution to this problem is to map the data points in another space $\mathcal{Z}$ and try to see if the data are linearly separable in that space. Figure 3.3 depicts a set of data points that are not linearly separable in the initial space, but can probably be linearly separable after being mapped to another space.

In this case we don't compute the inner product $\boldsymbol{x_n^T x_m}$ in space $\mathcal{X}$, but instead we compute the inner product $\boldsymbol{z_n^T z_m}$ in that space. The Lagrangian becomes

$$\mathcal{L}(\boldsymbol{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} y_n y_m a_n a_m \boldsymbol{z_n^T z_m}$$

and the decision function becomes

$$f(x) = sgn(\boldsymbol{w^T x} + b) = sgn(\sum_{n=1}^{N} a_n y_n \boldsymbol{z_n z} + b)$$

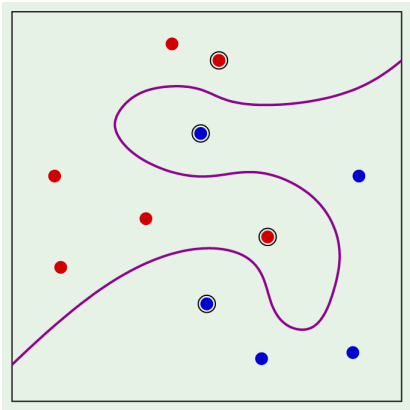


Figure 3.3

The kernel trick

The above technique is very useful if the data points are not linearly separable in one space but can be transformed in another space where they are linearly separable. This permits any new data point to be classified by simply transforming it to the new space and computing the new decision function. There is one major disadvantage though. The process of transforming the data points to another space and then computing the inner products can have significant costs.

As it turns out we don't need to pay that cost. The kernel trick is a method to avoid the explicit mapping [7]. We only need to define a function $K(\boldsymbol{x}, \boldsymbol{x'})$ and prove that this function is an inner product in some space.
The lagrangian then becomes

$$\mathcal{L}(\boldsymbol{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} y_n y_m a_n a_m K(\boldsymbol{x_n}, \boldsymbol{x})$$

and the decision function

$$f(x) = sgn(\boldsymbol{w^T x} + b) = sgn(\sum_{n=1}^{N} a_n y_n K(\boldsymbol{x_n}, \boldsymbol{x}) + b)$$

Some of the most popular SVM kernels are:

- Radial Basis Function (RBF): $K(\boldsymbol{x}, \boldsymbol{x'}) = e^{-\gamma ||\boldsymbol{x} - \boldsymbol{x'}||^2}$
- Polynomial kernel: $K(\boldsymbol{x}, \boldsymbol{x'}) = (c + a \cdot \boldsymbol{x^T x'})^d$
- Hyperbolic Tangent kernel: $K(\boldsymbol{x}, \boldsymbol{x'}) = tanh(c + a \cdot \boldsymbol{x^T x'})$

Soft-margin SVM

Soft-margin SVM is a formulation of the Support Vector Machine algorithm in order to be applicable in datasets where the data points are not linearly-separable no matter the kernel function selected. In that case, we are prepared to accept some outlier data points that are not correctly classified and we use a error metric function to determine the level of acceptance for such outliers. The mathematical equations are altered accordingly.

3.3 LIBSVM

LIBSVM [10] a library for Support Vector Machines training and classification. It is very popular and widely used for machine learning applications. LIBSVM attempts to bring the theoretical knowledge of Support Vector Machines to the real computing world.

3.3.1 SVM formulations

Soft-margin SVM is an SVM formulation, but there exist many more in an effort to expand the capabilities of the original algorithm. LIBSVM supports various SVM formulations for classification, regression and distribution estimation. The list of formulations supported are:

- C-Support Vector Classification
  This is the soft-margin SVM formulation.
- ν-Support Vector Classification
  This introduces and new parameter $\nu \in (0, 1]$. ν is an upper bound of the fraction of training errors and a lower bound of the fraction of support vectors.
- Distribution Estimation (One-class SVM)
  One-class SVM was proposed for estimating the support of a high-dimensional distribution.
- ϵ-Support Vector Regression (ϵ-SVR)

- ν-Support Vector Regression (ν-SVR)
  Similar to ν-SVC, for regression, it used a parameter $\nu \in (0, 1]$ to control the number of support vectors. The parameter ϵ in ϵ-SVR becomes a parameter here.

3.3.2 The quadratic problem - Sequential Minimal Optimization

A general form for one-variable SVM problems (C-SVC, ϵ-SVR) is the following.

$$\begin{aligned} \min_{\boldsymbol{a}} \quad & f(\boldsymbol{a}) \\ \text{subject to} \quad & \boldsymbol{y^T a} = \Delta \\ & 0 \leq a_t \leq C, t = 1, 2, \ldots, l \end{aligned} \tag{3.1}$$

where

$$f(\boldsymbol{a}) = \frac{1}{2}\boldsymbol{a}^T Q \boldsymbol{a} + \boldsymbol{p}^T \boldsymbol{a}$$

and $y_t = \pm 1, t = 1, 2, \ldots, l$.

The main problem in trying to solve Problem 3.1 is that matrix Q may be too large to be stored in the computer memory. There have been proposed many decomposition methods to tackle this difficulty. A decomposition method, splits the original problem into smaller ones, so that each smaller problem is possible to be stored and solved by the

computer. In the case of the SVM algorithms, such a method modifies only a subset of $\boldsymbol{a}$ per iteration, so only some columns of Q are needed. The subsets of variables that comprises of this smaller problem in each iteration is denoted as the working set B. The type of decomposition utilized by LIBSVM is called Sequential Minimal Optimization (SMO) and the smaller problems it creates contain only two variables.

Without going into too much detail about the exact implementation of the algorithm (see [10], subsection 4.1.1), we can describe the steps.

SMO algorithm

1. Set $k = 1$ for initial iteration
2. Find an initial feasible $\boldsymbol{a}^1$
3. In every iteration k, check the stop condition, which is to check if $\boldsymbol{a}^k$ is stationary, according to the KKT optimization problem conditions. If the condition is met stop the iteration, else proceed to step 4.
4. Select working set of two variables i, j (WSS algorithm)
5. Depending on the value, compared to 0, of a function $f(K(x, x'), i, j)$, solve a linear equations problem with variables a_i, a_j.
6. Update the value of $\boldsymbol{a}^{k+1}$ according to the results and go to step 3.

KKT stopping condition and WWS algorithm (see [10], subsection 4.1.2)

A feasible vector $\boldsymbol{a}$ is stationary point if and only if there exists a number b and two non-negative vectors $\boldsymbol{\lambda}$ and $\boldsymbol{\xi}$ such that

$$\nabla f(\boldsymbol{a}) + b\boldsymbol{y} = \boldsymbol{\lambda} - \boldsymbol{\xi}, \tag{3.2}$$

$$\lambda_i a_i = 0, \xi_i(C - a_i) = 0, \lambda_i \geq 0, \xi_i \geq 0, i = 1, 2, \ldots, l$$

where $\nabla f(\boldsymbol{a}) \equiv Q\boldsymbol{a} + \boldsymbol{p}$ is the gradient of $f(\boldsymbol{a})$.

This can be rewritten as

$$\nabla_i f(\boldsymbol{a}) + by_i = \begin{cases} \geq 0, & if a_i < C \\ \leq 0, & if a_i > 0 \end{cases}$$

which is equivalent to

$$\exists b : m(\boldsymbol{a}) \leq b \leq M(\boldsymbol{a})$$

where

$$m(\boldsymbol{a}) = \max_{i \in I_{up}(\boldsymbol{a})} -y_i \nabla_i f(\boldsymbol{a}) \text{ and } M(\boldsymbol{a}) = \min_{i \in I_{low}(\boldsymbol{a})} -y_i \nabla_i f(\boldsymbol{a})$$

and

$$I_{up}(\boldsymbol{a}) \equiv \{t | a_t < C, y_t = 1 \ or \ a_t > 0, y_t = -1\}$$

$$I_{low}(\boldsymbol{a}) \equiv \{t | a_t < C, y_t = -1 \ or \ a_t > 0, y_t = 1\}$$

That means that a feasible $\boldsymbol{a}$ is a stationary point if and only if

$$m(\boldsymbol{a}) \leq M(\boldsymbol{a})$$

and a suitable stop condition is

$$m(\boldsymbol{a}^k) - M(\boldsymbol{a}^k) \leq \epsilon$$

where ϵ is the tolerance.
The Working Set Selection algorithm steps are:

1. For all t, s define

$$a_{ts} \equiv K(t,t) + K(s,s) - 2K(t,s),$$

$$b_{ts} \equiv -y_t \nabla_t f(\boldsymbol{a}^k) + y_s \nabla_s f(\boldsymbol{a}^k) > 0$$

   and

$$\bar{a_{ts}} \equiv \begin{cases} a_{ts}, & if a_{ts} > 0 \\ \tau, & otherwise \end{cases}$$

2. Select

$$i \in arg \max_t \{-y_t \nabla_t f(\boldsymbol{a}^k) | t \in I_{up}(\boldsymbol{a}^k)\}$$

$$j \in arg \min t \{-\frac{b_{it}^2}{\bar{a}_{it}} | t \in I_{low}(\boldsymbol{a}^k), -y_t \nabla_t f(\boldsymbol{a}^k) < -y_i \nabla_i f(\boldsymbol{a}^k)\}$$

Other formulations

The two-variable SVM problems follow a similar procedure in order to perform the SMO algorithm. The exact mathematical equations can be found in [10], section 4.2.

3.3.3 Caching and Shrinking

LIBSVM uses two implementation tricks, i.e caching and shrinking, in order to speed-up the training process.

Shrinking

An optimal solution $\boldsymbol{a}$ of the SVM dual problem may contain some bounded elements. These elements may have already been bounded in the middle of the decomposition iterations. To save the training time, the shrinking technique tries to identify and remove some bounded elements, so a smaller optimization problem is solved. This is theoretically supported, showing that at the final iterations of the SMO algorithm only a small set of variables is still changed.

Caching

LIBSVM is using a Least-Recently-Used caching mechanism on software in order to reduce the number of computations needed. It is observed that especially in the last iterations of the SMO algorithms the values needed to be computed have already been done so in previous iterations. By utilizing this caching mechanism the execution time can be significantly slower.

The cache is implemented using a simple linked list of structures.

```
struct head_t
{
    head_t *prev, *next; // a circular list
    Qfloat *data;
    int len; // data[0,len) is cached in this entry
};
```

The structure points to the first `len` elements of a column. Modifying the linked list (additions, deletions, insertions) is easy enough, with the presence of pointers `prev` and `next`.

Chapter 4

Field Programmable Gate Arrays

An FPGA is a type of integrated circuit (IC) that can be programmed for different algorithms after fabrication. Modern FPGA devices consist of up to two million logic cells that can be configured to implement a variety of software algorithms. Although the traditional FPGA design flow is more similar to a regular IC than a processor, an FPGA provides significant cost advantages in comparison to an IC development effort and offers the same level of performance in most cases. Another advantage of the FPGA when compared to the IC is its ability to be dynamically reconfigured. This process, which is the same as loading a program in a processor, can affect part or all of the resources available in the FPGA fabric [19].

4.1 Architecture

Every FPGA chip is made up of a finite number of predefined resources with programmable interconnects to implement a reconfigurable digital circuit and I/O blocks to allow the circuit to access the outside world. The basic structure of an FPGA is composed of the following elements:

- Look-up table (LUT): This element performs logic operations.
- Flip-Flop (FF): This register element stores the result of the LUT.
- Wires: These elements connect elements to one another.
- Input/Output (I/O) pads: These physically available ports get data in and out of the FPGA.

Figure 4.1 shows how these elements combine to form the basic FPGA architecture.

In recent times the architecture of FPGAs has evolved with the inclusion of more advanced computational and data storage blocks, that increase the computational density and efficiency of the device. These are:

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates

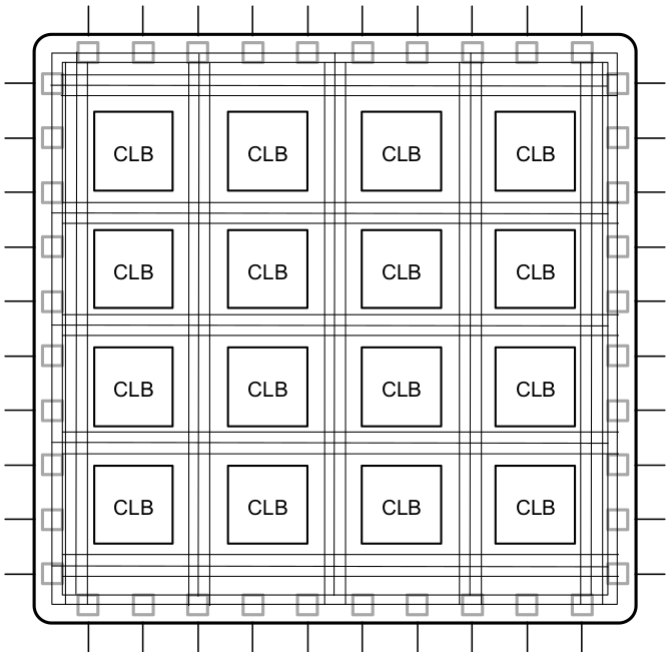


Figure 4.1: Basic FPGA architecture

- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

These new blocks, together with the traditional ones, form the modern FPGA architecture shown in Figure 4.2, which provides the FPGA with the flexibility to implement any software algorithm.

FPGA Components

Let's provide some more detail about the components of the modern FPGA architecture.

Lookup Table

The LUT is the basic building block of an FPGA and is capable of implementing any logic function of N Boolean variables. Essentially, this element is a truth table in which different combinations of the inputs implement different functions to yield output values.

The hardware implementation of a LUT can be thought of as a collection of memory cells connected to a set of multiplexers. The inputs to the LUT act as selector bits on the multiplexer to select the result at a given point in time (see Figure 4.3). A LUT can be used as both a function compute engine and a data storage element.

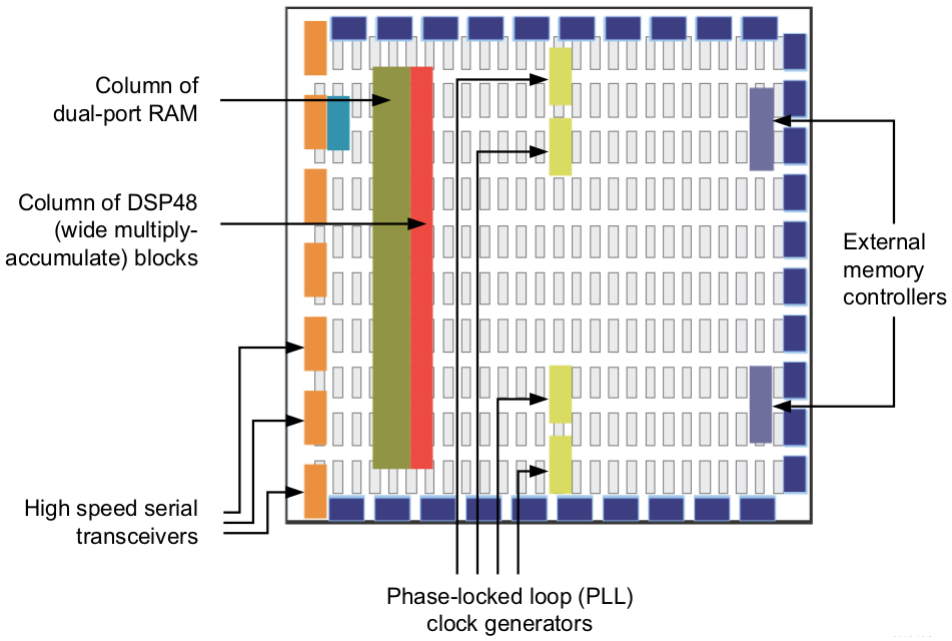


Figure 4.2: Modern FPGA architecture

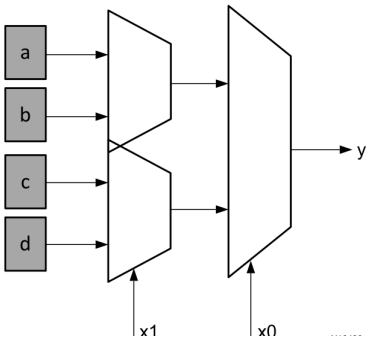


Figure 4.3: Lookup table representation

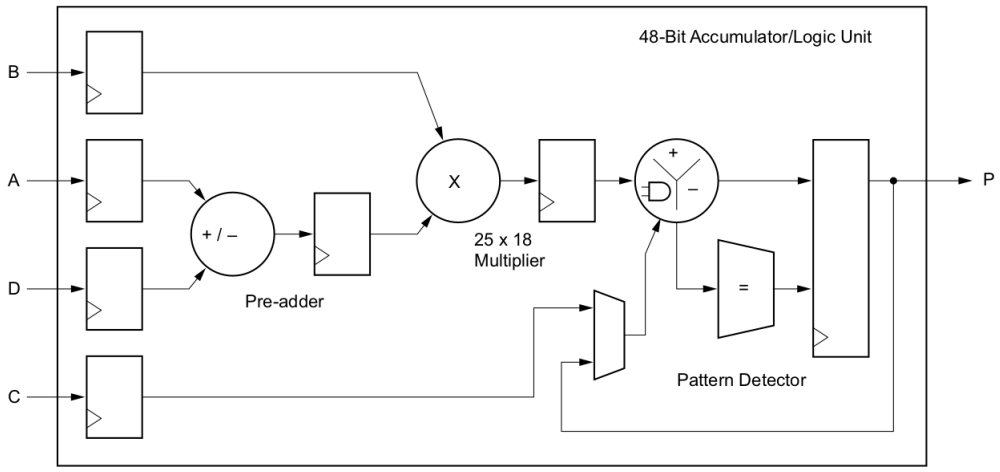


Figure 4.4: Structure of a DSP block

Flip-Flop

The flip-flop is the basic storage unit within the FPGA fabric. This element is always paired with a LUT to assist in logic pipelining and data storage. The basic structure of a flip-flop includes a data input, clock input, clock enable, reset, and data output. During normal operation, any value at the data input port is latched and passed to the output on every pulse of the clock. The purpose of the clock enable pin is to allow the flip-flop to hold a specific value for more than one clock pulse. New data inputs are only latched and passed to the data output port when both clock and clock enable are equal to one.

DSP Block

The most complex computational block available in modern FPGAs is the DSP block (see Figure 4.4). The DSP block is an arithmetic logic unit (ALU) embedded into the fabric of the FPGA, which is composed of a chain of three different blocks. The computational chain in the DSP is composed of an add/subtract unit connected to a multiplier connected to a final add/subtract/accumulate engine. This chain allows a single DSP unit to implement functions of the form:

$$P = B \cdot (A + D) + C$$

or

$$P += B \cdot (A + D)$$

Storage Elements

The FPGA devices usually include embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These ele-

ments are block RAMs (BRAMs), UltraRAM blocks (URAMS)[1], LUTs, and shift registers (SRLs).

4.2 FPGA Parallelism

The popularity and great utilization of FPGAs in the current era is based, among other reasons, upon the promise of ease of design and execution of highly parallelizable algorithms. In order to understand why this constitutes such a strong point of FPGA we have to compare the instruction execution process in both common processors and FPGAs.

Processor execution

Any segment of code written for CPU execution needs to be translated into low level instructions. Usually this process involves a compiler, who will translate the high-level language code into assembly code instructions. The assembly code, being close the specific architecture of the processor, describes the steps that need to be performed on hardware in order to perform the task intended. The thing is that, even though the task is well defined, its execution time is not constant, depending on many factors, such the location of the involving data (hard drive, memory, cache, etc.). This creates an additional need for optimization. Software engineers need to design their algorithms in such a way that the data needed during the execution are cached as much as possible. The added complexity in algorithm design is often overwhelming.

FPGA execution

The FPGA is an inherently parallel processing fabric capable of implementing any logical and arithmetic function that can run on a processor, while not being hindered by the restrictions of a cache, a unified memory space and shared arithmetic logic units.

The LUTs used for any computation are exclusive to this operation only. Unlike a processor, where all computations share the same ALU, an FPGA implementation instantiates independent sets of LUTs for each computation in the software algorithm. In addition to assigning unique LUT resources per computation, the FPGA differs from a processor in both memory architecture and the cost of memory accesses. In an FPGA implementation, the memory is arranged into multiple storage banks as close as possible to the point of use in the operation. This results in an instantaneous memory bandwidth, which far exceeds the capabilities of a processor.

[1] Xilinx specific

4.3 Applications

Initially FPGAs were used in telecommunications and networking. After some years, FPGAs found their way into consumer, automotive, and industrial applications [8]. In current times, FPGAs the utilization of FPGAs in data centers is really high. Their modern capabilities permit their usage for acceleration of demanding algorithms, such as search algorithms and machine learning training and inference.

Chapter 5

High Level Synthesis

In the past, the utilization of FPGAs was a challenging process, because the low-level FPGA design tools could be used only by engineers with a deep understanding of digital hardware design. A solution to this problem was introduced in the form of High Level Synthesis (HLS). HLS is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior. This creates some benefits involved with selection of HLS for FPGA programming [17]:

- Improved productivity for hardware designers
  Hardware designers can work at a higher level of abstraction while creating high-performance hardware.
- Improved system performance for software designers
  Software developers can accelerate the computationally intensive parts of their algorithms on a new compilation target, the FPGA.

5.1 HLS Phases

High-level synthesis includes the following phases:

- Scheduling
  Determines which operations occur during each clock cycle based on
  - Length of the clock cycle or clock frequency
  - Time it takes for the operation to complete, as defined by the target device
  - User-specified optimization directives

  If the clock period is longer or a faster FPGA is targeted, more operations are completed within a single clock cycle, and all operations might complete in one clock cycle. Conversely, if the clock period is shorter or a slower FPGA is targeted, high-level synthesis automatically schedules the operations over more clock cycles, and some operations might need to be implemented as multi-cycle resources.
- Binding
  Determines which hardware resource implements each scheduled operation. To im-

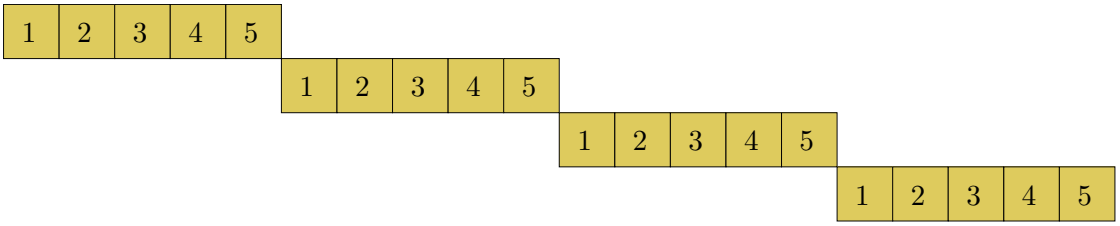


Figure 5.1: Execution of loop with no pipelining applied

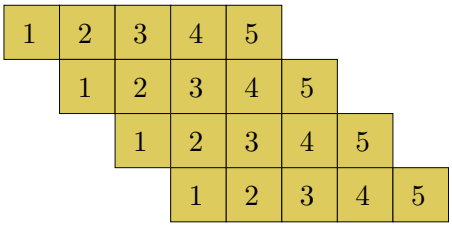


Figure 5.2: Execution of loop with pipelining applied

plement the optimal solution, high-level synthesis uses information about the target device.

- Control logic extraction
  Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design.

5.2 HLS techniques

High Level Synthesis is able to perform some techniques that greatly improve the performance of algorithm by exploiting the highly parallel design of FPGAs.

Pipelining

Pipelining allows the designer to avoid data dependencies and increase the level of parallelism in an algorithm hardware implementation [19].

We will explain what Pipelining does with an example. Let's suppose we have a set of n data(e.g an array) and we want to perform some operations on each element of this set in a loop. These operations may be numerical operations or data transfers. Let's also suppose, for ease of explaining, that each operations takes 1 cycle to be completed and there are 5 of these operations in the loop. The relevant term in this case is that the **Loop Latency (LL)** is 5. If we didn't do anything then each iteration of the loop would take 5 cycles to complete and as a result the whole loop would take $5 \cdot n$ cycles to complete (see Figure 5.1). During each clock cycle one operation is completed by the relevant hardware module. The problem is that at any given time only one module is active while the other ones are waiting for the next iteration of data to process.

Pipelining comes to solve this problem. It permits the next iteration to begin as soon

as the first first hardware module needed is freed from the previous iteration (see Figure 5.2). In our example, this is possible after only one cycle. The relevant term here is that the **Initiation Interval (II)** is 1. With pipelining the whole time needed for the execution of the loop would be $5 + (n - 1)$.

A more general type for computing the pipelined vs non-pipelined loop execution time in cycles would be

$$\text{Non-Pipelined: } n \cdot LL$$

$$\text{Pipelined: } LL + II \cdot (n - 1)$$

Considering, that the Loop Latency cannot change, most of the effort while designing an algorithm for FPGA execution goes into reducing the Initiation Interval as much as possible. A value of 1 for the II is considered ideal, because it reduces the execution time as much as possible, while at the same time not needing extra resources. This distinction is made because it is possible to make it 0. This means that all iterations are executing at the same time. This in HLS terms is called **Loop Unrolling** and is not always the best option for 2 reasons:

- The FPGA design must contain a hardware module for each element. This increases a lot the resource utilization. In our example the pipelined loop would need only 5 modules (one for each operation lasting a clock cycle), but would need $5 \cdot n$ modules, if we wanted to unroll the loop.
- The FPGA design must be able to process each data element at the same time.

Dataflow

Dataflow is another digital design technique, which is similar in concept to pipelining. The goal of dataflow is to express parallelism at a coarse-grain level. In terms of software execution, this transformation applies to parallel execution of functions within a single program.

The simplest case of parallelism is when functions work on different data sets and do not communicate with each other. In this case, HLS allocates FPGA logic resources for each function and then runs the blocks in independently. The more complex case, which is typical in software programs, is when one function provides results for another function. This case is referred to as the consumer-producer scenario.

Dataflow can be achieved with two use models for the consumer-producer scenario. In the first use model, the producer creates a complete data set before the consumer can start its operation. Parallelism is achieved by instantiating a pair of BRAM memories arranged as memory banks ping and pong. Each function can access only one memory bank, ping or pong, for the duration of a function call. When a new function call begins, the HLS-generated circuit switches the memory connections for both the producer and the consumer. This approach guarantees functional correctness but limits the level of achievable parallelism to across function calls.

In the second use model, the consumer can start working with partial results from the producer, and the achievable level of parallelism is extended to include execution within a function call. The hardware modules for both functions are connected through the use of a first in, first out (FIFO) memory circuit. This memory circuit, which acts as a extended to include execution within a function call. The hardware modules for both functions are connected through the use of a first in, first out (FIFO) memory circuit. This memory circuit, which acts as a queue in software programming, provides data-level synchronization between the modules. At any point during a function call, both hardware modules are executing their programming. The only exception is that the consumer module waits for some data to be available from the producer before beginning computation. In HLS terminology, the wait time of the consumer module is referred to as the **interval or initiation interval (II)**.

5.3 Xilinx Vivado HLS

The Xilinx Vivado HLS tool synthesizes a C function into an IP block that you can integrate into a hardware system [17].
The Vivado HLS design flow is:

1. Compile, execute (simulate), and debug the C algorithm.
2. Synthesize the C algorithm into an RTL implementation, optionally using user optimization directives.
3. Generate comprehensive reports and analyze the design.
4. Verify the RTL implementation using a push-button flow.
5. Package the RTL implementation into a selection of IP formats.

5.3.1 Directives and Pragmas

Directives and Pragmas are two sides of the same coin. In essence, they are instructions to the Vivado HLS tool in order to optimize the hardware design. Directives are specified in a configuration file, whereas pragmas are specified in the source code.

Following there is a brief presentation of some common directives that are used in the hardware implementation of this diploma thesis.

array_partition

Partitions an array into smaller arrays or individual elements. This partitioning:

- Results in RTL with multiple small memories or multiple registers instead of one large memory.
- Effectively increases the amount of read and write ports for the storage.
- Potentially improves the throughput of the design.
- Requires more memory instances or registers.

dataflow

Specifies that dataflow optimization be performed on the functions or loops, improving the concurrency of the RTL implementation. All operations are performed sequentially in a C description. Data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation. It is possible for the operations in a function or loop to start operation before the previous function or loop completes all its operations. When dataflow optimization is specified, Vivado HLS:

- Analyzes the dataflow between sequential functions or loops.
- Seeks to create channels (based on ping-pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which in turn:
- Decreases the latency
- Improves the throughput of the RTL design

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, Vivado HLS attempts to minimize the initiation interval and start operation as soon as data is available.

interface

Specifies how RTL ports are created from the function description during interface synthesis. The ports in the RTL implementation are derived from:

- Any function-level protocol that is specified.
- Function arguments
- Global variables (accessed by the top-level function and defined outside its scope)

Function-level handshakes:

- Control when the function starts operation.
- Indicate when function operation:
  - Ends
  - Is idle
  - Is ready for new inputs

pipeline

- Function pipelining
- Loop pipelining

resource

Specifies the resource (core) to implement a variable in the RTL. The variable can be any of the following:

- array
- arithmetic operation
- function argument

Vivado HLS implements the operations in the code using hardware cores. When multiple cores in the library can implement the operation, you can specify which core to use with this directive.

stream

By default, array variables are implemented as RAM:

- Top-level function array parameters are implemented as a RAM interface port.
- General arrays are implemented as RAMs for read-write access.
- In sub-functions involved in dataflow optimizations, the array arguments are implemented using a RAM ping-pong buffer channel.
- Arrays involved in loop-based dataflow optimizations are implemented as a RAM ping-pong buffer channel.

If the data stored in the array is consumed or produced in a sequential manner, a more efficient communication mechanism is to use streaming data, where FIFOs are used instead of RAMs.

unroll

Transforms loops by creating multiples copies of the loop body. A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations may also be impacted by logic inside the loop body (for example, break or modifications to any loop exit variable). The loop is implemented in the RTL by a block of logic representing the loop body, which is executed for the same number of iterations.

Chapter 6

Hardware

The acceleration of the algorithm presented in this diploma thesis is based upon the utilization of a specific FPGA accelerator card. This is the **Xilinx® Alveo™ U200 Data Center accelerator card**.

6.1 Overview

The Xilinx Alveo™ Data Center accelerator cards are a PCI Express® compliant cards designed to accelerate compute-intensive applications such as machine learning, data analytics, and video processing in a server or workstation [20].

On the Xilinx device, a platform consists of a static region and a dynamic region. The static region of the platform provides the basic infrastructure for the card to communicate with the host and hardware support for the kernel. It includes the following features:

- Host Interface (HIF): PCIe endpoint to enable communication with external PCIe host
- Direct Memory Access (DMA): XDMA IP and AXI Protocol Firewall IP
- Clock, Reset, and Isolation (CRI): Basic clocking and reset for card bring-up and operation.Reset and Dynamic Function eXchange isolation structure are required for isolation during partial bitstream download.
- Card Management Peripheral (CMP): Peripherals responsible for board health and diagnostics,debug, and programming
- Card Management Controller (CMC): UART/I2C communication to satellite controller(MSP432), QSFP, sensors and manages firmware updates from the host (over PCIe)
- Embedded RunTime Scheduler (ERT): Schedule and monitor compute units during kernel execution.

The dynamic region is the place where the accelerated kernels are designed to be stored and executed. All the algorithmic complexity comes from manipulating the resources of this region.

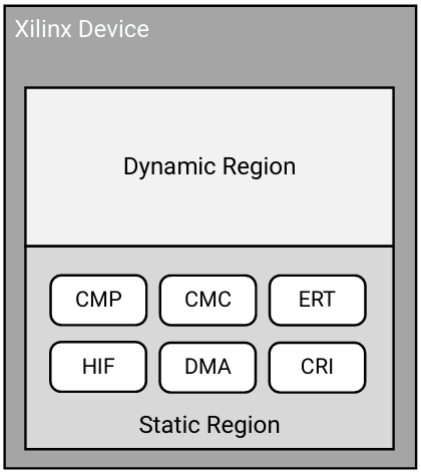


Figure 6.1: Dynamic and static regions in a platform

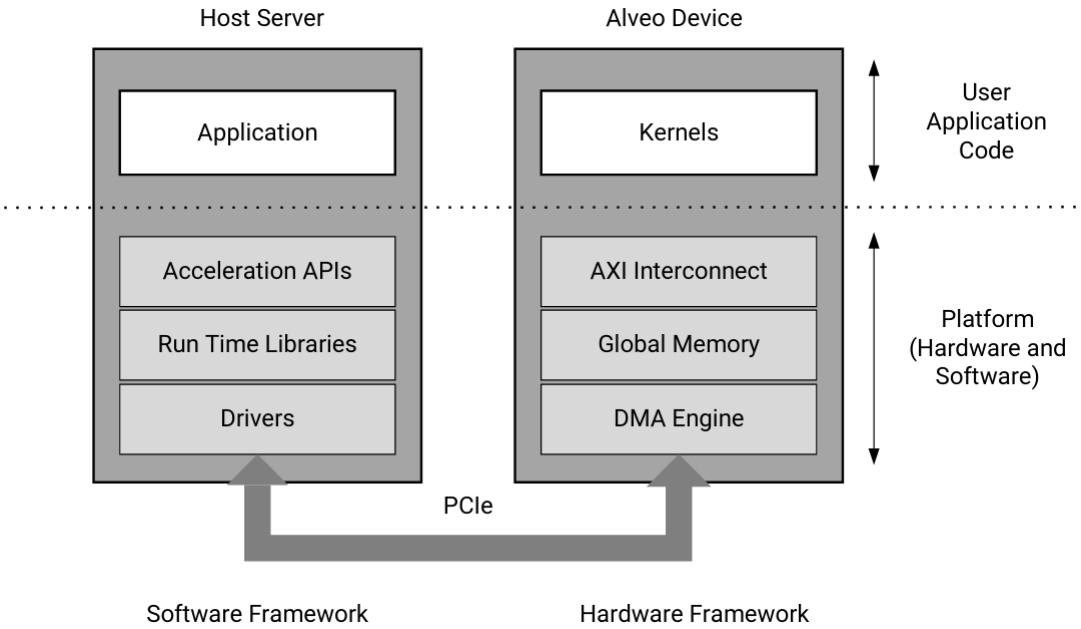


Figure 6.2: Alveo cards overview

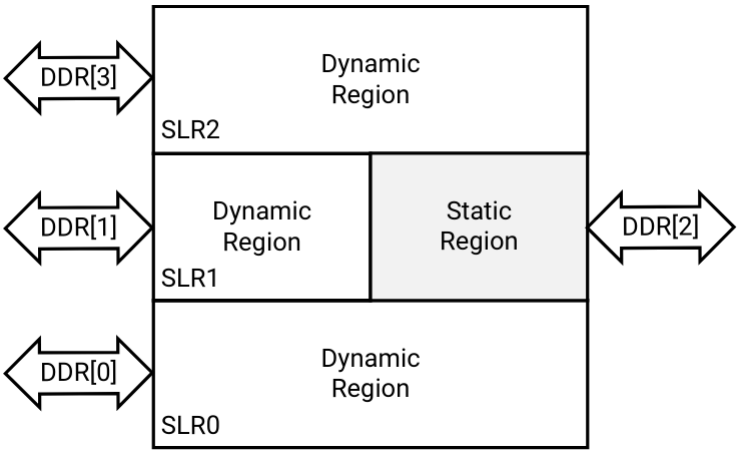


Figure 6.3: U200 floorpan

Resource	SLR0	SLR1	SLR2
LUTs	355K	160K	355K
Registers	723K	331K	723K
BRAMs	638	326	638
URAMs	320	160	320
DSPs	2265	1317	2265

Table 6.1: U200 block resources

6.2 Alveo U200 accelerator card

The Alveo U200 accelerator card is one of the available accelerator cards by Xilinx. It is a custom-built UltraScale+ FPGA that runs optimally (and exclusively) on the Alveo architecture. It features the XCU200 FPGA, which uses Xilinx stacked silicon interconnect (SSI) technology to deliver breakthrough FPGA capacity, bandwidth, and power efficiency. This technology allows for increased density by combining multiple super logic regions (SLRs). The XCU200 comprises three such SLRs. The device connects to 16 lanes of PCI Express® that can operate up to 8 GT/s (Gen3). It also connects to four DDR4 16 GB, 2400 MT/s, 64-bit with error correcting code (ECC) DIMMs for a total of 64 GB of DDR4. The device connects to two QSFP28 connectors with associated clocks generated on board [21]. The default clock to run the accelerator is 300 MHz.

Figures 6.3 and 6.4 give an idea of the structure of the device. The static and dynamic regions are shown across the FPGA SLRs, along with the available DDR memory connections associated with each SLR.
Table 6.1 shows the available resources in the dynamic region of each SLR.

As previously mentioned, the Alveo U200 card has a total of four available DDR memory banks. All but DDR[2] are located in the dynamic region. In addition, it is possible to use the device logic resources for small, fast,on-chip memory accesses as PLRAM.

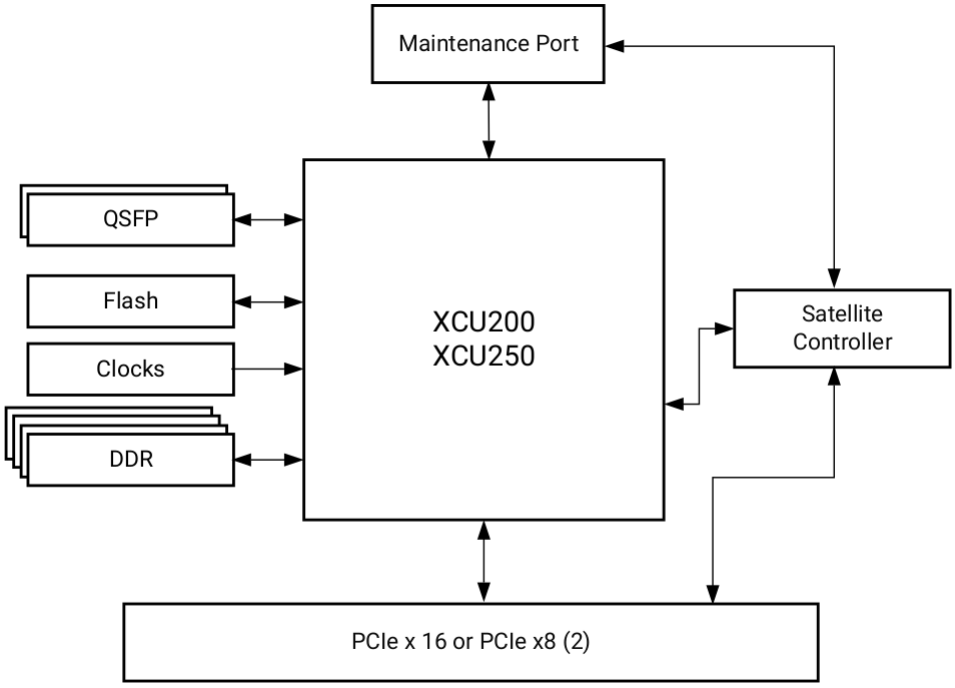


Figure 6.4: U200 diagram

If kernels are factories, then global memory banks are the warehouses through which goods transit to and from the factories. The SLRs are like distinct industrial zones where warehouses preexist and factories can be built. While it is possible to transfer goods from a warehouse in one zone to a factory in another zone, this can add delay and complexity. Using multiple DDRs helps balance the data transfer loads and improves performance. This comes with a cost, however, as each DDR controller consumes device resources.

Hands-on description

All of the above are technical details. The following is a list explaining how some of the U200 features provide great opportunities for algorithm acceleration on the FPGA.

- 4 DDR banks: The presence of 4 individual DDR banks permits the concurrent execution of 4 R/W transfer operations at the same time. This creates makes possible the even bigger acceleration of an algorithm, since we can divide the task at hand in 4 kernels, that can execute at the same, without one interacting with the other and without one kernel creating delays for the other.
- 512-bit wide memory-kernel transfers: This features permits the transfer of more data in a single clock cycle. We can exploit this by using 512-bit wide data types, such as `ap_uint<512>`. This creates speedup relative to the number of original values that can fit into the 512-bit margin (e.g 8 `double` or 16 `float` values).
- AXI burst transfers: The first read or write request to global memory is expensive, but subsequent contiguous operations are not. Transferring data in bursts hides the

memory access latency and improves bandwidth usage and efficiency of the memory controller.

The above features are mentioned again when we describe the design choices for the FPGA kernel code in Subsection 7.3.2.

Chapter 7

Accelerator Implementation

7.1 Introduction

FPGAs, as has already been mentioned, provide new opportunities to accelerate older applications. This happens because they are able to execute highly parallel code, due to the fact that they can place on the hardware chip and utilize many instances of the same compute module (e.g an adder, a multiplier, etc.). That means that the only restriction applied is the number of total resources of the FPGA and the possibilities that are offered are great. In order to fully take advantage of the capabilities of FPGA design, we first need to determine the part of the code that is best suited for FPGA execution. That part has to be:

- computationally intensive
- highly parallelizable

Both are important to be present. It has to make sense to dispatch the execution of the code to the FPGA, as in most cases there is some overhead in doing so. Dispatching code that is not that computationally intensive would most certainly result in slower execution.

7.2 Profiling and Hardware Function Selection

The process with which we can determine the part of a algorithm/program that is most computationally intensive and thus best suited for hardware acceleration is called **profiling**.

In order to perform this task one can select among a variety of tools. Our choice was `gprof`. This tool is able to analyze the performance of a program by inserting instrumentation code automatically into the program code during compilation. The output consists of two parts: the flat profile and the call graph. The flat profile gives the total execution time spent in each function and its percentage of the total running time. Function call counts are also reported. Output is sorted by percentage, with hot spots at the top of the list [9].

Table 7.1: Profiling of svm-train

functions \ datasets	a9a	skin	ijcnn1	w8a	Avg.
dot	79.03	37.41	63.71	72.9	63.26%
kernel_rbf	8.18	21.63	11.66	12.11	13.4%
get_Q	4.51	20.95	10.29	5,65	10.35%
select_working_set	5.31	14.45	8.31	5.08	8.29%

The second part of the output is the textual call graph, which shows for each function who called it (parent) and who it called (child subroutines).

Both parts of the output are equally important, as we will shortly see.

7.2.1 Original Code

Our implementation is based on version 3.24[1] of the LIBSVM library.

7.2.2 Profiling

Using `gprof` we get a timing report from various executions of the svm-train program using different datasets. The results are presented in Table 7.1. In order to not show not so useful data, we only present the functions of the report that are most called and take most of the execution time.

7.2.3 Hardware Function

In this subsection, I will try to explain the process that went behind the selection of the hardware function. I will not go into detail about the implementation of each attempt. It is logical that each dataset creates a different division of execution time among the functions, due to differences mainly in the number of features. Nevertheless, the data of the table can paint a picture of what is the reality of the situation.

Attempt 1

The function that takes most of the execution time is the dot product function named `dot`. This function, as its name implies, takes as arguments two pointers to the training vector data and returns their dot product.

In this first attempt to accelerate the training process of the LIBSVM, I tried to create an FPGA kernel that would only compute the dot product of two training vectors. The achieved execution times were not satisfactory. The problem was, and it is backed by the timing profiles, that this function has a very small execution time, so the overhead of dispatching the execution to the FPGA was too much.

[1] https://github.com/cjlin1/libsvm/releases/tag/v324

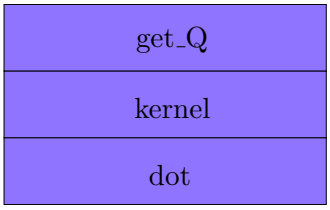


Figure 7.1: Calling stack of most time-consuming functions

Attempt 2

Upon further inspection of the original code and the `gprof` profiles, I understood the details of the calling stack of the `dot` function. This is shown in Figure 7.1

The `kernel` function is an overloaded function, depending on the initial program parameters, that takes the dot product of two training vectors and computes the SVM kernel. The available options are: `RBF, tanh, linear and polynomial`.

The fact that all calls of the `dot` function are coming for the `get_Q` function means that the latter is a better candidate for hardware acceleration. The relevant code inside `get_Q` is as follows.

```
for(j = start; j < len; j++) {
data[j] = (float)(y[i]*y[j]*
    (this->*kernel_function)(i,j));
}
```

The existence of this loop with count `len - start` (with `len` in the order of the number of training vectors) and a variable `i` that stays constant throughout creates for a nice setting for FPGA execution. What this code actually does is compute row i of matrix Q, first appearing in Equation 1.5 of Subsection 1.2.1 and of course later explained with more detail in Chapter 3. The loop starts from index `start`, because values preceding that and starting from index `0` are cached in software and are available instantly.

The initial approach was not to compute the whole loop as is. After Attempt 1 and only computing one dot product in the FPGA kernel, the next step was to compute the dot products of the whole row and afterwards compute the kernel functions in the host side code. This gave better results, but of course there was still room for improvement.

Attempt 3 (final)

The whole computation of the above loop was dispatched to the FPGA. This gave the best possible results for the given implementation of the SVM algorithm by the LIBSVM library. The following sections provide more insight into how the acceleration of this loop was possible utilizing the U200 accelerator card.

7.3 Accelerator Design

In order to explain the design choices that we made we have to take into account:

- the mechanics of the original algorithm
- the capabilities provided to us by utilizing the Alveo™ U200 accelerator card
- the aspect of retaining the existing functionality

Expanding on the last item, our goal from the beginning was to alter the original code as little as possible. Our attempt was to accelerate the original algorithm using the available tools and not make modifications that would may give significant speedups, but would alter core parts of it. The results of our FPGA version would have to match the results of the original software version.

We were able to produce two versions of the kernel code that are doing exactly that. The first version stores data in the FPGA global memory in `double` format, exactly like the original code, and the second version stores them in `float` format. The first version produces identical results to the original software, while the second version trades some accuracy with speed (more on the design specifics can be found in Subsection 7.3.2 and on the performance results in Chapter 8). We considered other versions, looking to further exploit this trade-off, but the loss in accuracy was deemed too much, which would be only be avoided with a whole restructuring of the original code. This effort would also require a more detailed accuracy analysis of the training, something that was out of the scope of our work.

Our implementation does only one basic change to the original source code (along with some other necessary additions in order to facilitate that change), which is to substitute the code of the loop in Section 7.2.3, with a call to our `Host` function called `callRowKernel` and add 3 function calls needed to translate the logic of the original program for FPGA execution.

The design choices will be presented in the following manner. We will provide the relevant information regarding the original implementation in software and the capabilities of the accelerator card and then explain our design choices. Subsection 7.3.1 will deal with the `Host` side code and Subsection 7.3.2 with the `Kernel` side code.

7.3.1 Host Code

The first thing we need to address is the way the training vector data was stored in the original code. The authors, in that version of the code, had decided to put more weight on sparse data. This lead to the usage of linked lists to store the data, with each element storing the `index` and `value` of the non-zero dimensions of the training vectors. Consequently, the `dot product` function has to traverse the relevant linked lists.

If the `index` of both current nodes is the same, it multiplies their `values` and adds this to a sum. Depending on the number of indices that have a non-zero value for both

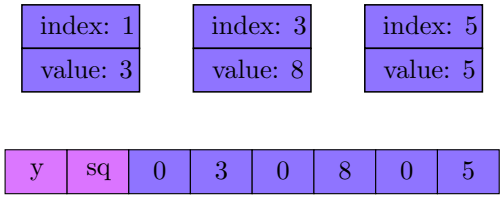


Figure 7.2: Linked list to array conversion - 6 feature training vector

training vectors, we have a different number of multiplications and additions. This creates an unpredictability on the computation, that would not make possible the parallelization of more that one dot product computation, which was the target in the FPGA kernel code. The need for repetitiveness inside the `Kernel` code made us abandon that way and instead store the data inside the `DMA global memory` of the accelerator card in sequential fashion using arrays, padding with zeros where needed.

Furthermore, to cover the case of an SVM formulation using classification labels and maybe using the `RBF Kernel` we added 2 elements at the beginning of our arrays. The first storing y_i and the second storing the sum of squares $\sum_{i=0}^{dimensions} x_i^2$ for the given training vector (see Figure 7.2).

One of the major advantages of using the Alveo™ U200 is the fact that its `global memory` has 4 banks. This permits `R//W` operations to happen from 4 different kernels at the same time. To properly use this feature we divided the training vector data in 4 arrays to be stored in the `global memory`. To give an example of how that is happening, let's consider a dataset with 1 million training vectors. The relevant data of the first 250 thousand vectors would be stored in Bank 0, of the next 250 thousand vectors in Bank 1, etc. In the case where the total number of training vectors is not divisible exactly by 4 we store the data remaining training vectors in the last bank.

Another major advantage is the `512-bit` width of the transfer bus between the FPGA and the `global memory` of the accelerator card. This makes possible `512-bit` transfers in one clock cycle. In our algorithm, this translates to the transfer of 8 `doubles` per clock cycle on the first version or 16 `floats` in the second version. To accommodate this feature, we extend the dimensions of the training vector to a multiple of 8 (or 16 respectively), padding with zeros when needed. That way, transfers in the `Kernel` code would never cross training vector dimension boundaries.

Without explaining how the `Kernel` code works (see Subsection 7.3.2), we need to explain what it does, so that we can describe the process of calling it in the `Host` side. Each of the 4 kernels is assigned to one of the 4 banks of the `global memory`. Each kernel takes as input parameters the training vector i (supposing we are performing the computation of row i of matrix Q), a `start` parameter (denoting the first vector in the bank that we want to compute the SVM kernel function for - this is closely related to the `start` variable of the original code) and a `products` parameter (denoting the number of training vectors that we want to compute the SVM kernel function for, starting beginning

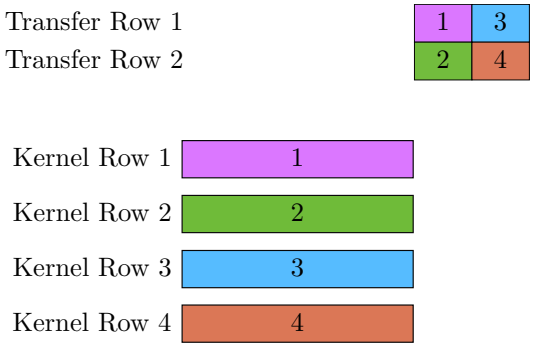


Figure 7.3: Initial approach to parallel kernel execution

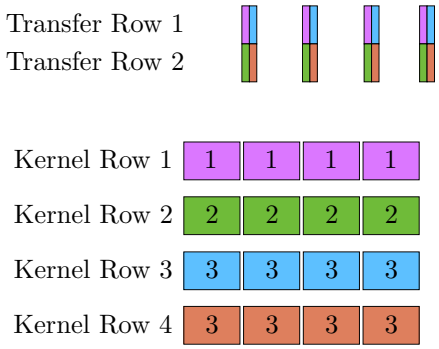


Figure 7.4: Final approach to parallel kernel execution

from `start` and return the results). Carefully selecting these parameters for the 4 kernels we can partition the `len - start` computation of the original loop into more chunks that can be executed in parallel, thus achieving a good speedup.

The initial approach was to partition the computation in 4 chunks, one for each kernel. This was fairly logical to do, since we can only execute up to 4 kernels in parallel. The attempt was successful for small datasets, but for large datasets the overhead of transferring the results from the FPGA global memory back to the host after the kernels had finished executing was not permitting the acceleration that we had anticipated (see Figure 7.3).

To tackle this problem we further partitioned the computation designated for each kernel into more chunks. That way the transfer of the results happens in parallel with the next kernel execution in line (see Figure 7.4). Regarding the size of this chunks, it was observed that we achieved acceptable results for sizes around 125 thousand vectors. If we set the limits to a lower value the parallelization of the kernel executions was not optimal, because the execution times were rather small and the overhead of kernel execution was big relative to the whole time of execution.

The final item we had to address from the original source code was `swapping`. The algorithm at various points during the execution performs a `Shrinking` procedure. This procedure marks some training vectors as unnecessary for the rest of the training process. To do that it swaps these vectors with others from the "end" of the list and reduces the "working length". Using a linked list makes this quite easy, by manipulating their

pointers. What we had to do to update the data in the `global memory` of the accelerator card, was keep track of those swaps, and before a new row computation was needed, make the necessary changes to the arrays and migrate them to the `global memory`, replacing the old sequence of values. This creates some overhead, compared to the original software version, but it happens so rarely that it doesn't affect the acceleration of the training.

7.3.2 Kernel Code

The functionality of the code was first mentioned in Subsection 7.3.1. What each kernel does is that it starts reading the training vector values from the global memory, does the necessary computations and returns the results back to the global memory. From there they are transferred back to the host.

This process of reading from the memory and writing back to it, and considering that only one DMA transfer operation can be happening at any given time (read or write), sets a lower bound regarding the time complexity of the whole task. As mentioned before, the `data bus` is able to transfer 8 `doubles` or equivalently 16 `floats` per clock cycle. We took advantage of this feature, not only for the reading process, but for the writing process as well. The LIBSVM algorithm expects float values to be returned, so we make `writes` of 16 `floats` at a time, so as to not disrupt the reading process that much. The resulting latency of our implementation is approximately $n * d + \frac{n}{16}$ cycles, where n is the number of training vectors to compute the SVM kernel function for and d is the number of dimensions divided by 8 (or 16 in the case of the second version), as they were selected in the `Host` side code. For example, to compute a row of 1 million elements of 32 dimensions each, each kernel would take a little more than 4.0625 (or 2.0625 for the second version) million clock cycles.

In order to achieve this result, where the actual time of execution is only determined by the time needed to read the data and write the results back to the global memory, we made use of the `pipeline`, `dataflow` and `unroll` directives. In the code these are specified using `#pragma HLS PIPELINE`, `#pragma HLS DATAFLOW` and `#pragma HLS UNROLL` respectively, in the required areas.

Another feature that permitted the lowest possible number of cycles was utilizing burst reads and writes between the FPGA and the `DMA memory`. The guidelines suggested, and we implemented that, that in order to achieve that all the reads and the writes should be contained withing a single loop, in order to show to the HLS tool that these transfers are to happen in burst.

The `pipeline` directive permits the parallel execution of a loop's body, starting each iteration as soon as possible, even before the previous one has finished. The `unroll` directive creates a unique module for every iteration of a loop in order for the loop to be executed all at once concurrently. The `dataflow` directive, along with the usage of `hls::streams`, facilitates the passing of data from one hardware module to the next (in code terms, output from one function that is input to another), before the first module

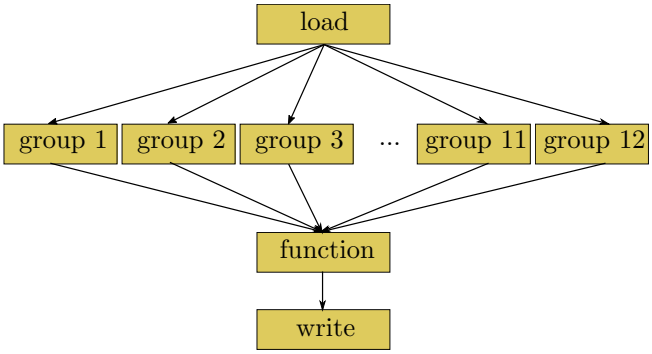


Figure 7.5: Kernel dataflow

has finished its execution. To better explain the `dataflow` directive usage we will describe the passing of data through the hardware modules in our design. Figure 7.5 provides a visual representation of the functions/modules defined in each kernel.

Module `load` reads data from the `global memory`; 8 `doubles` or 16 `floats` on every clock cycle. We know the total dimensions for every training vector, so we know how to group those reads per training vector. Every clock cycle, each of the 8 (or 16) values is multiplied with the corresponding value from the `base vector` (the vector with index i). To accomplish the multiplication of all the respective dimensions in one clock cycle we make use of the `unroll` directive, by adding `#pragma HLS UNROLL` in the loop code.

Afterwards, a tree-style addition of the 8 (or 16) results starts (3 levels or 4 levels respectively) and the result of these is streamed to one of the 12 `group` modules. These functions start adding the incoming values in sequential manner in groups of size $\frac{dimensions}{8}$ (or $\frac{dimensions}{16}$). These sums are the dot products for every training vector with the `base vector`. The sum is streamed to the `function` module, which computes the SVM kernel function. From there the results are streamed one by one to the `write` module, which groups them by 16 and writes them back to the `global memory`.

The interesting part of the dataflow model is the presence of the 12 `group` modules. In our attempt to make the kernels as general as possible in order to be able to train datasets with different number of features using the same FPGA bitstream, we had to abandon the notion of a tree-style addition for the $\frac{dimensions}{8}$ (or $\frac{dimensions}{16}$) values coming from the `load` module, as the Vivado HLS compiler would simply not pipeline the whole process, not being able to determine the depth of the tree at compile time.

For that reason, a sequential addition process was selected. The problem was that each addition takes more clock cycles to be completed, while the `load` module is pipelined and can produce an output value in every clock cycle. In order to not have the values of the following training vectors wait idle in a queue and stall the whole pipeline, we thought about feeding these values to a different module. Each of these `group` modules is computing the dot product for a different training vector. The trick is that by the time the first value of the 13th training vector is ready to passed to a `group` module, the addition process of the 1st will have been completed. The math that supports this is simple. The

compile reports showed that each addition takes at most 12 cycles to be completed. In that case the dot product of a training vector with the base vector takes $12 \cdot \frac{dimensions}{8}$ (or $12 \cdot \frac{dimensions}{16}$) cycles to be completed. In that time, the `load` module has produced exactly that many values, corresponding to 12 different training vectors. At the end, the number of cycles needed to make an addition dictated the number of modules needed to compute the dot products in parallel.

This is, in my opinion, the single most important design feature in the whole work of the diploma thesis. Previously, we had the above mentioned restriction, where the HLS tool would not synthesize correctly the tree-style addition of an unknown number of numbers. That had made us create a different executable for datasets with different number of features. That way, generalization was not possible. Our goal from the beginning was to provide a single executable, able to accelerate the training for many different datasets. That was what we achieved at the end.

Up to now, we have only described the process of computing the dot product, but the data stored for each training vector contain also its label `y` and its sum of squares `sq`. These, after being multiplied and added respectively with the corresponding values of the base vector, are streamed directly from the `load` module to the `function` module. The latter computes one of the following SVM kernel functions:

- Linear: $y \cdot dot$
- Polynomial: $y \cdot (gamma \cdot dot + coef)^{degree}$
- RBF: $y \cdot e^{-gamma \cdot (sq - 2 \cdot dot)}$
- tanh: $y \cdot tanh(gamma \cdot dot + coef)$

where $gamma, degree, coef$ are parameters of the SVM algorithm.

Appendix A contains the kernel code, along with some comments to guide the reader in understanding its functionality.

7.4 Resource Utilization

The accelerator card introduces some limitations in the amount of resources we can utilize. Apart from the total number of resources that are available, there exist some other practices that are recommended in order to achieve better performance. One such practice that we took into account while designing was that each FPGA kernel should be contained in one Super Logic Region (SLR)and not cross the boundaries between them. Considering the available resources per SLR (first shown in Table 6.1) and our design of 4 FPGA kernels, we are bound by the restrictions shown in Table 7.2.

The only parameter that can be changed and affects the resource usage of the kernels is the maximum number of dimensions per training vector supported. We found out that the maximum number of features that can supported by the `double` FPGA kernel is 8000,

Table 7.2: U200 resource restrictions per kernel

Resource	SLR0	SLR1	SLR2	Limit/Kernel
LUTs	355K	160K	355K	177.5K
Registers	723K	331K	723K	361.5K
BRAMs	638	326	638	319
URAMs	320	160	320	160
DSPs	2265	1317	2265	1132

Table 7.3: Resources per kernel - double version

Resource	Used	Available	Utilization
BRAM_18K	278	319	87.1%
DSP48E	426	1132	37.6%
FF	93777	361686	26%
LUT	62764	177415	35.4%
URAM	8	160	5%

while the maximum number of features that can be supported by the `float` FPGA kernel is 65000. Tables 7.3 and 7.4 show in detail the resource usage in these two "maximum" cases.

As we can see the resource that imposes the limit in both cases is the BRAM. BRAMs are storage units in the FPGA and that's why the float version is able to support a higher number of features, because the representation of float is smaller than that of a double.

Table 7.4: Resources per kernel - float version

Resource	Used	Available	Utilization
BRAM_18K	314	319	98.4%
DSP48E	391	1132	34.5%
FF	93777	361686	23.1%
LUT	62764	177415	31.8%
URAM	8	160	5%

Chapter 8

Performance Evaluation

The objective of this chapter is to present the speedups achieved using our implementation compared to the original software version. The FPGA speedups are relative to a multithreaded execution on the CPU that utilizes all 4 available cores. The CPU on which we run the tests was the AMD Ryzen™ 2200G, a CPU with a base clock speed of 3,5GHz. We try to explore how the training set size and the number of features affect these speedups.

In order to do that, we did a grid-like exploration, defined by different training set sizes and number of features. To achieve that in an objective manner we created some custom datasets. The original dataset was Epsilon[1]. This is a dense dataset that contains 400000 training vectors with 2000 features each. What we did was a take a subset of this dataset with 20000 training vectors. For each number of features that we wanted to test (5, 10, 25, 50, 75, 100, 200, . . . , 2000), we removed the features that we didn't need and we copied this new base set of 20000 training vectors many times in order to create new custom datasets with (20000, 40000, . . . , 500000) training vectors. The final step was to measure execution times for 1 row computation on the CPU and the FPGA.

Figures 8.1, 8.2 and 8.3 contain information about the execution time measured for every set of parameters (number of dimensions and training size) for the 3 different versions we checked (double FPGA version, float FPGA version and multithreaded CPU version). The black points denote the actual measurements and the lines are fitted to the data in each case. The fitting of the lines is almost perfect for all 3 versions, with the multithreaded one having only some minor deviances, that are mostly created by measurement accuracy errors. There is a linear relationship between the number of features, the training size and the actual execution time of one row computation.

This relationship creates an interesting situation when it comes to the actual speedup between the FPGA versions and multithreaded CPU one. Supposing the fitted line of the FPGA version for a given number of features is $f1(x) = ax + b$ and the fitted line for the multithreaded CPU version is $f2(x) = cx + d$, where x is the variable of the training size. In that case, both b and d have the role of the basic overhead of the execution on

[1]https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.html#epsilon

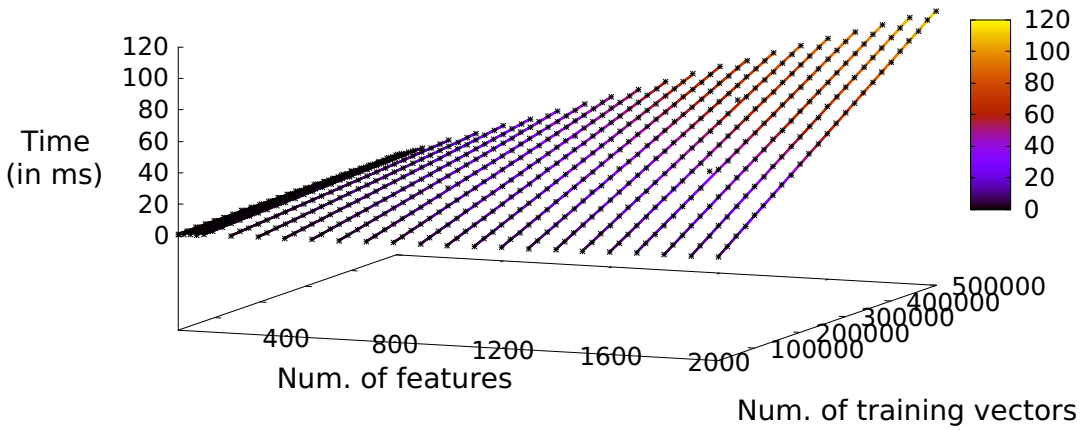


Figure 8.1: Double version: How the training size affects the execution time

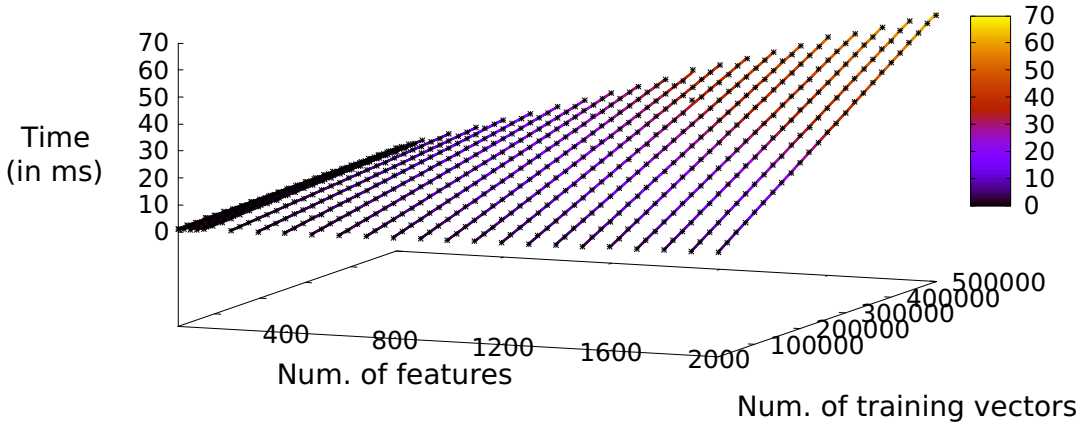


Figure 8.2: Float version: How the training size affects the execution time

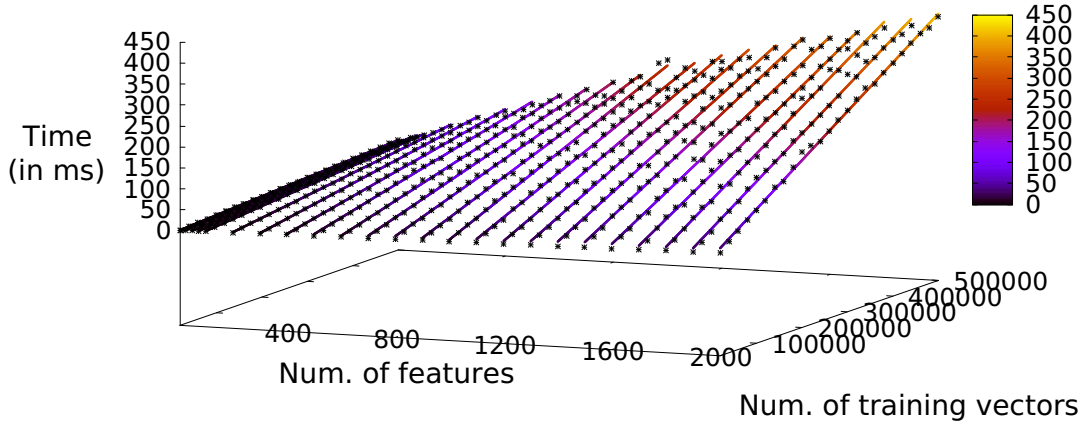


Figure 8.3: Multiple threads: How the training size affects the execution time

the platform. The equation of the speedup would then be $f(x) = \frac{cx+d}{ax+b}$. This function (supposing that it is not constant) is a hyperbola and has some interesting traits:

- $\lim_{x \to -\infty} f(x) = \lim_{x \to +\infty} f(x) = \frac{c}{a}$
- $\lim_{x \to -\frac{b}{a}^-} f(x) = \begin{cases} +\infty, & f1(-\frac{b}{a}) < 0 \\ -\infty, & f1(-\frac{b}{a}) > 0 \end{cases}$
- $\lim_{x \to -\frac{b}{a}^+} f(x) = \begin{cases} -\infty, & f1(-\frac{b}{a}) < 0 \\ +\infty, & f1(-\frac{b}{a}) > 0 \end{cases}$

In our case, since we only deal with positive numbers and the nature of the problem dictates that $a, b, c, d > 0$, the above traits determine the direction from which the values come before settling towards the $\frac{c}{a}$ value. If $f1(-\frac{b}{a}) < 0$, then the values $f(x)$ are lower and are increasing as x increases. If $f1(-\frac{b}{a}) > 0$, then the values $f(x)$ are higher and are decreasing as x increases.

Figures 8.4 and 8.5 contain the relevant information about the speedup of the two FPGA versions compared to the multithread CPU execution. We can observe the convergence of the graph to the "limit" value as the training size becomes bigger for every different number of features. Furthermore, the speedup values are growing as x grows.

We can also observe that the maximum speedup that can be achieved varies different for every different number of features. In Figure 8.6 we have the graphs of the functions of the slope of the lines of Figures 8.1, 8.2, 8.3, that show the execution time of the different version we measured. In the same way that the functions of speedup for every number of features are fractions of the linear functions of execution time, the function of the maximum speedup by number of features (see Figure 8.7) is a fraction of these

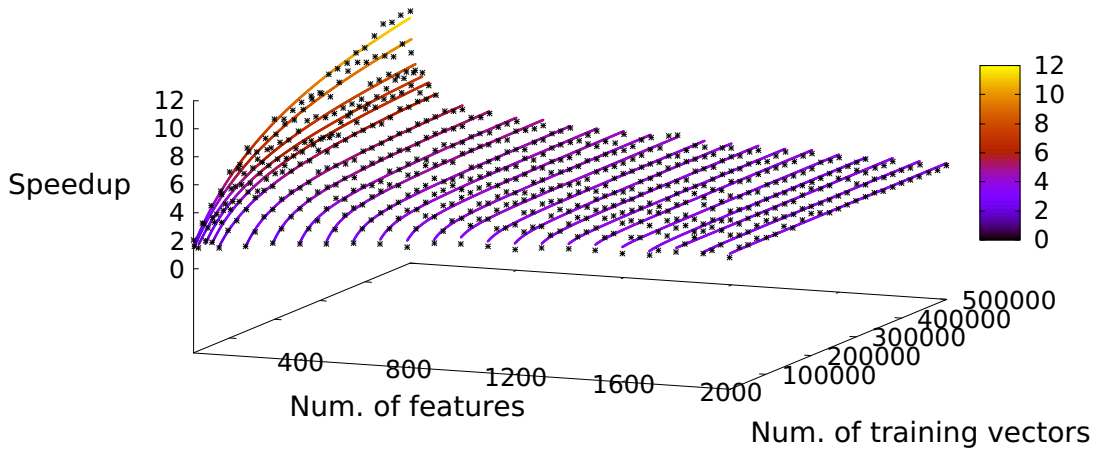


Figure 8.4: Double version: How the training size affects the speedup

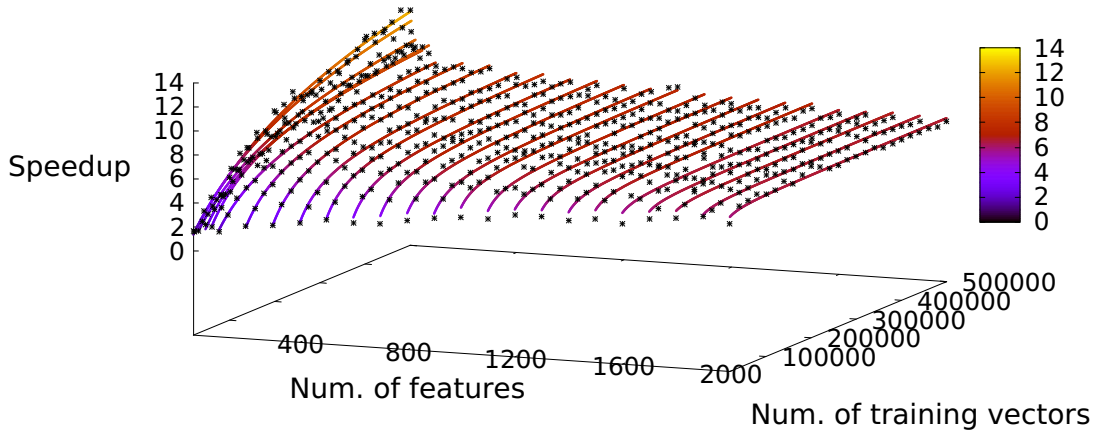


Figure 8.5: Float version: How the training size affects the speedup

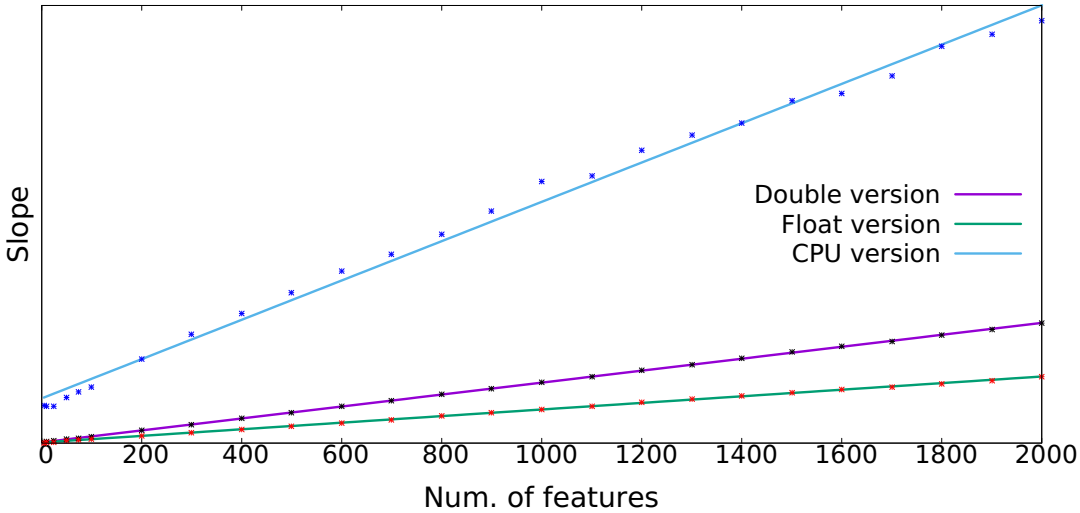


Figure 8.6: Slope of execution time lines by number of features

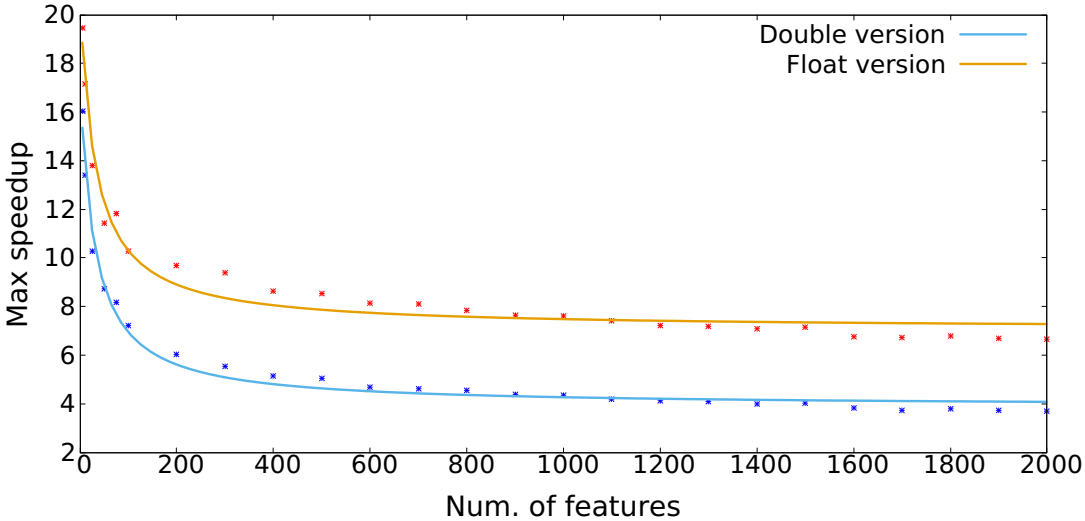


Figure 8.7: Maximum speedup by number of features

linear functions of the slopes. This relationship defines the shape of the graph in Figure 8.7 (again it's a hyperbola), that shows that for small number of features the maximum speedup is higher. The limits are about 3.5x and 7x for the double and the float version respectively.

Chapter 9

Related Work

In this chapter, we will briefly present work similar to ours. To the best of our knowledge and according to the list of "Interfaces and Extensions to LIBSVM" available on the library website[1], there has not been implemented a direct extension to LIBSVM, utilizing FPGAs. In that regard our work unique. There have been many implementations of the SVM algorithm for FPGAs, for both training and inference. In this chapter, we will only present some of the drawbacks of the implementation of LIBSVM regarding its potential for FPGA acceleration and how some other work has tried to tackle them.

Sequential Minimal Optimization is not well-scalable for huge data applications. In [11] Stochastic Gradient Descent is used as an alternative. This work also experiments with both single-precision floating point and fixed-point(5 bits integer and 20 bits fractional part) numerical representations. Their speedups seem to be very high, but their limitation is the low number of features supported by their design.

Instead of replacing SMO altogether there have been efforts to improve it for hardware acceleration. One of the disadvantages of the conventional SMO implementation used in LIBSVM is the need of data from only 2 row computations in each iteration. Caching further reduces this amount at times and only one new row computation is needed per iteration. This prevents the parallelization of more computations in the FPGA and thus it is technically a bottleneck of the original algorithm. The work in [12] addresses this limitation, by creating a variant of SMO called Hybrid Working Set (HWS), that creates working sets of bigger size of which the computations are grouped in columns, thus increasing the spatial locality of data. This work also supports training of training vectors with a higher number of features (with all the limitations that this enforces on the speedup).

The original SMO algorithm has an additional inefficiency. It checks the optimality of the remaining samples based on the assumption that the current two optimized samples satisfy the optimality. Thus it may identify samples satisfying the optimality as violating ones, and vice versa, which leads to additional iterations. Keerthiet al. [15] uses the boundaries of the sample subsets to select the samples to be optimized in the modified SMO (MSMO) algorithm, which surpasses the heuristic selection method of the SMO

[1]https://www.csie.ntu.edu.tw/ cjlin/libsvm/

algorithm. The boundaries are also used to check the optimality of the samples, which avoids the optimality-satisfaction assumption in the SMO algorithm and consequently requires fewer iterations and performs more efficiently. The work in [14] utilizes this modified version of the SMO algorithm in order to create a design which energy efficiency is an important aspect of.

The SVM kernels supported in LIBSVM are not all well-tailored for parallel hardware execution. Functions such as *exp* and *tanh* do not exploit all the capabilities of reconfigurable architecture. In [13] an implementation is proposed utilizing the Hardware Friendly Kernel (HFK). As it name implies, this kernel is better suited for hardware parallelization, having the advantage of being able to be computed with only shifts and additions rather than multiplications. This work also produces exciting speedups, but is again limited by the supported number of features (up to 64) of their design.

Finally, we have to make a reference to a work utilizing techniques similar to ours. In [16] they are utilizing High Level Synthesis in order to perform training with the SVM algorithm on a Zynq device. The difference in this case is that they are using a variant of the SVM algorithm, called Least Square SVM (LS-SVM), which has a lower computational complexity, as it solves a set of linear equations instead of a quadratic programming for standard SVM.

Reading the above brief reference of related work, it is evident that there exist many ways the original SVM algorithm can be improved and customized in order to be executed on FPGAs. These possibilities for improvement create a trade-off between speed, accuracy, utilization of hardware resources and wide support of datasets with different characteristics.

Chapter 10

Conclusion

In this diploma thesis, we present the results of our work in an attempt to accelerate the LIBSVM library for Machine Learning training on FPGAs.

Our effort was not centered on improving the original algorithm by ways of making modifications to it or replacing core parts of it, but instead utilizing High Level Synthesis in order to exploit as many of the capabilities of the FPGA accelerator card (Xilinx® Alveo™ U200 Data Center) as possible. Extra care was taken in order to parallelize the FPGA kernel code to a point where the only bottleneck was memory transfer operations to and from the DMA memory of the accelerator card, a bottleneck that could not be avoided, since, by providing support for training of a large amount of data, we could not store the necessary values in the space restricted FPGA local memory. The timing experiments show that, compared to a multithreaded CPU execution on a Ryzen™ 3 2200G, a CPU with a base clock speed of 3.5 GHz, we can achieve speedups of about 7x in the general case of our fastest version and up to 14x in some edge cases. This edge cases refer to datasets with few number of features and a substantial number of training vectors.

10.1 Future Improvements

The biggest bottleneck of the algorithm is coming from the SMO algorithm, due to the fact that only one row computation is asked from the FPGA at each time. I would argue that if anyone wanted to accelerate even more the algorithm, they would have to improve that part. As shown in Chapter 9 there have been efforts to substitute the SMO algorithm with another decomposition algorithm, more suited to parallel execution.

Another point of improvement would be the exploration of utilization of different data representations in the FPGA kernel. In our work, we have shown that it is possible to use a float representation of data, that doubles the speedup while also not losing in accuracy that much at the same time. There have been efforts in Machine Learning research to train networks with `INT8` representation of data. Such an improvement would require huge changes to the original algorithm of the LIBSVM library in order to not compromise the accuracy of the training. It is even more probable that it would require a completely new

implementation of the SVM algorithm. In any case, the results would be even better, as the speedup would be higher and the resource utilization would be lower, making possible the training of datasets with an even higher number of features.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press, 2016.

[2] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". Machine Learning 20, 273-297, 1995.

[3] Philippe Coussy and Adam Morawiec. High-Level Synthesis: From Algorithm to Digital Circuit. Springer, 2008

[4] Tom Mitchell. Machine Learning. McGraw Hill, 1997.

[5] Learning from Data: Course Notes, Lectures 14-15, California Institute of Technology, 2012.

[6] Support Vector Machine https://en.wikipedia.org/wiki/Support_vector_machine

[7] The kernel trick
https://en.wikipedia.org/wiki/Kernel_method#Mathematics:_the_kernel_trick

[8] Field Programmable Gate Arrays https://en.wikipedia.org/wiki/Field-programmable_gate_array

[9] The gprof utility
https://en.wikipedia.org/wiki/Gprof

[10] Chih-Chung Chang and Chih-Jen Lin, LIBSVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011. Software available at

[11] Felipe Fernandes Lopes, João Ferreira and Marcelo Fernandes. Parallel Implementation on FPGA of Support Vector Machines Using Stochastic Gradient Descent. Electronics. 8. 10.3390/electronics8060631, 2019.

[12] Sriram Venkateshan, Alap Patel and Kuruvilla Varghese. Hybrid Working Set Algorithm for SVM Learning With a Kernel Coprocessor on FPGA. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on. 23. 2221-2232. 10.1109/TVLSI.2014.2361254, 2015.

[13] Daniel Holanda Noronha, Matheus Torquato and Marcelo Fernandes. A Parallel Implementation of Sequential Minimal Optimization on FPGA. Microprocessors and Microsystems. 69. 10.1016/j.micpro.2019.06.007, 2019.

[14] L. Feng, Z. Li and Y. Wang, VLSI Design of SVM-Based Seizure Detection System With On-Chip Learning Capability, IEEE Transactions on Biomedical Circuits and Systems, vol. 12, no. 1, pp. 171-181, Feb. 2018, doi: 10.1109/TBCAS.2017.2762721.

[15] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, Improvements to Platt's SMO algorithm for SVM classifier design, Neural Comput., vol. 13, pp. 637–649, 2001.

[16] M. Ning, W. Shaojun, P. Yeyong and P. Yu, Implementation of LS-SVM with HLS on Zynq, 2014 International Conference on Field-Programmable Technology (FPT), Shanghai, 2014, pp. 346-349

[17] Xilinx® Inc., Vivado Design Suite User Guide: High-Level Synthesis (UG902 v2019.2), January 13, 2020

[18] Xilinx® Inc., Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393 v2019.2), February 28, 2020

[19] Xilinx® Inc., Introduction to FPGA Design with Vivado High-Level Synthesis (UG998 v1.1), January 22, 2019

[20] Xilinx® Inc., Alveo Data Center Accelerator Card Platforms: User Guide (UG1120 v1.2), June 26, 2020

[21] Xilinx® Inc., Alveo U200 and U250 Data Center Accelerator Cards Data Sheet (DS962 v1.2.1), December 9, 2019

Appendices

Appendix A

Kernel Code

A.1 Load Function/Module

```
void row_load(...)
{
    uint512 bvector[MAX_DIMENSIONS / 8];
   #pragma HLS RESOURCE variable=bvector core=XPM_memory uram
load_base:
    for (int i = 0; i < dimensions / 8; i++) {
       #pragma HLS LOOP_TRIPCOUNT min=dim/8 max=dim/8
       #pragma HLS PIPELINE II=1
         bvector[i] = base[i];
    }
```

The above part reads the data of the base vector, which corresponds to the line of matrix Q that we want to compute.

```
    int j = 0;
    int s = 0;
load_vectors:
    for (int i = 0; i < products * dimensions / 8; i++) {
       #pragma HLS LOOP_TRIPCOUNT min=prod*dim/8 max=prod*dim/8
       #pragma HLS PIPELINE II=1
         uint512 buf = data[i];
```

This part is the initial part of the loop that reads the data of the other training vectors and performs the initial processing. By including the `reads` in a `for loop` we notify the HLS tool that the reads need to happen in burst mode.

```

         double y, square;
         double r[8];
       #pragma HLS ARRAY_PARTITION variable=r complete

         for (int m = 0; m < 8; m++) {
            #pragma HLS LOOP_TRIPCOUNT min=8 max=8
            #pragma HLS UNROLL

```

```
27            uint64 rbase = bvector[j].range((m + 1)* 64 - 1, m * 64);
28            uint64 rvector = buf.range((m + 1)* 64 - 1, m * 64);
29            double base = *(double *)(&rbase);
30            double vector = *(double *)(&rvector);
31
32            if (j == 0 && m == 0) {
33                y = base * vector;
34                r[0] = 0;
35            }
36            else if (j == 0 && m == 1) {
37                square = base + vector;
38                r[1] = 0;
39            }
40            else {
41                r[m] = base * vector;
42            }
43        }
```

The above part is when we make the necessary computations between each training vector and the base vector. We multiply the features and the value `y`, and we add the sums of the squares of the features. Notice the complex index arithmetic, in order to process the data coming in 512 bits that need to be divided in `double`s.

In order for the computation to happen in one clock cycle we add the `unroll` directive, which creates a different hardware module for each iteration of the loop. The trick is that, in order to be able to write `array r` at different memory spots at one cycle, we need to use the `partition` directive on that array. This makes sure that each element of the array is stores in registers and a BRAM, thus operations can happen individually

```
44        r[0] += r[1];
45        r[2] += r[3];
46        r[4] += r[5];
47        r[6] += r[7];
48        r[0] += r[2];
49        r[4] += r[6];
50        r[0] += r[4];
```

Here is the tree-style addition in 3 levels for the double version. In the float version we add 16 values together in 4 levels.

```
51
52        if (j == 0) {
53           #pragma HLS occurrence cycle=dim/8
54            ap_uint<128> pair;
55            uint64 ry = *(uint64 *)(&y);
56            uint64 rsquare = *(uint64 *)(&square);
57            pair(63, 0) = ry;
58            pair(127,64) = rsquare;
59            fStream << pair;
60        }
```

Variable `j` is a counter to check when we start reading data from a new training vector. If `j == 0` that means that the 2 values of this set of data are `y` and `sq`, the product of

ys and the sum of sums between the training vector and the base vector. In that case, it need to be streamed to the `function` module.

```
61
62          if (s == 0) {
63              gStream0 << r[0];
64          }
65          else if (s == 1) {
66              gStream1 << r[0];
67          }
68          else if (s == 2) {
69              gStream2 << r[0];
70          }
71          else if (s == 3) {
72              gStream3 << r[0];
73          }
74          else if (s == 4) {
75              gStream4 << r[0];
76          }
77          else if (s == 5) {
78              gStream5 << r[0];
79          }
80          else if (s == 6) {
81              gStream6 << r[0];
82          }
83          else if (s == 7) {
84              gStream7 << r[0];
85          }
86          else if (s == 8) {
87              gStream8 << r[0];
88          }
89          else if (s == 9) {
90              gStream9 << r[0];
91          }
92          else if (s == 10) {
93              gStream10 << r[0];
94          }
95          else if (s == 11) {
96              gStream11 << r[0];
97          }
98
99          j++;
100         if (j == dimensions / 8) {
101             j = 0;
102             s++;
103             if (s == 12)
104                #pragma HLS occurrence cycle=12
105                 s = 0;
106         }
107     }
108 }
```

At last, depending on the counter `s` at the time, the result of the tree-style addition is being streamed to one of the 12 `group` modules, that perform the final addition to calculate the dot product.

A.2 Group Function/Module

```
1  void row_group(...)
2  {
3      int j = 0;
4      double sum = 0;
5  group_loop:
6      for (int i = 0; i < products * dimensions / 96; i++) {
7         #pragma HLS LOOP_TRIPCOUNT min=prod*dim/96 max=prod*dim/96
8         #pragma HLS PIPELINE II=12
9
10         double res = gStream.read();
11         sum += res;
12
13         j++;
14         if (j == dimensions / 8) {
15             sStream << sum;
16             j = 0;
17             sum = 0;
18         }
19     }
20 }
```

This code creates 12 different modules, as explained in Subsection 7.3.2. The purpose is to not delay the pipeline, while we wait for an addition operation to complete.

A.3 Function Function/Module

```
1  void row_function(...)
2  {
3      int s = 0;
4  function_loop:
5      for (int i = 0; i < products; i++) {
6         #pragma HLS LOOP_TRIPCOUNT min=prod max=prod
7         #pragma HLS PIPELINE II=1
8
9          ap_uint<128> pair = fStream.read();
10         uint64 ry = pair.range(63,0);
11         uint64 rsquare = pair.range(127,64);
12         double y = *(double *)(&ry);
13         double square = *(double *)(&rsquare);
```

We first receive the `y` and `sq` values.

```
14
15          double tres;
16          if (s == 0) {
17              tres = sStream0.read();
18          }
19          else if (s == 1){
20              tres = sStream1.read();
21          }
22          else if (s == 2){
23              tres = sStream2.read();
24          }
25          else if (s == 3){
26              tres = sStream3.read();
27          }
28          else if (s == 4){
29              tres = sStream4.read();
30          }
31          else if (s == 5){
32              tres = sStream5.read();
33          }
34          else if (s == 6){
35              tres = sStream6.read();
36          }
37          else if (s == 7){
38              tres = sStream7.read();
39          }
40          else if (s == 8){
41              tres = sStream8.read();
42          }
43          else if (s == 9){
44              tres = sStream9.read();
45          }
46          else if (s == 10){
47              tres = sStream10.read();
48          }
49          else if (s == 11){
50              tres = sStream11.read();
51          }
52          double res = tres;
```

Afterwards, we receive the dot product depending on the `group` module it was assigned to.

```
53
54          if (type == 0) { //linear
55              res = y * res;
56          }
57          else if (type == 1) { //polynomial
58              res = y * pow(gamma * res + coef, degree);
59          }
60          else if (type == 2) { //RBF
61              res = y * exp(-gamma * (square - 2 * res));
```

```
62          }
63          else if (type == 3) { //sigmoid
64              res = y * tanh(gamma * res + coef);
65          }
66
67          float fres = (float)res;
68          bundleStream << fres;
69          s++;
70          if (s == 12)
71             #pragma HLS occurrence cycle=12
72              s = 0;
73      }
74  }
```

At last, we compute the SVM kernel and stream the result to the `write` module.

A.4 Write Function/Module

```
1  void row_write(...)
2  {
3  write_loop:
4      for (int p = 0; p < products/16; p++) {
5         #pragma HLS LOOP_TRIPCOUNT min=prod/16 max=prod/16
6         #pragma HLS PIPELINE II=16
7          uint512 s;
8      bundle_loop:
9          for (int i = 0; i < 16; i++) {
10             float f = bundleStream.read();
11             uint32 rraw = *(uint32 *)(&f);
12             s((i + 1)* 32 - 1, i * 32) = rraw;
13         }
14         row[p] = s;
15     }
16 }
```

The `write` module is simple enough. It bundles the `float` results in groups of 16, in order to utilize the full 512-bit width of the bus for memory-kernel transfers.

Γλωσσάριο

Ελληνικός όρος	Αγγλικός όρος
Μηχανές Διανυσμάτων Υποστήριξης	Support Vector Machines
Μηχανική Μάθηση	Machine Learning
Σύνθεση Υψηλού Επιπέδου	High Level Synthesis
βήμα επανάληψης	iteration
γραμμική παλινδρόμηση	linear regression
δείκτης	index
διάνυσμα	vector
ελαχιστοποίηση	minimization
ετικέτα	label
κανονικοποίηση	regularization
κατηγοριοποίηση	classification
κλάση	class
μονοπύρηνο	single-core
νοημοσύνη	intelligence
πίνακας	matrix
παραγοντοποίηση	decomposition
πολυνηματική	multi-threaded
πολυπύρηνο	multi-core
πυρήνας	kernel
στόχος	target
συνάρτηση απόφασης	decision function
σύνολο δεδομένων	dataset
τετραγωνική	quadratic