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

Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Διπλωματικη Εργασια

FPGA-Acceleration of Machine Learning Algorithm, a case study using Gaussian Naive Bayes

ΓΕΩΡΓΙΟΣ Β. ΤΖΑΝΟΣ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων
Αθηνα, Μαρτιος 2019

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

Διπλωματικη Εργασια

FPGA-Acceleration of Machine Learning Algorithm, a case study using Gaussian Naive Bayes

Γεώργιος Β. Τζάνος

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28η Μαρτίου 2019.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Γεώργιος Γκούμας	Κιαμάλ Πεκσμετζή
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθηνα, Μαρτιος 2019

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

.....
Γεώργιος Β. Τζάνος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ζούμε σε μια εποχή που η μηχανική μάθηση και γενικότερα η τεχνητή νοημοσύνη αποτελεί αναπόσπαστο κομμάτι της προσπάθειάς μας να επιλύσουμε προβλήματα με ευφυείς προσεγγίσεις εκμεταλλευόμενοι την δυνατότητά μας να εκπαιδεύσουμε ένα σύστημα. Η μηχανική μάθηση μπορεί να οριστεί ως το φαινόμενο κατά το οποίο ένα σύστημα βελτιώνει την απόδοσή του, κατά την εκτέλεση μιας συγκεκριμένης εργασίας, χωρίς να υπάρχει ανάγκη να προγραμματιστεί εκ νέου. Μαζί όμως με την άνθιση της Μηχανικής Μάθησης και την εύρεση συνεχώς νέων και πιο αποδοτικών αλγορίθμων, η διαρκώς αυξανομένη ανάγκη για επεξεργασία όλο και μεγαλύτερου όγκου δεδομένων μας οδηγεί σιγά σιγά στα όρια των κλασικών λύσεων επεξεργασίας. Με τον διαφαινόμενο κορεσμό να πλησιάζει όλο και περισσότερο, μεγάλο κομμάτι της ερευνάς έχει στραφεί στην αναζήτηση εναλλακτικών λύσεων που εξαιτίας της διαφορετικής φύσης του προσφέρουν και διαφορετικά οφέλη. Μια από τις ποιοτικότερες εναλλακτικές έναντι ενός κλασικού επεξεργαστή, αποτελεί το FPGA, το οποίο σε συγκεκριμένες εργασίες, καταφέρνει τόσο καλύτερη χρονική επίδοσή λόγω της ικανότητάς του να πραγματοποιεί παράλληλους υπολογισμούς όσο και καλύτερης ενεργειακή επίδοση λόγω της χαμηλής ισχύος που καταναλώνει.

Στα πλαίσια αυτής της διπλωματικής δημιουργήσαμε έναν επιταχυντή υλικού για τον αλγοριθμό του Gaussian Naive Bayes με στόχο την καλύτερη δυνατή χρονική επίδοση. Προσπαθήσαμε μέσω αυτής της μελέτης να παρουσιάσουμε τα οφέλη της χρήσης ενός FPGA σε ενσωματωμένα συτήματα χαμηλής ενέργειας αλλά και να παρουσιάσουμε μια μεθοδολογία δημιουργίας ενός ετερογενούς κατανεμημένου συστήματος με χρήση πολλαπλών FPGA. Η διπλωματική αυτή αποτελεί παράλληλα κομμάτι μιας συνολικότερης προσπάθειας για την δημιουργία βιβλιοθηκών οι οποίες θα περιέχουν βασικές εργασίες ενός Data Center υλοποιημένες σε FPGA. Για τις παραπάνω ενέργειες εγίνε χρήση των πλακετών Zedboard και Pynq-Z1 και αξιολογήσαμε τα αποτελέσματα μας χρησιμοποιώντας τον αλγόριθμο του Gaussian Naive Bayes που αποτελεί μια ευρέως χρησιμοποιούμενη τεχνική Μηχανικής Μάθησης για στατιστική ταξινόμηση. Για τον προγραμματισμό των FPGA έγινε μελέτη και αξιοποίηση των δυνατοτήτων της Σύνθεσης Υψηλού Επιπέδου προκειμένου να υλοποιήσουμε έναν αποδοτικό επιταχυντή υλικού.

Λέξεις Κλειδιά

gaussian naive bayes, μηχανική μάθηση, pynq , zedboard, apache spark, σύνθεση υψηλού

επιπέδου, υπολογιστικό σύμπλεγμα, υπολογιστική επιτάχυνση

Abstract

We live at a time that machine learning is an integral part of our effort to solve problems with intelligent approaches by taking advantage of our ability to train a system. Machine Learning can be defined as the phenomenon where a system improves its performance, while it is executing a particular task, without the need for re-programming. But with the bloom of Machine Learning and the ever-increasing need, for more and more data processing, is slowly leading us, to the limits of classical processing solutions. Thats' why a big part of the research is trying to find alternatives solutions, that due to their different nature, offer and different benefits. One of the most qualitative alternative to a classic processor is FPGA which, in specific tasks, is capable of better performance in time due to its ability to perform parallel calculations and better performance in energy because of the low power it consumes.

In the context of this diploma, we implemented a Hardware accelerator on Gaussian Naive Bayes algorithm trying to optimize its time performance. We have attempted to present the benefits of using a FPGA in embedded low energy systems and to present a method for creating a heterogeneous distributed system using multiple FPGAs This diploma is also part of a larger effort to create libraries that will contain basic tasks of a Data Center implemented in FPGA. For the needs of our work we used the Zedboard and Pynq-Z1 boards and evaluated our results using the Gaussian Naive Bayes algorithm, a widely used Machine Learning technique for statistical classification. We have studied and exploited the capabilities of High-Level Composite in order to implement an efficient hardware accelerator for the programming of FPGA.

Keywords

gaussian naive bayes, mechanical learning, pynq, zedboard, apache spark, high level synthesis, computational cluster, computational acceleration

Στην Οικογένειά μου

Ευχαριστίες

Θα ήθελα καταρχάς να ευχαριστήσω τον καθηγητή κ. Δ.Σούντρη για την επίβλεψη αυτής της διπλωματικής εργασίας και για την εμπιστοσύνη που έδειξε σε μένα καθ΄ όλη την διάρκεια αυτής της διατριβής. Οι εκπαιδευτικές ευκαιρίες που μου προσέφερε ήταν καταλυτικές για μένα σε όλα τα επίπεδα. Επίσης ευχαριστώ ειλικρινά τον Μεταδιδακτορικό Ερευνητή Χριστόφορο Κάχρη για την καθοδήγησή του, τις συμβουλές του και την ενθάρρυνσή του σε όλα τα στάδια αυτής της προσπάθειας. Τέλος θα ήθελα να ευχαριστήσω πρώτα απ΄ όλα την οικογένεια μου για την αγάπη τους, την διαρκή τους υποστήριξη και την πίστη τους σε μένα, τους φίλους μου που ήταν πάντα δίπλα μου και την Έλενα για την αγάπη της και την υποστήριξή της.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Πηγαίος Κώδικας

Κεφάλαιο 1

Εισαγωγή

1.1 Αντικείμενο της διπλωματικής

Την τελευταία δεκαετία, παρατηρείται μια συνεχής αύξησης της χρήσης των συσκευών FPGA για όλο και περισσοτέρους και διαφορετικούς λόγους. Ενσωματώνουν στην ίδια πλακέτα τόσο ενσωματωμένο επεξεργαστή ARM όσο και ψηφίδα προγραμματιζόμενης λογικής. Το FPGA αντικαθιστά με τον καιρό τόσο τους επεξεργαστές γενικής χρήσης, όσο και τους ASICs. Μπορεί αρχικά να είχαν εμφανιστεί ως προϊόντα χαμηλής ενέργειας, σήμερα όμως χρησιμοποιούνται σε αρκετά πολύπλοκες συσκευές με αυξημένες απαιτήσεις σε επιδόσεις.

Τα ενσωματωμένα συστήματα είναι, εξειδικευμένα συστήματα υπολογιστών, τα οποία εκτελούν μια συγκεκριμένη λειτουργία. Αποτελούν ένα συνδυασμό υλικού και λογισμικού μέρους. Μάλιστα, είναι κοινή πρακτική για τα ενσωματωμένα συστήματα η τοποθέτηση πάνω στην ίδια επιφάνειας πυριτίου, επεξεργαστικών πυρήνων, μνημών, περιφερειακών, διασυνδέσεων I/O και μερικές φορές αναλογικών/ψηφιακών κυκλωμάτων, καταλήγοντας σε ολόκληρα συστήματα πάνω στο ίδιο υλικό πυριτίου (System-on-Chip, SoC). Συνήθως, τα ενσωματωμένα συστήματα είναι τμήμα ενός μεγαλύτερου συστήματος ή προϊόντος και τα συναντάμε παντού. Αν και οι περισσότεροι από εμάς γνωρίζουμε ότι καθημερινά κατασκευάζονται εκατομμύρια υπολογιστές σε όλον τον κόσμο, αυτό που ίσως δεν ξέρουμε είναι ότι κατασκευάζονται πολλά περισσότερα (δισεκατομμύρια) ενσωματωμένα συστήματα για ένα πλήθος διαφορετικών λειτουργιών. Συνεπώς η χρήση του FPGA με αποδοτικό τρόπο στα ενσωματωμένα συστήματα χαμηλής κατανάλωσης χρήζει μεγάλης σημασίας καθότι μπορούν να να βελτιώσουν την απόδοση ενός συστήματος δραματικά.

Στην σημερινή εποχή ο αριθμός των συνδεδεμένων συσκευών καθώς και ο όγκος των δεδομένων συνεχώς αυξάνονται, όπως αυξάνεται και ρυθμός με τον οποίο αυτά τα δεδομένα στέλνονται από τις συσκευές αυτές στα Κέντρα Δεδομένων. Τα FPGA λοιπόν, μπορούν να αποτελέσουν μια λύση σ΄ αυτό το φαινόμενο. Πιο συγκεκριμένα, είναι συσκευές πυριτίου οι οποίες μπορούν να επαναπρογραμματιστούν δυναμικά με τέτοιο τρόπο που να μπορεί να βελτιστοποιηθεί η χρήση τους σε εφαρμογές όπως η μηχανική μάθηση, η επεξεργασία εικόνας, η κωδικοποίηση, η συμπίεση και η ανάλυση δεδομένων. Αυτή η προσαρμοστικότητα των FPGA ανοίγει τον δρόμο για ταχύτερη επεξεργασία και καλύτερη ενεργειακή απόδοση ενώ

παράλληλα η χρήση τους μειώνει τις ανάγκες σε χώρο, παροχή ενέργειας και συστημάτων ψύξης στα Κέντρα Δεδομένων.

1.2 Δομή της διπλωματικής

Η διπλωματική αυτή οργανώνεται ως εξής:

Το **2ο κεφάλαιο** περιέχει μια περιγραφή του θεωρητικού υπόβαθρου που απαιτείται. Συγκεκριμένα υπάρχουν πληροφορίες για το Apache Spark, το Zynq-7000 All-Programmable SoC, το Pynq-Z1 και την Σύνθεση Υψηλού Επιπέδου που είναι απαραίτητες κατά την ανάγνωση της διπλωματικής εργασίας.

Το **3ο κεφάλαιο** κάνει μια περιγραφή του Gaussian Naive Bayes που αποτελεί τον αλγόριθμο μηχανικής μάθησης που εξετάσαμε ενώ παράλληλα κάνει και μια ανασκόπηση στο μαθηματικό υπόβαθρο του αλγορίθμου καθώς και πώς αυτός λειτουργεί αλγοριθμικά.

Το **4ο κεφάλαιο** παρουσιάζει εκτενώς και βήμα-βήμα την υλοποίησή του επιταχυντή υλικού που αναπτύξαμε για την πλακέτα Zedboard και τα αποτελέσματα του εγχειρήματος αυτού.

Το **5ο κεφάλαιο** αποτελεί μια εισαγωγή στην πλακέτα Pynq-Z1 και παρουσιάζει την πορεία που ακολουθήσαμε στην προσπάθειά μας να ενσωματώσουμε τον επιταχυντή υλικού που δημιουργήσαμε στο περιβάλλον της Python ώστε να μπορεί να εκτελεστεί στο PYNQ-Z1.

Το **6ο κεφάλαιο** παρουσιάζει την ενσωμάτωση του επιταχυντή με το Apache Spark, ύστερα από μερικές παραμετροποιησείς στην υλοποίηση του προηγούμενου κεφαλαίου.

Το **7ο κεφάλαιο** τέλος, συζητά σημαντικά συμπεράσματα που είναι πολύτιμα για τον αναγνώστη, καθώς και ιδέες για τη βελτίωση και επέκταση του υπάρχοντος έργου στο μέλλον.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

2.1 FPGA

2.1.1 Εισαγωγή

Τα αρχικά FPGA[1] σημαίνουν Field Programmable Gate Array. Είναι ημιαγωγικές συσκευές οι οποίες έχουν τη δυνατότητα να προγραμματίζονται και να επαναπρογραμματίζονται καθώς περιέχουν προγραμματιζόμενα λογικά μπλοκ και διασυνδεδεμένα κυκλώματα. Όταν μια πλακέτα περιέχει FPGA τότε ο κατασκευαστής έχει προκαθορίσει την εργασία που θα εκτελείται σε αυτό, αλλά μπορεί αργότερα να επαναπρογραμματιστεί από τον εκάστοτε σχεδιαστή, προκειμένου να εξυπηρετήσει τις δικές του ανάγκες. Ένα FPGA μπορεί να είναι τόσο απλό όσο και μια πύλη AND αλλά και τόσο πολύπλοκο όσο ένας πολυπύρηνος επεξεργαστής. Προκειμένου να προγραμματιστεί, χρησιμοποιείται Hardware Description Language όπως η Verilog και η VHDL μέσω των οποίων δημιουργείται ένα αρχείο bit, το οποίο κάνει το configuration της συσκευής. Σε κάθε προγραμματισμό της συσκευής όμως το configuration που γίνεται μέσω του bit αρχείου αποθηκεύεται στη SRAM, συνεπώς οποιαδήποτε αποσύνδεση της συσκευής από την πηγή τροφοδοσίας προκαλεί απώλεια της συγκεκριμένης ρύθμισης που έχει γίνει.

2.1.2 Αρχιτεκτονική

Η αρχιτεκτονική του FPGA τώρα, αποτελείται από τρεις βασικές μονάδες:

- **Τα προγραμματιζόμενα Λογικά Μπλοκ που εκτελούν λογικές συναρτήσεις.**

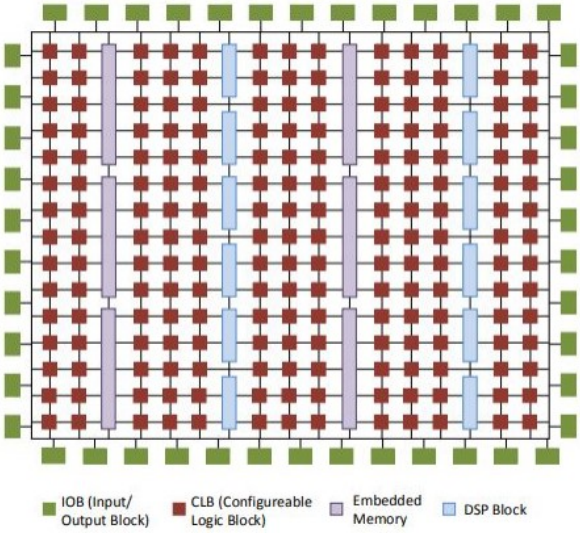
  Τα προγραμματιζόμενα Λογικά Μπλοκ, εκτελούν βασικούς υπολογισμούς και διαθέτουν αποθηκευτικά στοιχεία που χρησιμοποιούνται σε ψηφιακά συστήματα. Τα νεότερα FPGA διαθέτουν μια ετερογενή μίξη από διάφορα μπλοκ όπως μπλοκ μνήμης και πολυπλέκτες.

- **Οι προγραμματιζόμενες συνδέσεις μεταξύ των κυκλωμάτων που εκτελούν τις εργασίες.**

Η προγραμματιζόμενη Δρομολόγηση, εγκαθιστά την επικοινωνία μεταξύ του λογικού μπλοκ και του I/O μπλοκ προκειμένου να ολοκληρώσει τη σχεδίαση του χρήστη για το FPGA.

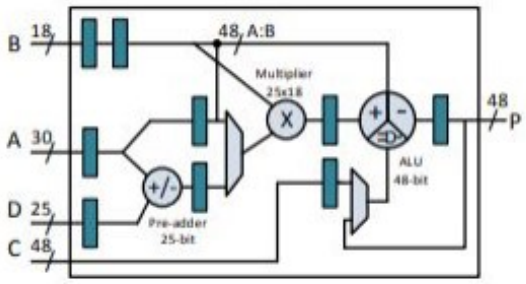
- **Και τα μπλοκ εισόδου, εξόδου (I/O Blocks) που βοηθούν στην επικοινωνία του chip με εξωτερικές συσκευές.**

  Τα προγραμματιζόμενα I/O, χρησιμοποιούνται για να επικοινωνήσουν τις λογικές μονάδες και τις συνδέσεις δρομολόγησης με εξωτερικά στοιχεία.



Σχήμα 2.1: Αρχιτεκτονική FPGA

Επίσης τα FPGA εξαιτίας της συνεχιζόμενης ανάγκης για εκτέλεση όλο και πιο πολύπλοκων εργασιών, διαθέτουν πλέον μονάδες όπως είναι οι ALUs, BlockRAMs, πολυπλέκτες, DSP-48 και μικροεπεξεργαστές



Σχήμα 2.2: Δομή Xilinx DSP48E1

Η Ροή που ακολουθεί ο προγραμματισμός ενός FPGA αποτελείται από τα παρακάτω βήματα:

- **Design Entry (Είσοδος Σχεδίασης):** Κατά την είσοδο της σχεδίασης (Design Entry) ο χρήστης μπορεί να καθορίσει τον τρόπο με τον οποίο θα προγραμματίσει το

FPGA. Αυτό μπορεί να συμβεί μέσω σχηματικής σχεδίασης το οποίο είναι για περιπτώσεις που ο χρήστης ασχολείται καθαρά με το υλικό (hardware), ενώ αν ο χρήστης σκέφτεται περισσότερο αλγοριθμικά επιλέγει να γράψει σε HDL.

- **Design Synthesis (Σύνθεση Σχεδίασης):** Κατά το κομμάτι της σχεδίασης της σύνθεσης , μεταφράζεται η HDL σε μορφή κατανοητή για τη συσκευή. Η διαδικασία της σύνθεσης ελέγχει για τυχόν λάθη στον κώδικα της HDL και κάνει ανάλυση της ιεραρχίας της σχεδίασης (Αυτό εξασφαλίζει ότι η σχεδίαση ακολουθεί έναν αποδοτικό τρόπο χρήσης των διαθέσιμων πόρων).

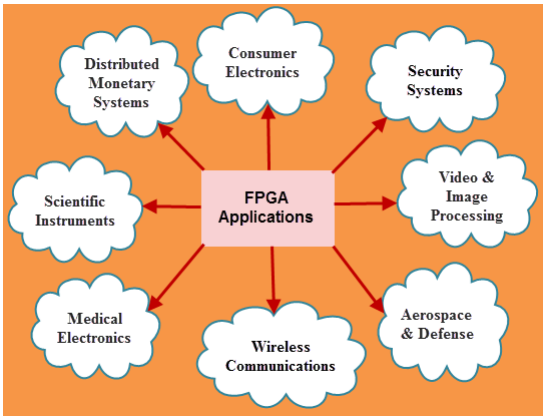
- **Design Implementation (Υλοποίηση Σχεδίασης)**: Κατά την υλοποίηση της σχεδίασης λαμβάνουν χώρα κατά σειρά τρείς διεργασίες:

  - **Translate**: αυτή η διαδικασία συγχωνεύει όλες τις input netlist στο αρχείο του logic design και οι θύρες αντιστοιχίζονται σε φυσικά στοιχεία όπως pins, switches κτλ.
  - **Map**: αυτή η διαδικασία ταιριάζει τη σχεδίαση στους διαθέσιμους πόρους της συσκευής.
  - **Place and Route**: αυτή η διαδικασία τοποθετεί και δρομολογεί τη σχεδίαση έτσι ώστε να ανταποκρίνεται στους χρονικούς περιορισμούς. Και τέλος γίνεται η δημιουργία του bitstream που σετάρει την συσκευή (Generate Programming File).

2.1.3 Χρησεις του FPGA

- Χρησιμοποιείται σαν κοντρόλερ συσκευών, για κωδικοποίηση επικοινωνιών και σαν φίλτρο.

- Χρησιμοποιούνται στον έλεγχο μηχανών, στα δίκτυα, και στα συστήματα καμερών παρακολούθησης και όλα αυτά γιατί συνδυάζουν την πολύπλοκη παράλληλη επεξεργασία με το χαμηλό κόστος και τη χαμηλή κατανάλωση ενέργειας.



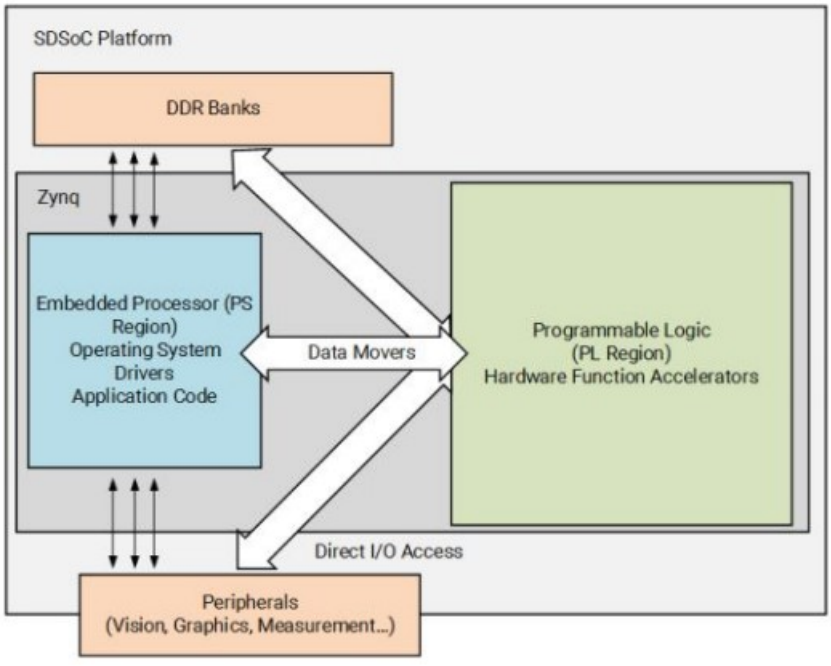
Σχήμα 2.3: Εφαρμογές FPGA

- Χρησιμοποιούνται στη δημιουργία μεγάλων συστημάτων υλικού συνδυάζοντας πολλά FPGA μαζί.

2.2 SDSoC & High Level Synthesis

2.2.1 Περιβάλλον SDSoC

Το περιβάλλον του SDSoC[2] παρέχει ένα framework για τον προγραμματισμό εφαρμογών υλοποιημένων στο hardware χρησιμοποιώντας γλώσσες προγραμματισμού υψηλού επιπέδου όπως η C και η C++. Έτσι ο χρήστης δε χρειάζεται πλέον να προγραμματίζει σε HDL αλλά γράφοντας σε C/C++ μπορεί να υλοποιήσει, όποια εφαρμογή επιθυμεί να επιταχύνει, χρησιμοποιώντς FPGA. Το IDE που χρησιμοποιείται στο SDSoC είναι ένα Eclipse-based environment που διαθέτει compilers ειδικούς για την υλοποίηση των hardware functions στις πλακέτες της Xilinx. Οι compilers του συστήματος αναλύουν το πρόγραμμα και καθορίζουν τη ροή των δεδομένων ανάμεσα στο software και τις hardware functions, παράγουν τα IP (Layout designs of integrated circuits) του υλικού και ρυθμίζουν την επικοινωνία μεταξύ συστήματος επεξεργασίας και υλικού. Επίσης παρέχεται η δυνατότητα εγκατάστασης λειτουργικού συστήματος Linux στη συσκευή. Τα παραπάνω καθιστούν το FPGA μια αυτόνομη οντότητα.



Σχήμα 2.4: Πλατφόρμα SDSoC

Χρησιμοποιώντας system-on-a-chip συσκευές της Xilinx όπως το Zynq-7000 είναι δυνατόν να υλοποιηθούν εφαρμογές στους επιταχυντές υλικού, εκτελεσμένες αρκετές φορές πιο γρήγορα σε σχέση με μια CPU/GPU έχοντας πλεονέκτημα και σε ό,τι αφορά την κατανάλωση ενέργειας.

Οι δυνατότητες που έχουν οι συσκευές FPGA γίνονται ευδιάκριτες σε περιπτώσεις υλοποίησης μιας αρκετά απαιτητικής εφαρμογής με ανεξάρτητους υπολογισμούς που μπορούν να πραγματοποιηθούν παράλληλα. Τότε θα παρατηρήσουμε πως μπορούμε να πετύχουμε την ιδανική ισορροπία ανάμεσα σε απόδοση και ενέργεια. Έτσι λοιπόν μέσω του SDSoC ακόμα και ένας προγραμματιστής που δε γνωρίζει από σχεδίαση υλικού, μπορεί να γράψει κώδικα σε C/C++ αφήνοντας αυτό το κομμάτι να υλοποιηθεί αυτόματα από τον compiler του συστήματος, ενώ ο ίδιος προγραμματίζει όπως θα έκανε για μια software εφαρμογή. Υπάρχουν φυσικά οδηγίες που αποτελούν κατευθυντήριες οδηγίες για τον compiler και τον τρόπο με τον οποίο θα υλοποιήσει την hardware function όσο και την επικοινωνία της με το software

Το SDSoC δίνει τη δυνατότητα τόσο για software εφαρμογές μέσω του ARM όσο και hardware μέσω του FPGA, επιτρέποντας παράλληλα την επικοινωνία μεταξύ των δύο. Για να γίνει λοιπόν, όσο το δυνατόν πιο αποδοτικά, η υλοποίηση της εφαρμογής μας, ο σχεδιαστής μπορεί αρχικά να γράψει όλη την εφαρμογή για εκτέλεση από τον ARM και κάνοντας profiling να εντοπίσει τις πιο απαιτητικές εργασίες και να τις μεταφέρει στο FPGA.

Κατά την υλοποίηση μιας hardware συνάρτησης είναι δυνατόν με συγκεκριμένες ντιρεκτίβες (HLS/SDS pragmas) στον compiler να προσαρμοστούν οι επιταχυντές καθώς και ο τρόπος μεταφοράς των δεδομένων ανάμεσα σε software και hardware ανάλογα με τις ανάγκες της εφαρμογής, ώστε να επιτευχθεί μέγιστη δυνατή απόδοση.

2.2.2 High Level Synthesis

Η Υψηλού Επιπέδου Σύνθεση - HLS[3], αποτελεί ένα εργαλείο το οποίο επιτρέπει στους σχεδιαστές υλικού να δημιουργήσουν και να ελέγξουν το υλικό αποτελεσματικά δίνοντας ταυτόχρονα υψηλότερο επίπεδο αφαίρεσης. Επίσης, δίνει τη δυνατότητα επαλήθευσης της σχεδίασης σε επίπεδο C, κάτι το οποίο επιτρέπει στο σχεδιαστή να επικυρώσει την ορθή λειτουργία του αλγορίθμου πολύ πιο γρήγορα και πολύ πιο εύκολα, συγκριτικά με μια γλώσσα περιγραφής υλικού HDL.

Κατά την ανάπτυξη λοιπόν εφαρμογών στο hardware και της βελτιστοποίησής τους, μπορεί πλέον να χρησιμοποιηθεί η Σχεδίαση Υψηλού Επιπέδου (High Level Synthesis). Ο προγραμματισμός των FPGA μέσω της HLS είναι μια σημαντική καινοτομία για πολλούς λόγους, κάποιοι από τους οποίους είναι:

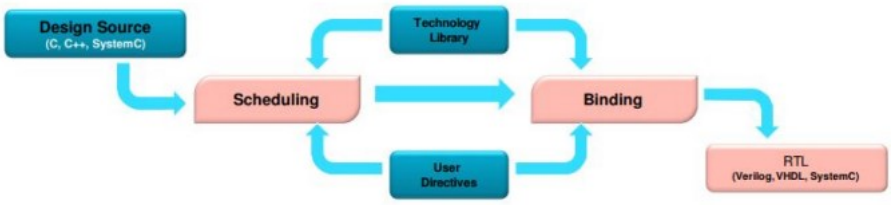
- Μεγάλα project είναι δύσκολο να γραφτούν σε γλώσσα περιγραφής υλικού HDL
- Πλέον απαιτητικές εργασίες που θα εκτελούνταν από την CPU μπορούν να μεταφερθούν στους επιταχυντές υλικού στο FPGA ρυθμίζοντας την επικοινωνία με τη CPU πολύ πιο εύκολα.
- Η HLS κάνει τον προγραμματισμό του υλικού πολύ πιο απλό και εύκολο. Ο προγραμματιστής μπορεί πλέον να περάσει απ' ευθείας στο αλγοριθμικό κομμάτι της υλοποίησής του αφού η HLS αναλαμβάνει τον RTL σχεδιασμό.
- Η συντήρηση μεγάλων project είναι πιο εύκολη με αυτόν τον τρόπο.

Προκειμένου να ολοκληρωθεί η λογική σύνθεση του σχεδιασμού απαιτείται RTL περιγραφή, κάτι το οποίο αναλαμβάνει αυτόματα η σύνθεση υψηλού επιπέδου. Αυτή δίνει τη δυνατότητα στο σχεδιαστή να κάνει αλγοριθμική περιγραφή του σχεδιασμού σε γλώσσα υψηλού επιπέδου (SystemC, C/C++). Ο σχεδιαστής αναπτύσσει τη λειτουργικότητα της μονάδας και το πρωτόκολλο διασύνδεσης και στη συνέχεια η HLS αναλαμβάνει το μετασχηματισμό του κώδικα σε πλήρως χρονομετρημένες υλοποιήσεις RTL, καθορίζοντας όλες τις απαραίτητες πληροφορίες για τον κύκλο εκτέλεσης του υλικού.

Ο βασικός κορμός λειτουργίας της HLS απαρτίζεται από τα παρακάτω βήματα:

- Δημιουργεί την RTL υλοποίηση από κώδικα γραμμένο σε C/C++.
- Εξάγει τον τρόπο εκτέλεσης και τη ροή δεδομένων από τον πηγαίο κώδικα.
- Υλοποιεί τη σχεδίαση, βασισμένη τόσο σε προεπιλεγμένες οδηγίες όσο και σε οδηγίες που έχει δώσει ο χρήστης για την υλοποίηση στο hardware.

Η HLS αποτελείται από δύο βασικές διεργασίες, το scheduling και το binding.



Σχήμα 2.5: Scheduling & Binding

Scheduling

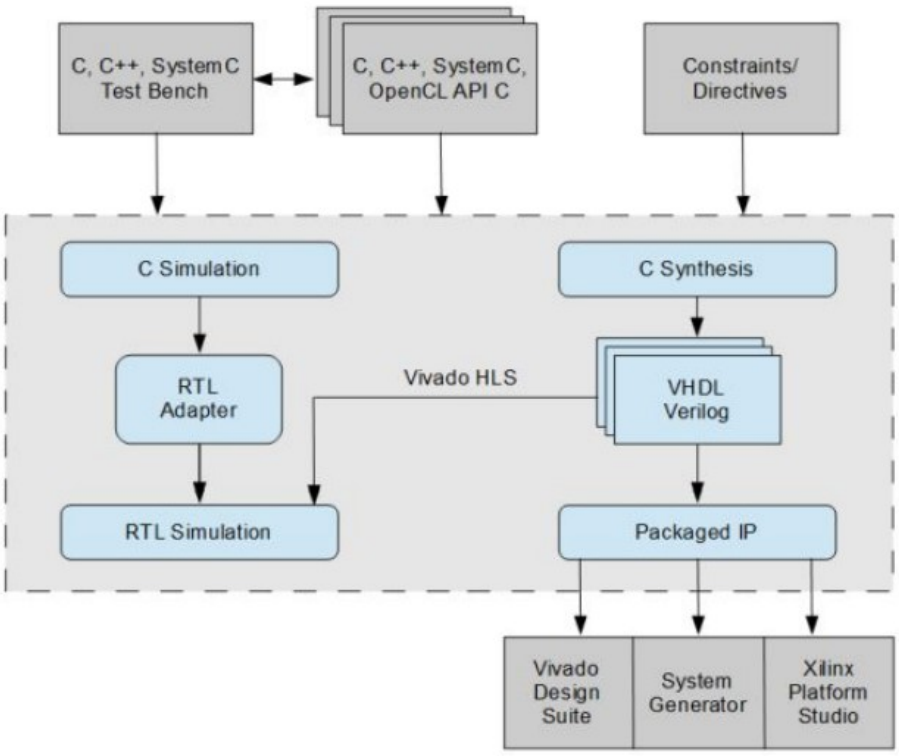
Κατά το scheduling γίνεται η μετάφραση του κώδικα C σε RTL και καθορίζεται ποιες διεργασίες θα εκτελεστούν σε κάθε κύκλο ρολογιού. Το scheduling έχει άμεση σχέση με τη συχνότητα του ρολογιού, την αβεβαιότητά του, καθώς και με τις οδηγίες που θα έχει εφαρμόσει ο χρήστης για την υλοποίηση του σχεδιασμού.

Binding

Είναι η διαδικασία του συσχετισμού του προγραμματισμένου λειτουργικού με τους φυσικούς πόρους της συσκευής. Η χρήση των φυσικών πόρων της συσκευής μπορεί να επηρεάσει τον προγραμματισμό των εργασιών, συνεπώς το binding ανατροφοδοτεί συνεχώς το scheduling μέχρι να ολοκληρωθεί η ταυτόχρονη ορθή χρήση των πόρων αλλά και η ορθή λειτουργία του αλγορίθμου.

Design Flow

Η ροή σχεδιασμού (Design Flow) φαίνεται στο παρακάτω σχήμα:



Σχήμα 2.6: Design Flow

Η ροή περιλαμβάνει τα εξής στάδια:

Δεδομένα εισόδου στο HLS:

Το αρχείο αυτό περιέχει τα data εισόδου προκειμένου στο τέλος της εκτέλεσης να συγκριθούν με τα δεδομένα εξόδου και να γίνει επαλήθευση της λειτουργίας της συνάρτησης.

Λειτουργία επαλήθευσης:

Γίνεται επαλήθευση της λειτουργίας του κώδικα πριν παραχθεί ο κώδικας RTL μέσω του testbench.

Σύνθεση:

Γίνεται ανάλυση και επεξεργασία του κώδικα,των οδηγιών του χρήστη, καθώς και των περιορισμών προκειμένου να δημιουργηθεί μια περιγραφή RTL του κώδικα.

Προσομοίωση και Αξιολόγηση:

Μετά τη δημιουργία του μοντέλου του RTL δοκιμάζεται η ορθή λειτουργία σε RTL επίπεδο για να προσδιοριστεί κατά πόσο έχουν τηρηθεί οι προδιαγραφές που έχουν αρχικά οριστεί.

Εξαγωγή RTL:

Τελευταίο βήμα είναι η εξαγωγή του υλικού για τη χρήση του σε ένα πραγματικό σύστημα. Η εξαγωγή γίνεται συνήθως σε μορφή IP.

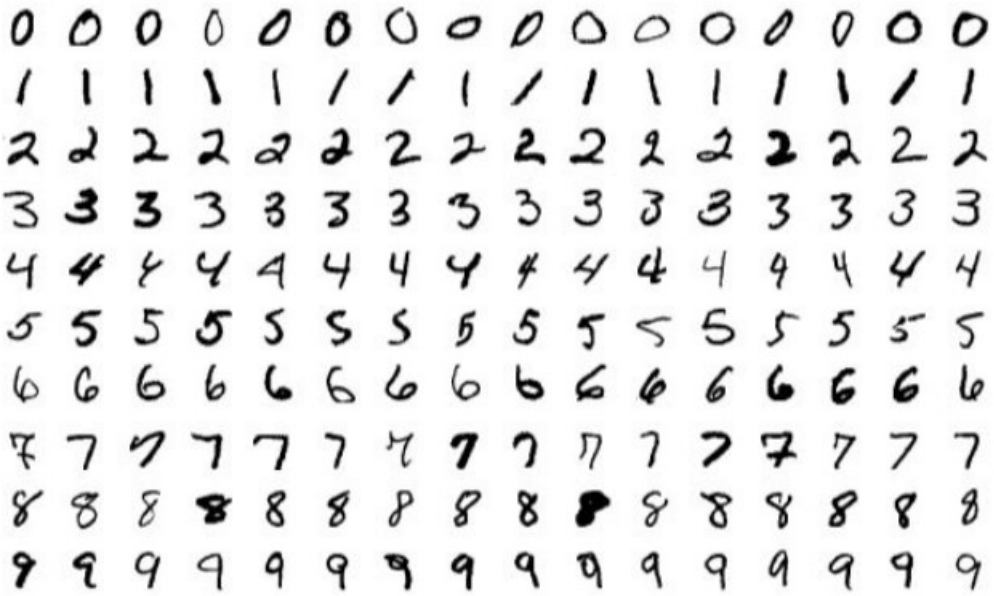
Στις πλακέτες της Xilinx, το Vivado HLS κατά την παραπάνω διαδικασία αποφασίζει με βάση προεπιλεγμένες ρυθμίσεις για τον τρόπο με τον οποίο θα κάνει χρήση του υλικού και των διαθέσιμων πόρων. Αυτό έχει ως συνέπεια πολλές φορές να δίνεται μεγαλύτερη έμφαση στην εξοικονόμηση πόρων σε σχέση με την απόδοση. Τότε ο σχεδιαστής, με συγκεκριμένες οδηγίες, που έχει διαθέσιμες η HLS, μπορεί να επέμβει άμεσα στον τρόπο αξιοποίησης του υλικού και να επιτύχει την ισορροπία που επιθυμεί ανάμεσα σε εξοικονόμηση πόρων και απόδοση. Αυτό επιτυγχάνεται μέσω των pragmas.

Πίνακας 2.1: Πίνακας HLS Pragmas

Ντιρεκτίβα	**Περιγραφή**
PIPELINE	Μειώνει το II για μια συνάρτηση ή ένα βρόχο επιτρέποντας την ταυτόχρονη εκτέλεση εργασιών.
UNROLL	Μετασχηματίζει τους βρόχους δημιουργώντας πολλαπλά αντίγραφα του σώματος του βρόχου σε επίπεδο RTL, το οποίο επιτρέπει την επικάλυψη μέρους ή όλων των επαναλήψεων του βρόχου.
ARRAY_PARTITION	Διαχωρίζει έναν πίνακα σε μικρότερους πίνακες ή μεμονωμένα στοιχεία. Αυτό αυξάνει αποτελεσματικά το ποσό των θυρών ανάγνωσης και εγγραφής για την αποθήκευση, βελτιώνοντας την απόδοση του σχεδίου.

2.3 MNIST

Το MNIST[4], είναι μια μεγάλη βάση δεδομένων που αποτελείται από χειρόγραφα ψηφία που χρησιμοποιείται συχνά για την εκπαίδευση διαφόρων συστημάτων επεξεργασίας εικόνας. Επίσης είναι ευρέως διαδεδομένη ως ένας τρόπος για εκπαίδευση και πρόβλεψη στο πεδίο της μηχανικής μάθησης. Το μοντέλο ταξινόμησης αυτό αποτελείται από 784 χαρακτηριστικά και 10 ετικέτες χρησιμοποιώντας μέχρι 40 χιλιάδες διαθέσιμα δείγματα εκπαίδευσης, για ένα πρόβλημα αναγνώρισης χειρόγραφων ψηφίων. Το αρχικό αρχείο δεδομένων περιέχει εικόνες σε γκρι κλίμακα από ψηφία που έχουν σχεδιαστεί με το χέρι, από το μηδέν έως το εννέα. Κάθε εικόνα έχει 28 pixel ύψος και 28 pixel πλάτος, δηλαδή συνολικά 784 pixels. Κάθε pixel έχει μια μοναδική τιμή συνδεδεμένη με αυτό, υποδεικνύοντας την φωτεινότητα αυτού του pixel, με υψηλότερους αριθμούς να σημαίνουν πιο σκούρο. Αυτή η τιμή είναι ένας ακέραιος μεταξύ του 0 και του 255, συμπεριλαμβανομένων των ακραίων τιμών.



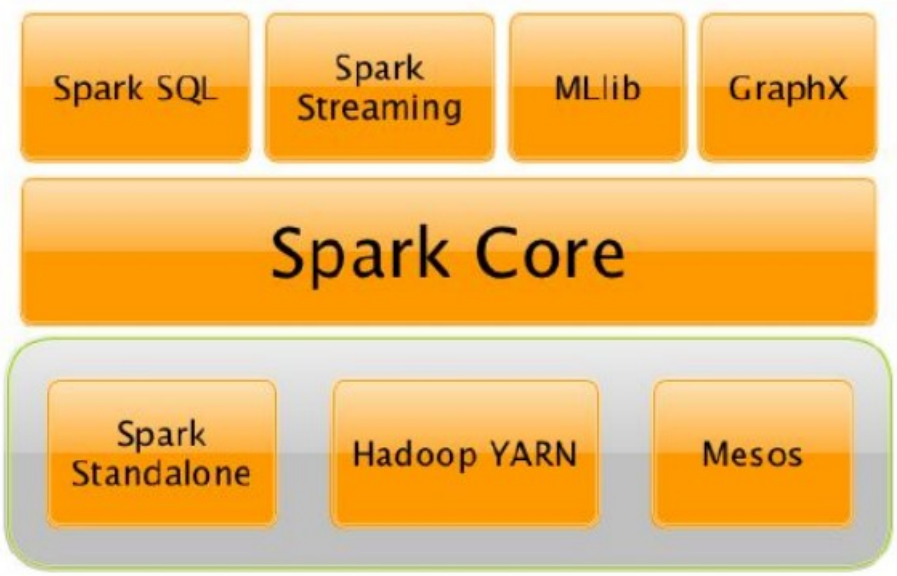
Σχήμα 2.7: MNIST

Η διαδικασία ταξινόμησης απαιτεί τα δεδομένα να διαχωρίζονται σε σύνολο εκπαίδευσης και δοκιμών. Και τα δύο σύνολα δεδομένων (train set, test set) έχουν 785 στήλες. Η πρώτη στήλη, που ονομάζεται ετικέτα, είναι το ψηφίο που σχεδιάστηκε από το χρήστη. Οι υπόλοιπες στήλες περιέχουν τις (επεξεργασμένες) τιμές των pixel (χαρακτηριστικά) της σχετικής εικόνας. Ο στόχος του Gaussian Naive Bayes είναι να εκπαιδεύσει ένα μοντέλο που να μπορεί να προβλέπει την ετικέτα (κλάση) ενός δείγματος του συνόλου δοκιμών μόνο μέσα από τα χαρακτηριστικά του συγκεκριμένου δείγματος.

2.4 Apache Spark

2.4.1 Εισαγωγή

Η πλατφόρμα Apache Spark[5], είναι ένα γρήγορο και γενικού σκοπού σύστημα υπολογισμών σε συστάδα υπολογιστών (cluster). Παρέχει υψηλού επιπέδου APIs στις γλώσσες προγραμματισμού Java, Scala, Python και R και μια βελτιστοποιημένη μηχανή η οποία υποστηρίζει γράφους γενικής εκτέλεσης. Επίσης παρέχει ένα σύνολο εργαλείων υψηλότερου επιπέδου όπως το Spark SQL, για επεξεργασία σε SQL δομημένων δεδομένων, MLib για μηχανική μάθηση, GraphX για επεξεργασία γράφων και Spark Streaming για επεξεργασία ροών (stream) σε πραγματικό χρόνο (live data streams)[6].



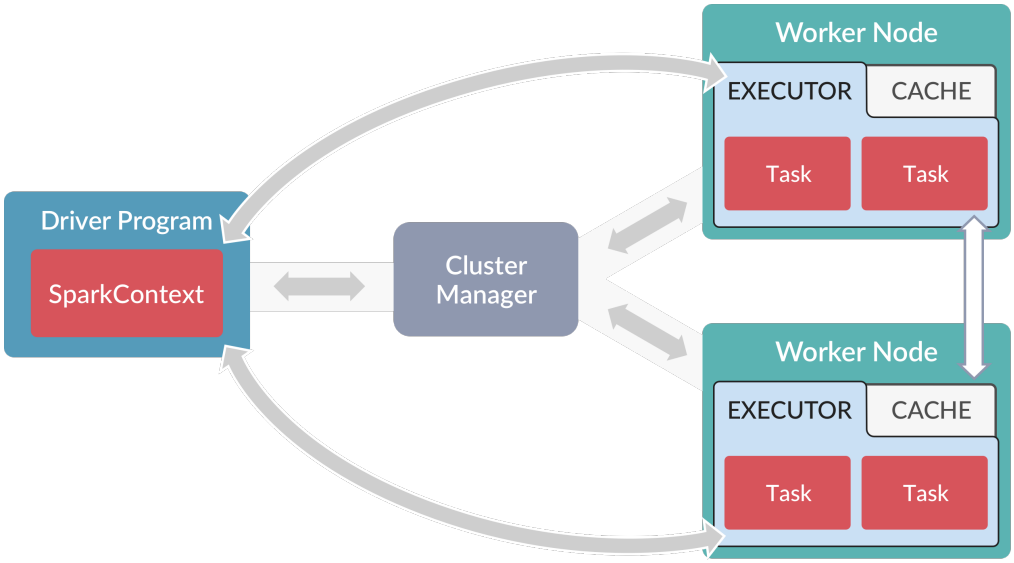
Σχήμα 2.8: Spark

Το Spark δημιουργήθηκε για να καλύψει περιπτώσεις που η χρήση του **MapReduce** στην πλατφόρμα Hadoop δεν ήταν τόσο αποδοτική. Μια τέτοια περίπτωση είναι οι επαναληπτικές εργασίες (iterating jobs) που συναντώνται σε αλγορίθμους μηχανικής μάθησης όπου η ίδια συνάρτηση εφαρμόζεται συνεχώς στο ίδιο σύνολο δεδομένων με σκοπό την βελτίωση μιας παραμέτρου.

2.4.2 Αρχιτεκτονική

Το Spark χρησιμοποιεί master/worker αρχιτεκτονική. Υπάρχει ένας οδηγός (driver) που επικοινωνεί με έναν συντονιστή που ονομάζεται master και διαχειρίζεται τους workers, στους οποίους τρέχουν οι executors. Ο master είναι ένα Spark instance που συνδέεται με έναν cluster manager, ο οποίος δεσμεύει cluster κόμβους για να χρησιμοποιήσει τους executors που βρίσκονται στους cluster κόμβους. Οι workers αποτελούν επίσης Spark instances μέσα στα οποία βρίσκονται οι executors που εκτελούν τις εργασίες. Η επικοινωνία μεταξύ του master και των workers γίνεται μέσω του Block Manager instances. Ο driver και οι executors τρέχουν τις δίκες τους Java διεργασίες. Μπορούν να τρέξουν όλες στην ίδια μηχανή (οριζόντια συστάδα) ή σε ξεχωριστές μηχανές (κατακόρυφη συστάδα) ή σε μικτή διαμόρφωση μηχανής. Μια Spark εφαρμογή τρέχει σαν ανεξάρτητο σετ από διεργασίες σε ένα cluster οι οποίες ρυθμίζονται από ένα SparkContext αντικείμενο, στο κυρίως πρόγραμμα, τον driver. Συγκεκριμένα για να τρέξει σε ένα cluster, το SparkContext μπορεί να συνδεθεί σε διαφόρους

τύπους cluster managers (Standalone , Yarn Mesos) οι οποίοι δεσμεύουν τους απαραίτητους πόρους. Μόλις συνδεθεί το Spark χρειάζεται τους executors στους κόμβους του cluster. Οι executors, είναι διεργασίες που εκτελούν υπολογισμούς και αποθηκεύουν δεδομένα για την εφαρμογή που θέλει να εκτελέσει ο χρήστης, Στην συνεχεία στέλνει τον κωδικά της εφαρμογής στους executors και τέλος το SparkContext μοιράζει τις εργασίες σε αυτούς προκειμένου να τις εκτελέσουν.



Σχήμα 2.9: Spark Application a.k.a Driver

Μια εφαρμογή στο Spark εκτελείται σε τρία στάδια:

- Δημιουργείται γράφος RDD δηλαδή ένας DAG (Directed Acyclic Graph) των RDD , ο οποίος αντιπροσωπεύει ολόκληρο τον υπολογισμό.
- Δημιουργείται ένας γράφος (DAG) σταδίων, ο οποίος είναι ένα λογικό στάδιο εκτέλεσης που βασίζεται στο γράφο RDD. Τα στάδια δημιουργούνται με το σπάσιμο του γράφου RDD σε τυχαία όρια.
- Με βάση το σχέδιο, χρονοδρομολογούνται και εκτελούνται καθήκοντα στους workers.

Resilient Distributed Datasets

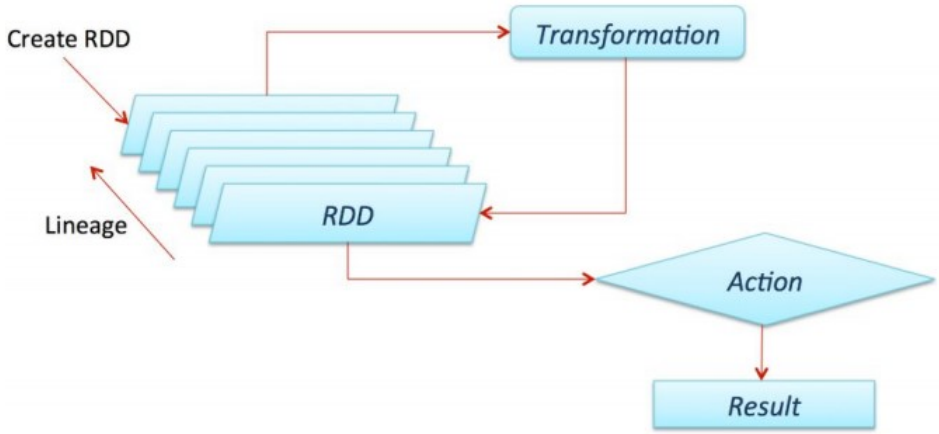
Το RDD[7] είναι μια συλλογή αντικείμενων με επιτρεπτό μόνο το διάβασμα (read-only collection) χωρισμένη στο σύνολο των workers με την δυνατότητα να ξαναχτιστεί, αν ένα από τα χωριζόμενα μέρη χαθεί, εξ ου και το Resilient στην ονομασία του. Τα στοιχειά RDD δεν βρίσκονται σε φυσική μνήμη όμως διατηρείται η πληροφορία υπολογισμού τους, χρησιμοποιώντας αποθηκευμένα δεδομένα. Έτσι είναι εγγυημένη η αξιοπιστία τους. Συνεπώς τα RDD μπορούν να ανακατασκευαστούν αν αποτύχουν οι κόμβοι.

Το RDD έχει δυο τύπους λειτουργίας:

- Transformations (Μετασχηματισμοί): Είναι διεργασίες όπως η map, filter, flatMap, mapPartition οι οποίες εφαρμόζονται πάνω στα RDD που έχουν διαμοιραστεί στους workers χωρίς όμως να αλλοιώσουν τα υπάρχοντα RDD άλλα δημιουργώντας καινούργια με βάση τα αρχικά RDD. Η δημιουργία των νέων RDD όμως δεν γίνεται σε πρώτο χρόνο μετά την εκτέλεση της

διεργασίας άλλα αντίθετα το Spark περιμένει μέχρι να του δοθεί σαν εντολή κάποια ενεργεία (Action) πάνω στο νέο RDD. Τότε μόνο δημιουργείται το RDD γι΄ αυτό και οι μετασχηματισμοί στο Spark αναφέρονται ως οκνηροί.

- Actions (Ενέργειες): Είναι λειτουργίες όπως η count, first, reduce οι οποίες επιστρέφουν τιμές μετά από υπολογισμούς πάνω σε κάποιο RDD.



Σχήμα 2.10: RDD

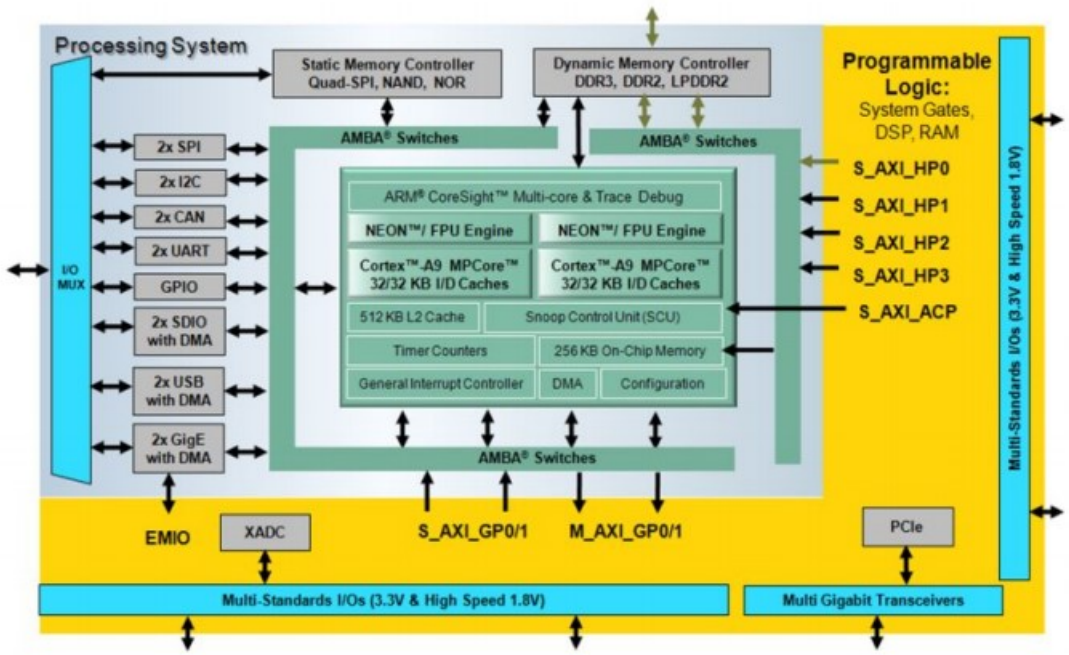
MLib(Machine Learning)

Το MLlib (Machine Learning)[8], το οποίο μας αφορά και στα πλαίσια αυτής της διπλωματικής εργασίας, αποτελεί μια κλιμακούμενη βιβλιοθήκη μηχανικής μάθησης η οποία παρέχει μια σειρά βελτιστοποιημένων αλγορίθμων για ομαδοποίηση, ταξινόμηση και παλινδρόμηση. Η βιβλιοθήκη μπορεί να εφαρμοστεί σε εφαρμογές του Spark υλοποιημένες σε Java, Scala και Python κάτι που την καθιστά εύχρηστη στην ενσωμάτωση της σε ποικίλες εφαρμογές.

2.5 Zynq-7000 All Programmable SoC

2.5.1 Επισκόπηση

Το Zynq-7000 All Programmable SoC (AP SoC)[9] αποτελεί μια συσκευή η οποία συνδυάζει τον προγραμματισμό ενός ARM επεξεργαστή (Processing System) με την ύπαρξη ψηφίδας προγραμματιζόμενης λογικής FPGA (Programmable Logic) διατηρώντας ταυτόχρονα υψηλή απόδοση και χαμηλή κατανάλωση ενέργειας. Ο ARM Cortex A9 είναι η καρδιά του PS (Processing System), το οποίο διαθέτει on-chip μνήμη, εξωτερική μνήμη και μια πληθώρα I/O περιφερειακών. Επίσης το γεγονός ότι διαθέτει λειτουργικότητες CPU, ASSP, DSD, ASIC του δίνει τη δυνατότητα να είναι ιδιαιτέρως αποτελεσματικό σε εφαρμογές που έχουν όχι μόνο υψηλές απαιτήσεις απόδοσης αλλά και απαιτήσεις χαμηλής κατανάλωσης. Ο επεξεργαστής είναι το πρώτο στοιχείο που τίθεται σε λειτουργία δίνοντας έτσι τη δυνατότητα για μια λογισμικο-κεντρική προσέγγιση της ρύθμισης της προγραμματιζόμενης λογικής. Το αρχείο ρύθμισης της προγραμματιζόμενης λογικής (PL) αναφέρεται ως bitstream.



Σχήμα 2.11: Αρχιτεκτονική Zynq

Ένα Zynq αποτελείται κυρίως από την προγραμματιζόμενη λογική και το σύστημα επεξεργασίας[10],

Programmable Logic

- CLB
- BRAM
- DSP slices
- DSD48E1
- XADC
- CMT

Proccesing System

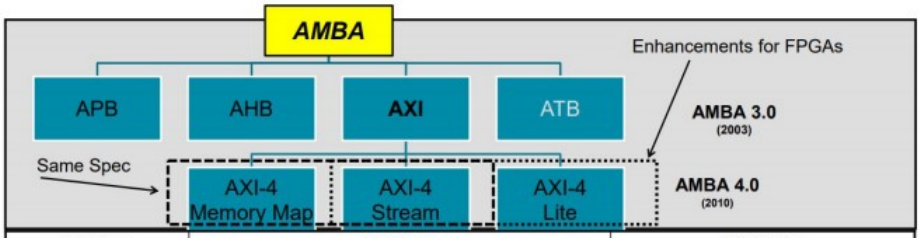
- APU
  - CPU
- Διασυνδέσεις Μνήμης
  - DDR Controller
  - Quad SPI Controller
  - SMC Static Memory Cotroller
- I/O Peripherals
  - GPIO
  - Gigabit Ethernet
  - USB Controller
  - SD/SDIO Controller
  - SPI Controller
  - CAN Controller
  - UART Controller
  - I2C Controller
  - DS MTO I/Os buffers
- Interconnect
  - Ocm Interconnect
  - Central Interconnect

2.5.2 Επικοινωνία μεταξύ PL και PS

Η μεγάλη δύναμη των Zynq είναι το σύστημα επικοινωνίας που υπάρχει ανάμεσα στο PS και στο PL, το οποίο δίνει τη δυνατότητα για εναλλαγή μεταξύ των δύο επιτρέποντας στον σχεδιαστή να εκμεταλλευτεί τον συνδυασμό τους. Ο μηχανισμός προκειμένου να συμβεί αυτό είναι ένα σύνολο προσδιορισμένων διασυνδέσεων και διεπαφών AXI. Το AXI[11] είναι κομμάτι του ARM AMBA μια οικογένεια micro-controller διαύλων. Όντας αυτή τη στιγμή στη δεύτερη έκδοση έχουμε το AXI4. Συγκεκριμένα το AXI4 χωρίζεται στους παρακάτω τύπους:

- **AXI4**: Για υψηλής απόδοσης απαιτήσεις που έχουν να κάνουν με το memory-mapping.
- **AXI4-Lite**: Για απλές, χαμηλής ταχύτητας διεκπεραιώσεις memory-mapped επικοινωνίας.
- **AXI4-Stream**: Για άμεση ροή δεδομένων χωρίς μεταφορά διευθύνσεων.

Κάθε διεπαφή αποτελείται από πολλαπλά AXI κανάλια. Έχουμε λοιπόν τους εξής τύπους διεκπεραίωσης:



Interface	Features	Similar to
Memory Map / Full (AXI4)	Traditional Address/Data Burst (single address, multiple data)	PLBv46, PCI
Streaming (AXI4-Stream)	Data-Only, Burst	Local Link / DSP Interfaces / FIFO / FSL
Lite (AXI4-Lite)	Traditional Address/Data—No Burst (single address, single data)	PLBv46-single OPB

Σχήμα 2.12: Δομή AXI

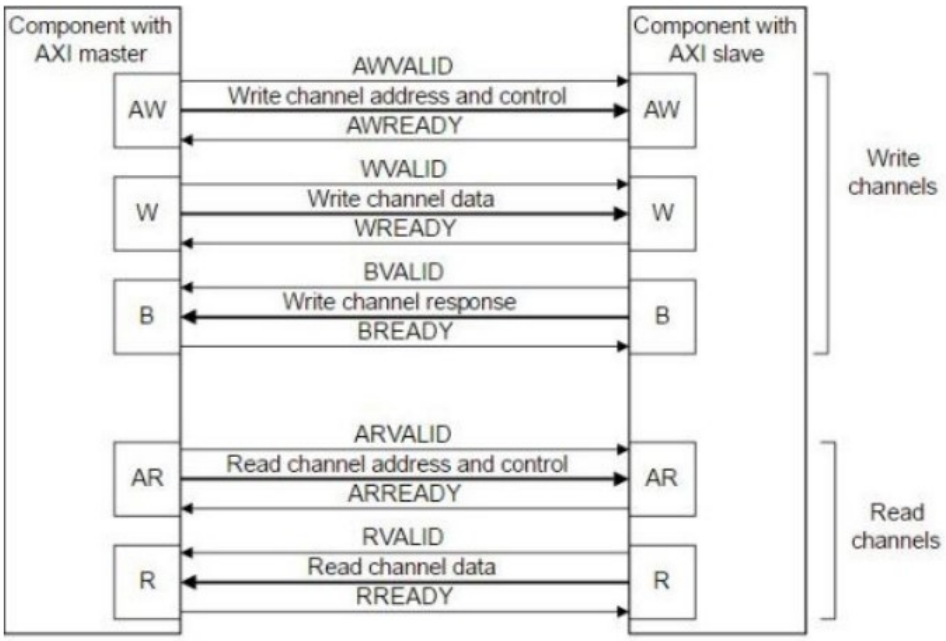
- General Purpose Ports: Ένας δίαυλος δεδομένων 32 bit, ο οποίος είναι κατάλληλος για επικοινωνίες χαμηλής και μέσης ταχύτητας μεταξύ του Συστήματος Επεξεργασίας και της Προγραμματιζόμενης Λογικής. Η διεπαφή είναι άμεση και δεν περιλαμβάνει προσωρινή αποθήκευση. Υπάρχουν τέσσερις γενικές διεπαφές γενικού σκοπού: το Σύστημα Επεξεργασίας είναι ο master των δύο, και η Προγραμματιζόμενη Λογική είναι ο master των άλλων δύο.
- Accelerator Coherency Ports: Μια ενιαία ασύγχρονη σύνδεση μεταξύ της Προγραμματιζόμενης Λογικής και του Snoop Control Unit (SCU) εντός του APU, με πλάτος διαύλου 64 bit. Αυτή η θύρα χρησιμοποιείται για να επιτευχθεί συνοχή μεταξύ των προσωρινών μονάδων APU και των στοιχείων εντός της Προγραμματιζόμενης Λογικής. Η Προγραμματιζόμενη Λογική είναι ο master.
- High Performance Ports: Οι τέσσερις διεπαφές AXI υψηλής απόδοσης περιλαμβάνουν FIFO buffers για να ικανοποιήσουν τη συμπεριφορά ανάγνωσης και εγγραφής 'burst' και υποστηρίζουν επικοινωνίες υψηλού ρυθμού μεταξύ της Προγραμματιζόμενης Λογικής και των στοιχείων μνήμης στο Σύστημα Επεξεργασίας. Το πλάτος δεδομένων είναι είτε 32 είτε 64 bit, και η Προγραμματιζόμενη Λογική είναι ο master και των τεσσάρων διεπαφών.

Τρόπος λειτουργίας του AXI

Το πρωτόκολλο AXI περιγράφει μία διεπαφή ανάμεσα σε ένα AXI master και ένα AXI slave. Έχουν τη μορφή IP (Layout designs of integrated circuits) πυρήνων οι οποίοι ανταλλάσσουν πληροφορίες μεταξύ τους. Οι memory mapped masters και slaves χρησιμοποιούνται για να δρομολογήσουν την επικοινωνία μεταξύ ενός ή παραπάνω masters και slaves. Τοσο το AXI4 όσο και AXI4-Lite διαθέτουν 5 κανάλια.

- Read Address Channel
- White Adress Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

Τα δεδομένα ρέουν και προς τις δύο κατευθύνσεις με το μέγεθος των δεδομένων να ποικίλει ανάλογα με τη διεπαφή που χρησιμοποιείται.



Σχήμα 2.13: AXI master-slave

Κεφάλαιο 3

Bayes

3.1 Θεώρημα Bayes

Από τους σημαντικότερους στόχους της πιθανοθεωρητικής προσέγγισης στη μηχανική μάθηση είναι η εύρεση της πιο πιθανής υπόθεσης του χώρου υποθέσεων H, δεδομένου ενός σώματος εκπαίδευσης D και της όποιας γνώσης ενδεχομένως διαθέτουμε για τις πιθανότητες των διαφόρων υποθέσεων $h \in H$. Η πιθανότητα ισχύος μιας υπόθεσης h δεδομένου ενός συνόλου στιγμιότυπων D δίνεται από νόμο του Bayes[12]:

$$P(h \mid D) = \frac{P(D \mid h)\, P(h)}{P(D)} \tag{3.1}$$

όπου:

- $P(h)$: η εκ των προτέρων πιθανότητα ισχύος της h, χωρίς να προηγηθεί παρατήρηση των δεδομένων του D.
- $P(D|h)$: η εκ των προτέρων πιθανότητα παρατήρησης των δεδομένων του D, με δεδομένη την υπόθεση h (πιθανοφάνεια - likelihood).
- $P(D)$: η εκ των προτέρων πιθανότητα παρατήρησης των δεδομένων του D. Ο συγκεκριμένος όρος απλοποιείται και δε συμμετέχει στους υπολογισμούς.
- $P(h|D)$: η ζητούμενη εκ των υστέρων πιθανότητα ισχύος της υπόθεσης h δεδομένης της παρατήρησης των δεδομένων του D.

Αν στον υπολογισμό της $P(h|D)$ δεν ενδιαφέρει ο ακριβής υπολογισμός της πιθανότητας αλλά το ποιά είναι η πιο πιθανή υπόθεση h δεδομένου του D ο παρανομαστής μπορεί να αγνοηθεί. Η αναζήτηση επομένως της πιο πιθανής υπόθεσης h δεδομένου του D ανάγεται στην εύρεση της υπόθεσης εκείνης με τη μεγαλύτερη εκ των υστέρων πιθανότητα (Maximum A-Posteriori ή MAP hypothesis). Ορίζουμε την υπόθεση αυτή ως:

$$h_{map} = argmax_{h \in H} P(D) = argmax_{h \in H} \left\{ \frac{P(h)P(D \mid h)}{P(D)} \right\} = argmax_{h \in H} \{P(h)P(D \mid h)\} \tag{3.2}$$

Για το πρόβλημα της ταξινόμησης, χρησιμοποιούμε διανυσματική αναπαράσταση των δεδομένων, δεχόμενοι ότι:

- Y : τυχαία μεταβλητή που δείχνει την κλάση ενός στιγμιότυπου.
- X : διάνυσμα τυχαίων μεταβλητών που δείχνει τις τιμές των παρατηρούμενων χαρακτηριστικών.
- c : μια συγκεκριμένη ετικέτα κλάσης.
- x : ένα συγκεκριμένο παρατηρούμενο διάνυσμα.

Ο σκοπός λοιπόν του πιθανολογικού αυτού ταξινομητή είναι με στοιχεία από x_0 έως x_n και με κλάσεις από c_0 έως c_k να υπολογίσει την πιθανότητα τα στοιχεία να ανήκουν σε κάθε μια από τις κλάσεις και να επιστρέψει την κλάση που συγκεντρώνει την μεγαλύτερη πιθανότητα. Έτσι λοιπόν για κάθε κλάση θέλουμε να υπολογίζουμε το $P(c_i|x_0, \ldots, x_n)$. Για να το κάνουμε αυτό χρησιμοποιούμε τον παρακάτω τύπο:

$$p(Y = c \mid X = x) = \frac{p(Y = c)p(X = x \mid Y = c)}{p(X = x)} \tag{3.3}$$

Άρα προκειμένου να βρούμε την κλάση y με είσοδο $X = x_1, \ldots, x_n$ χρησιμοποιώντας το κανόνα MAP έχουμε:

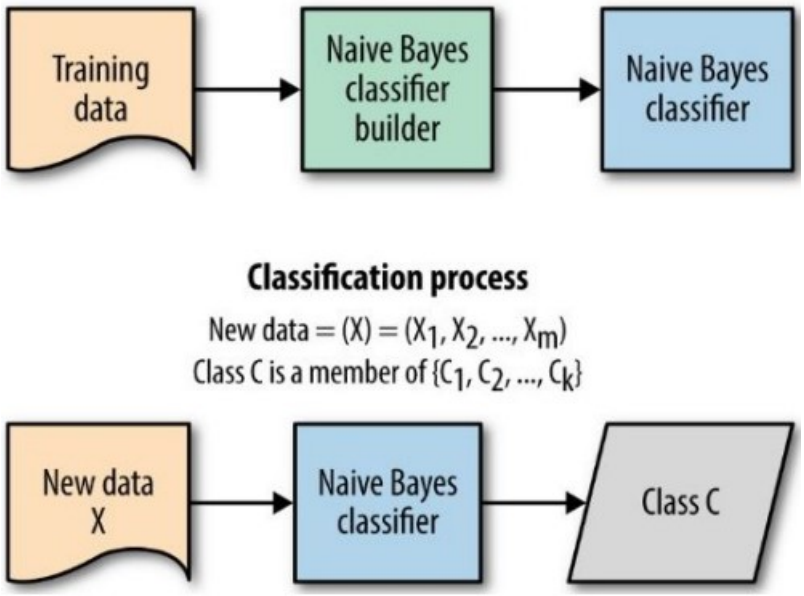
$$y = argmax_{\forall c} \{p(Y = c \mid X = x)\} = argmax_{\forall c} \{p(Y = c)p(X = x \mid Y = c)\} \tag{3.4}$$

3.2 Naive Bayes

Ο Naive Bayes ταξινομητής προσφέρει μια απλή πιθανοθεωρητική προσέγγιση στα προβλήματα μηχανικής μάθησης με επίβλεψη, όπου στόχος μας είναι να προβλέψουμε επακριβώς την κατηγορία-κλάση των στιγμιότυπων δοκιμής χρησιμοποιώντας ταξινομημένα στιγμιότυπα εκπαίδευσης που περιλαμβάνουν την πληροφορία της κλάσης που ανήκουν. Βασίζεται σε δυο σημαντικές εκλαϊκευτικές υποθέσεις. Πρώτον, υποθέτει ότι κάθε χαρακτηριστικό των στιγμιότυπων είναι στοχαστικά ανεξάρτητα των υπολοίπων, δεδομένης της κλάσης και δεύτερον ότι δεν υπάρχουν άλλα κρυφά χαρακτηριστικά που να επηρεάζουν την διαδικασία της πρόβλεψης. Κάτι το οποίο τις περισσότερες φορές δεν ισχύει εξ ου και το όνομα του αλγορίθμου (Naive = Αφελής). Ο οποίος όμως παρόλα αυτά τείνει να έχει καλά αποτελέσματα κατά την εφαρμογή του. Έτσι η πιθανότητα της σχέσης παραπάνω μετατρέπεται σε γινόμενο πιθανοτήτων.

Οπότε:

$$\begin{aligned} y &= argmax_{k \in (1,\ldots,|C|)} \{p(x_1, \ldots, x_n)\} \\ &= argmax_{k \in (1,\ldots,|C|)} \left\{ p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k) \right\} \end{aligned} \tag{3.5}$$



Σχήμα 3.1: Naive Bayes Classification Process

3.3 Gaussian Naive Bayes (GNB)

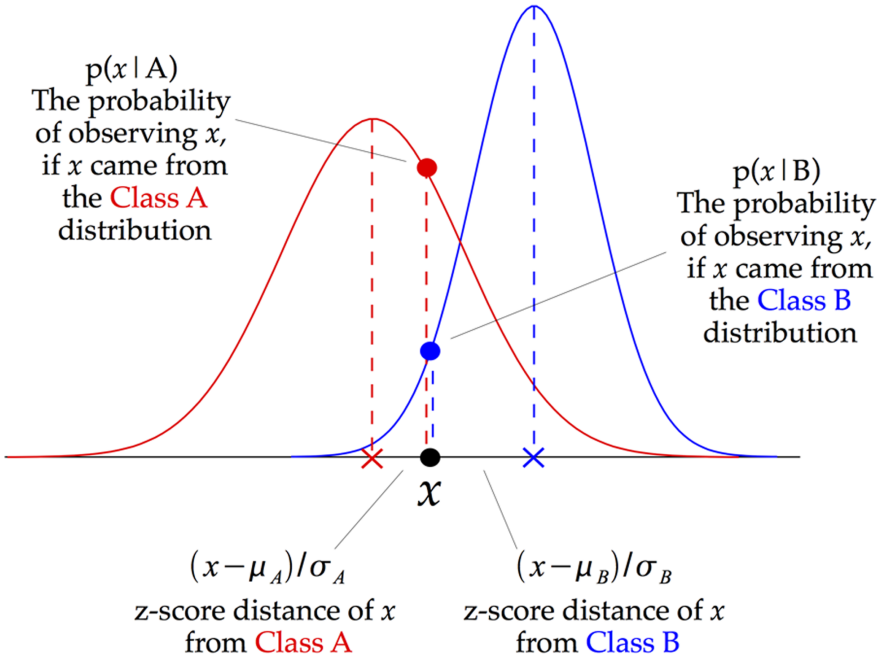
Σώμα Πρόβλεψης

Ο παράγοντας $p(C_k)$ υπολογίζεται βάσει της συχνότητας εμφάνισης της κλάσης C_k στα στιγμιότυπα του σώματος εκπαίδευσης. Οι δεσμευμένες πιθανότητες $p(x_i|C_k)$ υπολογίζονται ανάλογα με το αν το χαρακτηριστικό x_i είναι διακριτό ή συνεχές. Για τα διακριτά χαρακτηριστικά των διανυσμάτων, εκείνα δηλαδή που παίρνουν διακριτές τιμές, η πιθανότητα αυτή είναι ένας πραγματικός αριθμός μεταξύ 0 και 1 που αντιπροσωπεύει την πιθανότητα το εκάστοτε χαρακτηριστικό X_i να πάρει την τιμή x_i δεδομένης της κλάσης C_k. Για τα συνεχή χαρακτηριστικά, θεωρούμε ότι οι τιμές ακολουθούν μια γκαουσιανή κατανομή[13] (ξεχωριστή για κάθε χαρακτηριστικό), η οποία προσεγγίζεται από τα διανύσματα εκπαίδευσης. Η συνήθης θεώρηση είναι οι τιμές των χαρακτηριστικών είναι κανονικά κατανεμημένες. Οπότε για συνεχή χαρακτηριστικά έχουμε:

$$p(x_i \mid C_k) = \frac{1}{\sqrt{2\pi\sigma_{C_k}^2}} e^{-\frac{(x_i - \mu_{C_k})^2}{2\sigma_{C_k}^2}} \tag{3.6}$$

Σώμα Εκπαίδευσης

Το παραπάνω μοντέλο μας αφήνει ένα μικρό αριθμό παραμέτρων που θα εκτιμηθούν από το σώμα εκπαίδευσης. Για κάθε κλάση και συνεχές χαρακτηριστικό, χρειάζεται να εκτιμήσουμε την μέση τιμή και την διακύμανση της κατανομής που ακολουθούν οι τιμές του χαρακτηριστικού, δεδομένης της κλάσης.



Σχήμα 3.2: Gaussian Naive Bayes

Κεφάλαιο 4

Υλοποίηση Επιταχυντή

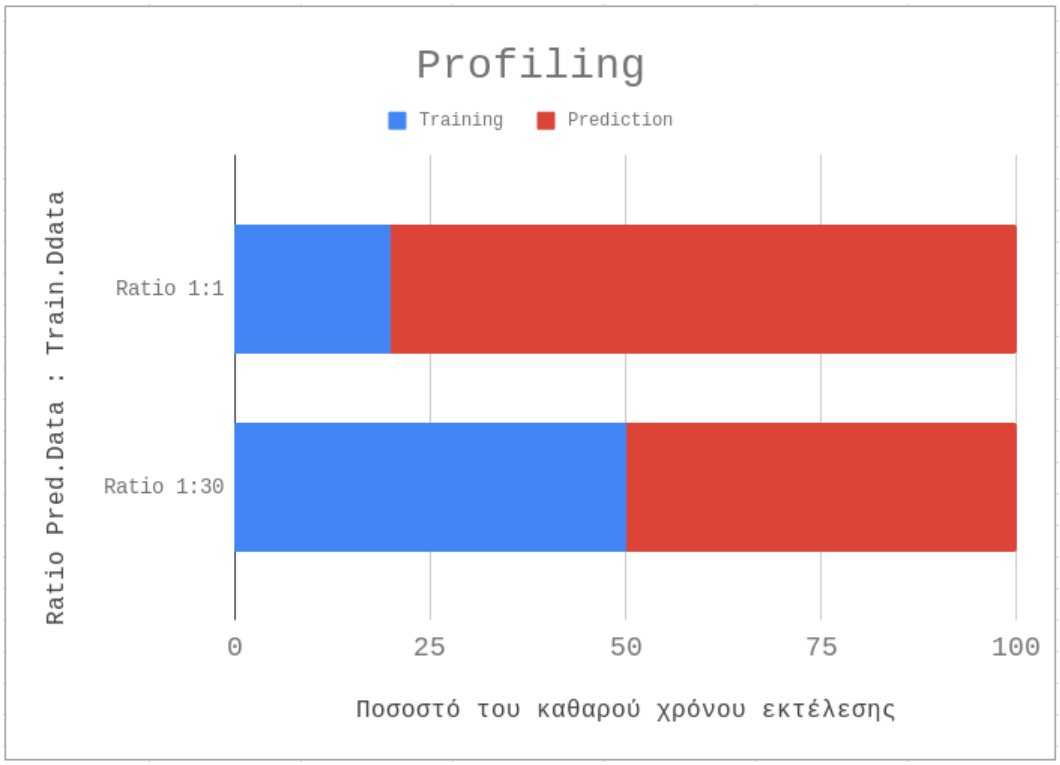
4.1 Αρχική Υλοποίηση

4.1.1 Profiling και Ανάλυση αλγορίθμου

Ξεκινώντας την μελέτη του επιταχυντή μας, αναπτύξαμε μια C υλοποίηση του αλγορίθμου Naive Bayes. Κατά την ανάλυση του αλγορίθμου παρατηρήσαμε εκείνα τα σημεία που θα άξιζε να εκμεταλλευτούμε προκείμενου να γίνει αποδοτική χρήση του επιταχυντή υλικού. Το γεγονός πως το κομμάτι της εκπαίδευσης του αλγορίθμου αποτελείται από αρκετές εμφωλευμένες επαναλήψεις για την δημιουργία του μοντέλου πρόβλεψης, μας δίνει την δυνατότητα αξιοποίησης τεχνικών παραλληλοποίησης και κατ' επέκταση επιτάχυνσης της συνάρτησης παρά τις σχετικά μη απαιτητικές μαθηματικές πράξεις που καλείται να υπολογίσει. Απ΄ την άλλη πλευρά το κομμάτι της πρόβλεψης του αλγορίθμου είναι ιδιαίτερα απαιτητικό στο κομμάτι των μαθηματικών υπολογισμών κάτι το οποίο κάνει την συγκεκριμένη συνάρτηση κατάλληλη για υλοποίηση στο υλικό με σκοπό την επιτάχυνση. Κατά το profiling παρατηρήσαμε ότι, παρότι το πιο απαιτητικό κομμάτι του αλγορίθμου είναι το κομμάτι της πρόβλεψης, το κομμάτι της εκπαίδευσης είναι εξίσου απαιτητικό για πολύ μεγάλο αριθμό δεδομένων εκπαίδευσής. Συγκεκριμένα παρατηρήσαμε ότι χωρίς τον χρόνο που χρειάζεται ο αλγόριθμος για να κάνει εξαγωγή των δεδομένων απ τα datasets, η πρόβλεψή καταλαμβάνει το 80% του χρόνου εκτέλεσης της εφαρμογής ενώ η εκπαίδευσή το 20% (για ίσο αριθμό δεδομένων εκπαίδευσης και πρόβλεψης). Τα ποσοστά όμως αυτά αλλάζουν για για αναλογία $(1:30)$ και σε αυτήν την περίπτωση, εκπαίδευση και πρόβλεψη χρειάζονται τον ίδιο χρόνο. Το γεγονός λοιπόν, πως η αναλογία, των δεδομένων εκπαίδευσης και πρόβλεψης, εξαρτάται από τις ανάγκες της εκάστοτε πραγματικού-χρόνου εφαρμογής και δεν είναι προκαθορισμένη, δημιουργεί την ανάγκη για επιτάχυνση και των δύο συναρτήσεων ώστε να καλυφθεί κάθε πιθανή περίπτωση].

Listing 4.1: Training Function

```
void NBtraining_accel(
                int DataPack,
                int n_per_class[N_class],
                float data[N_tr_data*N_feat],
                float priors[N_class],
                float means[N_class*N_feat],
                float variances[N_class*N_feat]) {
```



Σχήμα 4.1: Profiling

```
int i = 0 , j = 0, k = 0, class = 0, data_pointer[N_class + 1];
float var[N_feat], feature_means[N_class][N_feat], sums[N_feat
    ], ldata[N_feat];


data_pointer[0] = 0;
for (i = 0; i < N_class; i++){
        data_pointer[i+1] = (n_per_class[i]*N_feat +
            data_pointer[i]);
}
for (class = 0; class < N_class ; class++){
        for (i = 0; i < n_per_class[class]; i++){
                for ( j = 0; j < N_feat; j++){
                        if (i == 0) sums[j] = 0;
                        sums[j] += data[data_pointer[class] + j
                            + i*N_feat];
                }
        }

        for(i = 0; i < N_feat; i++){
                means[(class * N_feat) + i]=(sums[i]/(float)
                    n_per_class[class]);
        }
}
```

```
        for (class = 0; class < N_class ; class++){
                for ( i = 0 ; i < n_per_class[class]; i++ ){
                        for ( j = 0; j < N_feat; j++){
                                if (i == 0) var[j] = 0;
                                variances[(class * N_feat) + i] += (
                                    ldata[j] - feature_means[class][j])
                                     * (ldata[j] - feature_means[class
                                    ][j]);
                        }
                }

                for(i = 0; i < N_feat; i++){
                        variances[(class * N_feat) + i] = variances[(
                            class * N_feat) + i] / (float)n_per_class[
                            class];
                }
        }

        for (class = 0; class < N_class; class++){
                priors[class] = n_per_class[class]/(float)DataPack;
        }
}
```

Listing 4.2: Prediction Function

```
int NBprediction_accel( float data[N_feat],
                        float means[N_class*N_feat],
                        float variances[N_class*N_feat] ,
                        float priors[N_class] ) {

int i, j, prediction;
float threshold, numerator, max_likelihood, d_Pi;

d_Pi = 2 * M_PI ;
prediction = 0;
max_likelihood = -INFINITY;

for (i = 0; i < N_class; i++){
        for (j = 0; j < N_feat; j++){
            if (variances[i*N_feat + j] > threshold){
                numerator *= 1 / sqrt(d_Pi * variances[i*N_feat + j]))*
                    exp((-1*(data[j] - mean[i*N_feat + j])*(data[j] -
                    mean[i*N_feat + j])) / (2 * variances[i*N_feat + j
                    ]));
            }
            if (numerator > max_likelihood){
```

```
                max_likelihood = numerator;
                prediction = i;
            }
        }
}
return prediction;
}
```

Προκειμένου λοιπόν να επιταχύνουμε τις παραπάνω συναρτήσεις αλλά και να αξιολογήσουμε την απόδοση τους χρησιμοποιήσαμε το εργαλείο της Xilinx SDSoC. Μέσω του εργαλείου αυτού οι παραπάνω σε C υλοποιήσεις μεταφράζονται σε γλώσσα υλικού HDL και έτσι γίνεται η μεταφορά τους στην προγραμματιζόμενη λογική (PL). Για να μπορέσουμε να επιτύχουμε όμως αξιοπρεπή αποτελέσματα τα όποια να είναι συγκρίσιμα σε σχέση με έναν ενσωματωμένο επεξεργαστή ARM προστέθηκαν και ντιρεκτίβες HLS/SDS μέσω των οποίων μπορέσαμε να παρέμβουμε άμεσα στον τρόπο με τον οποίο επιθυμούμε να υλοποιηθεί ο επιταχυντής υλικού. Παρακάτω λοιπόν, γίνεται μια ανασκόπηση των σταδίων από τα οποία περάσαμε προκειμένου να καταλήξουμε στην τελική υλοποίηση που εμφανίζει και την μεγαλύτερη επιτάχυνση. Τα πρώτο και πιο εμφανές εμπόδιο που παρατηρήσαμε είναι πως το εργαλείο δεν ήταν σε θέση να μεταφέρει στο υλικό τις παραπάνω συναρτήσεις προειδοποιώντας ότι η συγκεκριμένη υλοποίηση δεν μπορεί να ολοκληρωθεί καθώς με το προεπιλεγμένο πρωτόκολλο (βλ. AXI4 Memory-Mapped) το FPGA δεν μπορεί να επεξεργαστεί κομμάτι δεδομένων, μεγαλύτερο από τα διαθέσιμα BRAMs του.

4.1.2 Απόδοση Αρχικής Υλοποίησης

Training & Prediction:

- Η μέτρηση είναι αδύνατη: Αδυναμία αντιγραφής δεδομένων στις BRAMs

4.2 Βελτιστοποίηση Πρώτη

4.2.1 Καθορισμός Διεπαφής Επικοινωνίας PS-PL

Για να ξεκινήσουμε λοιπόν τις βελτιστοποιήσεις, πρέπει πρώτα να καθορίσουμε τη επαφή επικοινωνίας μεταξύ του Συστήματος Επεξεργασίας και της Προγραμματιζόμενης Λογικής. Για να μπορέσουμε να επεξεργαστούμε ένα κομμάτι δεδομένων μεγαλύτερο από τα διαθέσιμα BRAMs μέσα στο FPGA, πρέπει να χρησιμοποιήσουμε πρωτόκολλο AXI4-Stream, αντί του πρωτοκόλλου AXI4 Memory-Mapped. Η χρήση του πρωτοκόλλου AXI-Stream δεν είναι φυσικά η μονή παραμετροποίηση που κάναμε θέλοντας να ξεφύγουμε από τον αυτόματο τρόπο με τον όποιο δημιουργεί την διεπαφή το SDSoC.

Listing 4.3: Training Function - Header file (accelerator.h)

```
#pragma SDS data copy(data[0:N_feat*Datapack])

#pragma SDS data access_pattern ( data:SEQUENTIAL, means:SEQUENTIAL,
    variances:SEQUENTIAL, priors:SEQUENTIAL )
```

```
#pragma SDS data mem_attribute ( data:CACHEABLE, means:CACHEABLE,
    variances :CACHEABLE, priors:CACHEABLE, n_per_class:CACHEABLE)

#pragma SDS data data_mover ( data:AXIDMA_SIMPLE, priors:AXIDMA_SIMPLE,
     variances:AXIDMA_SIMPLE, means:AXIDMA_SIMPLE )

#pragma SDS data sys_port ( data:ACP, means:ACP, variances:ACP, priors:
    ACP )

void NBtraining_accel( int DataPack,
                       int *n_per_class,
                       float *data,
                       float *priors,
                       float *means,
                       float *variances);
```

Listing 4.4: Prediction Function - Header file (accelerator.h)

```
#pragma SDS data copy(data[0:N_feat])

#pragma SDS data access_pattern ( data:SEQUENTIAL, means:SEQUENTIAL,
    variances:SEQUENTIAL, priors:SEQUENTIAL )

#pragma SDS data mem_attribute ( data:CACHEABLE, means:CACHEABLE,
    variances:CACHEABLE, priors:CACHEABLE )

#pragma SDS data data_mover ( data:AXIDMA_SIMPLE, priors:AXIDMA_SIMPLE,
     variances:AXIDMA_SIMPLE, means:AXIDMA_SIMPLE )

#pragma SDS data sys_port ( data:ACP, means:ACP, variances:ACP, priors:
    ACP )

int NBprediction_accel( float *data,
                        float *means,
                        float *variances,
                        float *priors );
```

Τα παραπάνω SDS pragmas τοποθετούνται ακριβώς πριν την δήλωση της συνάρτησης. Σκοπός τους είναι να καθορίσουν τον τρόπο επικοινωνίας ταυ FPGA με την CPU, τα DMA που θα χρησιμοποιήσουμε καθώς και τον προσδιορισμό της επαφής-πρωτοκόλλου για την μεταφορά των δεδομένων από τους DMAs στον πυρήνα.

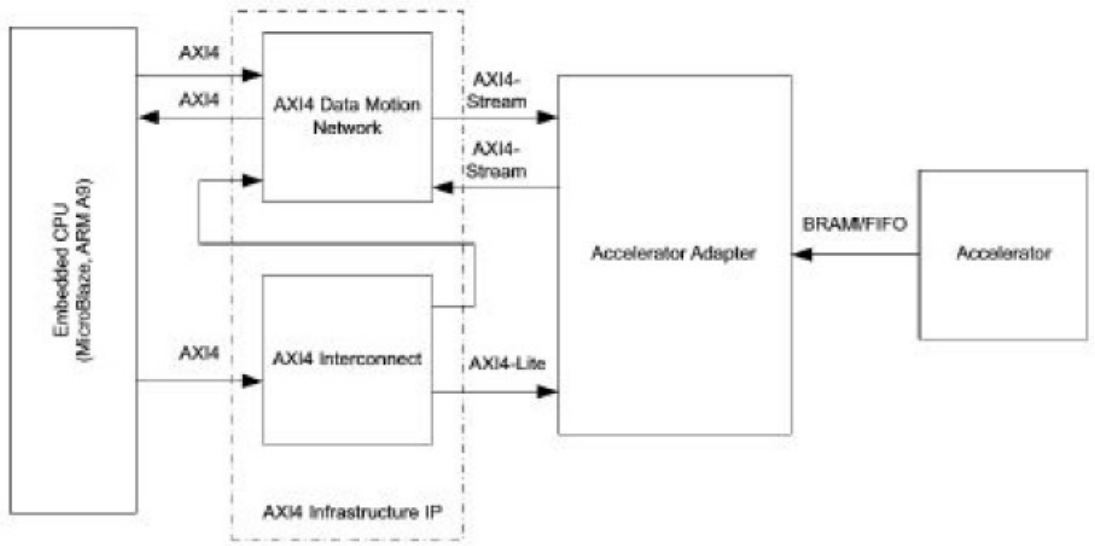
```
#pragma SDS data copy
```

Το πρώτο pragma **data copy** μας επιτρέπει να έχουμε μεταβλητό μέγεθος στον πινάκα εισαγωγής δεδομένων και μέσω της σταθεράς N_tr_data επιλεγούμε ως μέγιστη τιμή τις 2000 γραμμές δεδομένων. Έτσι κρατήσαμε το μέγεθος των δεδομένων κάτω από 8MB προκειμένου να αξιοποιήσουμε τον ταχύτερο τρόπο μεταφοράς δεδομένων του πρωτόκολλου AXI. Ουσιαστικά το συγκεκριμένο pragma

κάνει μια πλήρη αντιγραφή των δεδομένων μεταξύ της μνήμης του επεξεργαστή και της συνάρτησης του υλικού με την μεταφορά των δεδομένων να την αναλαμβάνει ο κατάλληλος data_mover.

#pragma SDS data access_pattern

Το pragma αυτό, μας επιτρέπει την χρήση διεπαφής AXI4-Stream προς την υποδομή AXI ανάμεσα στο υλικό και την CPU, και διεπαφή BRAM/FIFO προς τον πυρήνα επιτάχυνσης που κατασκευάσαμε. Το συγκεκριμένο IP χρησιμοποιείται για τη βελτίωση της επίδοσης σε επίπεδο συστήματος του επιταχυντή που βρίσκεται πάνω στο FPGA. Επίσης παρέχει έναν πολύ απλούστερο τρόπο επικοινωνίας της CPU με το FPGA, αποκρύπτοντας μεγάλο μέρος της πολυπλοκότητας. Ο πυρήνας μας επικοινωνεί με το accelerator adapter με διεπαφή FIFO διασφαλίζοντας ωστόσο πάντα ότι η ανάγνωση των δεδομένων γίνεται σειριακά έως ότου αυτά αποθηκευτούν σε BRAMs αφού η διεπαφή FIFO τα αποστέλλει με τέτοιο τρόπο στο πυρήνα. Συνολικότερα παρέχει μια AXI4-Stream διεπαφή για την μεταφορά των



Σχήμα 4.2: AXI-Interconnect

πινάκων προς την AXI υποδομή και μια AXI-LITE για την μεταφορά των αριθμών ανάμεσα στην CPU και το FPGA , καθώς και μια BRAM/FIFO προς τον επιταχυντή.

#pragma SDS data mem_attribute

Μέσω του **data mem_attribute** δηλώνουμε ότι τα ορίσματα έχουν αποθηκευτεί σε φυσική συνεχή μνήμη χρησιμοποιώντας sds_alloc() αντί για malloc().

#pragma SDS data data_mover

Το συγκεκριμένο pragma καθορίζει τον τύπο του μεταφορέα των δεδομένων, που χρησιμοποιείται για τη μεταφορά των πινάκων. Το AXI Direct Memory Access (AXI DMA) IP παρέχει άμεση πρόσβαση μνήμης υψηλής ταχύτητας, μεταξύ μνήμης και περιφερειακών τύπου AXI4-Stream. Πιο συγκεκριμένα, η λειτουργία Simple DMA παρέχει μια διαμόρφωση για απλές μεταφορές DMA σε κανάλια MM2S (Memory Mapped to Stream) και S2MM (Stream to Memory Mapped) που απαιτεί λιγότερη χρήση των πόρων του FPGA, ωστόσο οι πίνακες που μεταφέρονται με simple DMAs πρέπει να είναι 1D, μικρότεροι από 8 MB και επίσης να έχουν δεσμευτεί χρησιμοποιώντας την sds_alloc().

```
#pragma SDS data sys_port
```

Το τελευταίο pragma, καθορίζει τη θύρα μνήμης που παρέχει μια συνεκτική διεπαφή μεταξύ της Προγραμματιζόμενης Λογικής και της εξωτερικής μνήμης, για γρήγορη εκκαθάριση/ακύρωση της κρυφής μνήμης.

Σε αυτό το σημείο, δοκιμάζοντας εκ νέου την απόδοση της ενιαίας υλοποίησης που είχαμε δημιουργήσει, στην προσπάθειά μας να μεταφέρουμε στο υλικό τόσο το κομμάτι της εκπαίδευσης όσο και της πρόβλεψης, παρατηρούμε πως η μεταφορά μεγάλου όγκου δεδομένων δεν αποτελεί πλέον πρόβλημα, όμως οι υπολογιστικοί πόροι της συσκευής μας εξαντλήθηκαν κατά την διαδικασία μεταφοράς των συναρτήσεων στην Προγραμματιζόμενη Λογική, παρά την χρήση κάποιων βασικών pragmas HLS για την μείωση των πόρων.

Listing 4.5: Βασική υλοποίηση Training

```
void NBtraining_accel(
                int DataPack,
                int n_per_class[N_class],
                float data[N_tr_data*N_feat],
                float priors[N_class],
                float means[N_class*N_feat],
                float variances[N_class*N_feat]) {

    int i = 0, j = 0, k = 0, class = 0, data_pointer[N_class + 1];
    float sums[N_feat];

    data_pointer[0] = 0;
    for (i = 0; i < N_class; i++){
    #pragma HLS pipeline II=1
        data_pointer[i+1] = (n_per_class[i]*N_feat + data_pointer[i]);
    }

    for (class = 0; class < N_class ; class++){
        for (i = 0; i < n_per_class[class]; i++){
        #pragma HLS loop_tripcount min=1 max=250
            for ( j = 0; j < N_feat; j++){
                if (i == 0) sums[j] = 0;
                sums[j] += data[data_pointer[class] + j + i*N_feat];
            }
        }
        for(i = 0; i < N_feat; i++){
        #pragma HLS pipeline II=1
            means[(class * N_feat) + i]=(sums[i]/(float)n_per_class[
                class]);
        }
    }
    for (class = 0; class < N_class ; class++){
        for ( i = 0 ; i < n_per_class[class]; i++ ){
```

```
        #pragma HLS loop_tripcount min=1 max=250
            for ( j = 0; j < N_feat; j++){
                if (i == 0) var[j] = 0;
                variances[(class * N_feat) + i] += (ldata[j] -
                    feature_means[class][j]) * (ldata[j] -
                    feature_means[class][j]);
            }
        }
        for(i = 0; i < N_feat; i++){
        #pragma HLS pipeline II=1
            variances[(class * N_feat) + i] = variances[(class * N_feat
                )+ i] / (float)n_per_class[class];
        }
    }
    for (class = 0; class < N_class; class++){
    #pragma HLS pipeline II=1
            priors[class] = n_per_class[class]/(float)DataPack;
    }
}
```

Listing 4.6: Βασική υλοποίηση Prediction

```
int NBprediction_accel( float data[N_feat],
                        float means[N_class*N_feat],
                        float variances[N_class*N_feat] ,
                        float priors[N_class] ) {

    int i, j, prediction;
    float threshold, numerator, max_likelihood, d_Pi;

    d_Pi = 2 * M_PI ;
    prediction = 0;
    max_likelihood = -INFINITY;

    for (i = 0; i < N_class; i++){
        for (j = 0; j < N_feat; j++){
        #pragma HLS pipeline II=1
            if (variances[i*N_feat + j] > threshold){
                numerator *= 1 / sqrt(d_Pi * variances[i*N_feat + j]))*
                    exp((-1*(data[j] - mean[i*N_feat + j])*(data[j] -
                    mean[i*N_feat + j])) / (2 * variances[i*N_feat + j
                    ]));
            }
        }
        if (numerator > max_likelihood){
            max_likelihood = numerator;
            prediction = i;
```

```
        }
    }
    return prediction;
}
```

4.2.2 Αποδοση 1ης Βελτιστοποιησης

Training & Prediction:

- Η μέτρηση είναι αδύνατη: Μη επαρκείς πόροι

4.3 Βελτιστοποίηση Δεύτερη

4.3.1 Διαχωρισμός Συναρτήσεων

Σε αυτό το στάδιο λοιπόν αποφασίστηκε πως αφού οι δυο συναρτήσεις μαζί δεν είναι δυνατόν να μεταφερθούν με επιτυχία στην προγραμματιζόμενη λογική, μιας και ξεπερνούν τους διαθέσιμους υπολογιστικούς πόρους (DSPs , LUTs), θα πρέπει να δημιουργηθούν δυο διαφορετικοί επιταχυντές. Με αυτόν τον τρόπο και γνωρίζοντας απ΄ το profiling για ποια αναλογία training data : prediction data αξίζει να επιταχύνουμε ποια συνάρτηση, δίνουμε την επιλογή στον χρηστή ανάλογα με τις ανάγκες της εφαρμογής να μειώσει συνολικά τον χρόνο εκτέλεσης του προγράμματος του εκτελώντας κάθε φορά την συνάρτηση με το μεγαλύτερο χρόνο εκτέλεσης. Συγκεκριμένα γνωρίζουμε πως όταν η αναλογία είναι μεγαλύτερη από $(1 : 30)$ τότε παρότι το κομμάτι της πρόβλεψης είναι πολύ πιο απαιτητικό στον χρόνο εκτέλεσης του (για ίσα δεδομένα εκπαίδευσης και πρόβλεψης), η συνάρτηση η όποια αξίζει να επιταχύνουμε είναι η συνάρτηση της εκπαίδευσης. Ένας άλλος παράγοντας που συνηγόρησε υπέρ αυτής της απόφασης είναι το γεγονός πως τα δεδομένα εκπαίδευσης πρέπει να είναι αρκετά ώστε να εκπαιδεύσουμε σωστά το μοντέλο πρόβλεψης. Σε αντίθεση με τα δεδομένα πρόβλεψης που μπορεί να είναι από ένα έως όσα απαιτεί η εφαρμογή μας. Όποτε δεν είναι απίθανο να βρεθούμε σε νούμερα με αναλογίες όπως η παραπάνω. Παρατηρούμε πλέον ότι με τον διαχωρισμό των συναρτήσεων η κάθε μια ξεχωριστά είναι δυνατόν να μεταφερθεί στην προγραμματιζόμενη λογική αλλά οι επιδόσεις οι οποίες πετυχαίνουμε σε καμία περίπτωση δεν είναι σε θέση να συγκριθούν με έναν ARM επεξεργαστή. Έτσι γίνεται αντιληπτό πως θα πρέπει με τις κατάλληλες HLS ντιρεκτίβες να καθορίσουμε εμείς την αρχιτεκτονική και τον τρόπο υλοποίησης της συνάρτησης στο υλικό.

4.3.2 Αποδοση 2ης Βελτιστοποιησης

Training(2000 data lines):

- Accelerated Hardware: 159.000.000 cycles
- Software-Only: 182.000.000 cycles

Prediction(Per line of data):

- Accelerated Hardware: 4.093.000 cycles
- Software-Only: 2.220.000 cycles

4.4 Βελτιστοποίηση Τρίτη

4.4.1 Χρήση HLS Ντιρεκτίβων

Σε αυτό το το στάδιο σκοπός μας ήταν να αξιοποιήσουμε σε όσο το δυνατόν μεγαλύτερο βαθμό την αρχιτεκτονική του υλικού και να εκμεταλλευτούμε τις δυνατότητες παραλληλοποίησης που μας δίνει η τεχνολογία ενός FPGA μειώνοντας έτσι σε μεγάλο βαθμό τον χρόνο εκτέλεσης των συναρτήσεων μας. Ενώ μια CPU θα εκτελούσε κάθε μία από τις παραπάνω λειτουργίες σειριακά, τα εργαλεία σύνθεσης υψηλού επιπέδου της Xilinx θα επιχειρήσουν να τις χρονοδρομολογήσουν όσο συντομότερα και περισσότερο παράλληλα μας επιτρέπουν, οι λογικοί και φυσικοί-υλικοί περιορισμοί. Με τον όρο λογικούς περιορισμούς εννοούμε ότι τα εργαλεία δεν θα επιχειρήσουν π.χ. να παραβιάσουν τις εξαρτήσεις Read After Write και με τον όρο φυσικούς περιορισμούς εννοούμε ότι αν και μπορεί να υπάρχουν οδηγίες που μπορούν να εκτελεστούν ταυτόχρονα , λόγω της έλλειψης πόρων των FPGA ή λόγω της λανθασμένης διανομής των πόρων του FPGA, ενδέχεται να μην είναι σε θέση να επιτευχθεί παράλληλη εκτέλεση. Ο σκοπός μας είναι να ρυθμίσουμε σωστά το FPGA χρησιμοποιώντας εντολές υψηλού επιπέδου προκειμένου να ξεπεραστούν οι φυσικοί περιορισμοί.

4.4.2 Συνάρτηση Εκπαίδευσης - Αντιμετώπιση Προκλήσεων

Προκλήσεις

Στην συνάρτηση εκπαίδευσης κατά την ανάλυση του αλγορίθμου με σκοπό την μεταφορά του στο υλικό βρεθήκαμε αντιμέτωποι με τις εξής προκλήσεις:

- Η διπλή προσπελάση των δεδομένων για τον υπολογισμό του μέσου όρου και της διακύμανσης.
- Η αποδοτική χρήση του FPGA κατά την ανάγνωση/εγγραφή των δεδομένων.
- Η χρονοδρομολόγηση των σταδίων εξαγωγής του τελικού μοντέλου πρόβλεψης.
- Η εύρεση του ιδανικού μεγέθους πακέτου δεδομένων για αποστολή στον πυρήνα.

Η πρώτη πρόκληση, εντοπίζεται στην ανάγκη εκ νέου προσπέλασης των δεδομένων εκπαίδευσης για τον υπολογισμό της διακύμανσης ύστερα απ΄ τη πρώτη προσπέλαση για τον υπολογισμό του μέσου όρου. Η διπλή ανάγνωση των δεδομένων επιφέρει μεγάλη καθυστέρηση στον αλγόριθμο λόγω των πολλαπλών προσβάσεων στην DDR ή την κρυφή μνήμη.

$$\sigma^2 = \frac{\sum (X - \mu)^2}{N}$$

Σημ: Κατά την χρήση του βασικού τύπου, γίνεται εμφανές πως για τον υπολογισμό της διακύμανσης απαιτείται η ανάγνωση των χαρακτηριστικών X, που έχουν ήδη αναγνωστεί μια φορά για τον υπολογισμό του μέσου όρου μ.

Η δεύτερη πρόκληση, είναι ουσιαστικά η ανάγκη να προσαρμόσουμε την υλοποίηση μας στους συγκεκριμένους τρόπους ανάγνωσης και εγγραφής των δεδομένων από το FPGA, που βοηθούν στην αποδοτικότερη λειτουργία του επιταχυντή.

Η τρίτη πρόσκληση, έγκειται στην μέγιστη δυνατή παραλληλοποίηση των σταδίων υπολογισμού του κύριου υπολογιστικού βρόχου ώστε να γίνει σωστή χρήση των πόρων του FPGA.

Και τέλος η τέταρτη πρόκληση, είναι το γεγονός πως ο επιταχυντής υλικού πρέπει να υπολογίζει το μοντέλο πρόβλεψης, για μεγάλο αριθμό δεδομένων εκπαίδευσης. Ήταν αναγκαία λοιπόν, η ισορροπία ανάμεσά στην καθυστέρηση που θα επιφέρει μια επαναλαμβανόμενη κλήση του επιταχυντή, σε

περίπτωση κατάτμησης των δεδομένων, και στην καθυστέρηση που θα επιφέρει η προσπέλαση πολύ μεγάλου όγκου δεδομένων εκπαίδευσης, σε περίπτωση μη κατάτμησης των δεδομένων.

Αντιμετώπιση

Για να αντιμετωπίσουμε την πρώτη πρόκληση, έπρεπε να τροποποιήσουμε τους κλασικούς τύπους που χρησιμοποιούνται για τον υπολογισμό της διακύμανσης. Ο κύριος λόγος, για τον οποίο ήταν αναγκαία η τροποποίηση αυτή, ήταν γιατί οι βασικές φόρμουλες υπολογισμού μέσου όρου και διακύμανσης, μας αναγκάζανε να διαβάσουμε δυο φορές τα χαρακτηριστικά όλων των κλάσεων, καταργώντας την δυνατότητα μιας σειριακής ανάγνωσης των δεδομένων, δεδομένου ότι και η αντιγραφή τόσο πολλών δεδομένων στις BRAMs είναι αδύνατη. Έτσι λοιπόν μέσω της παρακάτω σχέσης,

$$\begin{aligned}
\sigma^2 &= \frac{\sum(X-\mu)^2}{N} \\
&= \frac{\sum(X^2 - 2X\mu + \mu^2)}{N} \\
&= \frac{\sum X^2}{N} - \frac{2\mu\sum X}{N} + \frac{N\mu^2}{N} \\
&= \frac{\sum X^2}{N} - 2\mu^2 + \mu^2 \\
&= \frac{\sum X^2}{N} - \mu^2
\end{aligned} \tag{4.1}$$

είμαστε σε θέση να διαβάσουμε μια φορά και μάλιστα σειριακά τα δεδομένα εκπαίδευσης. Επίσης επιτυγχάνεται ο ταυτόχρονος υπολογισμός του αθροίσματος των χαρακτηριστικών και του αθροίσματος των τετραγώνων των χαρακτηριστικών κάθε κλάσης, που είναι και οι όροι που απαιτούνται, τόσο για τον υπολογισμό του μέσου όρου όσο και της διακύμανσης.

Listing 4.7: Υπολογισμός Διακύμανσης - Αρχική Υλοποίηση

```
for (class = 0; class < N_class; class++){
        for (i = 0; i < n_per_class[class]; i++){
        #pragma HLS loop_tripcount min=1 max=250
            for ( j = 0; j < N_feat; j++){
                if (i == 0) sums[j] = 0;
                sums[j] += data[data_pointer[class] + j + i*N_feat];
            }
        }

        for(i = 0; i < N_feat; i++){
        #pragma HLS pipeline II=1
                means[(class * N_feat) + i] =(sums[i]/(float)
                    n_per_class[class]);
        }

}

for (class = 0; class < N_class; class++){
    for ( i = 0 ; i < n_per_class[class]; i++ ){
```

```
        #pragma HLS loop_tripcount min=1 max = 250
            for ( j = 0; j < N_feat; j++){
                #pragma HLS pipeline II=1
                    if (i == 0) var[j] = 0;
                    variances[(class * N_feat) + j] += (data[
                        data_pointer[class] + i*N_feat + j] -
                        feature_means[class][j]) * (data[data_pointer[
                        class] + i*N_feat + j] - feature_means[class][j
                        ]);
            }
    }
    for(i = 0; i < N_feat; i++){
    #pragma HLS pipeline II=1
        variances[(class * N_feat) + i] = variances[(class * N_feat) +
            i] / (float)n_per_class[class];
    }
}
```

Listing 4.8: Υπολογισμός Διακύμανσης - Τροποποιημένη Υλοποίηση

```
for (i = 0; i < n_per_class[class]; i++){
#pragma HLS loop_tripcount min=1 max=250

    for ( j = 0; j < N_feat; j++){
    #pragma HLS pipeline II=1
        sums[j] += data[data_pointer[class] + j + i*N_feat];
        sq_sums[j] += data[data_pointer[class] + j + i*N_feat] * data[
            dta_pointer[class] + j + i*N_feat];
    }

    for ( j = 0; j < N_feat; j++){
    #pragma HLS pipeline II=1
        means[(class * N_feat) + j ] = (sums[j]/(float)n_per_class[
            class]);
        sq_feature_means[j]=(sq_sums[j]/(float)n_per_class[class]);
    }

    for ( j = 0; j < N_feat; j++){
    #pragma HLS pipeline II=1
        variances[(class * N_feat) + j ] = sq_feature_means[j] - ((
            means[(class * N_feat) + j ])*(means[(class * N_feat) + j
            ]));
    }
}
```

Να σημειωθεί ότι η συγκεκριμένη τροποποίηση στον αλγόριθμο ήταν καταλυτική ρίχνοντας τους κύκλους ρολογιού ως και κατά 1/3 χαμηλότερα σε σχέση με την διπλή ανάγνωσή των δεδομένων.

Η αντιμετώπιση της δεύτερης πρόκλησης είναι ένα σύνθετο πρόβλημα κατά το οποίο έπρεπε να πάρουμε υπόψη αρκετές παραμέτρους που είχαν να κάνουν κυρίως με την προσαρμογή της υλοποίησής μας στις τεχνικές εκείνες κατά τις οποίες το FPGA αποδίδει με τον ταχύτερο τρόπο. Η υλοποίησή μας βασίστηκε στις εξής αρχές.

- Την σειριακή ανάγνωση και εγγραφή των πινάκων για την χρήση της διεπαφής BRAM/FIFO που παρέχει ο AXI4-Stream Accelerator.
- Την αποθήκευση των δεδομένων σε BRAMs για ταχύτερη πρόσβαση σε αυτά.
- Την αποφυγή συμφόρησης της μνήμης (memory bottlenecks).

Για την αξιοποίηση της διεπαφής AXI4-Stream, είναι απαραίτητο η ανάγνωση των δεδομένων να γίνεται σειριακά έως ότου αυτά αποθηκευτούν σε BRAMs. Η ίδια προϋπόθεση ισχύει και για την εγγραφή των δεδομένων (πρέπει να γίνεται σειριακά), στους πίνακες που αποτελούν την έξοδο του επιταχυντή, αφού η διεπαφή FIFO τα αποστέλλει κατά τέτοιο τρόπο στο πυρήνα. Έτσι λοιπόν έγιναν τα εξής βήματα. Δεδομένου ότι ο αλγόριθμος υπολογίζει τον μέσο όρο και την διακύμανση των χαρακτηριστικών ανά κλάση δημιουργούνται δυο πιθανές εκδοχές. Πρώτη εκδοχή, είναι η ανάγνωση όλων των δεδομένων του πίνακα και υπολογισμού των παραπάνω τιμών, για όλες τις κλάσεις, αφού πραγματοποιηθεί η προσπέλαση όλων των δεδομένων. Ενώ η δεύτερη εκδοχή, είναι η εύρεση των τιμών για κάθε μια κλάση ξεχωριστά και την αποθήκευση των αποτελεσμάτων σε πίνακες εξόδους που θα σταλούν στο σύστημά επεξεργασίας. Κατά την πρώτη εκδοχή συναντάμε τα εξής προβλήματα, χρήση υπερβολικής μνήμης για την αποθήκευση των μέσων ορών, των διακυμάνσεων και των ενδιάμεσων αποτελεσμάτων όλων των κλάσεων και αδυναμία χρήσης της διεπαφής AXI4-Stream και της σειριακής έγγραφης καθώς η αποθήκευση των δεδομένων θα γίνεται ανάλογά την κλάση στην οποία θα ανήκει η κάθε γραμμή δεδομένων, που έχουμε κάνει ανάγνωση. Συνεπώς οδηγούμαστε στην δεύτερη εκδοχή κατά την οποία τα δεδομένα προεπεξεργάζονται και έρχονται, ταξινομημένα ανά κλάση, στον επιταχυντή. Αυτό σημαίνει πως γνωρίζοντας από πόσες γραμμές αποτελείται κάθε κλάση, μπορούμε να διασχίσουμε σειριακά τους πίνακες. Έτσι υπολογίζουμε, για παράδειγμα, τον μέσο όρο και την διακύμανσή για την πρώτη κλάση και μόλις μεταβούμε στην επομένη κλάση αρχικοποιούμε ξανά τους πίνακες που χρησιμοποιήθηκαν, με σκοπό τον υπολογισμό των τιμών για την επόμενη κλάση.

Listing 4.9: Υπολογισμός Μοντέλου Πρόβλεψης

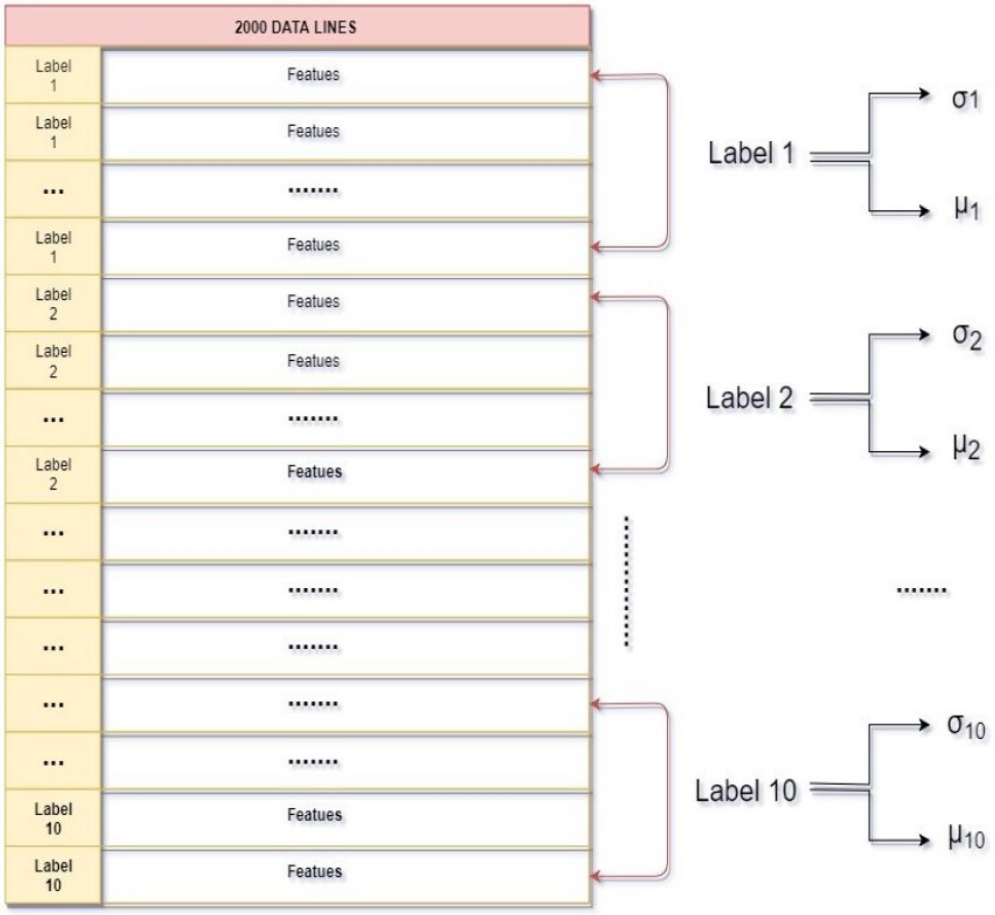
```
data_pointer[0] = 0;
for (class = 0; class < N_class; class++){
#pragma HLS pipeline II=1
        data_pointer[class+1] = (n_per_class[class]*N_feat +
            data_pointer[class]);

for (class = 0; class < N_class ; class++){
        pnt = data_pointer[class];
        for ( j = 0; j < N_feat; span>++){
        #pragma HLS pipeline II=1
                sums[j] = 0;
                sq_sums[j] = 0;
}


    //Code Block//
```

```
    for ( j = 0; j < N_feat; j++){
    #pragma HLS pipeline II=1
        means[(class * N_feat) + j ]=(sums[j]/(float)n_per_class[class
            ]);
        sq_feature_means[j]=(sq_sums[j]/(float)n_per_class[class]);
    }

    for ( j = 0; j < N_feat; j++){
    #pragma HLS pipeline II=1
        variances[(class * N_feat) + j ] = sq_feature_means[j] - ((
            means[(class * N_feat) + j ])*(means[(class * N_feat) + j
            ]));
        }
}
```



Σχήμα 4.3: Μορφή Πίνακα Δεδομένων Εκπαίδευσης

Σε αυτό το σημείο εντός της συνάρτησης του υλικού ορίζω πίνακες στους οποίους, θα είναι αποθηκευμένα τα δεδομένα μέχρι να ολοκληρωθούν οι υπολογισμοί, ώστε να είναι στην συνέχεια έτοιμα για αντιγραφή, στους πίνακες που θα αποσταλούν πίσω στο σύστημα επεξεργασίας. Με αυτόν

τον τρόπο λοιπόν έχοντας τα υπό επεξεργασία δεδομένα σε BRAMs μπορούμε να πετύχουμε την μέγιστη ταχύτητα προσπέλασής, εκμεταλλευόμενοι την εγγύτητα που προσφέρει η ανάγνωση τους από τη τοπική μνήμη του FPGA καθώς και την δυνατότητα ανάγνωσης τους και εγγραφής τους με οποίο τρόπο θέλουμε (σειριακά ή μη), χωρίς αυτό να επηρεάζει την απόδοση του επιταχυντή.

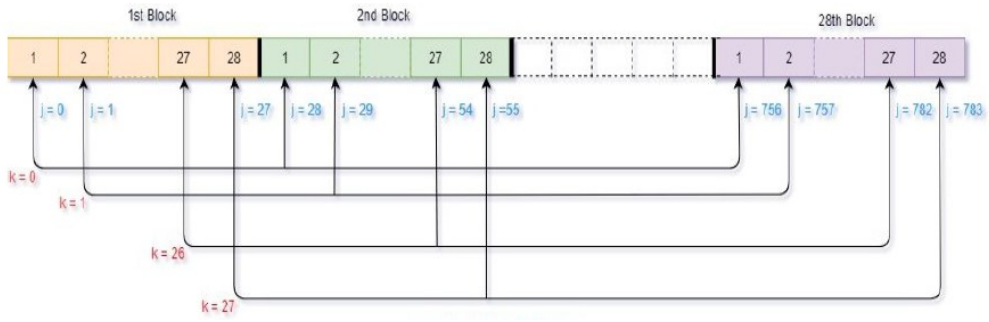
Listing 4.10: HLS pragmas in Training Function

```
void NBtraining_accel(
                int DataPack,
                int class_sum[N_class],
                float data[N_tr_data*N_feat],
                float priors[N_class],
                float means[N_class*N_feat],
                float variances[N_class*N_feat]) {

int i = 0 , j = 0, class = 0, k = 0, pnt, data_pointer[N_class + 1];
float var[N_feat], sums[N_feat], sq_sums[N_feat], feature_means[N_feat
    ], sq_feature_means[N_feat], ldata[N_feat];

#pragma HLS array_partition variable = data_pointer complete
#pragma HLS array_partition variable = var block factor=28
#pragma HLS array_partition variable = sums block factor=28
#pragma HLS array_partition variable = sq_sums block factor=28
#pragma HLS array_partition variable = feature_means block factor=28
#pragma HLS array_partition variable = sq_feature_means block factor=28
#pragma HLS array_partition variable = ldata block factor=28
```

Βλέπουμε πως επιλέξαμε να χωρίσουμε όλους τους τοπικούς πίνακες σε 28 κομμάτια από τα οποία το καθένα εμπεριέχει 28 στοιχεία. Δημιουργήσαμε εμφωλευμένες επαναλήψεις, κατά τις οποίες καταφέραμε, ο εσωτερικός βρόχος να ξεδιπλώνεται πλήρως. Πιο συγκεκριμένα, καθορίσαμε την ανάγνωση και την έγγραφη των δεδομένων ανά 28 στοιχεία αποκτώντας πρόσβαση σε ανεξάρτητη BRAM κάθε φορά, αποφεύγοντας την δημιουργία bottleneck από προηγουμένη ανάγνωση ή εγγραφή. Έχοντας λοιπόν το πλεονεκτήματα πως χρησιμοποιούμε την εσωτερική μνήμη του FPGA μπορούμε να αποκτήσουμε πρόσβαση στα δεδομένα και με μη σειριακό τρόπο και έτσι πετυχαίνουμε στις συγκεκριμένες επαναλήψεις II(Initiation Interval) = 1.



Σχήμα 4.4: BRAMs Partition

Για την τρίτη πρόκληση που είναι ουσιαστικά και το σημαντικότερο κομμάτι του αλγορίθμου

στόχος είναι η μέγιστη δυνατή παραλληλοποίηση των εργασιών ώστε να πετύχουμε και την μέγιστη δυνατή αξιοποίηση του FPGA. Παρατηρούμε ότι έχουμε δυο βρόχους. Ο πρώτος βρόχος δημιουργεί έναν πίνακα στον οποίο καταγράφουμε που αρχίζει και που τελειώνει κάθε κλάση, εντός του ταξινομημένου μονοδιάστατου πίνακα που εμπεριέχει τα δεδομένα εκπαίδευσης. Δημιουργήσαμε λοιπόν, αυτόν τον πίνακά για να επιτρέψουμε στον επιταχυντή υλικού μας να διαβάζει μια φορά και σειριακά τα δεδομένα εκπαίδευσης. Ταυτόχρονα για να παραληλλοποιήσουμε μια ακόμα ανεξάρτητη εργασία υπολογίζουμε και την συχνότητα εμφάνισης κάθε κλάσης στα δεδομένα, τα *priors* δηλαδή.

Listing 4.11: Υπολογισμός Priors

```
for (class = 0; class < N_class; class++){
#pragma HLS pipeline II=1
    data_pointer[class+1] = (class_sum[class]*N_feat + data_pointer[
        class]);
    priors[class] = class_sum[class]/(float)DataPack;
}
```

Στην συνέχεια περνάμε στον δεύτερο βρόχο της υλοποίησής μας, που είναι και ο κύριος βρόχος υπολογισμών. Συγκεκριμένα αποτελείται από 4 στάδια.

1.Αρχικοποίηση του αθροιστή των στοιχείων και του αθροιστή των τετραγώνων των στοιχείων.

Listing 4.12: Αρχικοποίηση Αθροιστών

```
for ( k = 0; k < 28; k++){
#pragma HLS pipeline II=1
     for (j = 0; j < N_feat; j+=28){
           sums[k+j] += 0;
           sq_sums[k+j] += 0;
      }
}
```

2. Για όλα τα στοιχεία μιας κλάσης, γίνεται αντιγραφή ανά γραμμή των δεδομένων εκπαίδευσης σε BRAMs και πρόσθεση των τιμών τους στους δυο συσσωρευτές.

Listing 4.13: Αντιγραφή δεδομένων σε BRAMs

```
for (i = 0; i < offset[class]; i++){
#pragma HLS loop_tripcount min=1 max=250

    for (j = 0; j < N_feat; j++){
    #pragma HLS pipeline II=1
              ldata[j] = data[data_pointer[class] + i*N_feat + j];
    }


    for ( k = 0; k < 28; k++){
    #pragma HLS pipeline II=1
         for (j = 0; j < N_feat; j+=28){
```

```
            sums[k+j] += ldata[k+j];
            sq_sums[k+j] += ldata[k+j] * ldata[k+j];
        }
    }
```

3. Υπολογισμός του μέσου όρου και του μέσου όρου των τετραγώνων των χαρακτηριστικών και ταυτόχρονα εκμεταλλευόμενοι την εγγύτητα των δεδομένων, που προσφέρει ο υπολογισμός των τιμών μέσα στην ίδια επανάληψή, υπολογίζουμε και την διακύμανση, μειώνοντας ακόμα περισσότερο τους κύκλους ρολογιού. Το συγκεκριμένο στάδιο σε πρώτη φάση αποτελούνταν από 2 στάδια, αρχικώς γινόταν ο υπολογισμός των μέσων όρων και στην συνέχεια της διακύμανσης. Ως καλύτερη λύση όμως επιλέγουμε να συγχωνεύσουμε αυτούς τους δύο βρόγχους και να οικοδομήσουμε ένα pipeline για τα δύο αυτά στάδια, πετυχαίνοντας υψηλή απόδοση, καθώς αυτό το pipeline μπορεί να επεξεργάζεται μια νέα είσοδο σε κάθε κύκλο ρολογιού, με σχεδόν τους ίδιους πόρους.

Listing 4.14: Υπολογισμός τιμής Μέσων Ορων και Διακύμανσης - Αρχικη υλοποίηση

```
for ( k = 0; k < 28; k++){
    for (j = 0; j < N_feat; j+=28){
    #pragma HLS pipeline II=1
        feature_means[j+k]=(sums[j+k]/(float)offset[class]);
        sq_feature_means[j+k]=(sq_sums[j+k]/(float)offset[class]);
    }
}
for ( k = 0; k < 28; k++){
    for (j = 0; j < N_feat; j+=28){
    #pragma HLS pipeline II=1
        var[k+j] = sq_feature_means[j+k] - ((feature_means[k+j])*(
            feature_means[k+j]));
    }
}
```

Listing 4.15: Υπολογισμός τιμής Μέσων Όρων και Διακύμανσης - Τροποιημένη υλοποίηση

```
for ( k = 0; k < 28; k++){
    for (j = 0; j < N_feat; j+=28){
    #pragma HLS pipeline II=1
        feature_means[j+k]=(sums[j+k]/(float)offset[class]);
        sq_feature_means =(sq_sums[j+k]/(float)offset[class]);
        var[k+j] = sq_feature_means - ((feature_means[k+j])*(
            feature_means[k+j]));
    }
}
```

4. Αντιγραφή με σειριακό τρόπο της διακύμανσης και του μέσου όρου στους πίνακες που θα αποστα-λούν πίσω στην CPU.

Listing 4.16: Σειριακή αντιγραφή δεδομένων για αποστολή πίσω στην CPU

```
for (j = 0; j < N_feat; j++){
```

```
#pragma HLS pipeline II=1
        means[(class * N_feat) + j ] = feature_means[j];
        variances[(class * N_feat) + j ] = sq_feature_means[j] - ((
            feature_means[j])*(feature_means[j]));
}
```

Παρατηρούμε λοιπόν πως επιτυγχάνουμε σε κάθε στάδιο, την ανάγνωση των νέων δεδομένων ανά κύκλο ρολογιού (Initiation Interval = 1) και αφού κάθε στάδιο πρέπει να περιμένει την ολοκλήρωση του προηγούμενου δεν μας δίνεται η δυνατότητα για παραπάνω παραλληλοποίηση.

Για να αντιμετωπίσουμε λοιπόν την τέταρτη πρόκληση έπρεπε να αναλογιστούμε τα εξής:

- Την χρήση του ταχύτερου μεταφορέα δεδομένων.
- Το κόστος της μεταφοράς δεδομένων από και προς το FPGA.

Αυτό που ήταν εξ αρχής γνωστό, είναι ότι δεν θα μπορούσαμε να μην έχουμε συγκεκριμένο όριο στο μέγεθος του πινάκα δεδομένων που θα εισαγάγαμε στον επιταχυντή καθότι η χρήση της λειτουργίας Simple DMA, που αποτελεί όπως είδαμε την πιο αποδοτική επιλογή, θα ήταν αδύνατη. Γνωρίζοντας έτσι, ότι το μέγεθος του πινάκα που εμπεριέχει τα δεδομένα εισαγωγής δεν θα πρέπει να ξεπερνά τα 8MB δημιουργήθηκε το ερώτημα για το ιδανικό μέγεθος πακέτου δεδομένων. Ύστερα λοιπόν, από πειραματική μελέτη παρατηρήσαμε το εξής, ότι η συχνή και αμφίδρομη επικοινωνία της συνάρτησης με το σύστημα επεξεργασίας έχει ως συνέπεια μεγάλη καθυστέρηση την όποια οι σχετικά απλοί μαθηματικοί υπολογισμοί, που υλοποιεί η συνάρτηση της εκπαίδευσης, δεν θα μπορούσε να αντισταθμίσει κάνοντας επεξεργασία ενός μικρού μεγέθους δεδομένων. Συγκεκριμένα συναντάμε το εξής φαινόμενο, μειώνοντας το μέγεθος των εισαγόμενων δεδομένων, μειώνεται ο χρόνος μεταφοράς τους στον επιταχυντή καθώς και οι κύκλοι επεξεργασίας τους. Μεγαλώνοντας όμως τον όγκο των δεδομένων, ενώ οι κύκλοι του επιταχυντή αυξάνονται σχεδόν ανάλογα δεν ισχύει το ίδιο και για τον χρόνο μεταφοράς των δεδομένων ο όποιος εμφανίζει μικρότερη και όχι ανάλογη αύξηση. Αυτό συμπερασματικά σημαίνει πως όσο μικρότερο πακέτο δεδομένων αποστέλλουμε στον επιταχυντή τόσο μεγαλύτερο κατακερματισμό των δεδομένων θα πρέπει να κάνουμε, συνεπώς τόσες περισσότερες φορές θα πρέπει να καλέσουμε τον επιταχυντή και η αύξηση αυτή των κλήσεων, θα σημαίνει περισσότερος χρόνος στην μεταφορά δεδομένων. Άρα παίρνοντας υπόψη μας τα παραπάνω και παρατηρώντας τις μετρήσεις που έχουμε κάνει για διαφορά μεγέθη πακέτων επιλέγουμε το μέγεθος των 2000 σειρών δεδομένων στην κάθε κλήση του επιταχυντή.

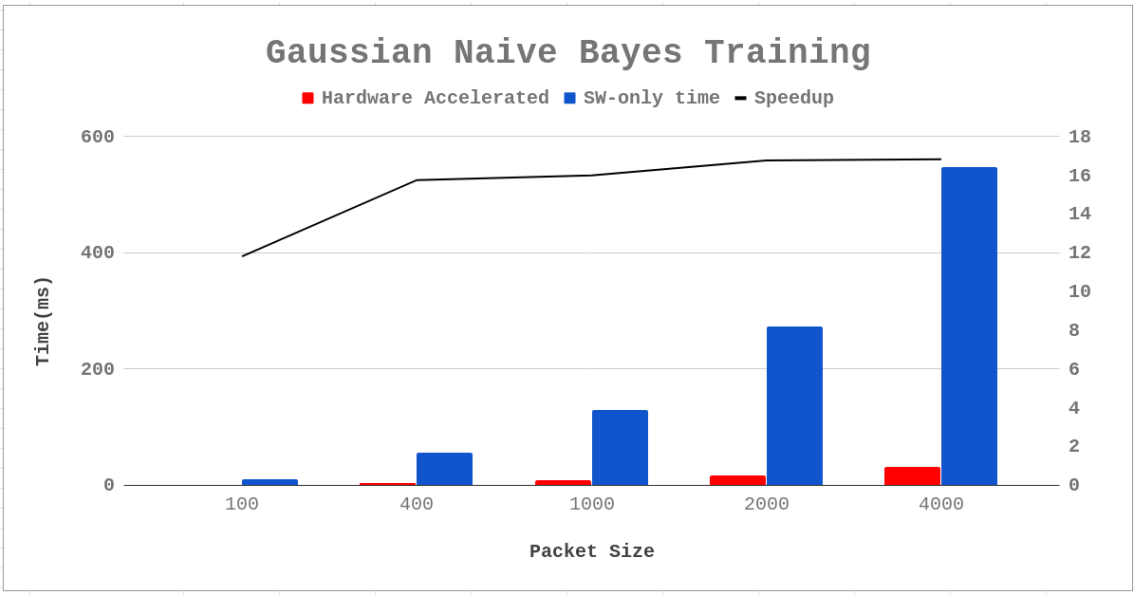
Κατώφλι κύκλων υλικού.

Κατά την αναζήτηση όλο και αποδοτικότερης υλοποίησης βρεθήκαμε μπροστά σε ένα εμπόδιο το όποιο δεν μπορέσαμε να ξεπεράσουμε. Συγκεκριμένα παρατηρήσαμε ότι ο επιταχυντής δεν μπορεί να ολοκληρώσει την συνάρτηση εκπαίδευσης σε λιγότερο από 10.000.000 κύκλους. Το κατώφλι αυτό οφείλεται στους κύκλους που απαιτούνται κατά την ανάγνωση των 8MB δεδομένων εκπαίδευσης. Η υλοποίηση μας λοιπόν φράσσεται από την ταχύτητα μεταφοράς των δεδομένων.

4.4.3 Συνάρτηση Πρόβλεψης - Αντιμετώπιση Προκλήσεων

Προκλήσεις

Κατά τον ίδιο τρόπο έγινε μελέτη και ανάλυση της συνάρτησης πρόβλεψης αντιμετωπίζοντας τις εξής προκλήσεις:



Σχήμα 4.5: Απόδοση Συνάρτησης Εκπαίδευσης για διαφορετικά πακέτα δεδομένων

- Γινόμενα κοντά στο 0 (underflow).
- Υψηλό υπολογιστικό φορτίο.
- Παραλληλοποίηση των υπολογισμών.

Η πρώτη πρόκληση οφείλεται στο γεγονός πως ο αλγόριθμος του Gaussian Naive Bayes θεωρεί όλα τα ενδεχόμενα μεταξύ τους ανεξάρτητα, όπως έχει προαναφερθεί και στην σχέση (3.5). Συνεπώς το αποτέλεσμα προκύπτει από γινόμενο αριθμών παρά πολύ κοντά στο μηδέν, το οποίο για μεγάλο όγκο δεδομένων οδηγεί αναπόφευκτα σε αδυναμία υπολογισμού και μηδενισμό ολοκλήρου του γινομένου (underflow).

Η δεύτερη πρόκληση προέρχεται από τον τύπο με τον όποιο υπολογίζεται η συνάρτηση πυκνότητας πιθανότητας στη Γκαουσιανή κατανομή.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{4.2}$$

Όπως παρατηρούμε θα πρέπει να υλοποιήσουμε αλγοριθμικά μια ιδιαίτερα απαιτητική παράσταση, η όποια προκαλεί μεγάλη καθυστέρηση κατά τον υπολογισμό της.

Η τρίτη και τελευταία πρόκληση είναι η αξιοποίηση της πολλαπλών πράξεων που θα υπολογίσει το υλικό προκειμένου να γίνουν ταυτόχρονα, χρησιμοποιώντας τον μέγιστο αριθμό DSP μπλοκ και LUT.

Αντιμετώπιση

Όπως γνωρίζουμε ήδη, από την μελέτη του Gaussian Naive Bayes δεν αρκεί μόνο ο υπολογισμός της συνάρτησης πυκνότητας πιθανότητας, αλλά απαιτείται και ο υπολογισμός του γινομένου των παραπάνω αποτελεσμάτων καθώς και η διαίρεση τους με τον όρο που υπάρχει στον παρανομαστή

προκειμένου να υπολογίσουμε την κλάση που συγκεντρώνει την μεγαλύτερη πιθανότητα (3.4). Έτσι λοιπόν εμφανίζεται ένα σημαντικό πρόβλημα το όποιο παρατηρείται από τις πρώτες κιόλας προσπάθειες πρόβλεψης στο συγκεκριμένο dataset ("MNIST"). Οι συνεχείς πολλαπλασιασμοί πολύ μικρών αριθμών σχεδόν κοντά στο 0 κάνουν το αποτέλεσμα να μηδενίζει υπολογίζοντας την πιθανότητα για όλες τις κλάσεις ίδια και ίση με 0. Εδώ λοιπόν έρχεται μια τροποποίηση στον αλγόριθμο μας προκειμένου να ξεφύγουμε από το φαινόμενο του underflow. Η λύση στο συγκεκριμένο πρόβλημα βασίζεται στην ιδέα πως δεν μας ενδιαφέρει ο ακριβής υπολογισμός της πιθανότητας κάθε κλάσης αλλά παρά μόνο η εύρεση της κλάσης με την μεγαλύτερη πιθανότητα. Συνεπώς μπορούμε πρώτον να απαλλαγούμε από τον παρανομαστή δεδομένου ότι είναι ο ίδιος για όλες τις κλάσεις και δεν αποτελεί ρυθμιστικό παράγοντα και δεύτερον να λογαριθμίσουμε τον αριθμητή ώστε να μετατρέψουμε τα γινόμενα σε αθροίσματα. Ενώ αρχικά υπολογίζαμε τον αριθμητή με αυτόν τον τρόπο:

$$y = p(C_k)\prod_{i=1}^{n} p(x_i \mid C_k) \tag{4.3}$$

Όπου,

$$p(x_i \mid C_k) = \frac{1}{\sqrt{2\pi\sigma_{C_k}^2}} e^{-\frac{(x_i-\mu_{C_k})^2}{2\sigma_{C_k}^2}} \tag{4.4}$$

Πλέον χρησιμοποιώντας τον φυσικό λογάριθμο $ln()$, και στα δύο μέλη, και δεδομένου ότι η λογαριθμική συνάρτηση είναι $1-1$ συνάρτηση, το μεγαλύτερο από τα $ln(y)$ θα είναι και αυτό που συγκεντρώνει την μεγαλύτερη πιθανότητα. Άρα πλέον καταλήγουμε στον εξής τύπο:

$$\begin{aligned} ln(y) &= ln(p(C_k)\prod_{i=1}^{n} p(x_i \mid C_k)) \\ &= ln(p(C_k) + ln(\prod_{i=1}^{n} p(x_i \mid C_k) \\ &= ln(p(C_k) + ln(\prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma_{C_k}^2}} e^{-\frac{(x_i-\mu_{C_k})^2}{2\sigma_{C_k}^2}}) \\ &= ln(p(C_k) + \sum ln(\frac{1}{\sqrt{2\pi\sigma_{C_k}^2}}) - \sum \frac{(x_i-\mu_{C_k})^2}{2\sigma_{C_k}^2} \end{aligned} \tag{4.5}$$

Με την συγκεκριμένη τροποποίηση αποφεύγουμε την ύπαρξη underflow στους υπολογισμούς μας μετατρέποντας έναν συσσωρευτή γινομένου σε έναν συσσωρευτή αθροίσματος χωρίς αυτό να επηρεάσει την ακρίβεια της συνάρτησης πρόβλεψης.

Listing 4.17: Συσσωρευτής - Αρχική Υλοποίηση

```
numerator *= 1 / sqrt(d_Pi * variances[i*N_feat + j]))*exp((-1*(data[j]
    - mean[i*N_feat + j])*(data[j] - mean[i*N_feat + j])) / (2 *
    variances[i*N_feat + j]));
```

Listing 4.18: Συσσωρευτής - Τροποποιημένη Υλοποιηση

```
numerator += log(1 / sqrt(d_Pi * variances[i*N_feat + j])))+((-1*(data[
    j] - mean[i*N_feat + j])*(data[j] - mean[i*N_feat + j])) / (2 *
    variances[i*N_feat + j]));
```

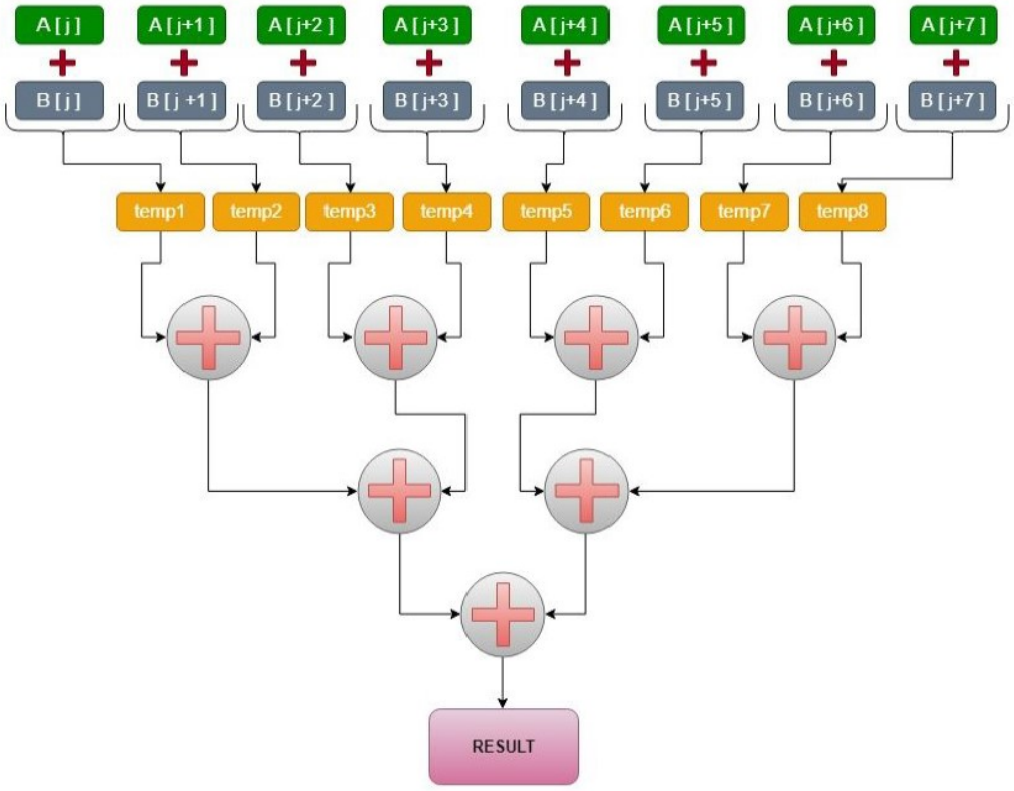
Η αντιμετώπιση της δεύτερης πρόκλησης βασίζεται στην αξιοποίηση όσο το δυνατόν περισσότερων πόρων του FPGA, παρεμβαίνοντας στον τρόπο με τον οποίο χρησιμοποιούνται οι DSPs πυρήνες. Επιδιώκουμε την παράλληλη χρησιμοποίησή τους αντί να αναμένουμε την επίλυση της παράστασης σειριακά. Έτσι λοιπόν διαμελίζουμε την συγκεκριμένη απαιτητική υπολογιστική παράσταση στις επιμέρους πράξεις που την αποτελούν αναγκάζοντας το εργαλείο να επιστρατεύσει μεγαλύτερο αριθμό DSPs και να επιτύχουμε αρκετά υψηλότερη επίδοση.

Listing 4.19: Διαμελισμός Συνάρτησης Πυκνότητας Πιθανότητας

```
for (j = 0; j < N_feat; j++){
#pragma HLS pipeline II=1
    if (lvar[i][j] > threshold){
        temp[0] = ldata[j] - lmean[i][j];
        temp[1] = temp[0] * temp[0];
        temp[2] = (-2) * lvar[i][j];
        temp[3] = temp[1] / temp[2];
        A[j] = temp[3];
        temp[4] = d_Pi * temp[2]/(-2);
        temp[5] = sqrt(temp[4]);
        B[j] =  log(1/temp[5]);
    }
}
for (j = 0; j < N_feat; j++){
#pragma HLS pipeline II=1
    numerator += A[j] + B[j];
}
```

Για την αντιμετώπιση της τρίτης πρόκλησης μπορεί να καταφέραμε να ξεφύγουμε προηγουμένως απ΄ το underflow αλλά ακόμα έχουμε την υλοποίηση ενός συσσωρευτή αθροίσματος. Παρατηρούμε λοιπόν ότι στην περίπτωση που περιμένουμε όλες τις προσθέσεις να εκτελεστούν σειριακά προσθέτουμε έναν παράγοντα $O(n)$ στην χρονική πολυπλοκότητα. Μπορούμε να ελαχιστοποιήσουμε αυτήν την καθυστέρηση αν αντικαταστήσουμε τους αθροιστές εν σειρά με μια πιο πολύπλοκη δομή αθροιστών η οποία μπορεί να εκτελέσει τα αθροίσματα με χρονική πολυπλοκότητα $O(log(n))$. Αντί να συσσωρεύουμε σειριακά κάθε αποτέλεσμα από τα επιμέρους αθροίσματα που έχουμε κατατμήσει παραπάνω στον ίδιο καταχωρητή με χρήση ενός αθροιστή, έχουμε την δυνατότητα να χρησιμοποιήσουμε πολλαπλούς αθροιστές διατεταγμένους σε δενδρική δομή. Αν οι πολλαπλασιασμοί που πρέπει να εκτελεστούν είναι n τότε το ύψος του δέντρου των αθροιστών θα είναι $O(log(n))$. Σε κάθε επίπεδο γίνεται χρήση ενδιάμεσων καταχωρητών για την αποθήκευση των μερικών αθροισμάτων με αποτέλεσμα στην κορυφή του δέντρου να υπολογίζεται το τελικό αποτέλεσμα. Το σύνολο της δομής των αθροιστών και των καταχωρητών απεικονίζεται στο σχήμα (4.6).

Σε αυτό το σημείο συνεπώς έχουμε να προσθέσουμε δυο πίνακες 784 στοιχείων ο καθένας, που οι τιμές που εμπεριέχουν αποτελούν τους όρους του αριθμητή. Μεσώ της δενδρικής αυτής δομής αθροίζουμε τους 2 πίνακες ανά οκτάδες μεταξύ τους δημιουργώντας ουσιαστικά $784/8 = 98$ δενδρικούς αθροιστές, αρχικοποιώντας τον συσσωρευτή με τον λογάριθμο της συχνότητας εμφάνισης κάθε κλάσης. Ωστόσο, η υλοποίηση 98 παραλλήλων δέντρων απαιτεί πολλούς πόρους υλικού και πολύπλοκη κατανομή της μνήμης. Έτσι, το εργαλείο HLS, μετά την ανάλυση του κώδικα της δομής του δέντρου, καθώς και των οδηγιών μας, καταφέρνει να υλοποιήσει μια pipelined έκδοση αυτού του



Σχήμα 4.6: Tree Adder

Σημ: Αντί για σειριακή άθροιση με χρήση ενός αθροιστή και ενός καταχωρητή, χρησιμοποιούμε πολλαπλούς καταχωρητές και αθροιστές σε παράλληλη δενδρική δομή με στόχο την όσο το δυνατόν περισσότερη παραλληλοποίηση των υπολογισμών.

δέντρου, καταφέρνοντας να δέχεται ένα νέο δεντρό ως είσοδο κάθε πέντε κύκλους ρολογιού και την ίδια στιγμή έχοντας επιλέξει να κάνουμε unroll με παράγοντα ίσο με 2 (factor = 2) τον κύριο βρόγχο που εμπεριέχει την επανάληψη που υλοποιεί το λογαριθμικό δεντρό μπορούμε να υπολογίζουμε σε ένα κύκλο 2 δεντρά παράλληλα. Το πλήρες ξετύλιγμα του κυρίου βρόχου δημιουργεί αδυναμία υλοποίησης του από το εργαλείο λόγω περιορισμένων πόρων άλλα δεν εμφανίζει και κάποια ιδιαίτερη βελτίωση στην απόδοση.

Listing 4.20: Δενδρική Υλοποίηση Αθροιστή

```
for (j = 0; j < N_feat; j += 8){
#pragma HLS pipeline II=5
        float temp1 = A[j] + B[j];
        float temp2 = A[j + 1] + B[j + 1];
        float temp3 = A[j + 2] + B[j + 2];
        float temp4 = A[j + 3] + B[j + 3];
        float temp5 = A[j + 4] + B[j + 4];
        float temp6 = A[j + 5] + B[j + 5];
        float temp7 = A[j + 6] + B[j + 6];
        float temp8 = A[j + 7] + B[j + 7];

        float level_1_1 = temp1 + temp2;
        float level_1_2 = temp3 + temp4;
        float level_1_3 = temp5 + temp6;
        float level_1_4 = temp7 + temp8;

        float level_2_1 = level_1_1 + level_1_2;
        #pragma HLS resource variable=level_2_1 core=FAddSub_nodsp

        float level_2_2 = level_1_3 + level_1_4;
        #pragma HLS resource variable=level_2_2 core=FAddSub_nodsp

        float level_3 = level_2_1 + level_2_2;
        #pragma HLS resource variable=level_3 core=FAddSub_nodsp

        numerator += level_3;
        #pragma HLS resource variable=numerator core=FAddSub_nodsp
}

if (numerator > max_likelihood){
    max_likelihood = numerator;
    prediction = i;
}
```

Φυσικά για να μπορέσει να πραγματοποιηθεί αποδοτικά η παραπάνω υλοποίηση έχουμε διαμελίσει και σε αυτήν την περίπτωση τους πινάκες που συμμετέχουν στους υπολογισμούς προκείμενου να ξεπεραστεί το όριο θύρας (το πολύ δύο θύρες ανάγνωσης/εγγραφής για κάθε on-chip μνήμη στο FPGA). Σε οποίες περιπτώσεις ήταν βέβαια εφικτό έγινε πλήρης διαμελισμός, ενώ σε άλλες έγινε με συγκεκριμένο μοτίβο το οποίο δημιουργεί όσο το δυνατόν λιγότερες συμφορήσεις στην μνήμη (bottlenecks).

Listing 4.21: HLS Pragmas in Prediction Function

```
#pragma HLS array_partition variable=A cyclic factor = 56
#pragma HLS array_partition variable=B cyclic factor = 56
#pragma HLS array_partition variable=temp complete
```

```
#pragma HLS array_partition variable=lvar cyclic factor = 56 dim=2
#pragma HLS array_partition variable=lpr complete
#pragma HLS array_partition variable=lmean cyclic factor = 56 dim=2
#pragma HLS array_partition variable=ldata cyclic factor = 56
```

Τέλος παρατηρούμε την χρονοδρομολόγηση που έγινε μεταξύ των σταδίων του υπολογισμού. Η υλοποίηση μας αποτελείται ουσιαστικά από 2 βρόχους. Κατά την πλήρη εκτέλεση του πρώτου βρόχου αντιγράφουμε τα δεδομένα στις τοπικές BRAMs, που θα χρησιμοποιηθούν για τους υπολογισμούς του δεύτερου βρόχου, προκείμενου να μπορούμε να επιτελέσουμε πολλαπλές προσβάσεις σε αυτές.

Listing 4.22: Αντιγραφή Δεδομένων σε BRAMs

```
for ( i = 0; i < N_class; i++ ){
#pragma HLS pipeline II=1
        lpr[i] = priors[i];
}
for ( i = 0; i < N_class; i++ ){
        for ( j = 0; j < N_feat; j++ ){
        #pragma HLS pipeline II=1
                lvar[i][j] = variances[i*N_feat + j];
                lmean[i][j] = means[i*N_feat + j];
                ldata[i] = data[i];
        }
}
```

Ενώ κατά τον δεύτερο και κύριο βρόχο υπολογίζουμε την πιθανότητα κάθε κλάσης σε 2 στάδια. Στο πρώτο στάδιο διαμελίζουμε σε ανεξάρτητους όρους, το τύπο υπολογισμού της συνάρτησης πιθανότητα, αποθηκεύοντας τους σε 2 πίνακες και στο δεύτερο υλοποιούμε τον συσσωρευτή σε δενδρική δομή. Έτσι πετυχαίνουμε υψηλότερη απόδοση και μεγαλύτερη παραλληλοποίηση στην χρήση των υπολογιστικών πόρων, αποφεύγοντας εξαρτήσεις που θα καθυστερούσαν την υλοποίηση μας.

Έτσι επιτύχαμε ως και 16.8 φορές γρηγορότερη εκτέλεση του αλγορίθμου της Εκπαίδευσης και ως και 14 φορές γρηγορότερη εκτέλεση του αλγορίθμου της Πρόβλεψης στο υλικό.

4.4.4 Απόδοση 3ης Βελτιστοποίησης

Training(2000 data lines):

- Accelerated Hardware: 10.700.000 cycles
- Software-Only: 182.000.000 cycles

Speedup = **x***16.8*

Prediction(Per line of data):

- Accelerated Hardware: 162.000 cycles
- Software-Only: 2.220.000 cycles

$$Speedup = \mathbf{x}14$$

4.5 Χαρακτηριστικά Υλοποίησης

4.5.1 Resource Utilization

Συνάρτηση Εκπαίδευσης

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	197	220	89,55
BRAM	56	140	40
LUT	37929	53200	71,3
FF	29271	106400	27,51

Σχήμα 4.7: Training Utilization

Συνάρτηση Πρόβλεψης

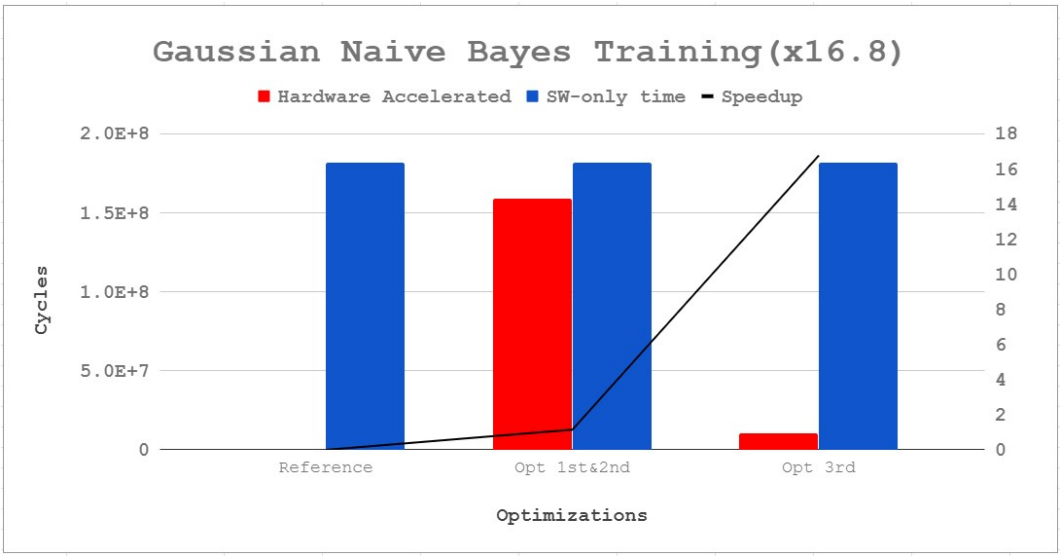
Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	96	220	43,64
BRAM	56	140	40
LUT	28524	53200	53,62
FF	25251	106400	23,73

Σχήμα 4.8: Prediction Utilization

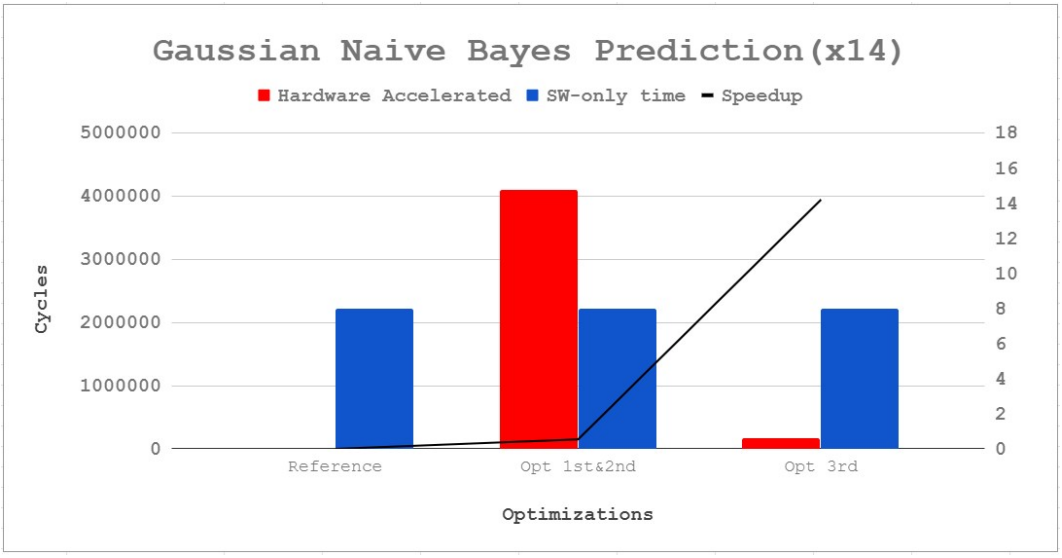
4.5.2 Γραφήματα Απόδοσης Βελτιστοποιήσεων

Συνάρτηση Εκπαίδευσης



Σχήμα 4.9: Βελτιστοποιήσεις Εκπαίδευσης

Συνάρτηση Πρόβλεψης



Σχήμα 4.10: Βελτιστοποιήσεις Πρόβλεψης

Κεφάλαιο 5

Pynq: Ενσωμάτωση με Python

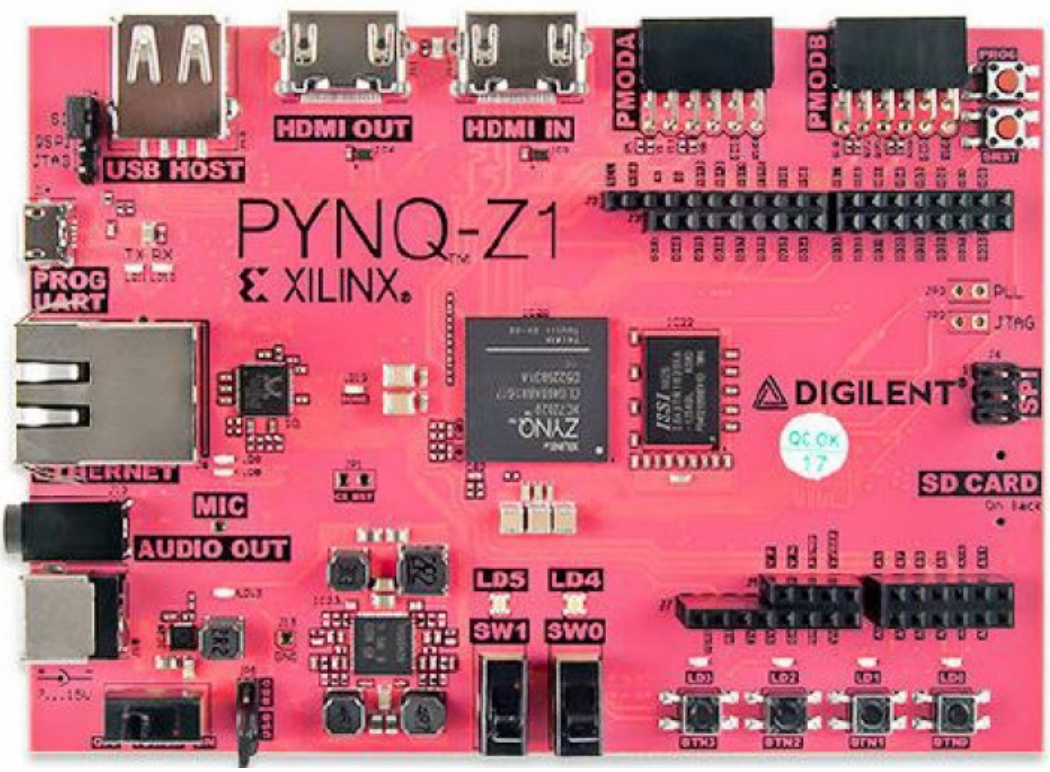
5.1 Εισαγωγή

Μέτα την επιτάχυνση του Gaussian Naive Bayes αλγορίθμου μέσω του SDSoC περιβάλλοντος υλοποιημένο σε C επιχειρήσαμε την ενσωμάτωση του επιταχυντή σε περιβάλλον Python. Δεδομένου ότι η Python είναι πλέον μια από τις πιο διαδεδομένες γλώσσες για Machine Learning και εφαρμογές τεχνητής νοημοσύνης καθώς περιέχει πολύ ισχυρές εντολές και βιβλιοθήκες, αυτό το επόμενο βήμα μας δίνει την δυνατότητα να δημιουργήσουμε μια εφαρμογή που μπορεί να ενσωματωθεί σε ένα σύγχρονο κέντρο δεδομένων και να αποτελέσει μια αποδοτική εναλλακτική τόσο σε απόδοση όσο και σε ενέργεια. Η μετατόπιση του υψηλού υπολογιστικού φορτίου από την CPU ενός συστήματος σε ένα ή περισσότερα FPGA μπορεί να επιφέρει σημαντική μείωση στο χρόνο εκτέλεσης των εφαρμογών. Το βασικό πρόβλημα που προκύπτει βεβαία στην συγκεκριμένη ιδέα είναι πως μέχρι στιγμής η Xilinx δεν έχει κάποιο παρόμοιο εργαλείο με το SDSoC που να είναι συμβατό με Python. Συνεπώς η ενσωμάτωση ενός υλικού επιταχυντή του οποίου η διεπαφή απαιτεί την γνώση των φυσικών διευθύνσεων των DMAs καθώς και πολλές ακόμα πληροφορίες που αφορούν το υλικό στο οποίο τρέχει, πάνω σε μια γλώσσα η οποία είναι απομακρυσμένη από το υλικό, όπως είναι η Python, είναι μια εργασία η οποία έπρεπε να γίνει χειροκίνητα. Το SDSoC ως τώρα, ήταν υπεύθυνο για την κατασκευή των διεπαφών υλικού και λογισμικού μεσώ των οποίων γίνεται η επικοινωνία μεταξύ του PS (Processing System) και του PL (Programmable Logic), βάση του χώρου διευθύνσεων που έχουν αποδοθεί από την κατασκευάστρια εταιρία (Xilinx). Οι λεπτομέρειες όμως για τον τρόπο υλοποίησης των παραπάνω διεπαφών είναι κάτι το οποίο αποκρύπτει το SDSoC, καθότι λειτουργεί με πλήρως αυτοματοποιημένο τρόπο, ενώ επίσης έχει εφαρμογή μόνο σε κώδικα γραμμένο σε C/C++.

Έτσι λοιπόν μπορεί η Xilinx να μην παρέχει ένα εργαλείο σαν το SDSoC συμβατό με Python ωστόσο παρέχει την πλακέτα Pynq-Z1 για την εξυπηρέτηση αυτού του σκοπού. Η πλακέτα Pynq-Z1[14] είναι κατάλληλη για επιτάχυνση εφαρμογών σε Python διότι παρέχεται για αυτήν ένα Image ανοιχτού κωδικά, το οποίο ενσωματώνει την Python και χρήσιμες βιβλιοθήκες της. Εδώ πρέπει να σημειώσουμε ότι παρότι το Image της πλακέτας περιεχέι ένα API χαμηλού επιπέδου, μέσω του οποίου ο κώδικας Python που προορίζεται να τρέξει στον ARM μπορεί να αλληλεπιδράσει με τον πυρήνα FPGA που υπάρχει στο υλικό, εμείς ακολουθήσαμε έναν άλλο τρόπο για να επικοινωνήσουμε και να ενσωματώσουμε το υλικό στην υλοποίηση μας. Κάνοντας reverse engineering στο SDSoC ώστε να βρούμε τον τρόπο με τον οποίο υλοποιεί την επικοινωνία του πυρήνα με την C και δημιουργεί το απαραίτητο bitstream, οδηγηθήκαμε σε ένα άλλο εργαλείο της Xilinx, το Vivado.

Τα βήματα που ακολουθήσαμε για να κάνουμε την ενσωμάτωση ήταν τα ακόλουθα:

- Συγγραφή σε Python τα κομμάτια του αλγορίθμου που προορίζονται για την CPU.
- Απομόνωση του IP του πυρήνα από το περιβάλλον του SDSoC και δημιουργία νέου bitstream συμβατού με το PYNQ-Z1
- Δημιουργία οδηγών που επιτρέπουν την αλληλεπίδραση με τον πυρήνα, και ενσωμάτωση τους στον κώδικα που γράψαμε στο πρώτο βήμα.



Σχήμα 5.1: Pynq-Z1

5.2 Υλοποίηση του GNB με χρήση Pynq-Z1

5.2.1 Βήμα 1ο - Δημιουργία driver και Overlay

Κατά το πρώτο στάδιο υλοποίησης του στόχου μας αντιλαμβανόμαστε πως πρωταρχικός στόχος είναι η εδραίωση επικοινωνίας μεταξύ του πυρήνα του FPGA και της CPU στην οποία τρέχει ο πηγαίος κώδικας της Python. Δεδομένου λοιπόν πως αποφασίσαμε να μην κάνουμε χρήση του ενσωματωμένου στο Pynq, Python API, έπρεπε να βρούμε έναν άλλο τρόπο για να επιτύχουμε την επιδιωκόμενη επικοινωνία. Έτσι ξεκινώντας από το SDSoC παρατηρήσαμε πως το εργαλείο μας δίνει την δυνατότητα πέρα απ την συγγραφή μιας ολοκληρωμένης εφαρμογής, στην οποία ενσωματώνουμε τον επιταχυντή υλικού και την δημιουργία μιας διαμοιραζόμενης βιβλιοθήκης που εμπεριέχει την συνάρτηση υλικού που έχουμε ήδη υλοποιήσει. Το εργαλείο μετά από αυτήν την διαδικασία παράγει μια precompiled library και άλλα αρχεία τα οποία εμπεριέχουν πληροφορίες για τις διευθύνσεις των DMAs. Το εγχείρημα μας όμως ακόμα δεν είναι ολοκληρωμένο καθότι παρουσιάζονται δυο βασικά προβλήματα που εμφανίζονται. Πρώτον το bitstream που δημιουργεί το SDSoC κατά την δημιουργία της διαμοιραζόμενης βιβλιοθήκης (Shared Library) δεν είναι συμβατό με το Pynq-Z1 αφού έχει δημιουργηθεί για Zedboard και δεύτερον

ακόμα και αν η βιβλιοθήκη που παρήγαγε το εργαλείο διέθετε όλες τις απαιτούμενες πληροφορίες για την επικοινωνία του λογισμικού με το υλικό, υπάρχει κάτι το οποίο δεν παράγει το SDSoC και το οποίο είναι πολύ σημαντικό για την υλοποίηση μας και αυτό δεν είναι άλλο από την C υλοποίηση της βιβλιοθήκης Overlay (που υπάρχει σε κώδικα Python, στο Python API). Συνεπώς η βιβλιοθήκη που δημιούργησε το SDSoC είναι ημιτελής και το Bitstream για άλλη συσκευή. Προκειμένου να διορθώσουμε την υπάρχουσα κατάσταση κάναμε τις εξής ενέργειες.

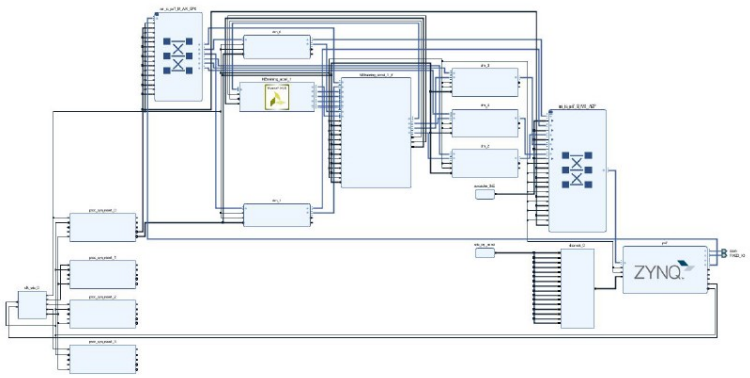
Χρησιμοποιώντας ως βάση το project που έχουμε ήδη δημιουργήσει με το SDSoC (δεδομένου ότι το Zedboard και το Pynq έχουν το ίδιο ολοκληρωμένο, κατ' έκταση και τους ίδιους ακριβώς πόρους) και αναζητούμε το αρχείο του project που το SDSoC στέλνει στο Vivado προκειμένου να δημιουργήσει το IP. Στην συνεχεία μέσα στο Vivado Project περιέχεται ο πυρήνας που κατασκευάσαμε, ωστόσο ο συγκεκριμένος έχει δημιουργηθεί για την πλακέτα Zedboard, συνεπώς αυτό που πρεπέι μα κάνομε είναι να ανοίξουμε το IP μας στο Vivado και να φτιάξουμε το bitstream του IP για το Pynq (device xc7z020-1clg400c). Αφού κατασκευάσουμε το bitstream για το PL του Pynq το κάνουμε export μαζί με το αντίστοιχο αρχείο Tcl που περιέχει τις φυσικές διευθύνσεις των DMAs και των υπολοίπων I/O που απαιτούνται για την επικοινωνία με τον υλικό επιταχυντή. Αυτήν την διαδικασία ακολουθούμε ουσιαστικά και για τα τους δυο πυρήνες, δηλαδή για το κομμάτι της εκπαίδευσης και για το κομμάτι της πρόβλεψης.

Στα Tcl αρχεία που παράγονται από το Vivado μπορούμε να δούμε τις ακριβείς φυσικές διευθύνσεις στις οποίες έχουν χαρτογραφηθεί οι DMAs και οι FIFO ACCELERATOR ADAPTER στην ακόλουθη μορφή.

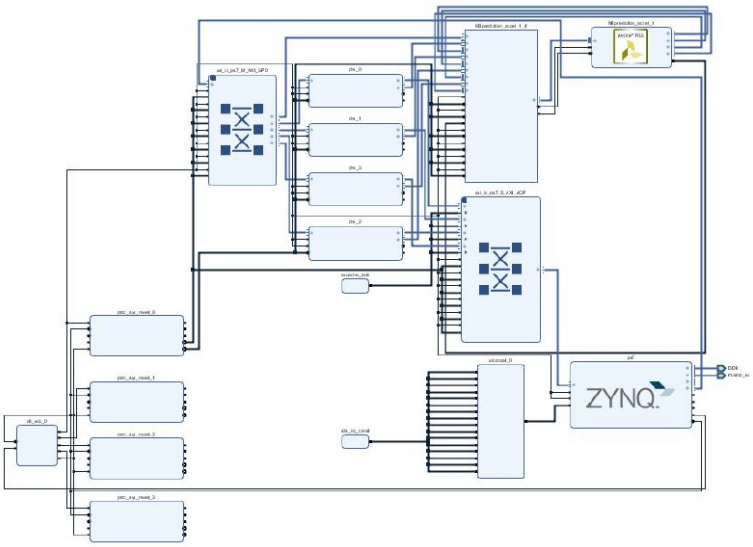
Listing 5.1: Κομμάτι Tcl Αρχείου

```
create_bd_addr_seg -range 0x00010000 -offset 0x40400000
[get_bd_addr_spaces ps7/Data] [get_bd_addr_segs dm_0/
S_AXI_LITE/Reg] SEG_dm_0_Reg

create_bd_addr_seg -range 0x00010000 -offset 0x43C00000
[get_bd_addr_spaces ps7/Data] [get_bd_addr_segs NBtraining_accel_1_if/
S_AXI/reg0] SEG_NBtraining_accel_1_if
```



Σχήμα 5.2: training.bd



Σχήμα 5.3: prediction.bd

Συνεπώς σε αυτό το στάδιο έχουμε καταφέρει να απομονώσουμε σε τέσσερα αρχεία, (Training.bit - Training.tcl & Prediction.bit - Prediction.tcl) τους πυρήνες και όλες τις απαραίτητες πληροφορίες για αυτούς αντίστοιχα.

Σε αυτό το σημείο έχοντας πλέον το κατάλληλα αρχεία .bit και .tcl επιστρέφουμε στην διόρθωση του προβλήματος της ημιτελούς βιβλιοθήκης που δημιουργεί το SDSoC. Αντιγράφοντας λοιπόν τον τρόπο με τον όποιο το εργαλείο δημιουργεί την απαιτουμένη precompiled βιβλιοθήκη κάνουμε χειροκίνητα την συγκεκριμένη διαδικασία, μέσω ενός makefile, ενσωματώνοντας στην βιβλιοθήκη μας και την C υλοποίηση του Overlay που απαιτείται για την φόρτωση του bitstream στο FPGA του Pynq. Επίσης θα πρέπει να αναφέρουμε πως η μεταγλώττιση των precompiled βιβλιοθηκών (NBtraining.so και NBprediction.so) που αποτελούν ουσιαστικά τους drivers για την επικοινωνία με τους πυρήνες που έχουμε δημιουργήσει, έγινε από τον μεταγλωττιστή που παρέχει το Image του Pynq και όχι από τους μεταγλωττιστές που χρησιμοποιεί το SDSoC. Έτσι είναι πλέον επιτυχής η επικοινωνία με το FPGA κατά την εκτέλεση του Python κώδικα από τον ARM του Pynq.

5.2.2 Βήμα 2ο - Χρήση CFFI και δημιουργία Python API

Σε αυτό το στάδιο προκειμένου να κάνουμε κλήση των C συναρτήσεων που περιέχονται στις βιβλιοθήκες που δημιουργήσαμε στο προηγούμενο βήμα, χρησιμοποιούμε κατά την συγγραφή του κώδικα Python την βιβλιοθήκη cffi. Η βιβλιοθήκη cffi μας επιτρέπει να αντικαταστήσουμε τις κλήσεις των C συναρτήσεων που βρίσκονται εντός των NBtraining.so και NBprediction.so με κλήσεις Python επιτρέποντας στην Python να καλέσει ή να εκτελέσει οποιοδήποτε κώδικα σε C είτε αυτός είναι σε source μορφή είτε είναι σε binary. Αφού λοιπόν καθορίσαμε τον τρόπο με τον οποίο θα εκτελέσουμε τις C συναρτήσεις που έχουμε κάνει wrap θα πρέπει να ρυθμίσουμε και το τρόπο με τον όποιο θα στείλουμε τα δεδομένα στις C συναρτήσεις. Το βασικό πρόβλημά εντοπίζεται στις διαφορετικές δομές δεδομένων που αναγνωρίζει κάθε γλώσσα. Συνεπώς παρουσιάζεται αδυναμία μεταφοράς των δομών δεδομένων της Python σε μορφή αναγνωρίσιμη από την C καθώς και αδυναμία δέσμευσης συνεχόμενης φυσικής μνήμης (όπως απαιτείται από τον πυρήνα του επιταχυντή). Την λύση σε αυτό το χάσμα δίνει και πάλι η βιβλιοθήκη cffi η οποία δημιουργεί buffers συνεχόμενης μνήμης καλώντας την συνάρτηση sds_alloc() και ταυτόχρονα μετατρέπει τα Python δεδομένα σε μορφή αναγνωρίσιμη από

την C τοποθετώντας τα στους buffers συνεχόμενης φυσικής μνήμης που προηγουμένως δημιούργησε. Στην συνέχεια για να αποστείλουμε τα buffers που μόλις δημιουργήσαμε στον επιταχυντή υλικού μας, χρησιμοποιούμε την διεύθυνση που έχουνε στην μνήμη και όχι ολόκληρη την δομή των δεδομένων. Για να αυτοματοποιήσουμε λοιπόν την συγκεκριμένη διαδικασία δημιουργήσαμε ένα Python API με σκοπό να μπορεί να γενικευτεί η χρήση του για οποιαδήποτε εφαρμογή.

Αναπτύξαμε έτσι μια mapper συνάρτηση, η οποία δεσμεύει και γεμίζει buffers συνεχούς μνήμης με τα δεδομένα εκπαίδευσης, προκειμένου να παραμείνουν εκεί για το υπόλοιπο της εκτέλεσης της εφαρμογής ενώ ταυτόχρονα ταξινομεί και τα δεδομένα με βάση την κλάση τους. Στην συνεχεία καλούμε μια άλλη mapper συνάρτηση η όποια στέλνει τους buffers που δημιουργήσαμε προηγουμένως σε κάθε επανάληψη. Οι buffers περιέχουν τις απαραίτητες πληροφορίες για τα δεδομένα που θα αποστείλουμε στον πυρήνα, όπως είναι η διεύθυνση συνεχόμενης μνήμης, που δεσμεύσαμε προηγουμένως, καθώς και πληροφορίες για το πλήθος των δεδομένων εκπαίδευσης, όταν μιλάμε για την συνάρτηση εκπαίδευσης. Τέλος δημιουργήσαμε μια συνάρτηση η όποια απελευθερώνει την μνήμη μετά την χρήση των buffers. Το Python API συνεπώς αποτελείται από τρεις βασικές κλήσεις:

- **cma (contiguous memory allocate):** Αυτή η κλήση χρησιμοποιείται για τη δημιουργία των buffers και την δέσμευση της συνεχούς μνήμης. Επίσης σε αυτό το σημείο φορτώνεται το Overlay και τα δεδομένα εκπαίδευσης εγγράφονται στους αντίστοιχους buffers. Χρησιμοποιώντας το cma, δημιουργείται και διατηρείται ένα νέο tumple, το οποίο περιέχει μόνο πληροφορίες σχετικά με αυτούς τους buffers (διευθύνσεις μνήμης, μεγέθη κ.λ.π).
- **NBtraining/NBprediction_kernel_accel:** Αυτή η κλήση χρησιμοποιείται για την εκτέλεση των hardware συναρτήσεων που έχουν υλοποιηθεί και επιταχυνθεί στο υλικό στέλνοντας προς επεξεργασία τους buffers που έχουμε δημιουργήσει κατά την κλήση της cma.
- **cmf (contiguous memory free):** Αυτή η κλήση χρησιμοποιείται ρητά για την απελευθέρωση όλων των δεσμευμένων buffers από τη μνήμη.

Δημιουργήσαμε λοιπόν ένα Python API ώστε να οδηγήσουμε το επιταχυντή υλικού το οποίο δεν είναι ανεξάρτητο της συγκεκριμένης εφαρμογής αλλά μπορεί με κάποιες παραμετροποιήσεις να κληθεί ως γενική βιβλιοθήκη για το PYNQ (Οι κώδικες βρίσκονται στο παράρτημα).

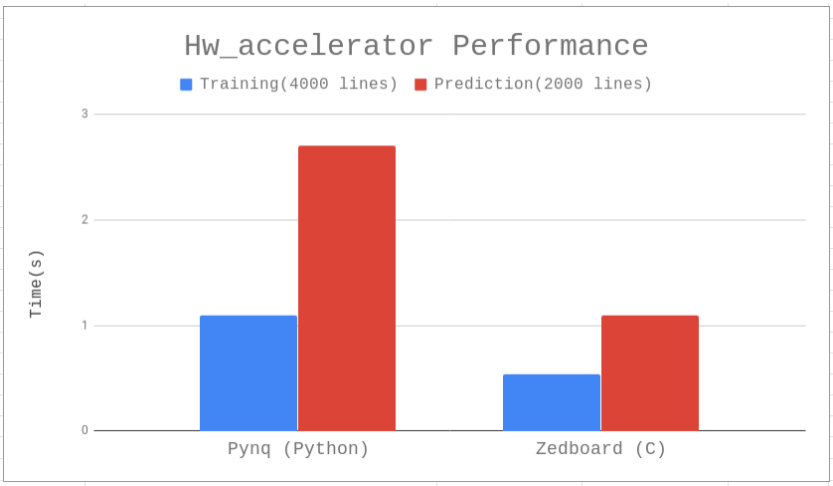
5.3 Επίδοση στο Pynq

Αφού πλέον έχει γίνει η ενσωμάτωση του πυρήνα στην Python μπορούμε να πάρουμε τις απαραίτητες μετρήσεις, συγκρίνοντας την C και την Python υλοποίηση. Παρατηρώντας τα αποτελέσματα των δυο υλοποιήσεων βλέπουμε αύξηση του χρόνου εκτέλεσης στην Python υλοποίηση γεγονός που οφείλεται στην φύση της προγραμματιστικής γλώσσας η όποια είναι δυναμική με ευελιξία στον τρόπο χρήση της. Αυτά τα στοιχεία την καθιστούν αργή αλλά ταυτόχρονα ιδιαίτερα λειτουργική για μηχανική μάθηση.

Συγκρίνοντας την Hardware accelerated υλοποίηση σε C σε σχέση με την Python παρατηρούμε προφανέστατα μια μείωση στην απόδοση η όποια οφείλεται στα μεγαλύτερα I/O overheads. Συγκεκριμένα βλέπουμε τις εξής μετρήσεις:

Πίνακας 5.1: Επίδοση Επιταχυντή σε Python & C

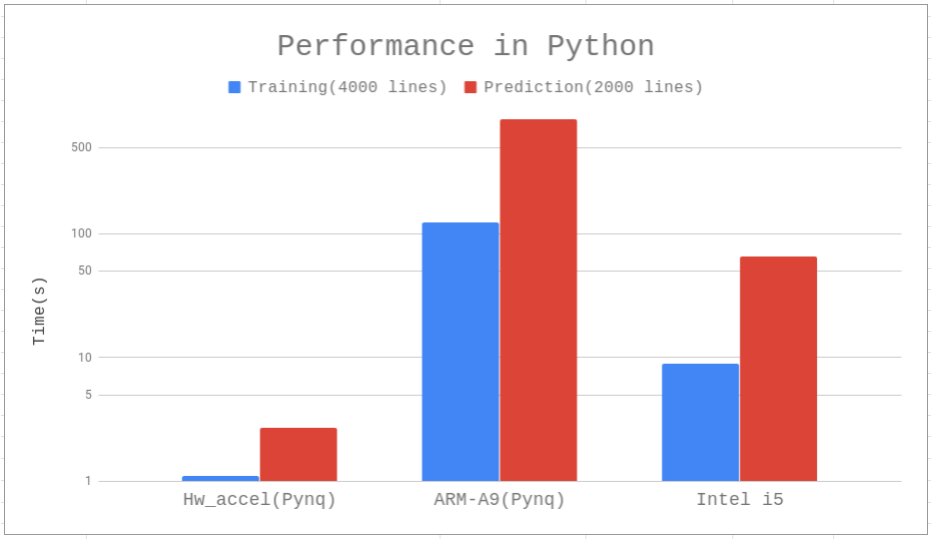
Hw_Accelerated	Training(4000 lines)	Prediction(2000 lines)
Pynq (Python)	1.1 sec	2.7 sec
Zedboard (C)	0.54 sec	1.1 sec



Συγκρίνοντας όμως αυτήν την φορά την Hardware accelerated υλοποίηση της Python με τον ARM και με μια CPU Intel i5 (4ης Γενιάς) παρατηρήσαμε εντυπωσιακά αποτελέσματα τα όποια μας ενθάρρυναν για το επόμενο βήμα της υλοποίησης μας που είναι η ενσωμάτωση του Spark στην εφαρμογή μας με σκοπό την δημιουργία ενός FPGA-cluster.

Πίνακας 5.2: Επίδοσεις σε περιβάλλον Python

Python Environment	Training(4000 lines)	Prediction(2000 lines)
HW_accelerated(Pynq)	1.1 sec	2.7 sec
ARM-A9	124 sec	850 sec
Intel i5	9 sec	66 sec



Κεφάλαιο 6

Ενσωμάτωση με Spark

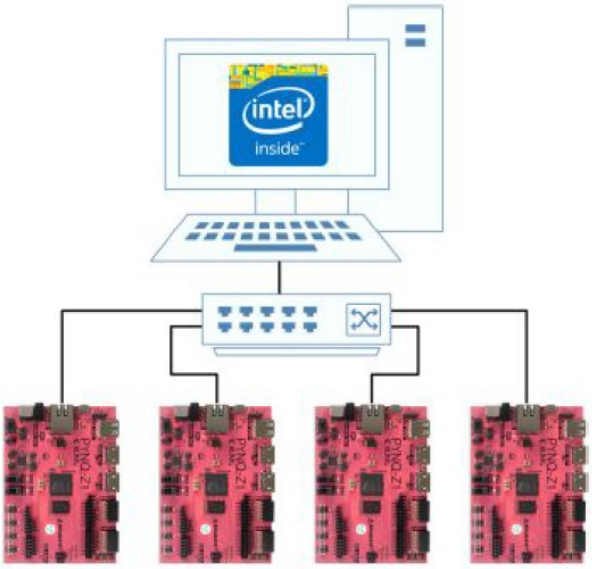
6.1 Εισαγωγή

Κατά την χρήση του Spark, ο αλγόριθμος Naive Bayes εμφανίζει εξαιτίας της φύσης του, δυνατότητες παραληλλοποίησης. Πιο συγκεκριμένα, κάθε μοντέλο πρόβλεψης μπορεί να υπολογιστεί σε διαφορετικό κόμβο της συστάδας, χρησιμοποιώντας ένα Spark **MapReduce** για κάθε επανάληψη του αλγορίθμου, ενώ κατά την πρόβλεψη χρησιμοποιούμε και πάλι τις συναρτήσεις **MapReduce** του Spark προκειμένου να υπολογίσουμε την μεγίστη πιθανότητα για κάθε γραμμή των ανεξάρτητων δεδομένων πρόβλεψης. Στην συνέχεια, βρίσκοντας το συνολικό άθροισμα σωστών και λάθος προβλέψεων μπορούμε να υπολογίσουμε την ακρίβεια των προβλέψεων μας. Έτσι λοιπόν μπορεί να γίνει ο υπολογισμός του μοντέλου πρόβλεψης και της μεγίστης εκ των υστέρων πιθανότητας MAP σε κάθε worker (κάνοντας χρήση της Προγραμματιζόμενης Λογικής).

Στα πλαίσια της θεωρητικής μελέτης της υλοποίησης ενός FPGA Cluster δημιουργήσαμε ένα single-node σύστημα χρησιμοποιώντας την stanadalone υλοποίηση του Spark δημιουργώντας μια οριζόντια συστάδα (που δεν έδινε την δυνατότητα παραληλλοποίησης των υπολογισμών) και κάναμε μετρήσεις. Σκοπός αυτής της προσπάθειας ήταν να παρουσιαστεί η μεθοδολογία μέσω της οποίας ένας χρήστης που διαθέτει το υλικό, να επεκτείνει το συγκεκριμένο σύστημα δημιουργώντας μια κατακόρυφη συστάδα, και από single-node να αξιοποιήσει μια πληθώρα από Pynq-Z1 στα όποια θα μπορεί να υπολογίσει παράλληλα το βαρύ υπολογιστικό κομμάτι ενός αλγορίθμου και εδώ συγκεκριμένα του αλγορίθμου του Gaussian Naive Bayes.

6.2 Υλοποίηση Spark-Pynq

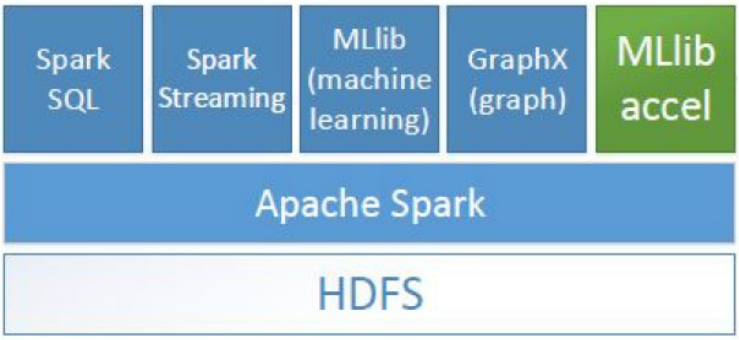
Σε αυτό το σημείο έπρεπε να κάνουμε κάποιες μετατροπές σε σχέση με τον κώδικα που έτρεχε κυρίως built-in Python εργαλεία κατά την εκτέλεση στο Pynq-Z1 καθώς πλέον οι βασικές map και reduce συναρτήσεις θα υλοποιηθούν από το Spark. Το βασικό πρόβλημα που έπρεπε να αντιμετωπίσουμε είναι η διατήρηση της φυσικά συνεχόμενης δεσμευμένης μνήμης που δημιουργούμε κατά την κλήση της συνάρτησης cma, του Python API. Η λύση που βρέθηκε είναι η ενσωμάτωση των διευθύνσεων μνήμης σε ένα νέο RDD το όποιο διατηρούμε (persist), προκειμένου οι διεργασίες που εκτελούνται αυτόματα από το Spark για την διαχείριση των RDD να μην επηρεάσουν τις διευθύνσεις των συνεχόμενων θέσεων που δημιουργήσαμε, μετά την επιστροφή των RDD από τις **MapReduce** συναρτήσεις. Στην συνέχεια, επειδή τα δεδομένα καθόλη την διάρκεια της εκπαίδευσης παραμένουν τα ιδία, είναι δηλαδή cached, καταφέραμε να υλοποιήσουμε ένα σχήμα που επιτρέπει την προσωρινή



Σχήμα 6.1: Συστάδα από Pynq-Z1

αποθήκευση του RDD σε συνεχή μνήμη, αποφεύγοντας τις μεταφορές εντός μνήμης κάθε φορά που ο επιταχυντής επικαλείται.

Στην συνεχεία θέλοντας να δημιουργήσουμε μια βιβλιοθήκη η όποια να ακολουθεί την λογική των συναρτήσεων μηχανικής μάθησης που έχουν ήδη υλοποιηθεί για Spark και εμπεριέχονται στην mllib βιβλιοθήκη, ενσωματώσαμε το API μας στην βιβλιοθήκη mllib_accel που ήδη εμπεριέχει την υλοποίηση για τη Λογιστική Παλινδρόμηση με τον αλγόριθμο Κατάβασης Πλαγιάς[15], ώστε να εκμεταλλευτούμε την Προγραμματιζόμενη Λογική που διατίθεται στους PYNQ workers.



Σχήμα 6.2: MLlib Accel

6.3 Επίδοση

Κατά την μέτρηση των επιδόσεων του single-node συστήματος που δημιουργήσαμε μέσω του Spark, συγκρίναμε εκ νέου το FPGA με έναν Intel i5 και αυτή την φορά παρατηρήσησαμε μεγάλη αύξηση στον χρόνο εκτέλεσης της εφαρμογής, γεγονός που οφείλεται στην χρήση του εργαλείου Spark. Η ετερογενής υλοποίηση που δημιουργήσαμε αυτή την φορά δεν είναι πιο αποδοτική σε σχέση

με την χρήση ενός τετραπύρηνου επεξεργαστή Intel i5 και αυτό είναι λογικό μιας και η αποδοτική της χρήση θα απαιτούσε την αξιοποίηση περισσότερων του ενός FPGA .

Η απόδοση όμως δεν αποτελεί αυτοσκοπό μιας και η υλοποίησή πραγματοποιήθηκε καθαρά για λόγους παρουσίασης της μεθοδολογίας χρήσης των FPGA σε ένα κατανεμημένο σύστημα.

Πίνακας 6.1: Επίδοσεις με Spark

Spark	Training(4000 lines)	Prediction(2000 lines)
HW_accelerated(Pynq)	35 sec	70 sec
Intel i5	9 sec	66 sec

Κεφάλαιο 7

Επίλογος

7.1 Συμπεράσματα

Στα πλαίσιο αυτής της διπλωματικής εργασίας παρατηρούμε πως οι επιταχυντές υλικού αποτελούν έναν εναλλακτικό τρόπο για αύξηση της απόδοσης των εφαρμογών Μηχανικής Μάθησης. Η δυνατότητα των FPGA για παράλληλη εκτέλεση πράξεων ξεφεύγοντας από τον σειριακό τρόπο εκτέλεση ενός κοινού επεξεργαστή είναι και ο λόγος για τον οποίο υπερτερούν έναντι των κλασικών τρόπων επεξεργασίας σε συγκεκριμένες εργασίες. Σε αυτό το έργο λοιπόν μελετήσαμε τον τρόπο με τον οποίο μπορεί να επιτευχθεί η δημιουργία ενός αποδοτικού επιταχυντή υλικού κάνοντας χρήση της Υψηλού Επιπέδου Σύνθεσης. Συγκεκριμένα η έρευνα μας εφαρμόστηκε πάνω στον αλγόριθμο του Gaussian Naive Bayes και βασίστηκε σε δυο μέρη. Στο πρώτο μέρος μετασχηματίσαμε δομικά τον κωδικά ώστε να αξιοποιήσουμε στο μέγιστο την δυνατότητα παραλληλοποίησης των υπολογιστικών τμημάτων του αλγορίθμου. Πιο συγκεκριμένα επιδιώξαμε την εφαρμογή μετασχηματισμών βρόχου στον κώδικα και την επιλογή των κατάλληλων οδηγιών HLS για την εκμετάλλευση του εγγενούς παραλληλισμού του αλγορίθμου. Το δεύτερο μέρος αποτελεί την ανάλυση και εύρεση της αποδοτικότερης διεπαφής επικοινωνίας μέσω των ντιρεκτίβων SDS. Τα αποτελέσματα αποδεικνύουν ότι η επιτάχυνση υλικού και συνεπώς η συν-σχεδίαση SW/HW είναι στην πραγματικότητα μια έγκυρη λύση όταν οι τεχνικές επιτάχυνσης του λογισμικού πληρούν τα όριά τους.

Επίσης αναπτύξαμε μια μεθοδολογία για την ενσωμάτωση των επιταχυντών υλικού σε συστήματα συστάδων. Τα πλαίσια ανάλυσης δεδομένων όμως, όπως το Spark, δεν υποστηρίζουν την απευθείας χρήση τέτοιων συσκευών, συνεπώς η ενσωμάτωση έγινε χειροκίνητα με βασική αλλαγή την αντικατάσταση της βιβλιοθήκης που εισάγεται.

Τα αποτελέσματα που προέκυψαν τελικά από αυτό το έργο παρουσιάζονται παρακάτω. Αρχικά, μέσω σωστής χρήσης της HLS και του SDSoC καταφέραμε να επιταχύνουμε τον αλγόριθμο μηχανικής μάθησης Gaussian Naive Bayes. Πιο αυγκεκριμένα παρατηρήθηκε εώς 16.8 φορές γρηγορότερη εκτέλεση της εκπαίδευσης του αλγορίθμου και έως 14 φορές γρηγορότερη εκτέλεση της πρόβλεψης του αλγορίθμου, σε σχέση με τη χρήση ενός ARM επεξεργαστή. Επιπλέον, κατά την ενσωμάτωση του επιταχυντή, σε μια Python εφαρμογή, επιτύχαμε καλύτερη απόδοση τόσο σε σχέση με τον ενσωματωμένο ARM επεξεργαστή του Pynq (Επιτάχυνση Εκπαιδευσης: $x112$, Επιτάχυνση Προβλεψης: $x311$) όσο και με έναν Intel $i5$ 4ης γενιάς (Επιτάχυνση Εκπαιδευσης: $x4.1$, Επιτάχυνση Προβλεψης: $x24$. Τέλος, παραθέτουμε τη μεθοδολογία αξιοποίησης πολλαπλών FPGA, από ένα κατανεμημένο σύστημα, κάνοντας χρήση του εργαλείου Spark.

7.2 Μελλοντική Εργασία

Η παρούσα εργασία θα μπορούσε να επεκταθεί με διάφορους τρόπους. Αρχικά, όσον άφορά το κομμάτι της υλοποίησης του επιταχυντή υλικού θα μπορούσαμε ίσως, να βρούμε ακόμα αποδοτικότερη υλοποίηση, επιτυγχάνοντας ακόμα υψηλότερα επίπεδα επιτάχυνσης. Παράλληλα θα μπορούσε να αναπτυχθεί μια μεθοδολογία για την αυτοματοποίηση, σε έναν βαθμό, της διαδικασίας δομικού μετασχηματισμού της υλοποίησης των αλγορίθμων που επιθυμούμε να επιταχύνουμε. Ως αποτέλεσμα, ο πυρήνας που δημιουργήσαμε μαζί με τυχόν μελλοντικούς επιταχυντές υλικού θα μπορούσε να αποτελέσει μια βιβλιοθήκη από IP μπλοκς, που θα συμπεριλαμβάνει όλους τους υπολογιστικά απαιτητικούς πυρήνες διαφόρων τεχνικών Μηχανικής Μάθησης.

Τέλος θα ήταν ενδιαφέρον να τροποποιήσουμε και να αναπτύξουμε το έργο μας σε διαφορετικές ετερογενείς πλατφόρμες, οι οποίες φιλοξενούν επεξεργαστές υψηλής απόδοσης σε συνδυασμό με FPGAs, όπως τα F1 instances της Amazon. Με αυτόν τον τρόπο θα μπορούμε να έχομε μια καθαρή εικόνα των δυνατοτήτων αξιοποίησης των FPGA και να μελετήσουμε τα όριά τους σε ένα πραγματικό σύστημα.

Κεφάλαιο 8

Δημοσίευση

Μέρος της διπλωματικής έχει δημοσιευτεί στο παρακατώ συνέδριο:

- Hardware Acceleration on Gaussian Naive Bayes Machine Learning Alogorithm , Georgios Tzanos, Christoforos Kachris, and Dimitrios Soudris. International Conference on Modern Circuits and Systems Technologies (MOCAST), 2019.

Βιβλιογραφία

[1] FPGA Field-programmable gate array,
https://en.wikipedia.org/wiki/Field-programmable_gate_array

[2] SDSoC Development Environment,
https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html

[3] Vivado Design Suite, User Guide, High-Level Synthesis

[4] THE MNIST DATABASE of handwritten digits,
http://yann.lecun.com/exdb/mnist/

[5] Apache Spark^{TM} - Lightning-Fast Cluster Computing,
https://spark.apache.org/

[6] Mastering Apache Spark 2,
https://jaceklaskowski.gitbooks.io/mastering-apache-spark

[7] Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, 2012.

[8] MLlib: Machine Learning in Apache Spark, Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Journal of Machine Learning Research (JMLR), 2016.

[9] Zynq-7000 All Programmable SoC,
https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[10] The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc, Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. Strathclyde Academic Media, 2014.

[11] AXI Reference Guide, 2017

[12] Bayes' Theorem,
M. Bayes and M. Price. An Essay towards Solving a Problem in the Doctrine of Chances. By the Late Rev. Mr.Bayes, F. R. S. Communicated by Mr. Price, in a Letter to John Canton, A. M. F. R. S. Philosophical Transactions, 53:370–418, 1763.

[13] A. H. Jahromi and M. Taheri, "A non-parametric mixture of Gaussian naive Bayes classifiers based on local independent features," in 2017 Artificial Intelligence and Signal Processing Conference (AISP), Shiraz, 2017, pp. 209–212.

[14] PYNQ - Python productivity for Zynq,
http://www.pynq.io/

[15] SPynq: Acceleration of Machine Learning Applications over Spark on Pynq, Christoforos Kachris, Elias Koromilas, Ioannis Stamelos, and Dimitrios Soudris. International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS), 2017.

Παράρτημα Α΄

Appendices

Α΄.1 HW accelerated Machine Learning Βιβλιοθήκη

Listing A.1: mllib_accel/PynqApp.py

```
from pyspark import SparkContext
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib_accel.classificationNB_Pynq import Naivebayes
from sys import argv
from time import time


def parsePoint(line):
    """
    Parse a line of text into an MLlib LabeledPoint object.
    """

    data = [float(s) for s in line.split(',')]

    return LabeledPoint(data[0], data[1:])


if __name__ == "__main__":

    dataset = "MNIST"
    _accel_ = 1

    train_file = "inputs/" + dataset + "_train.dat"
    test_file = "inputs/" + dataset + "_test.dat"

    with open(dataset, 'r') as f:
        for line in f:
            if line[0] != '#':
                parameters = line.split(',')
```

```
            numClasses = int(parameters[0])
            numFeatures = int(parameters[1])
f.close()

trainSet = []
with open(train_file, 'r') as f:
    for line in f:
        trainSet.append(parsePoint(line))
f.close()

testSet = []
with open(test_file, 'r') as f:
    for line in f:
        testSet.append(parsePoint(line))
f.close()

print("* NaiveBayes Application *")
print(" # train file: ".format(train_file))
print(" # test file: ".format(test_file))


NB = Naivebayes(numClasses, numFeatures, decision)


start = time()

NB.train(trainSet, _accel_)

end = time()

if _accel_:
    print("! Time running Naive Bayes train in hardware: {:.3f} sec
        ".format(end - start))
else:
    print("! Time running Naive Bayes train in software: {:.3f} sec
        ".format(end - start))

stats = ["Means","Variances","Priors"]
for i in range (3):
    NB.save("outputs/trainPack"+ stats[i] + ".txt", i)

start = time()
NB.test(testSet,_accel_)
end = time()
print("! Time running Naive Bayes test in software: {:.3f} sec".
    format(end - start))
```

```
    start = time()

    NB.test(testSet, _accel_)
    if _accel_:
        print("! Time running Naive Bayes test in hardware: {:.3f} sec"
            .format(end - start))
    else:
        print("! Time running Naive Bayes test in software: {:.3f} sec"
            .format(end - start))

    end = time()
```

Listing A.2: mllib_accel/classification.py

```
import numpy as np
import os
import platform
import re
import cffi
import random

from itertools import tee
from math import ceil
from pynq import Overlay

DataPackSizeMax = 2000
numClassesMax = 10
numFeaturesMax = 784

BS_SEARCH_PATH = os.path.dirname(os.path.realpath(__file__)) + "/
    overlays/"

ffi = cffi.FFI()

ffi.cdef("""
void overlay_download(char *bit_file);
void *sds_alloc(unsigned int size);
void sds_free(void *memptr);
void NBtraining_kernel(int DataPack, int *n_per_class, int data, float *
    priors, float *means, float *variances);
int NBprediction_kernel(int data, int means, int variances, int priors );
""")

LIB_SEARCH_PATH = os.path.dirname(os.path.realpath(__file__)) + "/
    drivers/"
if platform.machine() == "x86_64":
```

```
    # load 64bit ELF
    hw_tr = ffi.dlopen(LIB_SEARCH_PATH + "NBtraining_kernel64.so")
    hw_pr = ffi.dlopen(LIB_SEARCH_PATH + "NBprediction_kernel64.so")
elif platform.machine() == "i386":
    # load 32bit ELF
    hw_tr = ffi.dlopen(LIB_SEARCH_PATH + "NBtraining_kernel32.so")
    hw_pr = ffi.dlopen(LIB_SEARCH_PATH + "NBprediction_kernel32.so")
elif platform.machine() == "armv7l":
    # load 32bit ELF compiled for ARM
    hw_tr = ffi.dlopen(LIB_SEARCH_PATH + "NBtraining_kernel.so")
    hw_pr = ffi.dlopen(LIB_SEARCH_PATH + "NBprediction_kernel.so")
else:
    print("Machine type not supported. Exiting!")
    exit(1)


def cma_train(LabeledPoints):
    #------------------------
    #   Download Overlay.
    #------------------------

    hw_tr.overlay_download((BS_SEARCH_PATH + "NBtraining.bit").encode('
        ascii'))

    elements = []
    flatLabelpoints = []

    numLabeledPoints = len(LabeledPoints)

    numDataPacks = int(ceil(numLabeledPoints / DataPackSizeMax))
    DataPackSize = int(ceil(numLabeledPoints / numDataPacks))
    if bool(DataPackSize & 1):
        DataPackSize += 1
    paddingSize = (numDataPacks * DataPackSize) - numLabeledPoints

    overall_class = []

    chunklist = [LabeledPoints[i:i + DataPackSize] for i in range(0,
        numLabeledPoints, DataPackSize)]
    for i in range(numDataPacks):
        chunklist[i] = sorted(chunklist[i] , key = lambda x: x.label)
        flatLabelpoints.append(chunklist[i])
        trainRDDcount = list(map(lambda x: x.label, chunklist[i]))

        num_class = []
        for i in range(numClassesMax):
```

```
            num_class.append(trainRDDcount.count(float(i)))
        overall_class.append(num_class)

    c = 1
    # ------------------------------
    #   Allocate physically contiguous memory buffers.
    # ------------------------------
    data = ffi.cast("float *", hw_tr.sds_alloc(DataPackSize *
        numFeaturesMax * 4))
    LabeledPoints = [i for sublist in flatLabelpoints for i in sublist]

    buffers = []
    buffers.append(int(re.split("0x|>", str(data))[1], 16))
    buffers.append(DataPackSize * numFeaturesMax)
    buffers.append((DataPackSize - paddingSize) if c == numDataPacks
        else DataPackSize)
    buffers.append(overall_class[c-1])
    elements.append(buffers)

    i = 0
    for LabeledPoint in LabeledPoints:
        if i < int(DataPackSize):
            f = ffi.from_buffer(LabeledPoint.features.astype(np.float32
                ))
            features = ffi.cast("float *", f)
            offset_point = i * numFeaturesMax
            data[offset_point:offset_point + len(LabeledPoint.features)
                ] = features[0:len(LabeledPoint.features)]
        i += 1
        if i == int(DataPackSize):
            c += 1
            if c <= numDataPacks:
                # ------------------------------
                #   Allocate physically contiguous memory buffers.
                # ------------------------------
                data = ffi.cast("float *", hw_tr.sds_alloc(DataPackSize
                    * numFeaturesMax * 4))
                buffers = []
                buffers.append(int(re.split("0x|>", str(data))[1], 16))
                buffers.append(DataPackSize * numFeaturesMax)
                buffers.append((DataPackSize - paddingSize) if c ==
                    numDataPacks else DataPackSize)
                buffers.append(overall_class[c-1])
                elements.append(buffers)
                i = 0
    return elements
```

```
def trainingNB_kernel_accel(buffers):

    #------------------------
    #   Accelerator callsite.
    #------------------------

    DatapackSize = int(buffers[1] / (numFeaturesMax))

    numClasses = 10
    numFeatures = 784

    offset_buf = ffi.cast("int *", hw_tr.sds_alloc(numClassesMax * 4))
    offset_buf[0:numClassesMax] = buffers[3][0:numClassesMax]

    m_buf = ffi.new("float []", numClassesMax * numFeaturesMax)
    v_buf = ffi.new("float []", numClassesMax * numFeaturesMax)
    p_buf = ffi.new("float []", numClassesMax)

    hw_tr.NBtraining_kernel(DatapackSize, offset_buf, buffers[0], p_buf
        , m_buf, v_buf)

    v = np.frombuffer(ffi.buffer(v_buf, 4 * numClassesMax *
        numFeaturesMax), dtype = np.float32)
    m = np.frombuffer(ffi.buffer(m_buf, 4 * numClassesMax *
        numFeaturesMax), dtype = np.float32)
    p = np.frombuffer(ffi.buffer(p_buf, 4 *  numClassesMax), dtype = np
        .float32)


    variances = np.copy(np.reshape(v, (numClasses,numFeatures))[ : , :
        numFeatures])
    means= np.copy(np.reshape(m, (numClasses,numFeatures))[ : , :
        numFeatures])
    priors = np.copy(np.reshape(p, (numClasses))[:])

    trainPack = [(means), (variances), (priors)]

    return trainPack

def cmf_train(buffers):

    #------------------------
    #   Free previously allocated buffers.
    #------------------------
```

```
    hw_tr.sds_free(ffi.cast("void *", buffers[0]))

    return 0


def cma_predict(trainPack):

    # ----------------------------
    #    Download Overlay.
    # ----------------------------
    hw_pr.overlay_download((BS_SEARCH_PATH + "NBprediction.bit").encode
        ('ascii'))


    means = ffi.cast("float *", hw_pr.sds_alloc(numClassesMax *
        numFeaturesMax * 4))
    variances = ffi.cast("float *", hw_pr.sds_alloc(numClassesMax *
        numFeaturesMax * 4))
    priors = ffi.cast("float *", hw_pr.sds_alloc(numClassesMax * 4))

    address = []
    address.append(int(re.split("0x|>", str(means))[1], 16))
    address.append(int(re.split("0x|>", str(variances))[1], 16))
    address.append(int(re.split("0x|>", str(priors))[1], 16))

    m = np.reshape(list(trainPack[0]), numClassesMax * numFeaturesMax)
    v = np.reshape(trainPack[1], numClassesMax * numFeaturesMax)
    p = np.reshape(trainPack[2], numClassesMax )

    means[0:numClassesMax * numFeaturesMax] = m[0:numClassesMax *
        numFeaturesMax]
    variances[0:numClassesMax * numFeaturesMax] = v[0:numClassesMax *
        numFeaturesMax]
    priors[0:numClassesMax] = p[0:numClassesMax]

    return address

def predictionNB_kernel_accel(buffers, line):

    # ----------------------------
    #    Accelerator callsite.
    # ----------------------------

    numFeatures = 784

    data = ffi.cast("float *", hw_tr.sds_alloc(numFeaturesMax * 4))
```

```
    addr = int(re.split("0x|>", str(data))[1], 16)
    data[0:numFeatures] = line[0:numFeatures]

    prediction = hw_pr.NBprediction_kernel(addr, buffers[0], buffers
        [1], buffers[2])

    hw_pr.sds_free(ffi.cast("void *", addr))

    return prediction

def cmf_predict(buffers):

    #-------------------------
    #   Free previously allocated buffers.
    #-------------------------

    hw_pr.sds_free(ffi.cast("void *", buffers[0]))
    hw_pr.sds_free(ffi.cast("void *", buffers[1]))
    hw_pr.sds_free(ffi.cast("void *", buffers[2]))

return 0
```

Listing A.3: mllib_accel/Naivebayes.py

```
mport numpy as np
from math import exp ,sqrt ,pi, log , inf
from pyspark import RDD
from time import time
from itertools import tee
from functools import reduce
from .accelerators.Naivebayes import cma_train, cma_predict,
    predictionNB_kernel_accel, trainingNB_kernel_accel, cmf_train,
    cmf_predict

__all__ = ['Naivebayes']

class Naivebayes(object):
    """
    Multiclass Naive Bayes Model.
        :param numClasses:      Number of possible outcomes.
        :param numFeatures:     Dimension of the features.
    """


    def __init__(self, numClasses, numFeatures, trainPack = None ):
        self.numClasses = numClasses
        self.numFeatures = numFeatures
```

```
        self.trainPack = trainPack

    def train(self, trainRDD, _accel_ = 0):
        """
        Train a naive bayes model on the given data.

            :param trainRDD:        The training data, a list of
                LabeledPoint.

            :param _accel_:          0: SW-only, 1: HW accelerated (
                deafult: 0).
            :note:                   Labels used in naive bayes should
                be

                                     {0, 1, ..., k - 1} for k classes
                                         classification problem.
        """

        def array_red(a,b):
            adding = [tuple([tuple(row) if not isinstance(row,np.
                float32) else row for row in np.array(aa)+np.array(bb)
                ]) for aa, bb in zip(a, b)]
            return adding

        def sorting(trainRDD):
            trainRDD = sorted(trainRDD , key = lambda x: x.label)
            trainRDDcount = list(map(lambda x: x.label, trainRDD))

            offset=[]
            for i in range(self.numClasses):
                offset.append(trainRDDcount.count(float(i)))

            return trainRDD, offset

        def trainingNB_kernel(data, offset):
            offset_counter = 0

            priors = np.zeros((self.numClasses))
            variances = np.zeros((self.numClasses,self.numFeatures))
            means = np.zeros((self.numClasses,self.numFeatures))
            sums_x = np.zeros((self.numClasses,self.numFeatures))
            sq_sums_x = np.zeros((self.numClasses,self.numFeatures))
            for line in data:
                label = int(line.label)
                if offset_counter < offset[label]-1:
                    for i in range(0, self.numFeatures):
                        sums_x[label][i] += line.features[i]
```

```
                sq_sums_x[label][i] += line.features[i]*line.
                    features[i]
            offset_counter += 1
        else:
            priors[label] = offset[label]/sum(offset)
            for i in range(0, self.numFeatures):
                means[label][i] = sums_x[label][i] / offset[
                    label]
                variances[label][i] = (sq_sums_x[label][i] /
                    offset[label]) - (means[label][i]*means[
                    label][i])
            offset_counter = 0

    trainPack = [means, variances, priors]

    return trainPack

# For a multiclass classification with k classes, train k
    models (one per class).
print("* NaiveBayes Training *")
numBuffers = 1
if (_accel_):
    trainRDD = list(cma_train(trainRDD))
    numBuffers = len(trainRDD)
else:
    # Sort Data by type of class and counting overall features
        per class
    trainRDD, offset = sorting(trainRDD)


print("     #numBuffers:                {:d}".format(numBuffers)
    )
print("     #numClasses:                {:d}".format(self.
    numClasses))
print("     #numFeatures:               {:d}".format(self.
    numFeatures))
print("     #Accelerated:               {0}".format(bool(_accel_
    )))


#start = time()

if (_accel_):
    trainPackage = list(map(lambda data:
        trainingNB_kernel_accel(data), trainRDD))
    trainPackage = reduce(lambda a, b: array_red(a, b),
```

```
                trainPackage)
            trainPackage = [tuple([tuple(row/numBuffers) if not
                isinstance(row,np.float32) else row/numBuffers for row
                in np.array(aa)]) for aa in trainPackage]
            self.trainPack = list(trainPackage)
        else:
            trainPackage = trainingNB_kernel(trainRDD, offset)
            self.trainPack = list(trainPackage)
        #end = time()


        if (_accel_):
            map(cmf_train,trainRDD)

    def test(self, testRDD, _accel_ = 0):
        """
        Test a naive bayes model on the given data.
            :param testRDD:     The testing data, a list of
                LabeledPoint.
            :note:              Labels used in Naive Bayes should be
                                {0, 1, ..., k - 1} for k classes
                                    classification problem.
        """

        # Each example is scored against all k models and the model
            with highest probability
        # is picked to label the example.

        print("* NaiveBayes Testing *")


        if (_accel_):
            address = cma_predict(self.trainPack)
        else:
            testRDD = testRDD

        #start = time()

        if (_accel_):
            true = map(lambda data: 1 if data.label ==
                predictionNB_kernel_accel(address, data.features)
                else 0,testRDD)
        else:
            true = list(map(lambda data: 1 if data.label == self.
                predict(data.features) else 0, testRDD))
        true = reduce(lambda a, b: a + b,true)
```

```
            false = len(testRDD) - true

            #end = time()

            print("      # accuracy:                {:.3f} ({:d}/{:d})"
                .format(true / (true + false), true, true + false))
            print("      # true:                    {:d}".format(true))
            print("      # false:                   {:d}".format(false)
                )

            if (_accel_):
                cmf_train(address)

            return self

    def predict(self, data):
        """
        Predict values for a single data point using the model trained.
            :param features:    Features to be labeled.
        """
        prediction = 0
        max_likelihood = -inf
        for label in range(0, self.numClasses):
            numerator = log(self.trainPack[2][label])
            for j in range(0 , self.numFeatures):
                if  self.trainPack[1][label][j] < 0.00005:
                    numerator += 0.0
                else:
                    numerator += log(1 / sqrt(2 * pi * self.trainPack
                        [1][label][j])) + ((-1*(data[j] - self.
                        trainPack[0][label][j])**2) / (2 * self.
                        trainPack[1][label][j]))
            if numerator > max_likelihood :
                max_likelihood = numerator
                prediction = label

        return prediction

    def save(self, path, stats):
        """
        Save this model to the given path.
        """

        np.savetxt(path, self.trainPack[stats],newline='n')
```

```
    def load(self, path):
        """
        Load a model from the given path.
        """

self.trainPack = np.loadtxt(path)
```