ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

Επιτάχυνση του αλγορίθμου μηχανικής μάθησης k-Nearest Neighbors με τη χρήση FPGA

Διπλωματική Εργασία

του

Θεόδωρου Σταμπουλή

Θεσσαλονίκη, Φεβρουάριος 2019

Επιτάχυνση του αλγορίθμου μηχανικής μάθησης k-Nearest Neighbors
με τη χρήση FPGA

Θεόδωρος Σταμπουλής

Πτυχίο Ηλεκτρολόγου Μηχανικού και Μηχανικού Υπολογιστών, ΕΜΠ, 2016

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Κων/νος Μαργαρίτης

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28/02/2019

Κων/νος Μαργαρίτης	Θεόδωρος Κάσκαλης	Εμμανουήλ Ρουμελιώτης
....	....	....

Θεόδωρος Σταμπουλής

....

Περίληψη

Στην παρούσα διπλωματική εργασία γίνεται προσπάθεια επιτάχυνσης του αλγορίθμου μηχανικής μάθησης k-Nearest Neighbors με τη χρήση FPGA. Η αναπτυξιακή πλατφόρμα που χρησιμοποιήθηκε είναι η PYNQ-Z1 η οποία φέρει ολοκληρωμένο κύκλωμα της σειράς Zynq-7000 της Xilinx. Η πλατφόρμα αυτή υποστηρίζει την χρήση της γλώσσας υψηλού επιπέδου Python, για τον έλεγχο των επιταχυντών που εκτελούνται στο FPGA κομμάτι του κυκλώματος, μέσω του API που προσφέρει το ανοιχτού-κώδικα έργο PYNQ. Ο αλγόριθμος εφαρμογής ήταν αυτός που υλοποιείται από τη βιβλιοθήκη μηχανικής μάθησης Scikit-learn. Η βιβλιοθήκη αυτή τροποποιήθηκε ώστε να καλεί τους επιταχυντές υλικού που έχουν υλοποιηθεί στο FPGA. Προκειμένου να γίνουν οι ελάχιστες δυνατές τροποποιήσεις στον κώδικα της βιβλιοθήκης επιχειρήθηκε η επιτάχυνση συναρτήσεων του πακέτου επιστημονικών υπολογισμών NumPy, οι οποίες καλούνται από τη Scikit-learn. Η πρώτη συνάρτηση στην οποία εφαρμόστηκε η μεθοδολογία ανάπτυξης σε FPGA είναι η matmul και εκτελεί πολλαπλασιασμό πινάκων. Η ίδια μεθοδολογία ανάπτυξης εφαρμόστηκε και για τη συνάρτηση sort του πακέτου που υλοποιεί αλγόριθμο ταξινόμησης. Η επιτάχυνση που έχει επιτευχθεί τείνει στο 80% για μεγάλα σύνολα δεδομένων για τον αλγόριθμο k-NN. Δεδομένου ότι από το πακέτο Scikit-learn καλούνται και άλλες συναρτήσεις, εκτός αυτών που εκτελέστηκαν στο FPGA μια τέτοια επιτάχυνση θεωρείται ικανοποιητική. Μελλοντική επέκταση της εφαρμογής και σε άλλες συναρτήσεις του πακέτου NumPy αλλά και του πακέτου SciPy φαίνεται από τη παρούσα εργασία ότι είναι εφικτή.

Λέξεις Κλειδιά: FPGA, Μηχανική μάθηση, Αλγόριθμος k-κοντινότερων σημείων, επιταχυντές υλικού.

Abstract

The purpose of this master thesis is to develop a FPGA based implementation of the popular machine-learning algorithm k-Nearest Neighbors for PYNQ-Z1 development board. This board is based on a ZYNQ-7000 system on chip by Xilinx, which compines a dual-core Cortex-A9 processor with FPGA fabric and is the hardware platform for the PYNQ open-source framework. PYNQ project enables the control of accelarators running on an FPGA using the high-level language Python. K-NN implementation of machine learning library Scikit-learn is modified properly in order to call the designed FPGA accelarators. Scikit-learn is build on the numerical computation package Numpy. To prevent excessive modifications to the code of the library, which leads to major bugs, it was attempted to accelerate the functions of NumPy, which are called by Scikit-learn. The methodology was initially applied to matmul function, performing the classic algorithm of matrix multiplication. In respect to k-NN algorithm, the speedup achieved is close to 80% for big data sets. Considering the site-calls performed by Scikit-learn, this performance is acceptable. The above constitutes proof that expansion of this work to other functions of NumPy and SciPy packages is feasible.

Keywords: FPGA, Machine Learning, KNN, PYNQ, Scikit-learn, HW Accelerators,

Ευχαριστίες

Θα ήθελα να ευχαριστήσω την οικογένειά μου που με στήριξε κατά τη διάρκεια της εκπόνησης αυτής της διπλωματικής εργασίας, χωρίς αυτούς η μελέτη αυτή δε θα μπορούσε να ολοκληρωθεί. Επιπλέον θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Κωνσταντίνο Μαργαρίτη για την εμπιστοσύνη που έδειξε να μου αναθέσει την διπλωματική αυτή εργασία.

Τέλος θα ήθελα να την αφιερώσω στον πατέρα μου που με εμφύσησε την αγάπη του στις θετικές επιστήμες και ο οποίος πρόσφατα βγήκε πιο γερός από μια πολύ σοβαρή περιπέτεια υγείας.

Περιεχόμενα

1 Εισαγωγή xii

1.1 Πρόβλημα – Σημαντικότητα του θέματος xii

1.2 Σκοπός – Στόχοι xiii

1.3 Συνεισφορά xiv

1.4 Διάρθρωση της μελέτης xiv

2 Προγραμματιστικά εργαλεία και Hardware xv

2.1 Open-project PYNQ xv

2.1.1 Η αναπτυξιακή πλακέτα PYNQ-Z1.....xv

2.1.2 Βιβλιοθήκες του PYNQ.....xvi

2.1.3 Βιβλιοθήκες Hardware (Overlays).....xvii

2.2 High Level Synthesis xvii

2.2.1 Τεχνικές βελτιστοποίησης στο HLS.....xviii

2.3 SDSoC xxii

2.3.1 Δίκτυο Μεταγωγής Δεδομένων (Data Motion Network).....xxiii

2.4 7000 System-on-Chip xxiv

2.4.1 Η μονάδα επεξεργασίας (Processing System).....xxiv

2.4.2 Τμήμα προγραμματισμένης λογικής (Programmable Logic).....xxv

2.4.3 Οι διεπαφές PS-PL.....xxviii

2.5 Η βιβλιοθήκη μηχανικής μάθησης Scikit-learn xxix

2.5.1 Unsupervised Nearest Neighbors.....xxx

3 Παρουσίαση των αλγόριθμων xxx

3.1 Ο αλγόριθμος k κοντινότερων σημείων k-NN xxx

3.1.1 Περιγραφή του αλγορίθμου.....xxxi

3.2 Πολλαπλασιασμός Πινάκων στον k-NN xxxii

3.2.1 Πολλαπλασιασμός πινάκων με Υποπίνακες.....xxxiii

4 Υλοποίηση αλγορίθμων στο Hardware xxxiv

4.1 Μεθοδολογία ανάπτυξης xxxiv

4.2 Ανάλυση του κώδικα - Profiling xxxv

4.3 Πολλαπλασιασμός Πινάκων xxxvi

4.3.1 Ανάλυση των τρόπων υλοποίησης.....xxxvi

4.3.2 Αρχιτεκτονική του πυρήνα.....xxxix

4.4 Εξαγωγή των οδηγών xliv

4.5 Σύνδεση με Python xlvi
4.6 Εφαρμογή στη βιβλιοθήκη Scikit-learn xlvi
4.7 Bitonic Sort xlvii
5 Παρουσίαση Αποτελεσμάτων li
5.1 Εισαγωγή li
5.2 Πολλαπλασιασμός Πινάκων li
5.2.1 Standalone εφαρμογή li
5.2.2 Εφαρμογή στο PYNQ lii
5.3 Bitonic Sort liii
5.3.1 Standalone εφαρμογή liii
5.4 Επιτάχυνση του Knn liv
6 Επίλογος lv
6.1 Σύνοψη και συμπεράσματα lv
6.2 Μελλοντικές Επεκτάσεις lvi

Κατάλογος Εικόνων

2.1. Εικόνα: Η αναπτυξιακή πλακέτα PYNQ-Z1.....xv
2.2. Εικόνα: Σωλήνωση βρόχου (Loop Pipeline).....xix
2.3. Εικόνα: Παράδειγμα ξετυλίγματος βρόχου (Loop unrolling).....xx
2.4. Εικόνα: Διαίρεση πινάκων (Array partitioning).....xxi
2.5.Εικόνα: Παράδειγμα σωλήνωσης δεδομένων (Dataflow Pipeline).....xxi
2.6.Εικόνα: Διαδικασία σχεδίασης στο SDSoC.....xxii
2.7. Εικόνα: Δίκτυο μεταγωγής δεδομένων (Data motion network).....xxiii
2.8.Εικόνα: Hard και Soft processors στον Zynq-7000.....xxiv
2.9. Εικόνα: Σχεδιάγραμμα με τις υπομονάδες του Zynq-7000.....xxv
2.10. Εικόνα: Σχεδιάγραμμα του PL τμήματος.....xxvi
2.11. Εικόνα: Configurable Logic Block (CLB).....xxvi
2.12. Εικόνα: Απλοποιημένο διάγραμμα DSP48E1.....xxviii
2.13. Εικόνα: Διεπαφές PS-PL.....xxix
3.1. Εικόνα: Απλό παράδειγμα ταξινόμησης με k-NN.....xxxiii
3.2. Εικόνα: Σχηματική αναπαράσταση πολλαπλασιασμού με υποπίνακες.....xxxiv
4.1. Εικόνα: Profiling του k-NN.....xxxvi
4.2. Εικόνα: Buffers συνεχούς μνήμης.....xxxviii
4.3. Εικόνα: Απλό παράδειγμα που δείχνει την τοποθέτηση των στοιχείων κατά τον πολλαπλασιασμό υποπινάκων.....xxxix
4.4. Εικόνα: Πολλαπλασιασμός πινάκων με φόρτωση ολόκληρου του πίνακα Β.....xxxix
4.5. Εικόνα: Εφαρμογή τη οδηγίας array_partition στους πίνακες A, B.....xlii
4.6. Εικόνα: Σωλήνωση σε επίπεδο βρόχων.....xliii
4.7.Εικόνα: Αρχιτεκτονική του επιταχυντή.....xliv
4.8. Εικόνα: Ασύγχρονο και μη ασύγχρονο μοντέλο εκτέλεσης.....xlvi
4.9.Εικόνα: Δικτύωμα Bitonic sort.....l
4.10.Εικόνα: Αρχιτεκτονική του επιταχυντή Bitonic sort.....l
5.1.Εικόνα: Επιτάχυνση σε συνάρτηση με τον αριθμό στηλών.....li
5.2.Εικόνα: Χρόνοι εκτέλεσης και επιτάχυνση σε συνάρτηση με τον αριθμό στηλών.....lii
5.3. Εικόνα: Χρόνος εκτέλεσης και επιτάχυνση σε συνάρτηση με τον αριθμό των γραμμών του πίνακα Α.....liii
5.4. Εικόνα: Χρόνοι εκτέλεσης και επιτάχυνση σε συνάρτηση με τον αριθμό των σημείων του συνόλου δοκιμών.....lv

5.5. Εικόνα: Ακρίβεια αποτελεσμάτων που παρήγαγε ο k-NN...lv

Κατάλογος Πινάκων

4.1.Πίνακας: Κατανομή των πόρων του FPGA για τη συνάρτηση matmul_accel.............35

4.2.Πίνακας: Κατανομή των πόρων του FPGA για τη συνάρτηση bitonicsort_accel........42

5.1.Πίνακας: Χαρακτηριστικά του συνόλου δεδομένων "Optical Recognition of Handwritten Digits"...47

1 Εισαγωγή

1.1 Πρόβλημα – Σημαντικότητα του θέματος

Η μηχανική μάθηση είναι μια ταχύτατα αναπτυσσόμενη περιοχή της επιστήμης των υπολογιστών με μεγάλο εύρος εφαρμογών. Σε πολλές περιπτώσεις οι αλγόριθμοι μηχανικής μάθησης έχουν μεγάλες υπολογιστικές απαιτήσεις κάτι που δυσχεραίνει την εφαρμογή τους σε συστήματα με μειωμένους πόρους, όπως σε ενσωματωμένα συστήματα ή σε εφαρμογές όπου ο χρόνος απόκρισης είναι κρίσιμου χαρακτήρα, όπως στα πραγματικού χρόνου συστήματα (real time). Οι αλγόριθμοι αυτοί μπορούν να "ωφεληθούν" από τη υπολογιστική ισχύ των Field Programmable Gate Arrays (FPGAs). Ένα FPGA είναι ένα ολοκληρωμένο κύκλωμα με μεγάλο αριθμό λογικών πυλών, των οποίων οι συνδέσεις δεν είναι σταθερές αλλά μπορούν να μεταβληθούν. Μπορεί δηλαδή να "προγραμματισθεί" ανάλογα με την εφαρμογή που εκτελεί κάθε φορά, προσφέροντας καλύτερη απόδοση όσον αφορά την ταχύτητα εκτέλεσης αλλά και την κατανάλωση ενέργειας. Συγκριτικά με ένα Application Specific Integrated Circuit (ASIC) ένα FPGA διαθέτει μεγαλύτερη ευελιξία λόγω τις δυνατότητας επαναπρογραμματισμού και παρόμοια απόδοση.

Ο συνήθης τρόπος ανάπτυξης εφαρμογών που περιλαμβάνουν τη χρήση FPGA, είναι μέσω γλωσσών χαμηλού επιπέδου όπως οι C και C++, για την επικοινωνία με τον κυρίως επεξεργαστή και γλωσσών περιγραφής υλικού (Verilog-VHDL), για τον προγραμματισμό του ίδιου του FPGA. Τα προγραμματιστικά αυτά εργαλεία λόγω του ότι προϋποθέτουν μεγάλη εξειδίκευση χρησιμοποιούνται συνήθως από έμπειρους σχεδιαστές Hardware.

Στη διπλωματική αυτή εργασία επιχειρείται η χρήση δύο προγραμματιστικών εργαλείων με στόχο την απλοποίηση της διαδικασίας ανάπτυξης. Συγκεκριμένα χρησιμοποιούνται το ανοιχτού κώδικα λογισμικό PYNQ (Python Productivity for Zynq) και το High Level Synthesis (HLS) που παρέχεται μέσω της σουίτας SDSoC (Software Defined System-on-Chip). Το PYNQ δίνει τη δυνατότητα στον χρήστη να ελέγχει μέσω της γλώσσας υψηλού επιπέδου Python τους επιταχυντές που υλοποιούνται στο FPGA κομμάτι της εφαρμογής. Έτσι προκρίνεται η επέκταση της εκτέλεσης ρουτινών Python και στο υλικό. Το HLS δίνει εναλλακτική λύση στον προγραμματισμό του FPGA μέσω των γλωσσών C, C++, SystemC και OpenCl. Με αυτόν τον τρόπο προγραμματιστές που έχουν εμπειρία σε αυτές τις γλώσσες μπορούν να ωφεληθούν από τη χρήση αυτών των συσκευών.

1.2 Σκοπός – Στόχοι

Σκοπός της μελέτης είναι η επιτάχυνση του αλγορίθμου k-NN μέσω της χρήσης FPGA. Η υλοποίηση του αλγορίθμου αυτού από τη βιβλιοθήκη Scikit-learn επιλέχθηκε να είναι το αντικείμενο εφαρμογής της διαδικασίας ανάπτυξης σε FPGA. Η όλη διαδικασία συνοψίζεται στην σχεδίαση κατάλληλων βιβλιοθηκών, μέσω τον οποίων γίνεται η κλήση των επιταχυντών οι οποίοι είναι υλοποιημένοι στο κύκλωμα του FPGA. Οι βιβλιοθήκες αυτές στην ορολογία του πακέτου PYNQ ονομάζονται Overlays και αποτελούνται από το HW κομμάτι και το κομμάτι της οδήγησης (Drivers).

Οι γενικότεροι στόχοι που τέθηκαν είναι οι εξής:

α) Όσο το δυνατόν φυσικότερη εφαρμογή δηλαδή με τις λιγότερες δυνατές τροποποιήσεις στη βιβλιοθηκη Scikit-learn, καθώς είναι ένα υψηλής ποιότητας λογισμικό, δοκιμασμένο ενδελεχώς και τυχόν τροποποιήσεις εμπεριέχουν τον κίνδυνο σφαλμάτων.

β) Να ωφελούνται από την επιτάχυνση όσο το δυνατόν περισσότερες συναρτήσεις της βιβλιοθήκης.

γ) Σχεδίαση της Overlay, ώστε να είναι όσο το δυνατόν ποιο ελαστική ως προς τα ορίσματα που δέχεται.

Οι πρώτες δύο σχεδιαστικές απαιτήσεις καλύφθηκαν στοχεύοντας στην αντικατάσταση συναρτήσεων χαμηλά στην δομή του κώδικα, οι οποίες εκτελούν απλά το υπολογιστικό κομμάτι χωρίς να καλούν άλλες συναρτήσεις ή να εκτελούν ελέγχους ορισμάτων. Για την υλοποίηση του αλγορίθμου k-NN γίνονται κλήσεις συναρτήσεων του πακέτου επιστημονικών υπολογισμών NumPy. Με την αντικατάσταση στον κώδικα υπολογιστικά απαιτητικών συναρτήσεων του πακέτου NumPy επιτυγχάνεται η ελάχιστη παρέμβαση στην Scikit-learn, ενώ ωφελούνται όσες συναρτήσεις τις καλούν.

Η τρίτη σχεδιαστική απαίτηση είναι και η πιο δύσκολη να επιτευχθεί κυρίως λόγω πεπερασμένων πόρων που διαθέτει μια συσκευή FPGA. Επιπλέον η βέλτιστη εξαγωγή των δυνατοτήτων ενός FPGA επιτυγχάνεται όταν η αρχιτεκτονική των κυκλωμάτων με των οποίων υλοποιείται ένας αλγόριθμος είναι σταθερή. Σε γενικές γραμμές αυτό σημαίνει ότι όσο ποιο γενικού σκοπού είναι η υλοποίηση τόσο χαμηλότερη είναι η ταχύτητα εκτέλεσης.

1.3 Συνεισφορά

Στη μελέτη αυτή διερευνάται με ποιο τρόπο και σε ποιο βαθμό μπορεί να ωφεληθεί μια βιβλιοθήκη όπως η Scikit-learn και η NumPy από την επιτάχυνση που προσφέρει η εκτέλεση μέρους του κώδικά τους σε ένα FPGA.

1.4 Διάρθρωση της μελέτης

Στο κεφάλαιο 2 γίνεται παρουσίαση των προαπαιτούμενων για την κατανόηση της μεθοδολογίας ανάπτυξης που ακολουθήθηκε. Αναλύονται τα προγραμματιστικά εργαλεία που χρησιμοποιήθηκαν, καθώς και το Hardware στο οποίο έγινε η ανάπτυξη της εφαρμογής.

Στο κεφάλαιο 3 παρουσιάζονται ο υποψήφιος αλγόριθμος προς επιτάχυνση που στη περίπτωση αυτή είναι ο k-NN καθώς και η βιβλιοθήκη Scikit-learn.

Στο κεφάλαιο 4 γίνεται ανάλυση όλων των “βημάτων” που ακολουθήθηκαν κατά την όλη διαδικασία ανάπτυξης.

Στο κεφάλαιο 5 γίνεται παρουσίαση των αποτελεσμάτων.

Τέλος στο κεφάλαιο 6 συνοψίζονται τα συμπεράσματα. Επιπλέον δίδονται κατευθύνσεις για πιθανή επέκταση της μελέτης.

2 Προγραμματιστικά εργαλεία και Hardware

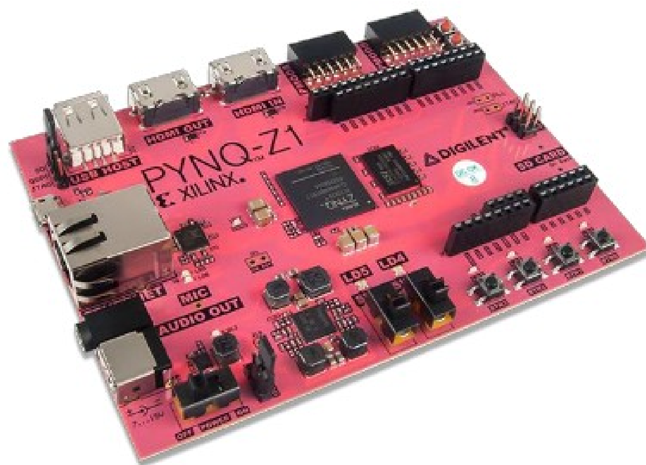
2.1 Open-project PYNQ

Το open-project PYNQ είναι ένα ανοιχτού-κώδικα framework που δίνει τη δυνατότητα χειρισμού επιταχυντών, οι οποίοι υλοποιούνται στο FPGA τμήμα του ολοκληρωμένου κυκλώματος μέσω της υψηλού επιπέδου γλώσσας Python. Με αυτό το τρόπο διευκολύνεται η διαδικασία σχεδίασης και υλοποίησης ενσωματωμένων συστημάτων, καθώς ένας προγραμματιστής μπορεί να χρησιμοποιήσει έτοιμες βιβλιοθήκες Hardware (Overlays). Αυτή τη στιγμή υπάρχουν συγκεκριμένες αναπτυξιακές πλακέτες που υποστηρίζουν το PYNQ. Η πλακέτα που χρησιμοποιήθηκε στη παρούσα διπλωματική είναι η PYNQ-Z1.

2.1.1 Η αναπτυξιακή πλακέτα PYNQ-Z1

Η αναπτυξιακή πλακέτα PYNQ-Z1 είναι σχεδιασμένη ώστε να υποστηρίζει το PYNQ. Τα κυριότερα χαρακτηριστικά της πλακέτας είναι τα παρακάτω:

- Ολοκληρωμένο κύκλωμα της σειράς ZYNQ-7000 με ενσωματωμένο επεξεργαστή 650MHz dual-core Cortex-A9. To Programmable Logic (PL) τμήμα είναι αντίστοιχο ενός Artix-7 FPGA και διαθέτει 13.300 logic slices, 630 KB block RAM, 220 DSP slices και analog-to-digital converter (XADC).
- Μνήμη 512MB DDR3 με 16-bit bus.
- Micro SD slot
- Θύρες USB και Ethernet
- 4 κομβία πίεσης, 2 διακόπτες, 4 LEDs, 2RGB LEDs.



Εικόνα 2.1: Η αναπτυξιακή πλακέτα PYNQ-Z1

Το λογισμικό που εκτελείται στην αναπτυξιακή πλακέτα PYNQ-Z1 περιλαμβάνει τα παρακάτω:

- Jupyter Notebook
- IPython kernel/packages
- Linux
- Base hardware library/API για το FPGA

Είναι απαραίτητη η επισήμανση ότι η σειρά Zynq-7000 είναι ολοκληρωμένα κυκλώματα τα οποία συνδυάζουν επεξεργαστή και FPGA στον ίδιο κρύσταλλο πυριτίου. Το τμήμα του επεξεργαστή αναφέρεται ως σύστημα επεξεργασίας (Processing System - PS) και το FPGA ως προγραμματιζόμενη λογική (Programmable Logic - PL)

2.1.2 Βιβλιοθήκες του PYNQ

Το λογισμικό PYNQ περιλαμβάνει διάφορες βιβλιοθήκες, ώστε να εξυπηρετεί τις λειτουργίες της πλακέτας. Υπάρχουν οδηγοί για τον έλεγχο έτοιμων επιταχυντών, περιφερειακών αλλά και soft-processors όπως ο MicroBlaze. Επίσης παρέχει έτοιμα APIs για τον χαμηλού επιπέδου έλεγχο λειτουργιών μιας overlay. Στις επόμενες παραγράφους αναλύονται οι βιβλιοθήκες που χρησιμοποιήθηκαν στη παρούσα διπλωματική.

- Xlnk - Η κλάση (class) Xlnk χρησιμοποιείται για την εξασφάλιση φυσικά συνεχών θέσεων μνήμης. Ένας επιταχυντής υλικού μπορεί να ωφεληθεί από την προσπέλαση θέσεων μνήμης που είναι συνεχόμενες. Αυτό συμβαίνει γιατί αποφεύγεται η χρονοβόρα μετάφραση των σελίδων από το χώρο εικονικής μνήμης. Σε συνδυασμό και με το πακέτο NumPy μπορεί να εξασφαλιστεί συνεχή μνήμη και για πίνακες ndarrays.
- Overlay - Είναι κλάση που χρησιμοποιείται για τη φόρτωση και έλεγχο Overlays στο PL. Για να χρησιμοποιηθεί η συγκεκριμένη κλάση θα πρέπει να παρέχεται ένα bitstream και ένα tcl αρχείο. Το bitstream αρχείο είναι αυτό που διαμορφώνει το FPGA ενώ το tcl παρέχει πληροφορίες σχετικά με τον επιταχυντή.
- C Foreign Function Interface (CFFI) - Το πακέτο PYNQ περιέχει όλες εκείνες τις λειτουργίες που επιτρέπουν τον έλεγχο των επιταχυντών που εκτελούνται στο PL. Επομένως δίνει τη δυνατότητα ο οδηγός του επιταχυντή να γραφεί στη γλώσσα Python. Αυτό βοηθάει στην αύξηση της παραγωγικότητας όμως προσθέτει καθυστέρηση. Στη παρούσα διπλωματική οι οδηγοί των επιταχυντών που σχεδιάστηκαν γράφτηκαν σε C, ώστε να γίνει εκμετάλλευση της υψηλότερης ταχύτητας εκτέλεσης. Το πακέτο CFFI

χρησιμοποιήθηκε για να γίνει η σύνδεση των οδηγών με τη γλώσσα Python, ώστε να μπορούν να καλούνται από τη τελευταία.

2.1.3 Βιβλιοθήκες Hardware (Overlays)

Οι βιβλιοθήκες Hardware (Overlays) βοηθούν τον χρήστη να επεκτείνει τις εφαρμογές του από το PS στο PL. Οι overlays μπορούν να χρησιμοποιηθούν για να επιταχύνουν τον κώδικα που εκτελείται ή για να διαμορφώσουν το PL προκειμένου να εκτελέσουν μια συγκεκριμένη εφαρμογή. Οι overlays μπορούν να φορτωθούν δυναμικά όπως και οι υπόλοιπες βιβλιοθήκες λογισμικού. Το PYNQ είναι ουσιαστικά μια διεπαφή Python που επιτρέπει να ελέγχεται το PL από τον κώδικα Python που τρέχει στο PS.

Μια Overlay περιλαμβάνει:

- Bitstream που χρησιμοποιείται για τη διαμόρφωση του FPGA.
- Ένα αρχείο Tcl που προσδιορίζει τον επιταχυντή.
- Python driver για τον έλεγχο του επιταχυντή.

2.2 High Level Synthesis

Το λογισμικό που χρησιμοποιήθηκε για τον προγραμματισμό του FPGA μέρους του κυκλώματος είναι το High Level Synthesis (HLS). Το HLS είναι μέρος της σουίτας SDSoC και η λειτουργία του είναι η μετατροπή μιας περιγραφής κώδικα C (C, C++, SystemC, OpenCL) σε περιγραφή Register Transfer Level (RTL), η σύνθεση της οποίας μπορεί να γίνει στο FPGA. Η χρήση υψηλότερου επιπέδου γλώσσας όπως η C++, σε σύγκριση με τις γλώσσες περιγραφής υλικού όπως οι VHDL-Verilog, προσφέρει πλεονεκτήματα όπως ευκολία στην ανάπτυξη εφαρμογών και μεγαλύτερη παραγωγικότητα. Επίσης δίνει τη δυνατότητα σε προγραμματιστές χωρίς προηγούμενη εμπειρία στην ανάπτυξη επιταχυντών υλικού να ωφεληθούν από τη χρήση αυτών των συσκευών.

Ο προγραμματισμός στο περιβάλλον του HLS επιτυγχάνεται μέσω οδηγιών (directives) που εισάγονται στον κώδικα C. Καθώς η εκτέλεση ενός προγράμματος σε C είναι από τη φύση του ακολουθιακή υπάρχουν πολλά directives, ώστε να εξαντλείται η δυνατότητα παράλληλης εκτέλεσης που μπορεί να προσφέρει το υλικό.

Τα βήματα που ακολουθούνται από το HLS κατά τη διαδικασία της σύνθεσης του κώδικα C σε RTL είναι τα παρακάτω:

- Χρονοπρογραμματισμός (Scheduling): Κατά τη διάρκεια αυτής της φάσης καθορίζεται ο ακριβής χρόνος εκτέλεσης των διεργασιών με βάση τις οδηγίες (directives) που εισάγει ο χρήστης. Όσο μεγαλύτερη είναι η περίοδος του ρολογιού, τόσες περισσότερες διεργασίες μπορούν να εκτελεστούν μέσα σε αυτή. Στη περίπτωση υψηλότερης συχνότητας οι διεργασίες “σπάζουν” σε περισσότερες περιόδους, ώστε να ικανοποιούνται οι χρονικές απαιτήσεις.
- Σύνδεση (Binding): Καθορίζονται τα στοιχεία hardware που θα εκτελέσουν κάθε διεργασία με βάση τους πόρους που έχει διαθέσιμη η συσκευή.
- Εξαγωγή της λογικής που ελέγχει το κύκλωμα (Control logic extraction): Ο έλεγχος της όλης εκτέλεσης οδηγείται από μια μηχανή πεπερασμένων καταστάσεων (finite state machine), η οποία συνθέτεται κατά τη φάση αυτή.

Για την αξιολόγηση της σύνθεσης ελέγχονται κάποιες βασικές μετρικές, τις οποίες εξάγει σε μορφή αναφοράς το HLS και είναι οι παρακάτω:

- Οι πόροι που χρησιμοποιήθηκαν από αυτούς που έχει διαθέσιμους η συσκευή FPGA. Τέτοιοι πόροι είναι οι LUTs, registers, BRAMs, DSP48s.
- Η καθυστέρηση (Latency) σε κύκλους ρολογιούς που απαιτείται, ώστε η συνάρτηση να δώσει αποτέλεσμα ή ένα loop να τελειώσει την εκτέλεση του.
- Το Διάστημα αρχικοποίησης (Initiation Interval - II) που είναι οι κύκλοι που απαιτούνται προκειμένου η συνάρτηση ή ένας βρόχος να μπορεί να δεχτεί νέα ορίσματα.

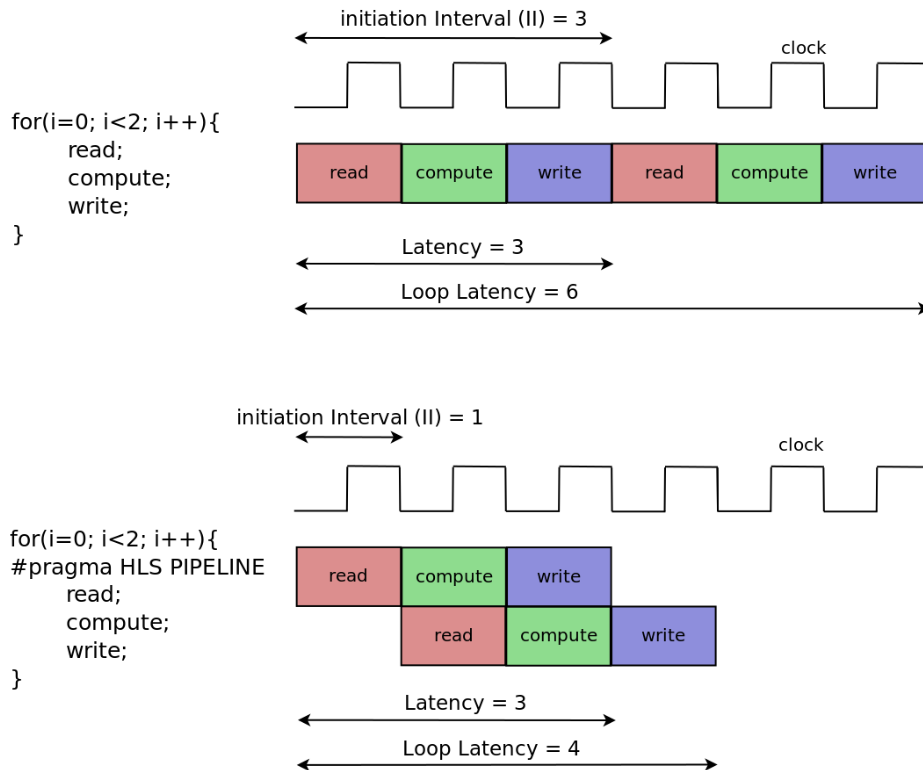
Η πιο σημαντική μετρική από την πλευρά της απόδοσης είναι το διάστημα αρχικοποίησης, το οποίο εκφράζει τη ρυθμαπόδοση (throughput). Το επιθυμητό είναι όσο το δυνατόν μικρότερο II ιδανικά ίσο με 1. Σε αυτή τη περίπτωση η συνάρτηση ή ο βρόχος επεξεργάζεται νέα δεδομένα σε κάθε κύκλο ρολογιού.

2.2.1 Τεχνικές βελτιστοποίησης στο HLS

Εδώ περιγράφονται οι βασικότερες τεχνικές προς επίτευξη καλύτερης απόδοσης της συνάρτησης που εκτελείται στο HW.

- Σωλήνωση βρόχου (Loop Pipelining) – Η εκτέλεση των βρόχων στις γλώσσες C/C++ γίνεται ακολουθιακά. Αυτό σημαίνει ότι πρέπει να τελειώσει μία επανάληψη του βρόχου για να ξεκινήσει η επόμενη. Στην τεχνική της σωλήνωσης βρόχου κάθε επανάληψη μοιράζεται σε διεργασίες που εκτελούνται σε έναν κύκλο ρολογιού και οι οποίες μπορούν να εκτελούνται παράλληλα. Για την εκμετάλλευση αυτής της τεχνικής το HLS διαθέτει την οδηγία #pragma HLS PIPELINE. Στο διάγραμμα παρακάτω

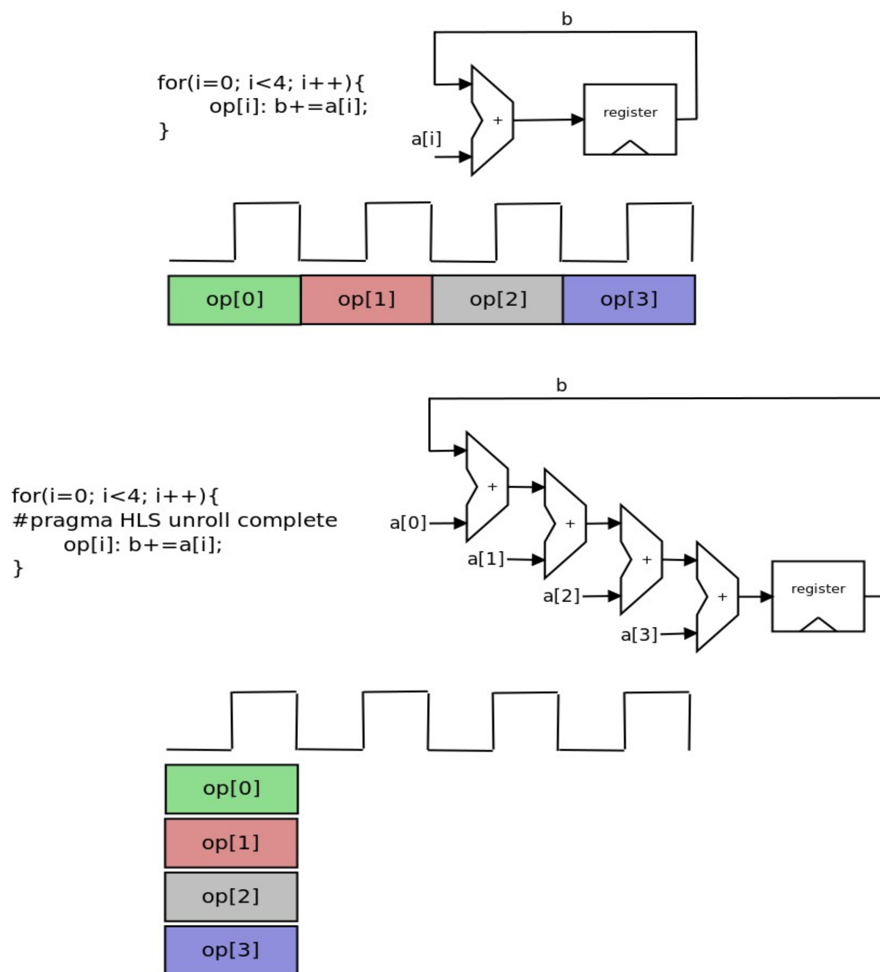
φαίνεται ένα απλό παράδειγμα σωλήνωσης βρόχου. Στην πρώτη περίπτωση, χωρίς την οδηγία PIPELINE, ο βρόχος εκτελείται ακολουθιακά με αποτέλεσμα να δέχεται νέα ορίσματα κάθε 3 κύκλους ρολογιού (II=3) και να ολοκληρώνει την εκτέλεση του σε 6. Στη δεύτερη περίπτωση με την εφαρμογή της σωλήνωσης, ο βρόχος δέχεται νέα ορίσματα σε κάθε κύκλο (II=1) και ολοκληρώνεται η εκτέλεση του μετά από 4.



2.2. Εικόνα: Σωλήνωση βρόχου (Loop Pipeline)

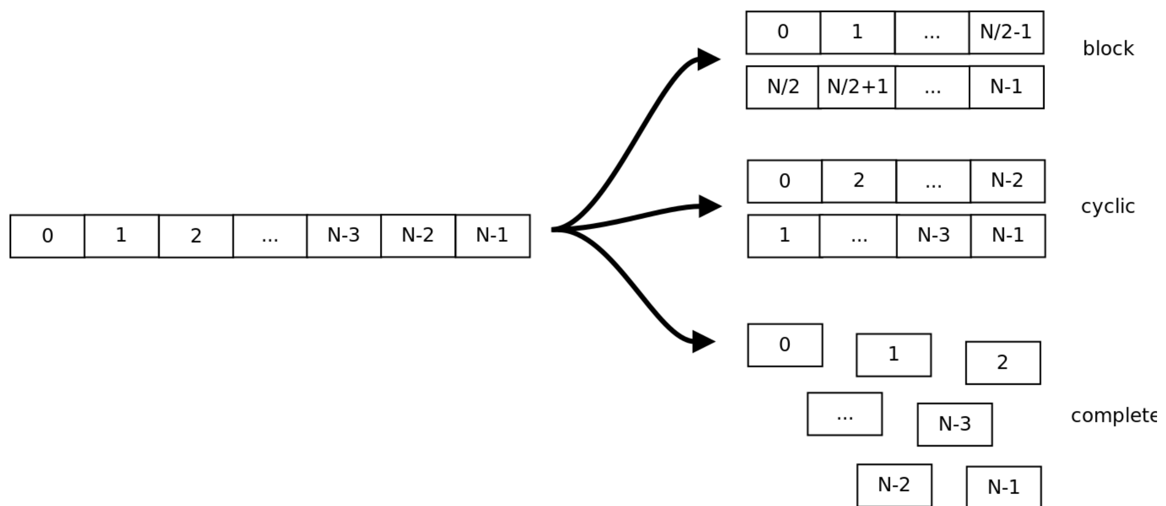
- Ξετύλιγμα βρόχου (Loop unrolling) – Η τεχνική του ξετυλίγματος βρόχου επιτρέπει την παράλληλη εκτέλεση περισσότερων του ενός διεργασιών του κυρίως σώματος του βρόχου. Η παράλληλη εκτέλεση επιτυγχάνεται με τη δημιουργία περισσοτέρων του ενός αντιγράφων των λογικών κυκλωμάτων που εκτελούν τις διεργασίες εντός του βρόχου. Η αντίστοιχη οδηγία του HLS είναι η #pragma HLS UNROLL [factor=N]. Αν N ισούται με τον αριθμό των επαναλήψεων του βρόχου τότε ο βρόχος ξετυλίγεται πλήρως (full unroll). Αν N είναι μικρότερο από τον αριθμό των επαναλήψεων ο βρόχος ξετυλίγεται μερικώς (partial unroll). Στην εικόνα 2.3 φαίνεται ένα παράδειγμα ξετυλίγματος βρόχου. Στην πρώτη περίπτωση ένας αθροιστής εκτελεί και τις 4 επαναλήψεις του βρόχου, μία σε κάθε κύκλο ρολογιού. Στην δεύτερη

περίπτωση η σύνθεση αποτελείται από 4 αθροιστές οι οποίοι εκτελούν σε μόλις ένα κύκλο ρολογιού όλο το βρόχο.



2.3. Εικόνα: Παράδειγμα ξετυλίγματος βρόχου (Loop unrolling)

- Διαμέριση πινάκων (Array partitioning) – Τα κυκλώματα των μνημών έχουν περιορισμένο αριθμό πυλών ανάγνωσης και εγγραφής. Αυτό μπορεί να οδηγήσει σε μείωση της απόδοσης αλγορίθμων με μεγάλο αριθμό προσπελάσεων στη μνήμη. Ο αριθμός των προσπελάσεων που μπορούν να συμβούν σε ένα κύκλο ρολογιού αυξάνεται στη περίπτωση που χρησιμοποιηθούν αντί του ενός κυκλώματος μνήμης, πολλά μικρότερα. Η οδηγία του HLS είναι η #pragma HLS ARRAY_PARTITION, ενώ υπάρχουν πολλές επιλογές για τον τρόπο διαίρεσης. Στο παρακάτω παράδειγμα φαίνονται οι τρεις βασικοί τρόποι διαίρεσης, εικόνα 2.4.



2.4. Εικόνα: Διαίρεση πινάκων (Array partitioning)

- Σωλήνωση ροής δεδομένων (Dataflow pipelining) – Όλες οι προηγούμενες τεχνικές πετύχαιναν παράλληλη εκτέλεση στο επίπεδο των τελεστών όπως πολλαπλασιασμών, προσθέσεων και προσπέλασης θέσεων μνήμης. Η τεχνική της σωλήνωσης στη ροή δεδομένων εφαρμόζεται στο επίπεδο των συναρτήσεων και των βρόχων. Η οδηγία είναι η #pragma HLS DATAFLOW. Παρακάτω φαίνεται ένα παράδειγμα της τεχνικής αυτής, εικόνα 2.5.

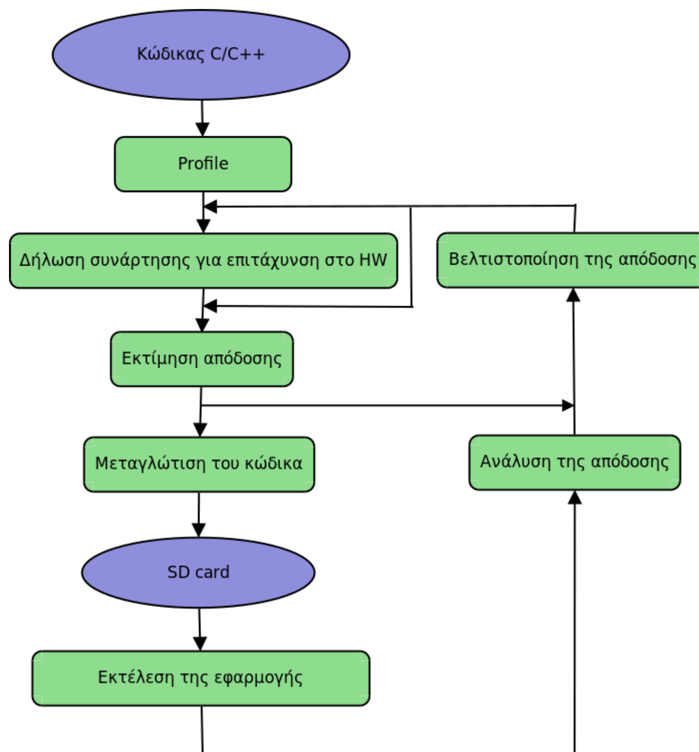
2.3 SDSoC

Η σουίτα SDSoC δίνει τη δυνατότητα προγραμματισμού των System-on-Chip με φυσικό τρόπο χωρίς να απαιτούνται μεγάλες τροποποιήσεις στον αρχικό κώδικα της εφαρμογής. Περιλαμβάνει μεταγλωττιστές που μετατρέπουν κώδικα γλώσσας C σε πλήρη συστήματα που εκτελούνται τόσο στο PS όσο και στο PL.

Τα βήματα για την ανάπτυξη εφαρμογών μέσω του SDSoC είναι τα παρακάτω:

- Ο προγραμματιστής ξεκινά με τον κώδικα C της εφαρμογής που θέλει να βελτιστοποιήσει. Με την μεταγλώττιση του πηγαίου κώδικα, ώστε να εκτελείται μόνο στον επεξεργαστή ARM, ο προγραμματιστής βεβαιώνεται ότι η εφαρμογή εκτελείται σωστά και παράγει τα αναμενόμενα αποτελέσματα.
- Στη συνέχεια μέσω profiling βρίσκει τα υπολογιστικά απαιτητικά κομμάτια, τα οποία είναι υποψήφια προς αποστολή στο FPGA. Τα τμήματα αυτά του κώδικα απομονώνονται μέσα σε συναρτήσεις.
- Τέλος εφαρμόζει τον SDSoC μεταγλωττιστή, ο οποίος παράγει κάρτα SD με την εφαρμογή.

Στην εικόνα 2.6 φαίνεται διάγραμμα που περιγράφει τη διαδικασία σχεδίασης.

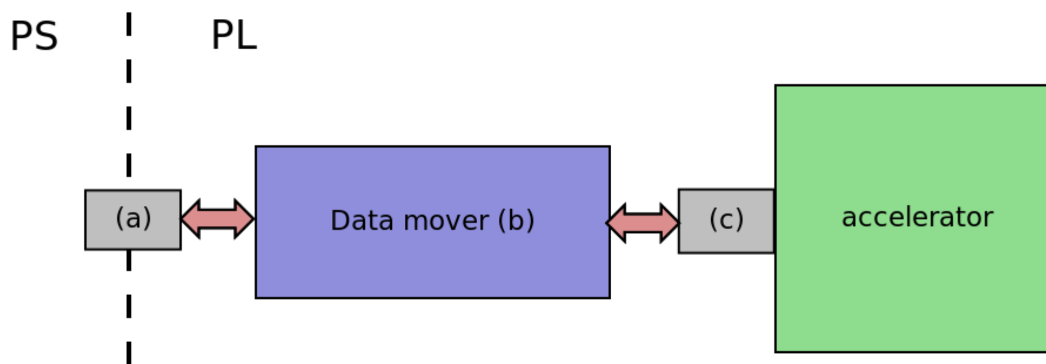


2.5.Εικόνα: Διαδικασία σχεδίασης στο SDSoC

2.3.1 Δίκτυο Μεταγωγής Δεδομένων (Data Motion Network)

Το SDSoC συνθέτει αυτόματα το δίκτυο μεταγωγής δεδομένων, το οποίο είναι υπεύθυνο για τη μεταφορά των δεδομένων από τις μνήμες του PS και τις εξωτερικές μνήμες στον πυρήνα και αντίστροφα. Αποτελείται από τρία στοιχεία

- Τη διεπαφή PS-PL (a).
- Τον μεταγωγέα (Data mover) μεταξύ του PS και του επιταχυντή (b).
- Την διεπαφή του επιταχυντή (c).



2.6. Εικόνα: Δίκτυο μεταγωγής δεδομένων (Data motion network)

Ένα από τα κριτήρια που ελέγχει το SDSoC προκειμένου να συνθέσει το κατάλληλο δίκτυο μεταγωγής δεδομένων είναι το είδος της μνήμης που προσπελάζει ο επιταχυντής. Η μνήμη που έχει κατανεμηθεί στον επιταχυντή μπορεί να είναι φυσικά συνεχής (contiguous memory) ή μη συνεχής σελιδοποιημένη (paged), cacheable ή non-cacheable. Η κατανομή φυσικά συνεχούς μνήμης γίνεται μέσω του API που παρέχει το SDSoC και το οποίο περιέχει και τις παρακάτω συναρτήσεις:

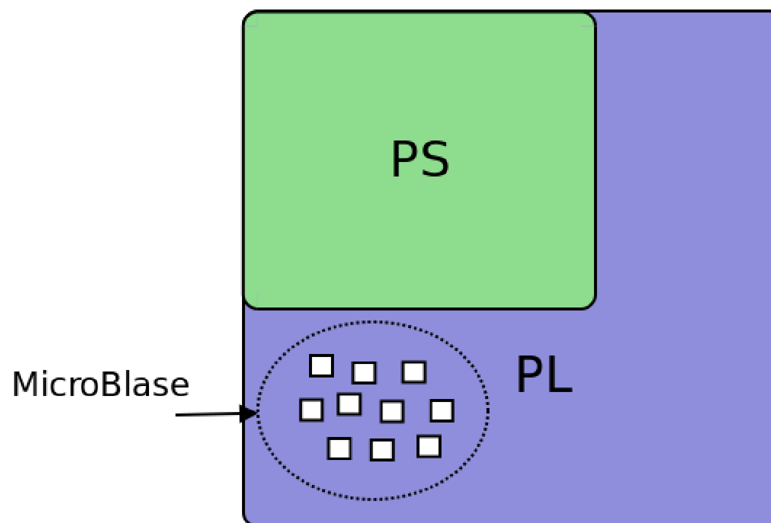
- void *sds_alloc(size_t size) – Κατανομή φυσικά συνεχούς μνήμης μεγέθους size bytes.
- void *sds_alloc_non_cacheable(size_t size) – Κατανομή φυσικά συνεχούς μνήμης. Στην περίπτωση αυτή δε γίνεται ενημέρωση της κρυφής μνήμης (cache).

2.4 7000 System-on-Chip

Το κύριο χαρακτηριστικό της σειράς Zynq-7000 είναι ότι συνδυάζει στο ίδιο ολοκληρωμένο κύκλωμα δύο διακριτές αρχιτεκτονικές, τη μονάδα επεξεργασίας (Processing System) και το τμήμα προγραμματισμένης λογικής (Programmable Logic). Η μονάδα επεξεργασίας είναι ένας ARM επεξεργαστής ενώ το τμήμα προγραμματισμένης λογικής είναι ένα κύκλωμα FPGA. Τα δύο αυτά τμήματα μπορούν να λειτουργούν σε συνεργασία ή ξεχωριστά.

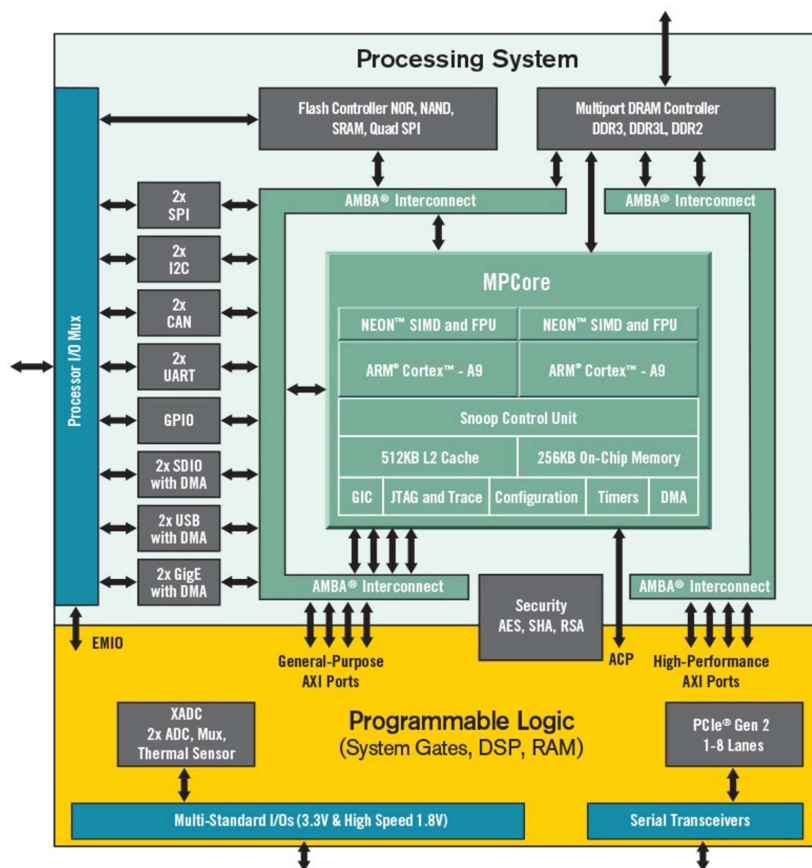
2.4.1 Η μονάδα επεξεργασίας (Processing System)

Η βασική μονάδα επεξεργασίας σε ένα κύκλωμα Zynq-7000 είναι ένας διπύρηνος επεξεργαστής ARM Cortex-A9. Ο επεξεργαστής αυτός χαρακτηρίζεται ως 'hard' επεξεργαστής, καθώς έχει υλοποιηθεί πάνω στο πυρίτιο και η αρχιτεκτονική των πυλών του είναι αμετάβλητη. Σε αντίθεση ένας 'soft' επεξεργαστής έχει υλοποιηθεί πάνω σε ένα σύστημα προγραμματισμένης λογικής (FPGA) με κατάλληλο συνδυασμό πυλών. Παράδειγμα 'soft' επεξεργαστή είναι ο Xilinx Microblaze. Η ύπαρξη του 'hard' επεξεργαστή δεν αποκλείει την χρήση ενός ή και πολλαπλών 'soft' επεξεργαστών ταυτόχρονα στο ίδιο κύκλωμα, εικόνα 2.8.



2.7.Εικόνα: Hard και Soft processors στον Zynq-7000

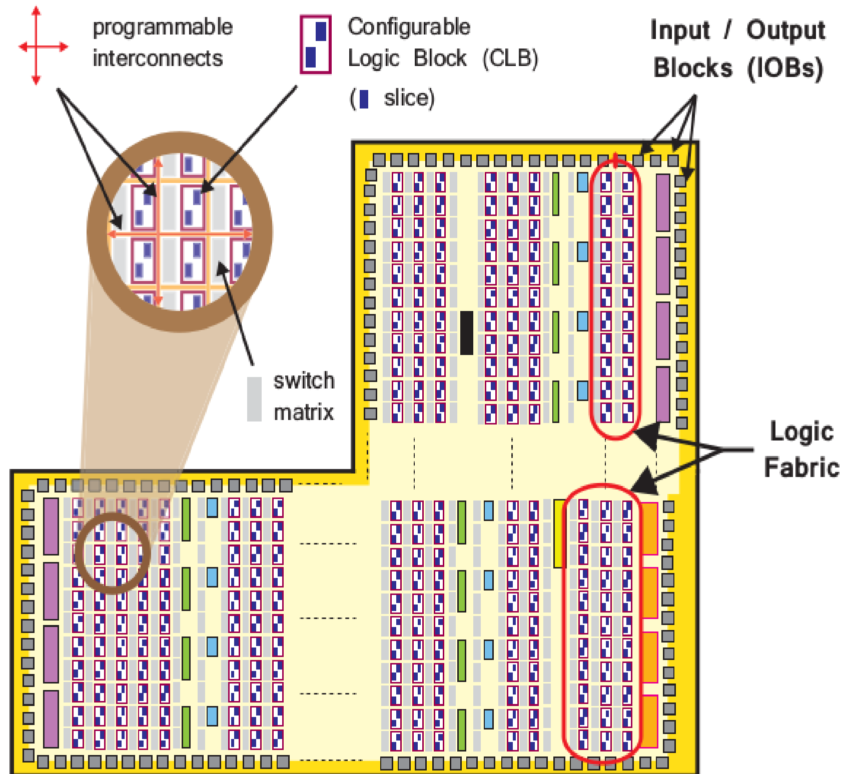
Τον επεξεργαστή ARM υποστηρίζουν και άλλες λογικές μονάδες οι οποίες αποτελούν το Application Processing Unit (APU). Εκτός του APU το PS ενσωματώνει και άλλες περιφερειακές μονάδες όπως κρυφές μνήμες (caches), διεπαφές για τη μνήμη και κύκλωμα για τη δημιουργία του σήματος ρολογιού, εικόνα 2.9.



2.8. Εικόνα: Σχεδιάγραμμα με τις υπομονάδες του Zynq-7000

2.4.2 Τμήμα προγραμματισμένης λογικής (Programmable Logic)

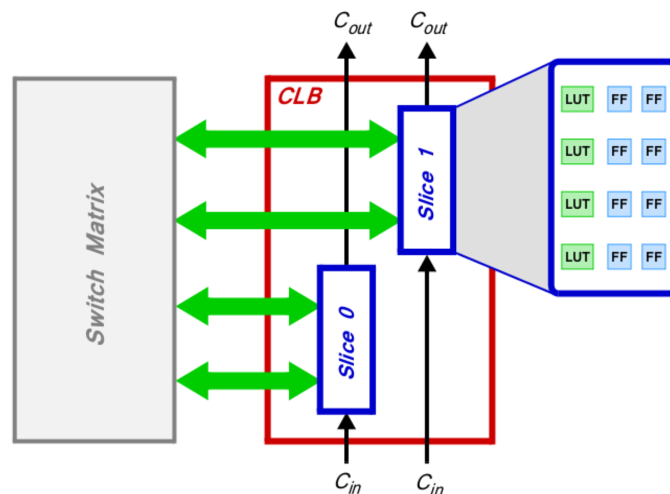
Το δεύτερο κυρίως τμήμα της αρχιτεκτονικής Zynq είναι η προγραμματιζόμενη λογική, η οποία βασίζεται στην αρχιτεκονική των Artix-7 και Kintex-7 FPGA της Xilinx. Το μεγαλύτερο μέρος του PL τμήματος αποτελούν γενικού σκοπού μονάδες επεξεργασίας τα διαμορφούμενα λογικά μπλόκα (Configurable Logic Blocks – CLB), τα οποία τοποθετούνται σε πίνακες των δύο διαστάσεων.



2.9. Εικόνα: Σχεδιάγραμμα του PL τμήματος

Στην εικόνα φαίνονται τα στοιχεία που αποτελούν το τμήμα PL και συνοπτικά είναι τα παρακάτω:

- Configurable Logic Blocks (CLBs) - Είναι το μικρότερο λογικό κύτταρο της δομής, συνδέεται με τα υπόλοιπα λογικά στοιχεία με αναδιαμορφωνόμενες καλωδιώσεις οι οποίες υλοποιούνται από τις μονάδες Switch Matrixes. Τα Lookup Tables (LUT), Flip-flops (FF) που συνθέτουν ένα CLB οργανώνονται σε δύο υπομονάδες Slice, όπως φαίνεται στην παρακάτω εικόνα.



2.10. Εικόνα: Configurable Logic Block (CLB)

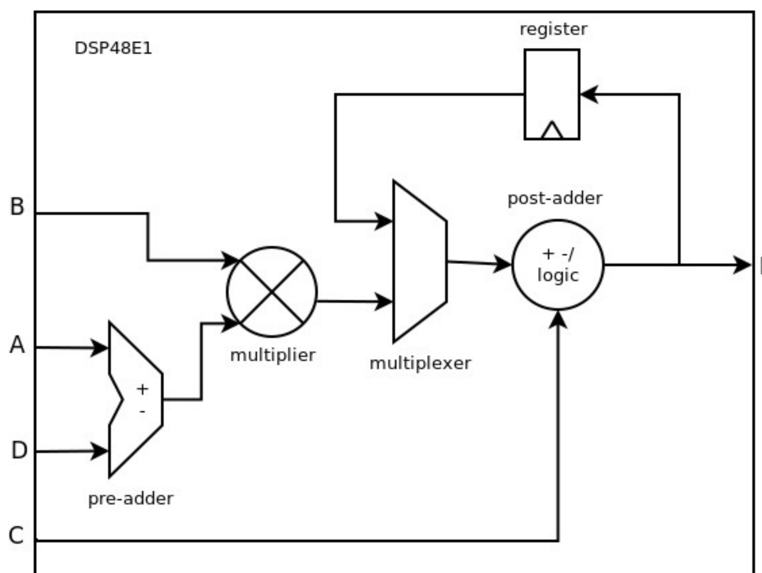
- Φέτες (Slices) - Οι υπομονάδες Slices των CLBs διαθέτουν πόρους για τη σύνθεση λογικών και ακολουθιακών κυκλωμάτων. Αποτελούνται από 4 lookup Tables και 8 Flip-Flops.
- lookup Tables (LUTs) - Περιέχονται στα Slices και μπορούν να διαμορφωθούν λογικές συναρτήσεις 6 μεταβλητών, σε μικρή μνήμη ROM ή RAM και σε καταχωρητές ολίσθησης (shift register). Επίσης μπορούν να συνδυαστούν ώστε να σχηματίσουν μεγαλύτερες λογικές μονάδες.
- Flip-Flops (FF) - είναι ακολουθιακά λογικά κυκλώματα που εκτελούν τη λειτουργία ενός καταχωρητή ολισθησης 1-bit με λειτουργία επανεκκίνησης (reset).
- Πίνακας Διακοπτών (Switch Matrix) - Δίπλα σε κάθε CLB βρίσκεται ένας Switch Matrix ο οποίος παρέχει τις συνδέσεις μεταξύ των στοιχείων εντός του CLB αλλά και με άλλους πόρους του PL.
- Λογικά κυκλώματα κρατουμένου (Carry logic) – Είναι απαραίτητα στοιχεία στην υλοποίηση αριθμητικών κυκλωμάτων που χρειάζονται ενδιάμεσα σήματα κρατουμένου για να λειτουργήσουν. Τέτοια κυκλώματα μπορούν να δημιουργηθούν με το συνδυασμό γειτονικών slices.
- Κυκλώματα εισόδου/εξόδου (Input/Output Blocks-IOBs) – Τα κυκλώματα αυτά χρησιμοποιούνται ως διεπαφές μεταξύ των λογικών στοιχείων του PL και των ακροδεκτών του κυκλώματος. Οι ακροδέκτες λειτουργούν ως σύνδεση με τα εξωτερικά λογικά κυκλώματα. Ένα IOB μπορεί να χειριστεί σήματα ενός bit ενώ είναι τοποθετημένα περιφερειακά του PL.

Εκτός των προηγούμενων στοιχείων το τμήμα προγραμματιζόμενης λογικής ενσωματώνει και μονάδες ειδικών εφαρμογών, οι οποίες είναι τμήματα μνήμης RAM (Block RAMs) για απαιτητικές εφαρμογές σε μνήμη και DSP48E1 για επιτάχυνση των αριθμητικών πράξεων. Τα παραπάνω στοιχεία βρίσκονται σε διάταξη στήλης και σε κοντινή απόσταση μεταξύ τους, ώστε να ικανοποιείται η απαίτηση της μεγάλης ταχύτητας στις εφαρμογές που χρησιμοποιούνται.

- Block RAMs – Κάθε BRAM μπορεί να αποθηκεύσει έως 36Kb πληροφορίας, ενώ διαμορφώνεται είτε σε μία RAM των 36Kb, είτε σε δύο ανεξάρτητες μνήμες RAM των 18Kb η κάθε μία. Εξ ορισμού το μήκος λέξης είναι 18 bits και με αυτή τη διαμόρφωση η μνήμη διαθέτει 2048 στοιχεία μνήμης, όμως το μήκος της λέξης μπορεί να μεταβληθεί ανάλογα με τις απαιτήσεις. Εναλλακτικά των BRAMs μπορούν να χρησιμοποιηθούν κατανεμημένες μνήμες οι, οποίες συντίθενται από LUTs που διανέμονται σε μια μεγάλη περιοχή του PL. Η διαφορά τους είναι ότι οι τελευταίες είναι

πολύ πιο αργές στην προσπέλασή λόγω της μεγάλης απόστασης των στοιχείων τους, όμως πλεονεκτούν όταν η μνήμη που καλούνται να υλοποιήσουν είναι μικρότερη σε μέγεθος.

- Τα στοιχεία DSP48E1 διαθέτουν μονάδες πολλαπλασιασμού (multipliers), πρόσθεσης (adders) καθώς και καταχωρητές (registers), ώστε να μπορούν να υλοποιήσουν μεγάλο πλήθος αριθμητικών πράξεων και λογικών συναρτήσεων. Μπορούν να υποστηρίξουν δυναμική μεταβολή της λογικής συνάρτησης που εκτελούν, δηλαδή σε κάθε κύκλο ρολογιού να μεταβάλουν τη διαμόρφωσή τους ανάλογα με τον υπολογισμό που καλούνται να εκτελέσουν. Τα DSP48E1 είναι βελτιστοποιημένα, ώστε να μπορούν να υποστηρίζουν τη μέγιστη συχνότητα ρολογιού. Τέλος μπορούν να συνδυαστούν, ώστε να υλοποιούν μεγαλύτερα κυκλώματα. Στην εικόνα 2.12 φαίνεται απλοποιημένο διάγραμμα μιας μονάδας DSP48E1.



2.11. Εικόνα: Απλοποιημένο διάγραμμα DSP48E1

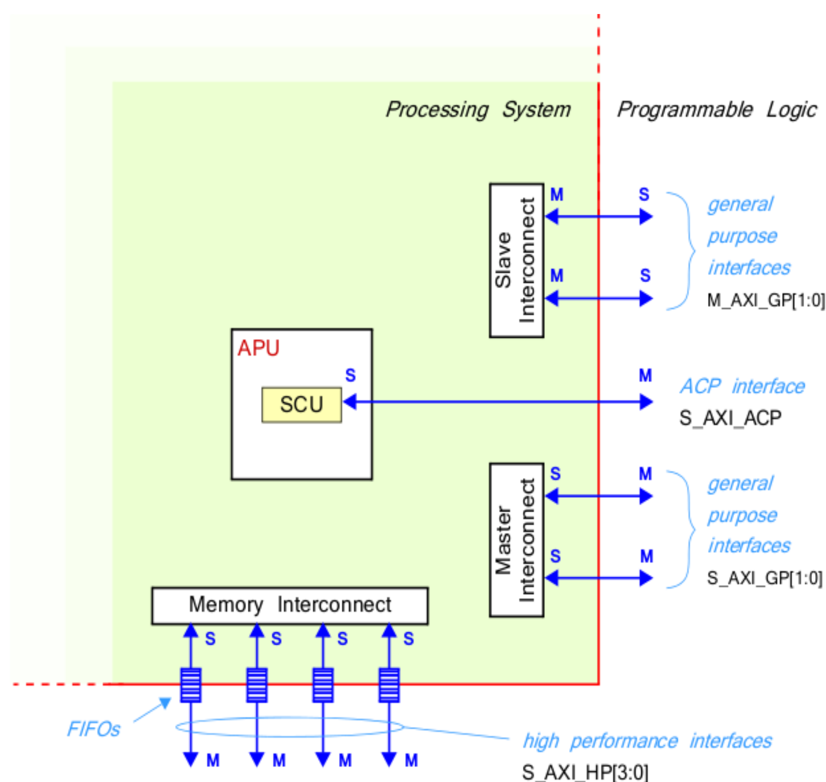
2.4.3 Οι διεπαφές PS-PL

Σημαντικό κομμάτι για την κατανόηση του τρόπου επικοινωνίας μεταξύ των PS και PL τμημάτων του κυκλώματος είναι οι διεπαφές που βασίζονται στο πρότυπο Advanced eXtensible Interface (AXI). Υπάρχουν τρεις διαφορετικοί τύποι τέτοιων διεπαφών:

- Γενικού σκοπού δίαυλοι AXI (General purpose AXI): Είναι δίαυλοι 32-bit και που προσφέρουν χαμηλή ή μέση ταχύτητα επικοινωνίας μεταξύ των PS-PL. Η διεπαφή υλοποιείται χωρίς ενδιάμεσο buffer (direct connection). Υπάρχουν τέσσερις

διαφορετικές τέτοιες διεπαφές όπου στις δύο είναι master το PS ενώ στις άλλες δύο το PL.

- Accelerator Coherency Port: Υπάρχει μόνο ένας τέτοιος δίαυλος 64-bit στο κύκλωμα και συνδέεται άμεσα με το Snoop Control Unit (SCU) μέσα στο APU, ώστε να διατηρεί συνοχή μεταξύ των κρυφών μνημών (caches) και του PL. Σε αυτόν το δίαυλο το PL είναι ο master.
- Υψηλής απόδοσης δίαυλοι (High Performance Ports): Είναι δίαυλοι 32 ή 64-bit, οι οποίοι περιλαμβάνουν δομές FIFOs, ώστε να μπορούν να υποστηρίξουν υψηλή ταχύτητα μεταφοράς δεδομένων. Και στους τέσσερις διαύλους master είναι το PL.



2.12. Εικόνα: Διεπαφές PS-PL

2.5 Η βιβλιοθήκη μηχανικής μάθησης Scikit-learn

Η Scikit-learn είναι μια βιβλιοθήκη μηχανικής μάθησης γραμμένη στη γλώσσα υψηλού επιπέδου Python. Είναι ανοιχτού-κώδικα και “πατάει” πάνω στα πακέτα NumPy, SciPy και matplotlib. Προσφέρει εργαλεία για διάφορους αλγορίθμους μηχανικής μάθησης, μεταξύ αυτών και του k-Nearest Neighbors μέσω των κλάσεων που περιέχονται στο πακέτο sklearn.neighbors. Οι κλάσεις αυτές μπορούν να χειριστούν είτε NumPy arrays είτε scipy.sparce matrices σαν είσοδο.

2.5.1 Unsupervised Nearest Neighbors

Η κλάση NearestNeighbors προσφέρει τρεις διαφορετικούς αλγορίθμους για την εφαρμογή του k-NN. Οι τρεις αυτοί αλγόριθμοι είναι οι Balltree, KDTree και brute-force. Ο ποιο απλός είναι ο τελευταίος από τους τρεις. Σε αυτόν υπολογίζονται όλες οι αποστάσεις μεταξύ των σημείων του συνόλου εκπαιδεύσεως και δοκιμών. Επομένως αν N είναι το πλήθος του συνόλου των δεδομένων και D η διάσταση του χώρου το προβλήματος (feature-space) τότε η υπολογιστική πολυπλοκότητα είναι

$$O[DN^2]$$

Στην περίπτωση μικρού συνόλου δεδομένων αυτή η προσέγγιση μπορεί να είναι ανταγωνιστική των άλλων δύο. Οι δύο άλλοι αλγόριθμοι χρησιμοποιούν tree-based δομές δεδομένων προκειμένου να μειώσουν την ποσότητα υπολογισμών που απαιτούνται.

3 Παρουσίαση των αλγόριθμων

3.1 Ο αλγόριθμος k κοντινότερων σημείων k-NN

Ο αλγόριθμος k-nearest neighbors είναι από τους πιο απλούς αλγορίθμους μηχανικής μάθησης. Ανήκει στην κατηγορία των supervised learning αλγορίθμων και παρόλο την απλότητά του μπορεί να δώσει καλά αποτελέσματα σε μεγάλη πληθώρα προβλημάτων. Υπάρχουν μέθοδοι για κατηγοριοποίηση (classification) και παλινδρόμηση (regression). Σε κάθε περίπτωση ο αλγόριθμος δέχεται ως είσοδο τα σημεία που αντιπροσωπεύουν το σύνολο δοκιμής (training set) και αυτά που αντιπροσωπεύουν το σύνολο εκπαιδεύσεως (testing set) στον ν-διάστατο χώρο του προβλήματος.

Όταν εφαρμόζεται η μέθοδος της κατηγοριοποίησης η έξοδος είναι η κλάση στην οποία ανήκει το σημείο του συνόλου δοκιμής. Η κατηγοριοποίηση του σημείου του συνόλου δοκιμής στην κατάλληλη κλάση γίνεται ελέγχοντας μεταξύ των k κοντινότερων σε αυτό σημείων του συνόλου εκπαιδεύσεως τη κλάση πλειοψηφίας, δηλαδή αυτή στην οποία ανήκουν τα περισσότερα σημεία.

Στην περίπτωση της παλινδρόμησης τα σημεία εξόδου δεν ανήκουν σε κλάσεις με διακριτές τιμές αλλά τους προσδιορίζεται μια ιδιότητα με συνεχή τιμή. Ο προσδιορισμός της τιμής αυτής για τα σημεία του συνόλου δοκιμής γίνεται υπολογίζοντας το μέσο όρο των τιμών των ιδιοτήτων των k κοντινότερων σημείων του συνόλου εκπαιδεύσεως.

3.1.1 Περιγραφή του αλγορίθμου

Για μια πιο μαθηματικώς αυστηρή περιγραφή του αλγορίθμου το σύνολο εκπαιδεύσεως αποτελείται από διανύσματα στο ν-διάστατο χώρο (feature space), τα οποία συνοδεύονται από την τιμή της κλάσης. Αν το σύνολο των τιμών των κλάσεων είναι το C ενώ το πλήθος του συνόλου εκπαιδεύσεως (training set) είναι m προκύπτουν οι δυάδες

$$(X_1, Y_1), (X_{2,} Y_2), \ldots, (X_m, Y_m) \in R^n x C$$

Έστω σημείο x του συνόλου δοκιμής (testing-set) και δοθέντος κατάλληλης μετρικής της απόστασης $\|.\|$ για τον ν-διάστατο χώρο του προβλήματος, ο αλγόριθμος αναζητά τα k κοντινότερα διανύσματα του συνόλου εκπαιδεύσεως (training set)

$$\|X_{(1)} - x\| \leq \ldots \|X_{(k)} - x\|$$

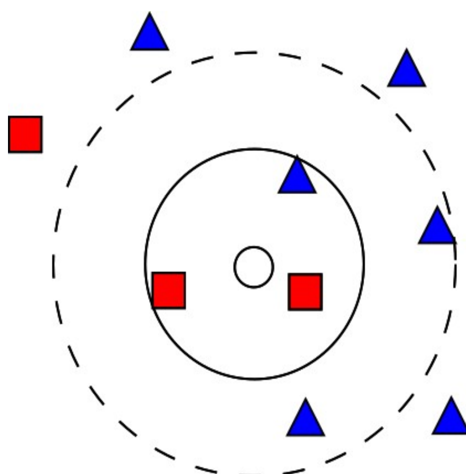
και εν συνεχεία προσδίδει στο σημείο x την ποιο πολυάριθμη κλάση μεταξύ των $X_{(1)}, \ldots, X_{(k)}$.

Η ποιο κοινή μετρική απόστασης που χρησιμοποιείται σε χώρους συνεχών μεταβλητών είναι η ευκλείδεια. Για διακριτές τιμές όπως στη περίπτωση αναγνώρισης κειμένων χρησιμοποιούνται αποστάσεις όπως οι Hamming.

Παραλλαγές του αλγορίθμου που περιγράφηκε μπορεί να προκύψουν δίνοντας βάρη στους k κοντινότερους γείτονες, έτσι ώστε να συνεισφέρουν ανάλογα με το πόσο κοντά βρίσκονται στο σημείο προς κατάταξη.

Η επιλογή της παραμέτρου k εξαρτάται από τα δεδομένα προς επεξεργασία. Σε γενικές γραμμές επιλογή μεγαλύτερου k μειώνει το φαινόμενο του θορύβου αλλά μειώνει και την ακρίβεια, καθώς κάνει τα όρια μεταξύ των κλάσεων δυσδιάκριτα. Καλές τιμές για το k μπορούν να βρεθούν χρησιμοποιώντας διάφορες τεχνικές. Στη περίπτωση επιλογής μεταξύ μόνο δύο κλάσεων μια καλή στρατηγική είναι η επιλογή περιττού k, ώστε να αποφεύγονται οι ισοπαλίες.

Στην εικόνα 3.1 φαίνονται δύο παραδείγματα του αλγορίθμου. Και στις δύο περιπτώσεις ζητείται να ταξινομηθεί το σημείο στο κέντρο που παριστάνεται από έναν κύκλο. Για k=3 τα κοντινότερα σημεία περιέχονται στον κύκλο με τη συνεχή διαγράμμιση και η πλειοψηφία τους είναι κόκκινα. Επομένως ταξινομείται στην κόκκινη κλάση. Στη δεύτερη περίπτωση k=5 και τα πέντε κοντινότερα σημεία περιέχονται στον κύκλο με τη διακεκομμένη διαγράμμιση. Σε αυτή τη περίπτωση όμως η πλειοψηφία των σημείων είναι μπλε επομένως το σημείο ταξινομείται στη μπλε κλάση.



3.1. Εικόνα: Απλό παράδειγμα ταξινόμησης με k-NN

3.2 Πολλαπλασιασμός Πινάκων στον k-NN

Στην πιο απλή εκδοχή του k-NN, μέρος του αλγορίθμου είναι ο υπολογισμός του πίνακα αποστάσεων μεταξύ των συνόλων εκπαίδευσης και του συνόλου δοκιμών. Έστω οι πίνακες των σημείων εκπαίδευσης και δοκιμών X, Y αντίστοιχα. Αν το πλήθος των στοιχείων του συνόλου εκπαιδεύσεως και του συνόλου δοκιμών είναι m και r αντίστοιχα, ενώ η διάσταση του χώρου του προβλήματος είναι n τότε οι διαστάσεις των δύο πινάκων είναι X[m][n], Y[r][n]. Η απόσταση μεταξύ των σημείων X_i, Y_j ορίζεται όπως παρακάτω

$$D_{ij} = \|X_i - Y_j\|$$

Για τον k-NN είναι αναγκαία μόνο η σχετική ταξινόμηση των αποστάσεων, επομένως στην περίπτωση που χρησιμοποιείται ως μετρική η ευκλείδεια απόσταση συμφέρει ο υπολογισμός του τετραγώνου της απόστασης.

$$D_{ij} = \|X_i - Y_j\|^2 = X_i X_i^T - 2 X_i Y_j^T + Y_j Y_j^T \Rightarrow$$

$$D = \underbrace{\begin{bmatrix} X_0 X_0^T & X_0 X_0^T \ldots & X_0 X_0^T \\ X_1 X_1^T & X_1 X_1^T \ldots & X_1 Y_1^T \\ & \vdots & \\ X_{m-1} X_{m-1}^T & X_{m-1} X_{m-1}^T \ldots & X_{m-1} X_{m-1}^T \end{bmatrix}}_{Array dimension m \times r} \text{-}2XY^T + \underbrace{\begin{bmatrix} Y_0 Y_0^T & Y_1 Y_1^T \ldots & Y_{r-1} Y_{r-1}^T \\ Y_0 Y_0^T & Y_1 Y_1^T \ldots & Y_{r-1} Y_{r-1}^T \\ & \vdots & \\ Y_0 Y_0^T & Y_1 Y_1^T \ldots & Y_{r-1} Y_{r-1}^T \end{bmatrix}}_{Array dimension m \times r}$$

Όπως φαίνεται στην παραπάνω αριθμητική παράσταση ο πίνακας D προκύπτει από την πρόσθεση τριών μερών. Για το πρώτο και τρίτο προσθετέο απαιτούνται m και r

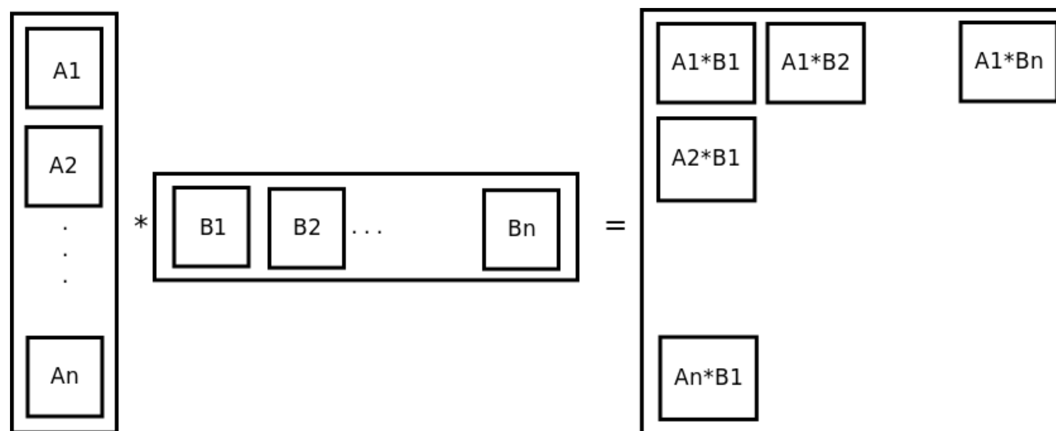
εσωτερικά γινόμενα διανυσμάτων αντίστοιχα ή m*n και r*n βαθμωτά γινόμενα. Το πιο βαρύ υπολογιστικά κομμάτι είναι ο πολλαπλασιασμός των πινάκων X, Y^T, καθώς σε αυτή τη περίπτωση απαιτείται ο υπολογισμός m*r εσωτερικών γινομένων ή m*n*r βαθμωτών γινομένων. Αναμένεται λοιπόν ο μεγαλύτερος χρόνος κατά τον υπολογισμό του πίνακα αποστάσεων να αναλώνεται στον πολλαπλασιασμό των δύο πινάκων.

3.2.1 Πολλαπλασιασμός πινάκων με Υποπίνακες

Ο τρόπος αυτός πολλαπλασιασμού συνίσταται στην διαίρεση των πινάκων A, B σε υποπίνακες ίδιου μεγέθους και στη συνέχεια πολλαπλασιασμού των υποπινάκων κατάλληλα ώστε να σχηματιστεί το τελικό γινόμενο. Παρακάτω παρατίθεται η μαθηματική διατύπωση όπου με $\mathbf{A_i}$ αναφέρονται οι υποπίνακες.

$$AB = \begin{bmatrix} \boldsymbol{A_1} \\ \boldsymbol{A_2} \\ \vdots \\ \boldsymbol{A_N} \end{bmatrix} \begin{bmatrix} \boldsymbol{B_1} & \boldsymbol{B_2} \dots \boldsymbol{B_M} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A_1 B_1} & \boldsymbol{A_1 B_2} \dots \boldsymbol{A_1 B_M} \\ \boldsymbol{A_2 B_1} & \boldsymbol{A_2 B_2} \dots \boldsymbol{A_2 B_M} \\ & \ddots \\ \boldsymbol{A_N B_1} & \boldsymbol{A_N B_2} \dots \boldsymbol{A_N B_M} \end{bmatrix}$$

Ο πολλαπλασιασμός με υποπίνακες μπορεί να προσφέρει ταχύτερες υλοποιήσεις καθώς εκμεταλλεύεται καλύτερα τις προσπελάσεις στη μνήμη λόγω τοπικότητας των δεδομένων. Στη περίπτωση των FPGAs μπορεί να αναπτυχθεί ένας βελτιστοποιημένος πυρήνας που να εκτελεί τον πολλαπλασιασμό πινάκων συγκεκριμένων διαστάσεων. Στη συνέχεια η γενίκευση για αυθαίρετες διαστάσεις πινάκων γίνεται με τον πολλαπλασιασμό των υποπινάκων. Στην παρακάτω εικόνα δίδεται ο τρόπος αυτός πολλαπλασιασμού.



3.2. Εικόνα: Σχηματική αναπαράσταση πολλαπλασιασμού με υποπίνακες

4 Υλοποίηση αλγορίθμων στο Hardware

4.1 Μεθοδολογία ανάπτυξης

Στην παράγραφο αυτή αναλύεται η μεθοδολογία που ακολουθήθηκε στη παρούσα διπλωματική. Το πρώτο απαραίτητο βήμα είναι η ανάλυση του κώδικα μέσω κατάλληλου λογισμικό, ώστε να εντοπιστούν τα υπολογιστικά απαιτητικά κομμάτια. Η ανάλυση αυτή έγινε με τη βοήθεια του πακέτου cProfile. Στη συνέχεια αφού εντοπίστηκε η υποψήφια συνάρτηση προς μεταφορά στο PL, ξαναγράφηκε στη γλώσσα C, ώστε να γίνει δυνατή η σύνθεση μέσω του HLS. Προκειμένου να είναι δυνατή η διαδικασία της βελτιστοποίησης πρέπει να γίνει σύγκριση των αποτελεσμάτων με αυτά που παράγει ο αντίστοιχος κώδικας, ο οποίος εκτελείται στον επεξεργαστή ARM. Με τη σύγκριση αυτή εκτός από την ορθότητα των αποτελεσμάτων ελέγχεται και η επιτάχυνση που επιτυγχάνεται. Το πρόγραμμα αυτό το οποίο εκτελεί τους δύο κώδικες στον επεξεργαστή και στον επιταχυντή, εξάχθηκε μέσω του λογισμικού SDSoC σαν αυτόνομη (standalone) εφαρμογή. Στη συνέχεια ακολουθήθηκε το “πακετάρισμα” των οδηγών του επιταχυντή σαν shared object ώστε να είναι δυνατή η κλήση τους από την Python.

Πρέπει να σημειωθεί ότι καθώς η αναπτυξιακή πλακέτα PYNQ-Z1 δεν υποστηρίζεται από το SDSoC χρειάστηκε να ακολουθηθεί η διαδικασία για την ανάπτυξη βασικής πλατφόρμας (base platform). Η βασική πλατφόρμα είναι το σχέδιο πάνω στο οποίο στηρίζονται οι υπόλοιπες υλοποιήσεις των εφαρμογών και περιέχει τους ορισμούς των ρολογιών στο PL, του υλικού που χειρίζεται τις εξαιρέσεις στο PL καθώς και των διεπαφών PS-PL.

4.2 Ανάλυση του κώδικα - Profiling

Κατά την ανάπτυξη ενός αλγορίθμου στο Hardware ο σχεδιαστής έρχεται αντιμέτωπος με μια σειρά κρίσιμων σχεδιαστικών αποφάσεων. Δεδομένου ότι έχει

αποφασισθεί ο αλγόριθμος και η βιβλιοθήκη στην οποία θα εφαρμοσθεί η μεθοδολογία ανάπτυξης και στην περίπτωση αυτή είναι ο k-NN και η Scikit-learn αντίστοιχα, το επόμενο ερώτημα είναι ποιο "σημείο" του αλγορίθμου θα "ωφεληθεί" από την επιτάχυνση που μπορεί να προσφέρει το FPGA. Προκειμένου να απαντηθεί το ερώτημα αυτό, είναι απαραίτητο να γίνει ανάλυση του κώδικα και των συναρτήσεων που καλεί ως προς το χρόνο που χρειάζεται κάθε συνάρτηση για να ολοκληρωθεί. Η ανάλυση αυτή έγινε με τη βοήθεια του πακέτου cProfile. Το αποτέλεσμα της ανάλυσης κατά την εκτέλεση του αλγορίθμου σε τεχνητό σύνολο δεδομένων φαίνεται στην παρακάτω εικόνα.

```
neigh = KNeighborsClassifier(n_neighbors=10, algorithm = 'brute')
neigh.fit(Train, Train_labels)
cProfile.run('predicted = neigh.predict(Test)','restats')

         583 function calls in 17.650 seconds

   Ordered by: cumulative time
   List reduced from 125 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.045    0.045   17.650   17.650 {built-in method builtins.exec}
        1    0.001    0.001   17.605   17.605 <string>:1(<module>)
        1    0.210    0.210   17.605   17.605 classification.py:133(predict)
        1    0.000    0.000   17.241   17.241 base.py:317(kneighbors)
        2    0.014    0.007   17.230    8.615 pairwise.py:1141(pairwise_distances_chunked)
        1    0.000    0.000   10.124   10.124 pairwise.py:1290(pairwise_distances)
        1    0.000    0.000   10.124   10.124 pairwise.py:1061(_parallel_pairwise)
        1    1.631    1.631   10.123   10.123 pairwise.py:164(euclidean_distances)
        1    0.000    0.000    8.466    8.466 extmath.py:149(safe_sparse_dot)
        1    8.466    8.466    8.466    8.466 {built-in method numpy.core.multiarray.dot}
```

4.1. Εικόνα: Profiling του k-NN

Η λειτουργία κάθε συνάρτησης εξηγείται ως εξής.

- Kneighbors – Η συνάρτηση αυτή επιστρέφει πίνακα με τους k κοντινότερους γείτονες για κάθε σημείο του συνόλου δοκιμών (testing set).
- Pairwise - Οι συναρτήσεις που περιέχουν στο όνομα τους το pairwise επιστρέφουν τον πίνακα αποστάσεων μεταξύ των training και testing sets, ενώ καλούνται διαδοχικά κάνοντας διάφορους ελέγχους στα ορίσματα που δέχονται.
- Euclidean-distances - Επιστρέφει τον πίνακα των ευκλείδειων αποστάσεων.
- safe_sparce_dot - Εκτελεί πολλαπλασιασμό πινάκων και μπορεί να χειριστεί με ασφάλεια τόσο αραιούς πίνακες του SciPy όσο και πυκνούς numpy-arrays.

Από τα παραπάνω είναι φανερό ότι ο περισσότερος υπολογιστικός χρόνος καταναλώνεται στον υπολογισμό του πίνακα αποστάσεων.

Προκειμένου να καλυφθούν οι στόχοι που τέθηκαν στη παράγραφο 1.1, δηλαδή η φυσική υλοποίηση και εφαρμογή σε όσο το δυνατόν περισσότερες συναρτήσεις,

επιλέχθηκε να επιταχυνθεί η συνάρτηση dot του πακέτου NumPy. Η συνάρτηση αυτή εκτελεί πολλαπλασιασμό πινάκων και καλείται από τη safe_sparce_dot. Όπως θα εξηγηθεί παρακάτω ο σχεδιασμός ενός επιταχυντή που θα εκτελεί πολλαπλασιασμό πινάκων στο υλικό και θα μπορεί να δέχεται ως όρισμα αυθαίρετο μέγεθος πινάκων είναι μεν εφικτός, όμως σε καμία περίπτωση δε θα μπορεί να προσφέρει την επιτάχυνση ενός πυρήνα που είναι σχεδιασμένος να δέχεται συγκεκριμένο μέγεθος πινάκων. Αυτό συμβαίνει γιατί στη δεύτερη περίπτωση μπορεί να γίνει εκμετάλλευση όλων των τεχνικών παράλληλων υπολογισμών που μπορεί να προσφέρει το FPGA. Επομένως η τρίτη απαίτηση της παραγράφου 1.1 που είναι και η ελαστικότητα ως προς τα ορίσματα μπορεί να ικανοποιηθεί μόνο μερικώς.

4.3 Πολλαπλασιασμός Πινάκων

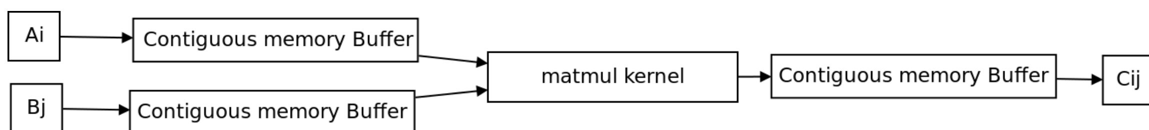
4.3.1 Ανάλυση των τρόπων υλοποίησης

Ο πολλαπλασιασμός πινάκων είναι ένας από τους ποιο κλασικούς αλγόριθμους που μπορούν να ωφεληθούν από την υλοποίηση τους στο Hardware. Για την αποδοτική του υλοποίηση θα πρέπει ο σχεδιαστής να έχει γνώση των πλεονεκτημάτων αλλά και των περιορισμών που προκύπτουν από την χρήση του FPGA. Όπως αναφέρθηκε και προηγουμένως για να γίνει πλήρη εκμετάλλευση των δυνατοτήτων παράλληλης επεξεργασίας, θα πρέπει να τεθούν περιορισμοί στο μέγεθος των πινάκων που μπορούν να δοθούν ως ορίσματα. Επομένως θα πρέπει να γίνει αρκετά στοχευμένη εφαρμογή. Ένας δεύτερος περιορισμός που προκύπτει είναι η πολύ μικρότερη ταχύτητα προσπέλασης των δεδομένων από τη συσκευή του FPGA σε σχέση με τον επεξεργαστή. Αυτό συμβαίνει γιατί το FPGA στερείται της ιεραρχίας μνημών, την οποία διαθέτει η επεξεργαστική μονάδα. Επιπλέον θα πρέπει τα δεδομένα προς επεξεργασία να βρίσκονται σε συνεχή μνήμη, ώστε να γίνεται ταχύτατη προσπέλαση μέσω DMA.

Οι πίνακες τους οποίους θα κληθεί να διαχειριστεί η συνάρτηση μπορεί να περιέχουν χιλιάδες στοιχεία. Επομένως είναι αδύνατο να μπορούν να φορτωθούν εξολοκλήρου στο FPGA λόγω περιορισμένων πόρων. Σε αυτή την περίπτωση η καλύτερη στρατηγική είναι ο πολλαπλασιασμός να γίνει χωρίζοντας τους αρχικούς πίνακες σε υποπίνακες. Οι υποπίνακες αυτοί έχουν τη διάσταση των πινάκων που δέχεται ο επιταχυντής και αποστέλλονται σε αυτόν στη σωστή σειρά έως ότου επιτευχθεί ο πολλαπλασιασμός.

Η μνήμη στην οποία αποθηκεύονται τα δεδομένα, τα οποία επεξεργάζεται στη συνέχεια ο πυρήνας είναι απαραίτητο να είναι συνεχής. Η φυσική συνέχεια της μνήμης εξασφαλίζει την ταχύτατη ανάγνωση και αντιγραφή των δεδομένων από τον πυρήνα. Σε αντίθετη περίπτωση σπαταλάτε χρόνος στη μετάφραση των εικονικών θέσεων μνήμης λόγω σελιδοποίησης.

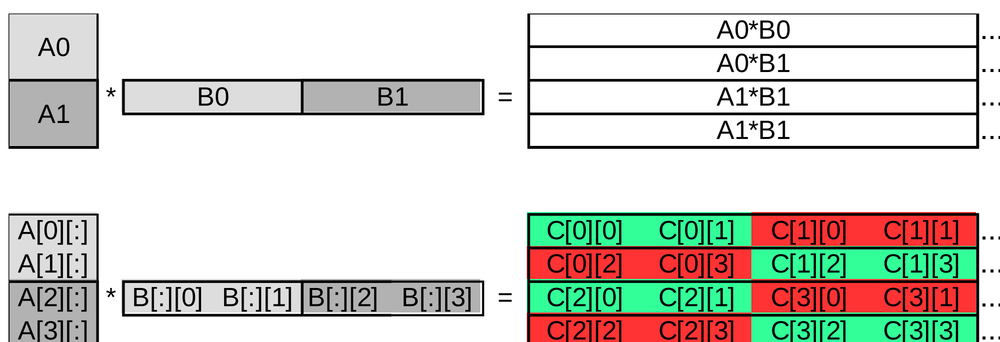
Η αρχική σχεδίαση ήταν πριν την τελική τους επεξεργασία τα τμήματα των πινάκων A, B να αποθηκεύονται σε buffers συνεχούς μνήμης και εν συνεχεία να στέλνονται προς επεξεργασία, όπως φαίνεται στην εικόνα 4.2.



4.2. Εικόνα: Buffers συνεχούς μνήμης

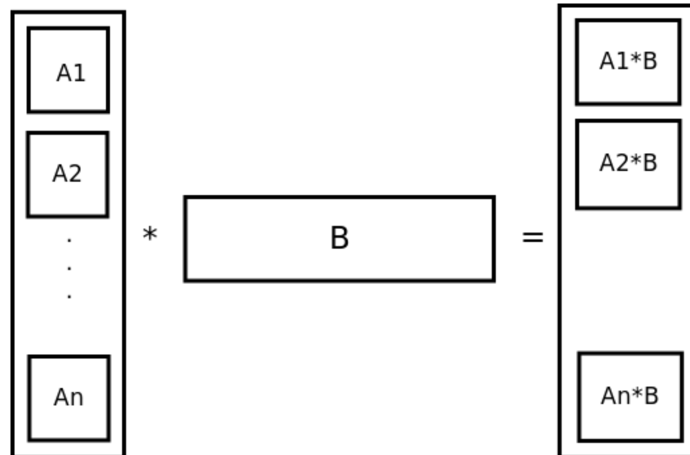
Αυτή η προσέγγιση αποδείχθηκε ότι δεν ήταν αποδοτική καθώς εισήγαγε μεγάλες καθυστερήσεις κατά την αντιγραφή των πινάκων από τη κύρια μνήμη προς τα buffers συνεχούς μνήμης.

Ένας επιπλέον περιορισμός ήταν η σειρά που τοποθετούνται τα στοιχεία του πίνακα C στη μνήμη. Αποδεικνύεται ότι κατά τον τρόπο αυτό του πολλαπλασιασμού με υποπίνακες, τα στοιχεία του πίνακα εξόδου, έστω C, δεν τοποθετούνται με τη σωστή σειρά και λόγο αυτού απαιτείται αναδιάταξη των στοιχείων στη μνήμη. Αποτέλεσμα είναι η εισαγωγή επιπλέον καθυστέρησης. Στην εικόνα 4.3 φαίνεται ένα απλό παράδειγμα που δείχνει πως τοποθετούνται τα στοιχεία του πίνακα C στην απλή περίπτωση όπου οι πίνακες A[4][N] και B[N][4] πολλαπλασιάζονται. Οι πίνακες είναι χωρισμένοι σε 2 υποπίνακες ο καθένας και τα αποτελέσματα του πολλαπλασιασμού των υποπινάκων τοποθετούνται στη συνεχή μνήμη κατά σειρά εξόδου, όπως φαίνεται στο πάνω μέρος της εικόνας. Στο κάτω μέρος με κόκκινο παριστάνονται τα στοιχεία του πίνακα που έχουν τοποθετηθεί λάθος και με πράσινο αυτά που έχουν τοποθετηθεί σωστά.



4.3. Εικόνα: Απλό παράδειγμα που δείχνει την τοποθέτηση των στοιχείων κατά τον πολλαπλασιασμό υποπινάκων

Προκειμένου να ξεπεραστούν τα προβλήματα που αναφέρθηκαν επιλέχτηκε οι πίνακες A, B και C, να φορτώνονται εξολοκλήρου σε Buffers συνεχούς μνήμης. Επιπλέον για να τηρηθεί η σωστή σειρά εγγραφής των αποτελεσμάτων στον πίνακα C χωρίς να υπάρχει ανάγκη επαναδιάταξης, αποφασίστηκε μόνο ο πίνακας A να στέλνεται τμηματικά προς επεξεργασία ενώ ο πίνακας B να φορτώνεται εξολοκλήρου στον επιταχυντή.



4.4. Εικόνα: Πολλαπλασιασμός πινάκων με φόρτωση ολόκληρου του πίνακα B

Ο πηγαίος κώδικας αυτού του τρόπου πολλαπλασιασμού φαίνεται παρακάτω. Η συνάρτηση bmatmul_fixed_B δέχεται σαν ορίσματα τους πίνακες A, B, C τον αριθμό των σειρών του πίνακα A (A_n_rows), τον αριθμό των στηλών των δύο πινάκων (AB_n_col) και τον αριθμό των σειρών του B (B_n_rows). Η συνάρτηση που εκτελείται στο HW είναι η matmul_accel. Ο βρόχος εκτελείται στον επεξεργαστή ARM ο οποίος καλεί την συνάρτηση HW. Η σταθερά MAX_ROWS_OF_SUB_A αναφέρεται στο μέγιστο αριθμό σειρών που μπορεί να έχει ο υποπίνακας A, με βάση τις προδιαγραφές του επιταχυντή.

```
void  bmatmul_fixed_B(float  *A,  float  *B,  float  *C,  int  A_n_rows,  int  A_n_col,  int
B_n_rows)
{
    int i;
    if (A_n_rows <= MAX_ROWS_OF_SUB_A){
      matmul_accel(A, B, C, A_n_rows%MAX_ROWS_OF_SUB_A, A_n_col, B_n_rows);
    }
    else if(A_n_rows%MAX_ROWS_OF_SUB_A==0){
       for(i=0; i<A_n_rows; i+=MAX_ROWS_OF_SUB_A){
          matmul_accel(&A[i*A_n_col],  B,  &C[i*B_n_rows],  MAX_ROWS_OF_SUB_A,  A_n_col,
B_n_rows);
        }
     }
     else{
        for(i=0; i<A_n_rows-MAX_ROWS_OF_SUB_A; i+=MAX_ROWS_OF_SUB_A){
```

```
            matmul_accel(&A[i*A_n_col], B, &C[i*B_n_rows], MAX_ROWS_OF_SUB_A, A_n_col, B_n_rows);
        }
      matmul_accel(&A[i*A_n_col], B, &C[i*B_n_rows], A_n_rows%MAX_ROWS_OF_SUB_A, A_n_col, B_n_rows);
     }
}
```

4.3.2 Αρχιτεκτονική του πυρήνα

Ο πυρήνας σχεδιάστηκε ώστε να δέχεται μέγιστο μέγεθος πινάκων A[256][64], B[512][64], με τον πίνακα B να είναι ανάστροφος σε σχέση με το συνήθη αλγόριθμο πολλαπλασιασμού. Ο πίνακας C[256][512] είναι το αποτέλεσμα του πολλαπλασιασμού των δύο ορισμάτων εισόδου. Οι πίνακες A, B αρχικά φορτώνονται στις μνήμες BRAM του FPGA, ώστε να είναι δυνατή η προσπέλαση των στοιχείων τους κατά τυχαίο τρόπο, όπως απαιτεί ο αλγόριθμος. Προκειμένου να υποστηρίζονται πίνακες μικρότερων διαστάσεων οι πίνακες “παραγεμίζονται” με 0.0 για τα στοιχεία στα οποία δεν αντιστοιχούν δεδομένα. Ο κώδικας με τα directives των HLS και SDSoC είναι ο παρακάτω.

```
#pragma SDS data access_pattern(A:SEQUENTIAL, B:SEQUENTIAL, C:SEQUENTIAL)
#pragma SDS data copy(A[0:rows_of_A*col_of_AB], B[0:rows_of_B*col_of_AB], C[0:rows_of_A*rows_of_B])
#pragma SDS data mem_attribute(A:PHYSICAL_CONTIGUOUS, B:PHYSICAL_CONTIGUOUS, C:PHYSICAL_CONTIGUOUS)
#pragma SDS data data_mover(A:AXIDMA_SIMPLE, B:AXIDMA_SIMPLE, C:AXIDMA_SIMPLE)
void matmul_accel(float *A, float *B, float *C, int rows_of_A, int col_of_AB, int rows_of_B)
{
    float buf_of_A[MAX_ROWS_OF_A][MAX_COL_OF_AB], buf_of_B[MAX_ROWS_OF_B][MAX_COL_OF_AB];
#pragma HLS array_partition variable=buf_of_A block factor=16 dim=2
#pragma HLS array_partition variable=buf_of_B block factor=16 dim=2

#pragma HLS DATAFLOW
LOOP1:for(int i=0; i<MAX_ROWS_OF_A; i++) {
         for(int j=0; j<MAX_COL_OF_AB; j++) {
#pragma HLS PIPELINE
                 if ((i<rows_of_A) & (j<col_of_AB)){
                         buf_of_A[i][j] = A[i * col_of_AB + j];
                 }
                 else{
                         buf_of_A[i][j] = 0.0;
                 }
         }
    }

 LOOP2:for(int i=0; i<MAX_ROWS_OF_B; i++) {
        for(int j=0; j<MAX_COL_OF_AB; j++) {
#pragma HLS PIPELINE
                if ((i<rows_of_B) & (j<col_of_AB)){
                        buf_of_B[i][j] = B[i * col_of_AB + j];
                }
                else{
```

```
                        buf_of_B[i][j] = 0.0;
                }
            }
        }

 LOOP3:for (int i = 0; i < MAX_ROWS_OF_A; i++) {
        for (int j = 0; j < MAX_ROWS_OF_B; j++) {
              float result = 0.0;
#pragma HLS PIPELINE
              for (int k = 0; k < MAX_COL_OF_AB; k++) {
                  float term = buf_of_A[i][k] * buf_of_B[j][k];
                  result += term;
              }
              if (i<rows_of_A & j<rows_of_B){
                  C[i * rows_of_B + j] = result;
              }
        }
    }
}
```

Τα SDS directives τοποθετούνται πριν από τον ορισμό της συνάρτησης και καθορίζουν τη σύνθεση του δικτύου μεταγωγής δεδομένων (Data motion network).

#pragma SDS data copy - Γνωστοποιεί ότι οι πίνακες θα αντιγραφούν από την κύρια μνήμη εντός PS στον επιταχυντή. Επιπλέον πληροφορεί το SDSoC για το αριθμό των δεδομένων που θα αντιγραφούν στο PL. Εναλλακτικά υπάρχει το directive zero_copy όπου σε αυτή τη περίπτωση ο επιταχυντής προσπελάζει τα δεδομένα απευθείας από τη μοιραζόμενη μνήμη (shared memory).

#pragma SDS access_pattern - δίνει τη πληροφορία στο SDSoC ότι οι προσπελάσεις των πινάκων A, B, C γίνονται κατά ακολουθιακό τρόπο. Έτσι ο συνθέτεται μια δομή FIFO για κάθε ένα από τος τρεις πίνακες.

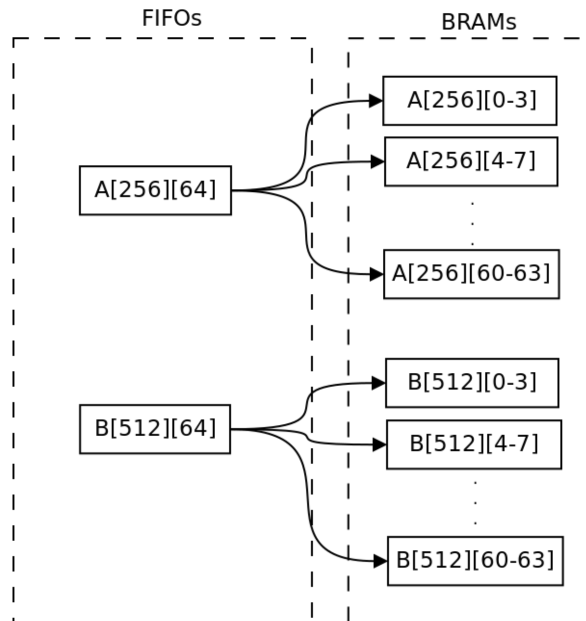
#pragma SDS mem_attribute - Δηλώνει αν οι πίνακες βρίσκονται σε contiguous/paged memory. Στην περίπτωση αυτή καθορίζεται ότι βρίσκονται σε contiguous memory.

#pragma SDS data_mover - καθορίζεται ο μεταγωγέας δεδομένων που θα χρησιμοποιηθεί για τη μεταφορά των δεδομένων. Εδώ είναι ο AXIDMA_SIMPLE.

Στον κυρίως κώδικα της συνάρτησης υπάρχουν τρεις βασικοί βρόχοι LOOP1, 2, 3. Οι δύο πρώτοι φορτώνουν του πίνακες A και B αντίστοιχα στους buf_of_A, buf_of_B, που είναι στοιχεία BRAMs εντός του επιταχυντή. Αυτό, όπως εξηγήθηκε και στη προηγούμενη παράγραφο, επιτρέπει την τυχαία προσπέλαση των δεδομένων από τον τρίτο βρόχο ο οποίος εκτελεί τον αλγόριθμο πολλαπλασιασμού πινάκων.

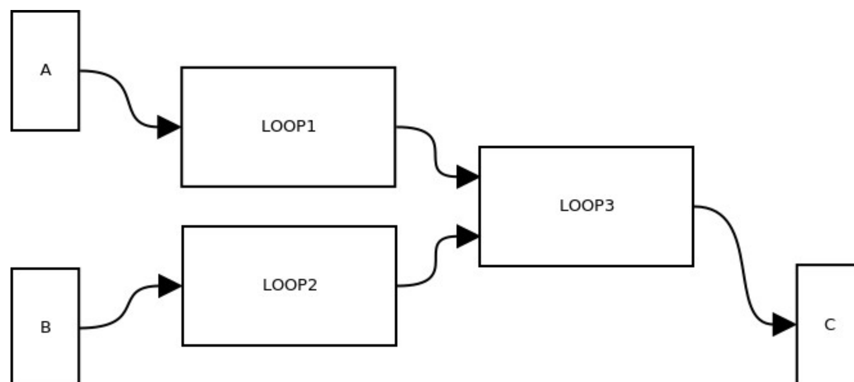
#pragma HLS array_partition - Στον ορισμό των buf_of_A, buf_of_B εφαρμόζεται το HLS directive array_partition. Λόγω αυτής της οδηγίας οι στήλες των δύο πινάκων μοιράζονται σε 16 διαφορετικά στοιχεία BRAMs, εικόνα 4.5. Κάθε

στοιχείο BRAM μπορεί να υποστηρίξει δύο προσπελάσεις στον ίδιο κύκλο ρολογιού. Επομένως μπορούν να υποστηριχθούν 32 προσπελάσεις στοιχείων των buf_of_A, buf_of_B ταυτόχρονα. Αυτό είναι απαραίτητο ώστε να επιτευχθεί η pipeline λειτουργία του κυκλώματος με όσο το δυνατόν μικρότερο διάστημα αρχικοποίησης (Initiation Interval).



Εικόνα 4.5: Εφαρμογή της οδηγίας array_partition

#pragma HLS DATAFLOW - Το HLS directive dataflow επιτρέπει τη σωλήνωση (pipeline) σε επίπεδο συναρτήσεων ή βρόχων. Στην περίπτωση των δύο πρώτων βρόχων τα δεδομένα που παράγονται είναι ασυσχέτιστα μεταξύ τους και έτσι εκτελούνται παράλληλα όπως φαίνεται και στην εικόνα 4.6.



4.6. Εικόνα: Σωλήνωση σε επίπεδο βρόχων

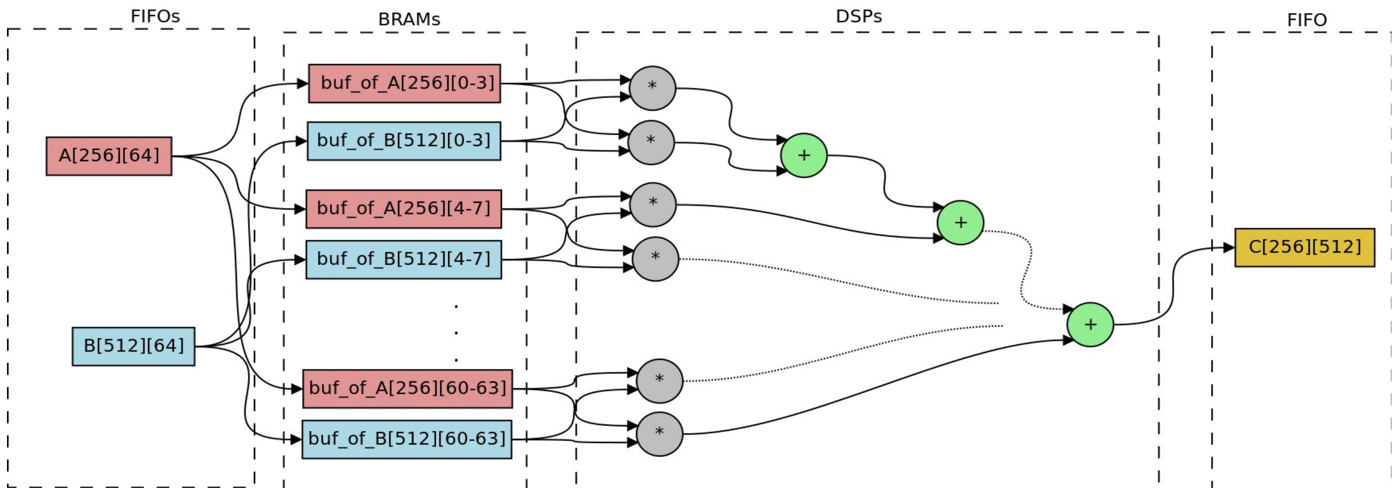
#pragma HLS PIPELINE - Το HLS directive PIPELINE επιτρέπει τη εκτέλεση των βρόχων με τη τεχνική της σωλήνωσης (PIPELINE). Στους πρώτους δύο βρόχους, οι οποίοι απλά φορτώνουν τους πίνακες A, B στις εσωτερικές μνήμες, δεν υπάρχει καμία εξάρτηση μεταξύ των δεδομένων, είτε εντός της ίδιας επανάληψης, είτε μεταξύ διαφορετικών επαναλήψεων. Επομένως II=1, που είναι και το επιθυμητό.

Στον τελευταίο βρόχο η οδηγία PIPELINE τοποθετείται πριν το τελευταίο for η λειτουργία του οποίου είναι να δίνει το εσωτερικό γινόμενο μεταξύ δύο στηλών των δύο πινάκων. Το HLS "ξεδιπλώνει" όλους τους βρόχους (loop unrolling) μετά την οδηγία PIPELINE, ούτως ώστε να επιτύχει τον πολλαπλασιασμό των στηλών ακολουθιακά. Προκείμενου να επιτευχθεί όσο το δυνατόν μικρότερο II θα πρέπει όσο το δυνατόν περισσότερα στοιχεία από τις στήλες προς πολλαπλασιασμό να είναι διαθέσιμα εντός ενός κύκλου ρολογιού. Ιδανικά καθώς το νούμερο των στηλών είναι 64 θα πρέπει να είναι και τα 64 στοιχεία διαθέσιμα εντός ενός κύκλου. Στην περίπτωση που εξετάζεται, όπως εξηγήθηκε και στο directive array_partition, επιλέχθηκε να είναι διαθέσιμα 32 στοιχεία ανά κύκλο ρολογιού προκειμένου να επαρκέσουν οι πόροι του FPGA σε στοιχεία DSPs. Αυτό κατά τη σύνθεση οδήγησε σε II=2. Η κατανομή των πόρων του FPGA που χρησιμοποιήθηκαν φαίνεται στον παρακάτω πίνακα.

4.1.Πίνακας: Κατανομή των πόρων του FPGA για τη συνάρτηση matmul_accel

	Αριθμός στοιχείων	Ποσοστό
BRAM_18K	96	34%
DSP48E	160	72%
FF	24130	22%
LUT	34040	63%

Ολόκληρη η αρχιτεκτονική του επιταχυντή όπως τον έχει συνθέσει το HLS φαίνεται στην εικόνα 4.7.



4.7.Εικόνα: Αρχιτεκτονική του επιταχυντή

4.4 Εξαγωγή των οδηγών

Για να επικοινωνήσει ο πυρήνας που βρίσκεται φορτωμένος στο FPGA είναι απαραίτητοι οι οδηγοί (Drivers). Η σουήτα SDSoC εξάγει αυτόματα τους οδηγούς της συνάρτησης που εκτελείται στο HW με βάση τις οδηγίες (directives) που δίνονται στον πηγαίο κώδικα. Η συνάρτηση που δηλώνεται ότι εκτελείται στο HW αντικαθίσταται από μια “stub” συνάρτηση, η οποία ρυθμίζει την αποστολή των ορισμάτων στον πυρήνα. Ο τροποποιημένος κώδικας από το SDSoC φαίνεται παρακάτω (Ο αντίστοιχος πηγαίος κώδικας παρουσιάστηκε στη παράγραφο 4.3.1). Η συνάρτηση matmul_accel έχει αντικατασταθεί αυτόματα από το SDSoC, με τη _p0_matmul_accel_1_noassync.

```
void  bmatmul_fixed_B(float  *A,  float  *B,  float  *C,  int  A_n_rows,  int  A_n_col,  int
B_n_rows)
{
    int i;
    if (A_n_rows <= MAX_ROWS_OF_SUB_A){
      _p0_matmul_accel_1_noasync(A, B, C, A_n_rows%MAX_ROWS_OF_SUB_A, A_n_col, B_n_rows);
    }
    else if(A_n_rows%MAX_ROWS_OF_SUB_A==0){
       for(i=0; i<A_n_rows; i+=MAX_ROWS_OF_SUB_A){
         _p0_matmul_accel_1_noasync(&A[i*A_n_col],  B,  &C[i*B_n_rows],  MAX_ROWS_OF_SUB_A,
A_n_col, B_n_rows);
        }
     }
     else{
        for(i=0; i<A_n_rows-MAX_ROWS_OF_SUB_A; i+=MAX_ROWS_OF_SUB_A){
         _p0_matmul_accel_1_noasync(&A[i*A_n_col],  B,  &C[i*B_n_rows],  MAX_ROWS_OF_SUB_A,
A_n_col, B_n_rows);
        }
```

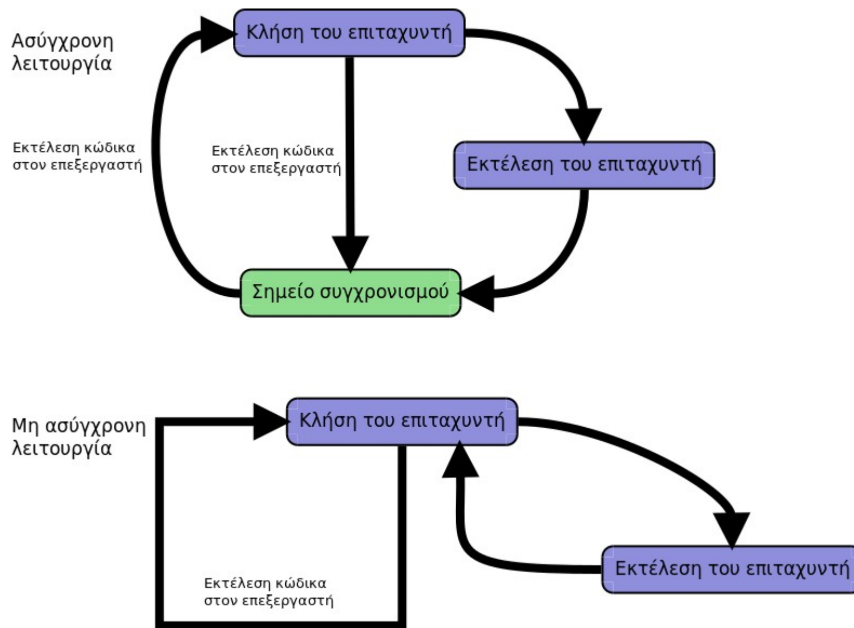
```
                _p0_matmul_accel_1_noasync(&A[i*A_n_col],   B,   &C[i*B_n_rows],   A_n_rows
%MAX_ROWS_OF_SUB_A, A_n_col, B_n_rows);
     }
}
```

Ο αυτοματοποιημένος κώδικας που δημιούργησε το SDSoC για τη συνάρτηση stub που παίζει τον ρόλο του οδηγού είναι ο παρακάτω:

```
void _p0_matmul_accel_1_noasync(float A[16384], float B[32768], float C[131072])
{
  switch_to_next_partition(0);
  int start_seq[1];
  start_seq[0] = 0;
  cf_request_handle_t _p0_swinst_matmul_accel_1_cmd;
   cf_send_i(&(_p0_swinst_matmul_accel_1.cmd_matmul_accel), start_seq, 1 * sizeof(int),
&_p0_swinst_matmul_accel_1_cmd);
  cf_wait(_p0_swinst_matmul_accel_1_cmd);

  cf_send_i(&(_p0_swinst_matmul_accel_1.A), A, 65536, &_p0_request_0);
  cf_send_i(&(_p0_swinst_matmul_accel_1.B), B, 131072, &_p0_request_1);
  cf_send_i(&(_p0_swinst_matmul_accel_1.load_B), &load_B, 4, &_p0_request_3);

cf_receive_i(&(_p0_swinst_matmul_accel_1.C),C, 524288, &_p0_matmul_accel_1_noasync_num_C,
&_p0_request_2);

  cf_wait(_p0_request_0);
  cf_wait(_p0_request_1);
  cf_wait(_p0_request_2);
  cf_wait(_p0_request_3);
}
```

Ο ρόλος της συνάρτησης cf_send_i είναι να στέλνει τον ακριβή αριθμό δεδομένων στον πυρήνα. Η συνάρτηση cf_receive_i λαμβάνει το αποτέλεσμα της συνάρτησης που στη περίπτωση αυτή είναι ο πίνακας C.

Εδώ πρέπει να αναφερθεί ότι το μοντέλο εκτέλεσης είναι μη ασύγχρονο. Αυτό "αντανακλάται" και στην προσθήκη στην ονομασία της stub συνάρτησης του συνθετικού "noasync". Σε αυτό το μοντέλο εκτέλεσης, ο επεξεργαστής περιμένει τον επιταχυντή να δώσει αποτελέσματα πριν συνεχίσει την εκτέλεση του υπόλοιπου βρόχου. Ο συγχρονισμός επιτυγχάνεται μέσω της συνάρτησης cf_wait στον κώδικα της stub συνάρτησης. Μέχρι την έξοδο της συνάρτησης stub ο επεξεργαστής παραμένει ανενεργός. Εναλλακτικά στο ασύγχρονο μοντέλο εκτέλεσης, αφού γίνει κλήση της συνάρτησης HW ο επεξεργαστής συνεχίζει την εκτέλεση μέχρι του σημείου συγχρονισμού στον κώδικα. Το ασύγχρονο και μη ασύγχρονο μοντέλο εκτέλεσης παρουσιάζεται στην εικόνα 4.8.



4.8. Εικόνα: Ασύγχρονο και μη ασύγχρονο μοντέλο εκτέλεσης

4.5 Σύνδεση με Python

Για τη σύνδεση με τη γλώσσα Python χρησιμοποιήθηκε το πακέτο CFFI. Προκείμενου να μπορεί να κληθεί μέσω του πακέτου CFFI, η συνάρτηση bmatmul_fixed_B εξάχθηκε σαν shared object. Στο παράρτημα Α υπάρχει ο πλήρης κώδικας σε Python ο οποίος καλεί τη συνάρτηση stub. Η λειτουργία των μεθόδων της κλάσης mmult_hw εξηγείται ως εξής:

- __init__- Στη γραμμή 37 γίνεται η δήλωση της συνάρτησης bmatmul_fixed_B, ώστε να μπορεί να κληθεί μέσω του πακέτου cffi σαν συνάρτηση Python. Στην αμέσως επόμενη γραμμή φορτώνεται το shared object στο οποίο έχει “πακεταριστεί” η συνάρτηση bmatmul_fixed_B.
- download_bitstream - Φορτώνει το bitstream αρχείο στο FPGA μέσω της κλήσης της κλάσης Overlay του PYNQ.
- Matmul - Είναι η συνάρτηση που εκτελεί τον πολλαπλασιασμό πινάκων.
- numpynq_buffer – Αρχικοποιεί τα buffers συνεχούς μνήμης.
- numpynq_buffers_reset – Διαγράφει τα buffers που δεν χρειάζονται. Λόγω του ότι τα buffers συνεχούς μνήμης βρίσκονται στο χώρο kernel του λειτουργικού συστήματος είναι σημαντικό να διαγράφονται μόλις πάψει η χρήση τους.

- numpynq_buffers_stats – Δίνει στατιστικά για τα buffers που χρησιμοποιούνται εκείνη τη στιγμή.

4.6 Εφαρμογή στη βιβλιοθήκη Scikit-learn

Ο κώδικας ο οποίος καλεί τους επιταχυντές οργανώθηκε σε πακέτο Python το οποίο ονομάστηκε numpynq. Ο κώδικας του αρχείου sklearn.utils.extmath.py τροποποιήθηκε ώστε να καλεί το πακέτο numpynq. Ποιο συγκεκριμένα η συνάρτηση του αρχείου που τροποποιήθηκε είναι η safe_sparce_dot.

```
1  from numpynq import matmul as nphw
2  def safe_sparse_dot(a, b, dense_output=False):
3      if sparse.issparse(a) or sparse.issparse(b):
4          ret = a * b
5          if dense_output and hasattr(ret, "toarray"):
6              ret = ret.toarray()
7          return ret
8      elif a.shape[1]<=64 and b.shape[1]<=512:
9          print("This matmul is taking place on HW!!!")
10         accel = nphw.mmult_hw()
11         return accel.matmul(a, np.transpose(b))
12     else:
13         return np.dot(a, b)
```

Στον αρχικό κώδικα του αρχείου προστέθηκαν οι σειρές 1, 8-11. Στη σειρά 8 γίνεται έλεγχος των διαστάσεων των πινάκων. Η σειρά 9 είναι απλώς ένα μήνυμα που γνωστοποιεί ότι ο πολλαπλασιασμός γίνεται στο PL. Στη σειρά 10 αρχικοποιείται η κλάση mmult_hw. Τέλος στη σειρά 11 γίνεται κλήση της συνάρτησης.

4.7 Bitonic Sort

Στο πλαίσιο της επιτάχυνσης της συνάρτησης πολλαπλασιασμού πινάκων του πακέτου NumPy επιχειρήθηκε η εφαρμογή και σε άλλη συνάρτηση του πακέτου. Η συνάρτηση αυτή είναι η numpy.sort(a, axis=-1, kind='quicksort', order=None), η οποία υλοποιεί έναν αλγόριθμο ταξινόμησης, εξ ορισμού τον quicksort. Δέχεται ως όρισμα έναν πίνακα numpy και παράγει ως έξοδο πίνακα ίδιων διαστάσεων με τα στοιχεία ως προς τη διάσταση που ορίζει ο χρήστης διατεταγμένα κατά αύξουσα/φθίνουσα σειρά.

Η συνάρτηση αυτή επιλέχθηκε καθώς είναι μέρος του αλγορίθμου που υλοποιεί η κλάση neighbors της Scikit-learn. Συγκεκριμένα εφαρμόζεται για την τοποθέτηση σε αύξουσα σειρά των k κοντινότερων γειτόνων.

Άντι του quicksort ο αλγόριθμος που αναπτύχθηκε ώστε να τρέχει στο HW είναι ο bitonicsort. Η επιλογή αυτή έγινε γιατί από τη φύση του ο αλγόριθμος έχει μεγάλες δυνατότητες παραλληλοποίησης και άρα είναι κατάλληλος να εκτελεστεί στο υλικό. Αντίθετα ο quicksort είναι καταλληλότερος να εκτελεστεί ακολουθιακά.

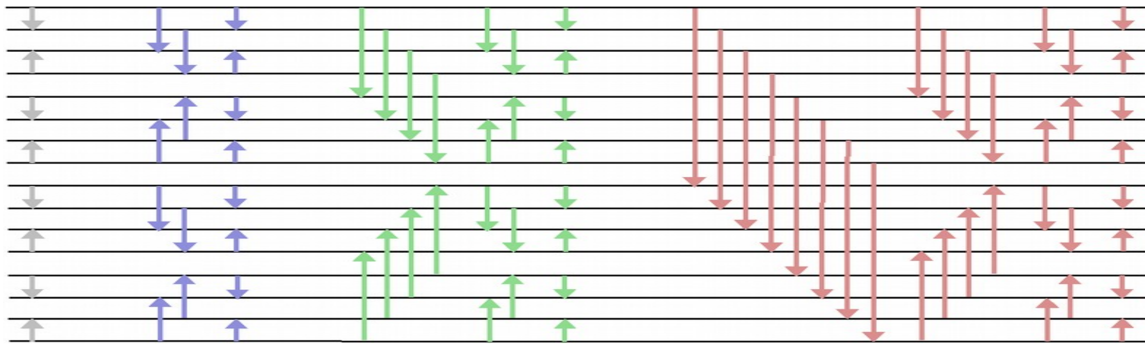
Ο κώδικας σε C++ είναι ο παρακάτω:

```
#pragma SDS data access_pattern( key:SEQUENTIAL, keyout:SEQUENTIAL)
void bitonicsort_accel(float key[N_train * N_neigh], float keyout[N_train * N_neigh],
unsigned char direction){
#pragma HLS DATAFLOW
    float buf_of_key[N_train][N_neigh];
#pragma HLS ARRAY_PARTITION variable=buf_of_key block factor=16 dim=2
    for(int i=0; i<N_train; i++) {
        for(int j=0; j<N_neigh; j++) {
#pragma HLS PIPELINE II=1
            buf_of_key[i][j] = key[i * N_neigh + j];
        }
    }

    const unsigned int passlut[N_steps] = {1, 2, 1, 3, 2, 1, 4, 3, 2, 1};
    const unsigned int stagelut[N_steps] = {0, 1, 1, 2, 2, 2, 3, 3, 3, 3};

    float reg[N_steps + 1][N_neigh];
#pragma HLS ARRAY_PARTITION variable=reg complete dim=0

    for (int i=0; i<N_train; i++){
#pragma HLS DEPENDENCE variable=buf_of_key inter false
        for (int j=0; j<N_neigh; j++){
            reg[0][j] = buf_of_key[i][j];
        }
#pragma HLS PIPELINE II=1
            for (unsigned int step = 0; step < N_steps; step++){
                for (unsigned int c = 0; c < (unsigned int) ((N_neigh + 1) >> 1); c++){
                    unsigned int stage = stagelut[step];
                    unsigned int pass = passlut[step];
                    unsigned char pseudo_direction = direction;
                    if ((((N_neigh + 1) >> 1) - 1) >> stage) & 1) pseudo_direction ^= 1;
                        unsigned int shift = pass - 1;
                        unsigned int distance = 1 << shift;
                        unsigned int mask = distance - 1;

                        unsigned int leftC = ((c >> shift) << (shift + 1)) + (c & mask);
                        unsigned int rightC = leftC + distance;
                        if (rightC < N_neigh){
                            float left = reg[step][leftC];
                            float right = reg[step][rightC];

                            float greater;
                            float lesser;
                            if (left > right){
                                greater = left;
                                lesser = right;
                            }
```

```
                            else{
                                greater = right;
                                lesser = left;
                            }

                            if (((c >> stage) & 1) ^ pseudo_direction){
                                 reg[step + 1][leftC] = lesser;
                                 reg[step + 1][rightC] = greater;
                            }
                            else{
                                 reg[step + 1][leftC] = greater;
                                 reg[step + 1][rightC] = lesser;
                            }
                        }
                    }
                }

    for (int j=0; j<N_neigh; j++){
        buf_of_key[i][j] = reg[N_steps][j];
    }
}
    for(int i=0; i<N_train; i++) {
        for(int j=0; j<N_neigh; j++){
#pragma HLS PIPELINE II=1
            keyout[i * N_neigh + j] = buf_of_key[i][j];
        }
    }
}
```

Ο επιταχυντής δέχεται ως είσοδο τον πίνακα key διαστάσεων 1024x16 (N_train=1024, N_neigh=16). Όπως και πριν χρησιμοποιείται το directive access_pattern=SEQUENTIAL ώστε να δημιουργηθούν οι κατάλληλες FIFO δομές για τους πίνακες εισόδου και εξόδου. Ο πρώτος βρόχος διαβάζει τον πίνακα εισόδου key και τον φορτώνει στις εσωτερικές μνήμες του πυρήνα buf_of_key. Ο buf_of_key συνοδεύεται από το directive array_partition, ώστε να δημιουργηθούν 16 διαφορετικές μνήμες BRAMs η οποίες αποθηκεύουν τις στήλες του. Με αυτό το τρόπο ο επόμενος κόμβος μπορεί να προσπελάσει ταυτόχρονα και τα 16 στοιχεία μιας σειράς του πίνακα buf_of_key.

Η οδηγία PIPELINE τοποθετήθηκε πριν από τους βρόχους που “ξεδιπλώνουν” όλο το δικτύωμα του bitonic sort. Το δικτύωμα φαίνεται στην εικόνα 4.9 όπου με βελάκια παριστάνονται τα λογικά κυκλώματα των συγκριτών.



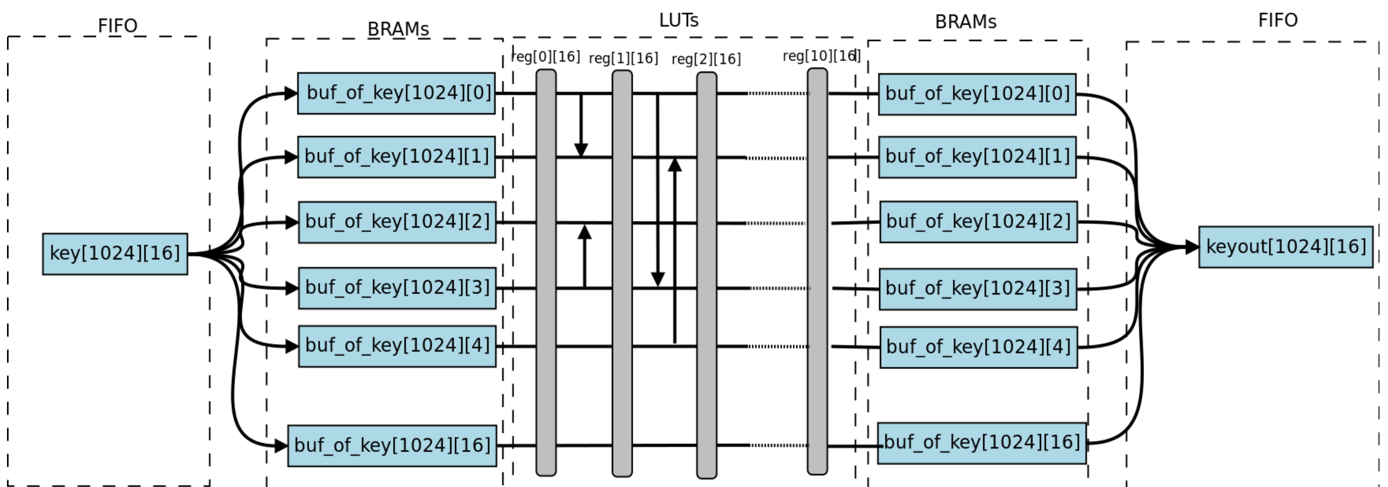
Εικόνα 4.9: Δικτύωμα Bitonic sort

Το κύκλωμα εκτελεί συστολική λειτουργία με II=1. Η κατανομή των πόρων του FPGA που χρησιμοποιήθηκαν φαίνεται στον πίνακα 4.2.

4.2.Πίνακας: Κατανομή των πόρων του FPGA για τη συνάρτηση bitonicsort_accel

	Αριθμός στοιχείων	Ποσοστό
BRAM_18K	32	11%
DSP48E	0	0%
FF	17074	16%
LUT	38394	72%

Η πλήρης αρχιτεκτονική του κυκλώματος φαίνεται στην εικόνα 4.9.



4.10.Εικόνα: Αρχιτεκτονική του επιταχυντή Bitonic sort

5 Παρουσίαση Αποτελεσμάτων

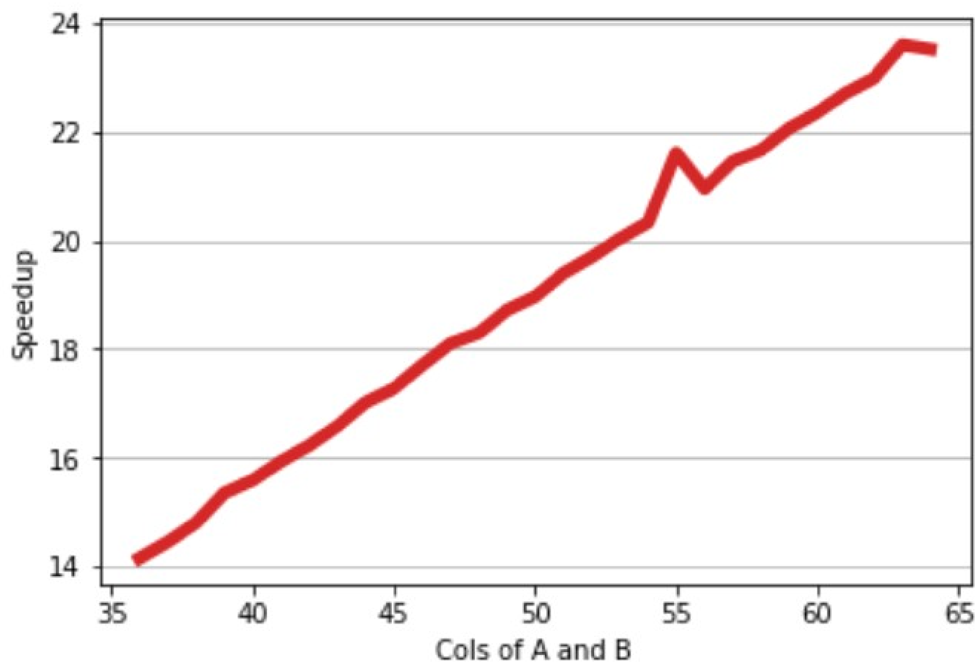
5.1 Εισαγωγή

Στο κεφάλαιο αυτό παρουσιάζονται τα αποτελέσματα που έδωσε η εκτέλεση των αλγορίθμων στο hardware. Η μετρική που εξετάστηκε είναι η επιτάχυνση που έχει επιτευχθεί σε σχέση με την εκτέλεση του αντίστοιχου αλγορίθμου στον επεξεργαστή ARM του ίδιου ολοκληρωμένου κυκλώματος.

5.2 Πολλαπλασιασμός Πινάκων

5.2.1 Standalone εφαρμογή

Αρχικά παρουσιάζεται η επιτάχυνση που έδωσε η συνάρτηση κατά την εκτέλεση της σαν standalone εφαρμογή. Έγινε εκτέλεση του αλγορίθμου με τον αριθμό των στηλών των δύο πινάκων που δέχεται σαν είσοδο να μεταβάλετε. Όπως είναι φανερό από το διάγραμμα στην εικόνα 5.1, η επιτάχυνση (speedup) είναι γραμμική συνάρτηση των στηλών των πινάκων εισόδου, ενώ ξεκινάει από ~15 και φτάνει στο ~23.5 για το μέγιστο αριθμό στηλών που μπορεί να δεχτεί.

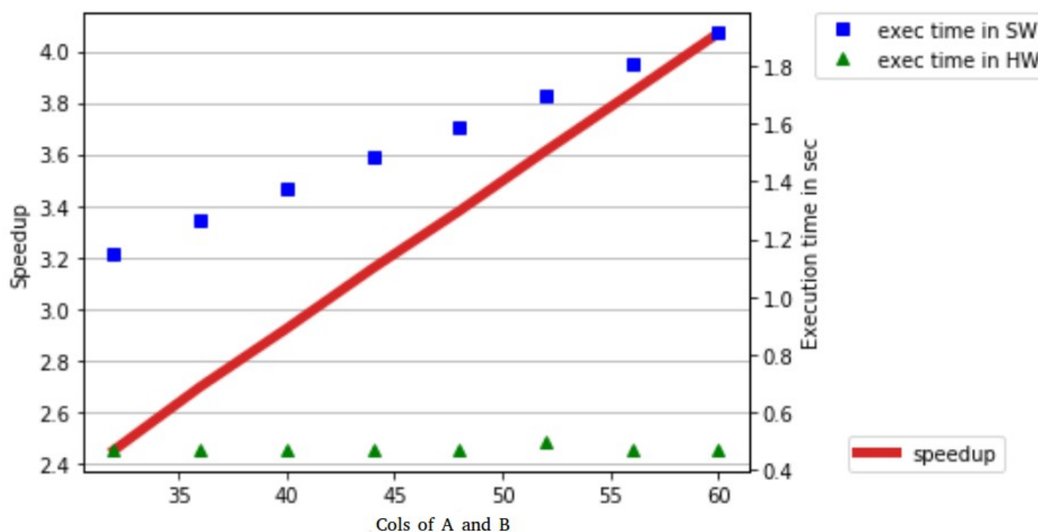


5.1.Εικόνα: Επιτάχυνση σε συνάρτηση με τον αριθμό στηλών

Όπως αναφέρθηκε και στη παράγραφο 3.2 ο αριθμός των πολλαπλασιασμών που απαιτούνται κατά την εκτέλεση του πολλαπλασιασμού δύο πινάκων A[n][m], B[m][k] είναι n*m*k. Ο επιταχυντής που τρέχει στο FPGA όμως εκτελεί σταθερό αριθμό βαθμωτών πολλαπλασιασμών που είναι ανάλογοι με το μέγιστο μέγεθος των πινάκων που δέχεται. Επομένως ο χρόνος εκτέλεσης στο FPGA είναι σταθερός ενώ ο χρόνος εκτέλεσης στον επεξεργαστή μεταβάλετε γραμμικά σε σχέση με τον αριθμό των στηλών.

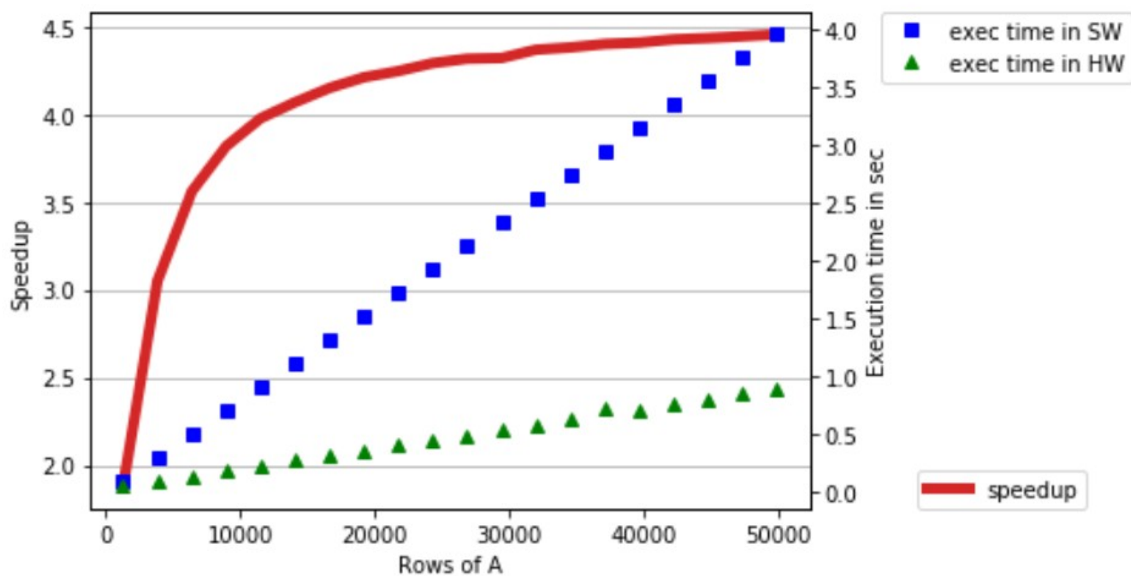
5.2.2 Εφαρμογή στο PYNQ

Εδώ παρουσιάζεται η επιτάχυνση που προσέφερε η συνάρτηση matmul_hw.matmul σε σχέση με τη συνάρτηση πολλαπλασιασμού πινάκων matmul του πακέτου NumPy. Στο πρώτο διάγραμμα δίδονται οι χρόνοι εκτέλεσης των δύο συναρτήσεων και η επιτάχυνση σε συνάρτηση με τον αριθμό στηλών των δύο πινάκων εισόδου, εικόνα 5.2.



Εικόνα 5.2: Χρόνοι εκτέλεσης και επιτάχυνση σε συνάρτηση με τον αριθμό στηλών

Στη συνέχεια έγινε έλεγχος της επιτάχυνσης που προσφέρει σε συνάρτηση με τον αριθμό των σειρών του πίνακα A.



5.4. Εικόνα: Χρόνος εκτέλεσης και επιτάχυνση σε συνάρτηση με τον αριθμό των γραμμών του πίνακα A

Αυτή τη φορά η επιτάχυνση ξεκινάει από ~2 και πλησιάζει το ~4.5 όσο αυξάνονται οι σειρές του πίνακα A. Καθώς η καθυστέρηση που εισάγετε από τη κλήση της βιβλιοθήκης cffi είναι σταθερή, είναι λογικό η επιβάρυνση στη συνολική απόδοση να μειώνεται καθώς αυξάνεται ο “ωφέλιμος” χρόνος υπολογισμού.

5.3 Bitonic Sort

5.3.1 Standalone εφαρμογή

Για τον αλγόριθμο Bitonic Sort ακολουθήθηκε η ίδια διαδικασία ανάπτυξης με τον πολλαπλασιασμό πινάκων. Κατά την εκτέλεση σαν standalone εφαρμογή του προγράμματος δοκιμών η επιτάχυνση που παρουσιάστηκε έτεινε στο 9.5. Το πρόγραμμα δοκιμών συνέκρινε την ταχύτητα εκτέλεσης του επιταχυντή με τον ίδιο αλγόριθμο που εκτελούνταν στον επεξεργαστή. Δυστυχώς κατά τη σύγκρισή με την αντίστοιχη υλοποίηση από τη NumPy αποδείχθηκε πάνω από τέσσερις φορές πιο αργός. Αυτό οφείλεται σε δύο λόγους. Ο ένας είναι η ταχύτητα της βιβλιοθήκης NumPy καθώς είναι ένα βελτιστοποιημένο πακέτο. Ο δεύτερος και σημαντικότερος λόγος είναι ότι, ακόμα και για πολύ μεγάλους πίνακες εισόδου, ο χρόνος που απαιτείται για το καθαρά

υπολογιστικό κομμάτι του αλγορίθμου είναι συγκρίσιμος με τον χρόνο που σπαταλάτε στην κλήση της συνάρτησης μέσω Python.

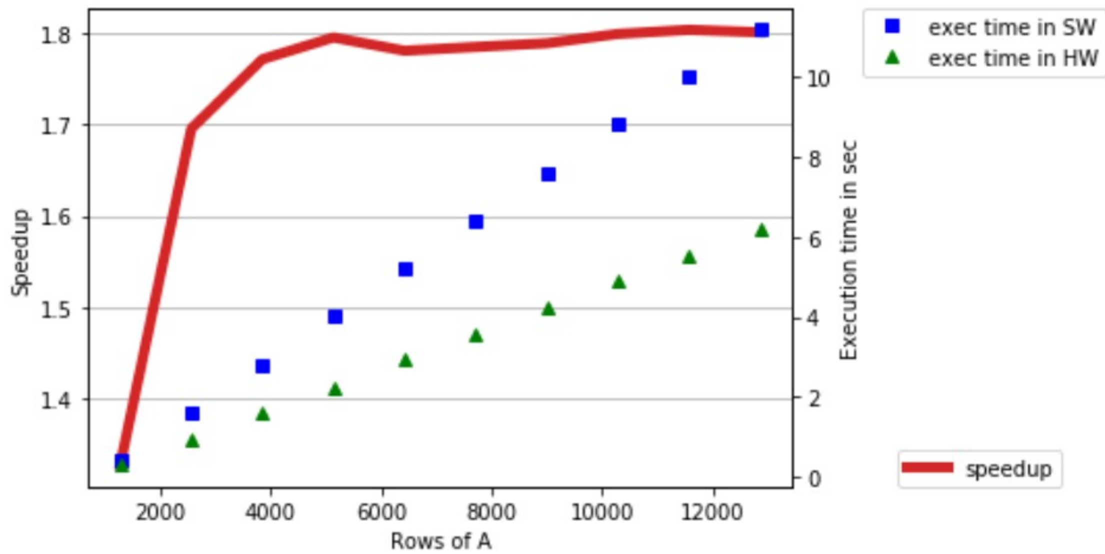
5.4 Επιτάχυνση του Knn

Το σύνολο δεδομένων στο οποίο δοκιμάστηκε ο αλγόριθμος είναι το “Optical Recognition of Handwritten Digits”, το οποίο ανήκει στα έτοιμα σύνολα δεδομένων που υπάρχουν στο πακέτο Scikit-learn. Κάθε στοιχείο του συνόλου είναι μια φωτογραφία ψηφίου 8x8. Τα υπόλοιπα χαρακτηριστικά του συνόλου φαίνονται στον πίνακα 5.1.

5.1 Πίνακας:Χαρακτηριστικά του συνόλου δεδομένων “Optical Recognition of Handwritten Digits”

Κλάσεις	10
Δείγματα ανα κλάση	~180
Σύνολο δειγμάτων	1797
Διαστάσεις	64
Τύπος	Ακέραιοι 0-16

Ως σύνολο εκπαιδεύσεως χρησιμοποιήθηκαν τα πρώτα 512 δείγματα του συνόλου. Τα υπόλοιπα δείγματα χρησιμοποιήθηκαν ως σύνολο δοκιμών. Προκειμένου να δοκιμαστεί η αποτελεσματικότητα της υλοποίησης και σε μεγαλύτερα σύνολα τα δείγματα του συνόλου δοκιμών “πολλαπλασιάστηκαν” με τη συνάρτηση tile του πακέτου numpy. Η επιτάχυνση που παρατηρήθηκε ξεκινάει από το 1.3 για μικρά σύνολα δοκιμών και φτάνει περίπου στο 1.8 για μεγάλα σύνολα. Στην εικόνα 5.4 δίδεται η επιτάχυνση του αλγορίθμου k-NN σε συνάρτηση με τον αριθμό των σημείων του συνόλου δοκιμών.



5.5. Εικόνα: Χρόνοι εκτέλεσης και επιτάχυνση σε συνάρτηση με τον αριθμό των σημείων του συνόλου δοκιμών

Ενδεικτικά παρατίθενται τα αποτελέσματα κατά την εκτέλεση του k-NN με σύνολο εκπαιδεύσεως τα 512 πρώτα σημεία και σύνολο δοκιμών τα υπόλοιπα του συνόλου δεδομένων.

```
Classification report for classifier KNeighborsClassifier(algorithm='brute', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=3, p=2,
           weights='uniform'):
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       126
           1       0.89      0.77      0.83       130
           2       0.82      0.94      0.88       125
           3       0.92      0.89      0.91       130
           4       0.99      0.96      0.98       130
           5       0.97      0.95      0.96       130
           6       0.96      0.98      0.97       130
           7       0.99      0.99      0.99       128
           8       0.86      0.87      0.87       123
           9       0.89      0.91      0.90       128

   micro avg       0.93      0.93      0.93      1280
   macro avg       0.93      0.93      0.93      1280
weighted avg       0.93      0.93      0.93      1280
```

5.6. Εικόνα: Ακρίβεια αποτελεσμάτων που παρήγαγε ο k-NN

6 Επίλογος

6.1 Σύνοψη και συμπεράσματα

Στην παρούσα διπλωματική επιταχύνθηκε η εκτέλεση του αλγορίθμου k-NN της βιβλιοθήκης μηχανικής μάθησης Scikit-learn με τη χρήση FPGA. Αποδείχθηκε ότι οι αλγόριθμοι αυτοί μπορούν να ωφεληθούν από τη χρήση των συσκευών αυτών. Το ολοκληρωμένο κύκλωμα που χρησιμοποιήθηκε (Zynq-7000) είναι προσανατολισμένο για εφαρμογή σε ενσωματωμένα συστήματα, όμως ωφέλεια μπορεί να προκύψει και σε μεγαλύτερα συστήματα όπως συστοιχίες FPGAs με πολύ περισσότερους διαθέσιμους πόρους. Αναμένεται μάλιστα η εφαρμογή σε τέτοιου είδους συστήματα να δίνει πολύ καλύτερα αποτελέσματα.

Η ευκολία που προσφέρει η χρήση της γλώσσας Python είναι ένα ακόμα όφελος που προκύπτει από την παρούσα υλοποίηση. Η βιβλιοθήκη overlay που αναπτύχθηκε μπορεί να κληθεί σαν οποιαδήποτε βιβλιοθήκη Python. Κατά αυτόν τον τρόπο ο χρήστης δεν χρειάζεται να ασχοληθεί με λεπτομέρειες που αφορούν την υλοποίηση.

Τέλος αξίζει να αναφερθεί ότι λόγω της επιλογής να επιταχυνθεί η συνάρτηση NumPy.matmul η οποία εκτελεί πολλαπλασιασμό πινάκων, εκτός του k-NN αύξηση της ταχύτητας εκτέλεσης μπορεί να παρατηρηθεί και σε άλλους αλγορίθμους του ίδιου πακέτου που την καλούν.

6.2 Μελλοντικές Επεκτάσεις

Η πρώτη επέκταση είναι η εφαρμογή της μεθοδολογίας και σε άλλες συναρτήσεις του πακέτου NumPy. Αυτό επιχειρήθηκε για την συνάρτηση NumPy.sort η οποία εκτελεί αλγόριθμο ταξινόμησης. Κατά τη διαδικασία της βελτιστοποίησης στην οποία εκτελέστηκε πρόγραμμα δοκιμών (testbench) η επιτάχυνση κυμάνθηκε στο ~9.5. Δυστυχώς όμως δεν υπήρξαν ανάλογα αποτελέσματα και όταν η συνάρτηση καλούνταν μέσω της διεπαφής Python. Επομένως χρειάζεται προσεκτική επιλογή των αλγορίθμων που θα βελτιστοποιηθούν. Καλοί υποψήφιοι είναι αυτοί που παρουσιάζουν μεγάλο λόγο υπολογισμών ανά λέξη εισόδου.

Όσον αφορά τη βελτιστοποίηση της απόδοσης της υπάρχουσας υλοποίησης η μείωση της καθυστέρησης κατά την κλήση των οδηγών είναι μια κατεύθυνση που θα μπορούσε να εξεταστεί. Η βιβλιοθήκη CFFI μέσω της οποίας γίνεται η “συγκόλληση”

της Python με τους οδηγούς του επιταχυντή παρουσιάζει μεγάλη προγραμματιστική ευκολία, με κόστος όμως αυξημένη καθυστέρηση. Η χρήση γρηγορότερων επεκτάσεων όπως Cython και ctypes θα μπορούσε να βελτιώσει περαιτέρω την απόδοση.

Βιβλιογραφία

[1] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC, First Edition, Strathclyde Academic Media, 2014.

[2] Vivado Design Suite, AXI Reference Guide, 2017

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

[4] Vivado Design Suite, User Guide, High-Level Synthesis, 2017

[5] Vivado Design Suite Tcl Command Reference Guide, February 02, 2018

[6] SDSoC Environment User Guide, December 20, 2017

[7] SDSoC Environment Profiling and Optimization Guide, December 20, 2017

[8] SDSoC Programmers Guide, July 2, 2018

[9] SDSoC Environment Platform Development Guide, April 4, 2018

[10] Sdx Pragma Reference Guide, December 20, 2017

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

[12] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer, Parallel Programming for FPGAs 2018

[13] 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.

[14] scikit-learn user guide, scikit-learn developers, Nov 21, 2017

[15] CFFI Documentation Release 1.11.5, Armin Rigo, Maciej Fijalkowski, Feb 27, 2018

[16] Python productivity for Zynq (Pynq) Documentation, Release 2.2, Xilinx, Sep 19, 2018

[17] k-nearest neighbors, 30 January 2019, https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

[18] UCI Machine Learning Repository, Optical Recognition of Handwritten Digits Data Set *https://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits*

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

Παράρτημα A - Διεπαφή Python

```
1  import cffi
2  from pynq import Overlay,PL,Xlnk
3  import numpy as np
4  import os
5
6  ROWS_OF_A=256
7  ROWS_OF_B=512
8  COL_OF_AB=64
9
10 BS_SEARCH_PATH = os.path.dirname(os.path.realpath(__file__))
11
12 BITFILE = BS_SEARCH_PATH + "/overlay/matmul.bit"
13 LIBRARY = BS_SEARCH_PATH + "/drivers/libmatmul.so"
14
15
16 matmul_overlay = None
17
18
19 class mmult_hw:
20     """Class which controls matrix multiplication on hardware
21
22     Attributes
23     ----------
24     bitfile : str
25         Absolute path of bitstream file
26     libfile : str
27         Absolute path of shared library
28     overlay : Overlay
29         Gives access to bitstream overlay
30
31     """
32
33     def __init__(self):
34         self.bitfile = BITFILE
35         self.libfile = LIBRARY
36         ffi = cffi.FFI()
37         ffi.cdef("void bmatmul_fixed_B(float *A, float *B, float *C, int A_n_rows, int A_n_col,
   int B_n_rows);")
38         self.lib = ffi.dlopen(self.libfile)
39         self.xlnk = Xlnk()
40
41     def download_bitstream(self):
42         """Download the bitstream
43
44         Downloads the bitstream onto hardware using overlay class.
45         Also gives you access to overlay.
46
47         Parameters
48         ----------
49         None
50
51         Returns
52         -------
53         None
54
55         """
56         global matmul_overlay
```

```
57          matmul_overlay = Overlay(self.bitfile)
58          matmul_overlay.download()
59
60      def matmul(self, A, B):
61          if (A.shape[1]!=B.shape[1]):
62              raise RuntimeError("A and B cant multiply due to dimension mismatch")
63          if B.shape[0]>512 or A.shape[1]>64:
64              raise RuntimeError("Not proper arrays for HW multiplication")
65          (A_n_rows, A_n_col)=A.shape
66          B_n_rows=B.shape[0]
67          C=Xlnk().cma_array(shape=(A_n_rows, ROWS_OF_B), dtype=np.float32, cacheable=1)
68          pA=cffi.FFI().cast("float *", A.ctypes.data)
69          pB=cffi.FFI().cast("float *", B.ctypes.data)
70          pC=cffi.FFI().cast("float *", C.ctypes.data)
71          if any("cdata" not in elem for elem in [str(pA), str(pB), str(pC)]):
72                  raise RuntimeError("Unknown buffer type!")
73          self.lib.bmatmul_fixed_B(pA, pB, pC, A_n_rows, A_n_col, B_n_rows)
74          return C
75
76      def numpynq_buffer_free(self, buf):
77          self.xlnk.cma_free(buf)
78
79      def numpynq_cast(self, buf, dtype):
80          return cffi.FFI().cast(dtype , buf.ctypes.data)
81
82      def numpynq_buffer(self, shape, dtype=np.float32, cacheable=1):
83          return self.xlnk.cma_array(shape, dtype = dtype)
84
85      def numpynq_buffers_reset(self):
86          self.xlnk.xlnk_reset()
87
88      def numpynq_buffers_stats(self):
89          return self.xlnk.cma_stats()
```