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

Διπλωματικη Εργασια

Επιτάχυνση Συστημάτων Συστάσεων Βασισμένη σε FPGA

Κατσαντώνης Κωνσταντίνος

Επιβλέπων:
Δρ. Δημήτριος Σούντρης
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2017

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

Διπλωματικη Εργασια

Επιτάχυνση Συστημάτων Συστάσεων Βασισμένη σε FPGA

Κατσαντώνης Κώνσταντίνος

Επιβλέπων:
Δρ. Δημήτριος Σούντρης
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Νοεμβρίου 2017.

.....	.....	.....
Δημήτριος Σούντρης	Γεώργιος Γκούμας	Κιαμάλ Πεκμεστζή
Αν. Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2017

.....

Κατσαντώνης Κωνσταντίνος

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

Περίληψη

Η ραγδαία αύξηση των δεδομένων τα τελευταία χρόνια έχει οδηγήσει σε άνθιση τον κλάδο της μηχανικής μάθησης και γενικότερα της τεχνητής νοημοσύνης. Ωστόσο παρά της αυξανόμενες ανάγκες για επεξεργασία όλο και μεγαλύτερου όγκου δεδομένων στα κέντρα δεδομένων ο τρόπος που τα επεξεργαζόμαστε δεν έχει αλλάξει ιδιαίτερα εδώ και αρκετά χρόνια και ο μόνος λόγος που η επεξεργαστική ισχύς συμβαδίζει με την αύξηση των δεδομένων είναι ότι προς το παρών δεν μας έχει εγκαταλείψει ο νόμος του Moore κάτι που σύντομα ενδεχομένως να σταματήσει να ισχύει.

Το εναλλακτικό μονοπάτι μέχρι τώρα έναντι του κλασικού CPU server αποτελούν οι αρχιτεκτονικές που διαμοιράζουν το υπολογιστικό έργο εκτός απο τις CPU και σε GPU με τις τελευταίες να πετυχαίνουν εξαιρετικά αποτελέσματα όσον αφορά την χρονική επίδοση. Ωστόσο αν και οι χρονικές επιδόσεις των GPU είναι εξαιρετικές όσον αφορά την ενεργειακή απόδοση οδηγούν σε περαιτέρω αύξηση της κατανάλωσης ισχύος.

Σκοπός αυτής της διπλωματικής είναι να αναδείξει ένα ακόμα εναλλακτικό υπολογιστικό μονοπάτι που στοχεύει όχι στην αντικατάσταση των υπολοίπων αλλά στο να καλύψει κάποιες ανάγκες που οι υπάρχουσες μέθοδοι αδυνατούν. Στόχος αυτής της διπλωματικής είναι να δείξουμε πως μπορούν τα FPGAs να ενσωματωθούν στα κέντρα δεδομένων κάνοντας χρήση υπαρχουσών τεχνολογιών με στόχο ταυτόχρονα την αύξηση της ενεργειακής και χρονικής επίδοσης. Για τον σκοπό αυτό στην συγκεκριμένη διπλωματική χρησιμοποιήσαμε τον αλγόριθμο Εναλλασόμενων Ελάχιστων Τετραγώνων για Παραγοντοποίηση Πίνακα (Alternating Least Squares For Matrix Factorization) με στόχο να παρουσιάσουμε ένα διάγραμμα εργασιακής ροής σχετικά με το πώς κάποιος μπορεί να ξεκινήσει να τρέχει αλγορίθμους υποβοηθούμενους από FPGA σε ενα Datacenter.

Η διπλωματική αυτή εντάσσεται σε ένα γενικότερο πλαίσιο μιας προσπάθειας να κατασκευαστούν βιβλιοθήκες οι οποίες θα περιέχουν βασικές εργασίες ενός Datacenter και θα είναι επιταχυμένες σε FPGA και έτοιμες για χρήση από τον εκάστοτε προγραμματιστή.

Οι πλακέτες που χρησιμοποιήσαμε έιναι οι Xilinx ZedBoard, Xilinx Zc702 και Digilent Pynq Z1, ενώ όσον αφορά το λογισμικό χρησιμοποιήσαμε τα Xilinx SDSoC, Xilinc Vivado HLS,Apache Spark. Η ροη εργασίας που ακολουθήσαμε ήταν η εξής : αρχικά γράψαμε τον αλγόριθμο των εναλλασόμενων ελαχίστων τετραγώνων σε C και έπειτα καταγράψαμε το χρονικό του προφίλ. Στην συνέχεια με τη βοήθεια του Xilinx SDSoC και του Vivado HLS κατασκευάσαμε μερικούς πυρήνες επιτάχυνσης για το έντονο υπολογιστικά κομμάτι του αλγορίθμου. Δοκιμάσαμε την χρονική επίδοση των πυρήνων πάνω στο zedboard και στην συνέχεια την ενεργειακή τους επίδοση πάνω στο zc702. Στην συνέχεια επιλέξαμε τον καλύτερο απο τους πυρήνες τόσο ενεργειακά

όσο και χρονικά και τον ενσωματώσαμε σε περιβάλλον Python στην πλακέτα Pynq-Z1 κάνοντας χρήση των βιβλιοθηκών που παρέχονται για την τελευταία. Η μετάβαση απο C σε Python έγινε διότι επιζητούσαμε να δημιουργήσουμε μια υλοποίηση σε μια γλώσσα που να χρησιμοποιείτε ευρέως στα κενταρα δεδομένων απο τους προγραμματιστές. Τέλος για τον σκοπό της διπλωματικής κατασκευάσαμε ένα Cluster απο Pynqs το οποίο προσομοίωσε ένα μικρό κέντρο δεδομένων πάνω στο οποίο, με την βοήθεια του Apache Spark, τρέξαμε κατανεμημένα τον επιταχυμένο αλγόριθμο.

Περιεχόμενα

Περίληψη **i**

1 Εισαγωγή **1**
1.1 FPGAs με μια ματιά 1
1.2 Η Αρχιτεκτονική των FPGAs 1
1.3 Εφαρμογές των FPGAs 2
1.4 Μηχανική μάθηση και εκθετική αύξηση των δεδομένων 2
1.5 Μετάβαση απο το γενικού σκοπού υπολογιστικό υλικό σε προσαρμοσμένες αρχιτεκτονικές 4
1.6 FPGA ή GPU; 4
1.7 Ενσωμάτωση των FPGA στα DataCenter 5
1.7.1 Apache Spark 6
1.7.2 Vivado HLS 8
1.7.3 Zynq - 7000 10
1.7.4 ZedBoard 11
1.7.5 Pynq 11
1.7.6 Zc702 12

2 Συστήματα Συστάσεων **15**
2.1 Εισαγωγή 15
2.2 Συνεργατικό Φιλτράρισμα 15
2.2.1 Βασισμένο σε Μοντέλο 16
2.2.2 Βασισμένο σε Μνήμη 16
2.2.3 Παραγοντοποίηση Πίνανα 16
2.3 Content-Based Filtering 17
2.4 Παραγοντοποίηση Πίνακα με χρήση εναλλασσόμενων ελάχιστων τετραγώνων (ALS) 20
2.4.1 Διαίσθηση 22
2.4.2 Μαθηματική προσέγγιση 22

3 Συγγραφή Αλγορίθμου **25**
3.1 Τα σύνολα δεδομένων 25
3.2 Ο Αλγόριθμος 26
3.2.1 Βασικές Συναρτήσεις 28
3.2.2 Συνάρτηση Main 31
3.3 Απεικόνιση χρονικού προφιλ 33

4 Υλοποίηση Επιταχυντή **37**
4.1 Εισαγωγή 37
4.2 Προκλήσεις 39

4.3 Βασικές Παραδοχές Σχετικά με τον Πολλαπλασιασμό πινάκων 39
4.4 Διεπαφή CPU και FPGA 46
4.5 Πρώτη έκδοση πυρήνα (Version 1) 47
4.6 Δεύτερη Έκδοση Πυρήνα (Version 2) - Χρήση του IP AXI4-Stream Accelerator Adapter 55
4.7 Τρίτη Έκδοση Πυρήνα (Version 3) - Accelerator Adapter - Μεταβλητό Παράθυρο 58
4.8 Επίδοση χωρίς HLS εντολές 63
4.9 Επίδοση στα datasets 64
4.10 Ενεργειακή επίδοση 66
4.11 Κανονικοποίηση 69
4.12 Συμπερασματικές παρατηρήσεις 71

5 Ενσωμάτωση με Python **73**
5.1 Εισαγωγή 73
5.2 Xilinx's Python API στο Pynq - Z1 74
5.3 Απομόνωση του πυρήνα 75
5.4 Συγγραφή των οδηγών του υλικού 76
5.5 Συγγραφή του υπόλοιπου Software 80
5.6 Επίδοση στο Pynq 82
5.7 Συμπερασματικές Παρατηρήσεις 83

6 Ενσωμάτωση στο Spark **87**
6.1 Εισαγωγή 87
6.2 Περιγραφή του Cluster και Spark Configuration 88
6.3 Ο Αλγόριθμος 89
6.4 Επίδοση 91
6.5 Παρατηρήσεις 93

7 Σύνοψη **97**
7.1 Περίληψη 97
7.2 Μελλοντική δουλειά 98

1 Εισαγωγή

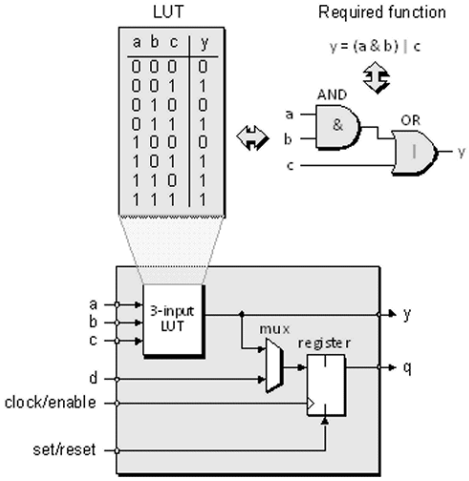
1.1 FPGAs με μια ματιά

Ένα FPGA (Field Programmable Array) είναι ένα ολοκληρωμένο κύκλωμα, το οποίο μπορεί να αναδιαρθρωθεί μετά την κατασκευή του, σε αντίθεση με τα παραδοσιακά ASICS (Application Specific Integrated Circuits), τα οποία δεν μπορούν να πάρουν κανένα είδος διαμόρφωσης μετά την κατασκευή τους. Μέχρι πρόσφατα, ο επαναπρογραμματισμός των FPGA γινόταν αποκλειστικά με τη χρήση μιας Γλώσσας Περιγραφής Υλικού (Hardware Definition Language - HDL) όπως Verilog ή η VHDL, αλλά πρόσφατα αναπτύχθηκαν εργαλεία High Level Synthesis (HLS) που επιτρέπουν την επαναπρογραμματισμό των FPGAs από γλώσσες υψηλού επιπέδου όπως η C και η C ++.

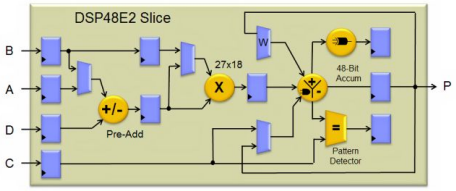
1.2 Η Αρχιτεκτονική των FPGAs

Τα FPGAs αποτελούνται κυρίως από πολλαπλές συστοιχίες διαμορφώσιμων λογικών μπλοκ (CLBs)[1], οι οποίες συνήθως αποτελούνται από κάποια LUTs[2] (Look Up Tables Figure 1.1), πλήρεις αθροιστές, πολυπλέκτες και Flip-Flops[1], όπως φαίνεται στο σχήμα 1.2. Επιπλέον περιέχουν I / O ακίδες για διασύνδεση με άλλες συσκευές και κανάλια δρομολόγησης που επιτρέπουν τις διασυνδέσεις μεταξύ των CLB. Ο προγραμματισμός ενός FPGA αφορά στην ακριβή διαμόρφωση των καναλιών δρομολόγησης, προκειμένου να επιτευχθεί η επικοινωνία μεταξύ των επιθυμητών CLB, με τον πιο αποτελεσματικό τρόπο, με στόχο την υλοποίηση μιας εφαρμογής. Η δρομολόγηση των καλωδίων, καθώς και η διαμόρφωση και η επιλογή των CLB πραγματοποιούνται από τα εργαλεία λογισμικού που προσφέρονται από κάθε εταιρεία και είναι στενά διασυνδεδεμένα με τα προϊόντα της. Τα σύγχρονα FPGA εκτός από τα παραδοσιακά CLB παρέχουν επίσης πιο εξειδικευμένα στοιχεία, όπως ολόκληρους μικροεπεξεργαστές, DSPs[4] και BRAMs (Σχήμα 1.1). Τα βασικά πλεονεκτήματα των FPGA έναντι των

συμβατικών επεξεργαστών είναι ότι σε συγκεκριμένες εφαρμογές μπορούν να επιτύχουν αυξημένη ταχύτητα εκτέλεσης [3] , αυξημένη απόδοση ανά watt, κυρίως λόγω της δυνατότητα εκτέλεσης ταυτόχρονων υπολογισμών. Από την άλλη όμως, ο προγραμματισμός τους είναι εξαιρετικά δύσκολος και απαιτεί μεγάλη εμπειρία και τεχνογνωσία.



(α′) πολύ απλό CLB, το οποίο αποτελείται από ένα LUT, έναν πολυπλέκτη και ένα flip-flop, που υλοποιεί την ακόλουθη λογική συνάρτηση: $y = (a\text{AND}b)\text{OR}c$



(β′) DSP48, χρησιμοποιείται για ταχεία εκτέλεση, αριθμητικών πράξεων της ακόλουθης μορφής: $P = B * (D + A)$, $P = B * A$ κλπ

Σχήμα 1.1: Δύο δομικά στοιχεία των σύγχρονων FPGAs

1.3 Εφαρμογές των FPGAs

Ένα FPGA είναι ικανό να επιλύσει οποιοδήποτε υπολογιστικό πρόβλημα και αυτό μπορεί εύκολα να αποδειχθεί από το γεγονός ότι τα FPGAs μερικές φορές χρησιμοποιούνται για την υλοποίηση μικροεπεξεργαστών όπως ο microblaze της Xilinx και ο επεξεργαστής Nios της Altera. Ωστόσο, το πλεονέκτημά τους έναντι των CPU είναι ότι αν προγραμματιστούν σωστά μπορούν να εκμεταλλευτούν την παράλληλη φύση κάποιων εφαρμογών, σε αντίθεση με τις CPU που για οποιδήποτε πρόβλημα, μπορούν να εκτελούν υπολογισμούς με διαδοχικό μόνο τρόπο.
Οι πιο συνηθισμένες εφαρμογές FPGA περιλαμβάνουν εφαρμογές στον τομέα της αεροδιαστημικής, ραδιοαστρονομίας [5] και της άμυνας, της ιατρικής ηλεκτρονικής, της επεξεργασίας εικόνας και video[5], των ενσύρματων και ασύρματων επικοινωνιών[5], της πρωτοτυποποίησης Asic και πολλά άλλα ...

1.4 Μηχανική μάθηση και εκθετική αύξηση των δεδομένων

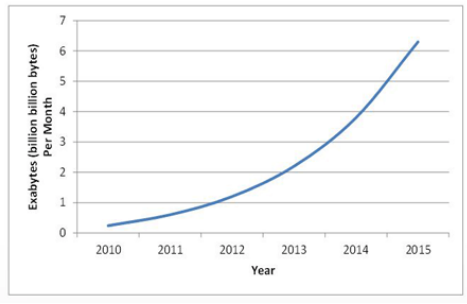
Μέχρι πρόσφατα, ο προγραμματισμός από πλευράς διακομιστή περιελάμβανε πολύπλοκους αλγόριθμους. Ωστόσο, η εκθετική αύξηση των δεδομένων [7] σε συνδυασμό

με την αυξημένη υπολογιστική ισχύ οδήγησε σε σημαντική αλλαγή. διακομιστές αντί να τρέχουν σύνθετους αλγορίθμους σε μικρό όγκο δεδομένων, τρέχουν πολύ απλούστερους σε τεράστιους όγκους δεδομένων[6]. Η πιο γνωστή κατηγορία τέτοιων αλγορίθμων είναι οι αλγόριθμοι μηχανικής μάθησης, οι οποίοι καταναλώνουν μεγάλο όγκο δεδομένων, προσπαθώντας να εντοπίσουν ή να συνάψουν συμπεράσματα, προκειμένου να εξυπηρετήσουν καλύτερα τους χρήστες ή ακόμα και να προβλέψουν τις μελλοντικές τάσεις. Αυτοί οι αλγόριθμοι επιτυγχάνουν εξαιρετικά καλύτερα αποτελέσματα από τους αλγόριθμους παλαιότερης γενιάς. Οι αλγόριθμοι μηχανικής μάθησης κατηγοριοποιούντε με βάση τον τρόπου με τον οποίο εκπαιδεύονται στις ακόλουθες κατηγορίες:

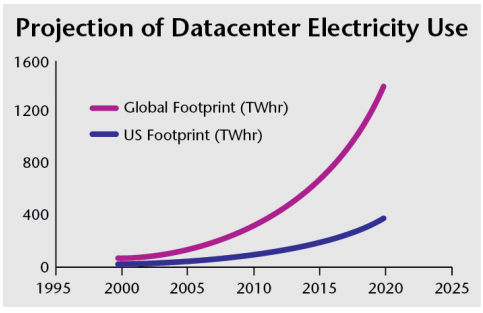
- *Supervised Learning*:Σε αυτή την περίπτωση δίνεται μια επιθυμητή έξοδος για κάθε επιθυμητή είσοδο και ο αλγόριθμος προσπαθεί να βρει τον γενικό κανόνα για την αντιστοίχιση των εισόδων στις αντίστοιχες εξόδους.
- *Unsupervised Learning*: Σε αυτή την κατηγορία ο αλγόριθμος προσπαθεί να εντοπίσει ένα μοτίβο μέσα στα δεδομένα.
- *Reinforcement learning*:Ο αλγόριθμος αλληλεπιδρά με ένα δυναμικό περιβάλλον και μέσω ενός συστήματος ποινών και ανταμοιβών-επιβραβεύσεων προσπαθεί να μάθει τον τρόπο επίτευξης ενός συγκεκριμένου στόχου.

και σύμφωνα με την επιθυμητή έξοδο:

- *classification*:Σε αυτή την περίπτωση ο αλγόριθμος προσπαθεί να ταιριάξει κάθε είσοδο σε μια κλάση-ομάδα.
- *regression*: Σε αυτή την κατηγορία ο αλγόριθμος προσομοιώνει μια πραγματική συνάρτηση αναθέτοντας σε κάθε είσοδο έναν πραγματικό αριθμό
- *Clustering*: ένα σύνολο εισόδων πρέπει να χωριστεί σε ομάδες-κλάσεις. Σε αντίθεση με την ταξινόμηση(classification), οι ομάδες δεν είναι γνωστές εκ των προτέρων, καθιστώντας αυτό συνήθως μια εργασία χωρίς επίβλεψη.



(α΄) εκθετική αύξηση των δεδομένων στην διάρκεια μιας πενταετίας

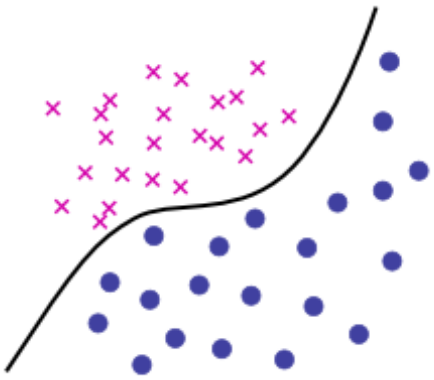


(β΄) η συνεχώς αυξανώμενη κατανάλωση ισχύος από τα data-centers στον κόσμο.

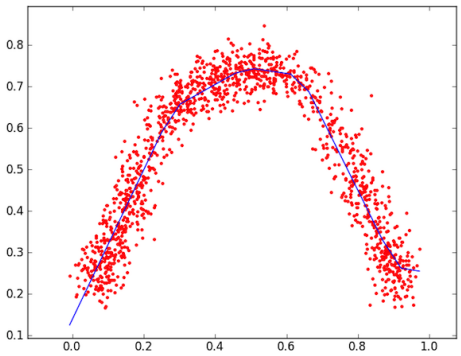
Σχήμα 1.2: Η ταυτόχρονη αύξηση των δεδομένων και της ενέργειας που απαιτείται για την επεξεργασία τους

1.5 Μετάβαση απο το γενικού σκοπού υπολογιστικό υλικό σε προσαρμοσμένες αρχιτεκτονικές

Η ανάγκη για όλο και περισσότερη επεξεργασία δεδομένων σε υψηλότερες ταχύτητες και χαμηλότερη κατανάλωση ενέργειας, σε συνδυασμό με το γεγονός ότι οι αλγόριθμοι γίνονται απλούστεροι από εκείνους του παρελθόντος, δημιουργούν την ανάγκη για νέους υπολογιστικούς πόρους πέρα από τους παραδοσιακούς επεξεργαστές, εδώ οι FPGAs και GPUs είναι οι πιο δημοφιλείς υποψήφιοι .Μέχρι στιγμής, η λύση ήταν η ίδια για όλα τα υπολογιστικά προβλήματα, ένα τεράστιο κέντρο δεδομένων με πολλά ικριώματα από εξυπηρετητές οι αποτελούνται από κάποιες CPU. Ο εναλλακτικός, προσαρμοσμένος υπολογισμός(custom computing) περιλαμβάνει τη δημιουργία μιας εξειδικευμένης λύσης για κάθε πρόβλημα, χρησιμοποιώντας εξειδικευμένο υλικό για πιο αποτελεσματικές λύσεις χρόνου και ενέργειας. Σύμφωνα με αυτή την προοπτική, δεν είναι πλέον βιώσιμη η μελέτη λογισμικού και υλικού ως δύο εντελώς ανεξάρτητες οντότητες.
Επί του παρόντος, η πιο κοινή εναλλακτική λύση στις CPU-only αρχιτεκτονικές είναι οι GPU, ως αυτοτελείς επεξεργαστές ή ως cooproccesors μαζί με τις CPU.



(α′) απεικόνιση ενός δισδιάστατου προβλήματος ταξινόμησης σε ομάδες(classification)



(β′) regression: Ο αλγόριθμος προσαρμόζει το μοντέλο (μπλε γραμμή) στις κόκκινες κουκίδες (δεδομένα)

1.6 FPGA ή GPU;

Η απλή απάντηση σε αυτή την ερώτηση είναι ότι οι FPGAs είναι αποδοτικές όσον αφορά την εξοικονόμηση ενέργειας και οι GPUs είναι οικονομικά αποδοτικές, αλλά η λήψη αποφάσεων που βασίζεται μόνο σε αυτό το γεγονός είναι παρακινδυνευμένη. Η σύγκριση των **υπολογιστικών δυνατοτήτων** δεν είναι απλή, καθώς χρησιμοποιούνται διαφορετικές μετρικές για κάθε συσκευή, GFLOPs (Giga Floating Point Operations) για GPUs και GMACS (Giga Multiply Accumulate) για FPGAs. Αυτό σημαίνει ότι οι GPU υπερέχουν στην επιτάχυνση αλγορίθμων που βασίζονται σε πράξεις κινητής υποδιαστολής, ενώ από την άλλη τα FPGA απαιτούν αλγόριθμους σταθερού υποδιαστολής(fixed point) προκειμένου να εκμεταλλευτούν τα πλεονεκτήματα

των bitwise λειτουργιών τους. Όταν εξετάζουμε τον παράγοντα της **ενεργειακής επίδοσης**, τα πράγματα είναι πιο ξεκάθαρα και τα FPGAs έχουν μεγάλο πλεονέκτημα καθώς μπορούν να προσφέρουν τεράστιες δυνατότητες επεξεργασίας με μεγάλη ενεργειακή απόδοση, μειώνοντας την ανάγκη για θερμική διαχείριση. Οι **I/O Διεπαφές** είναι ένα άλλο ισχυρό σημείο των FPGAs. Οι μονάδες GPU περιορίζονται αποκλειστικά σε PCIe, ενώ οι FPGA έχουν μεγάλη ευελιξία.

Ένα άλλο μεγάλο πλεονέκτημα των FPGAs είναι ότι παρέχουν την ικανότητα πρόβλεψης **του χρόνου εκτέλεσης** (**latency**) και μάλιστα με ακρίβεια χτύπου ρολογιού. Οι FPGA παρέχουν καθορισμένο χρονισμό σε επίπεδο μικροδευτερολέπτων (κύκλοι ρολογιού). Αυτό είναι ιδιαίτερα σημαντικό για κρυπτογράφηση, κωδικοποίηση ήχου, συγχρονισμό δικτύου ή εφαρμογές ελέγχου που πρέπει να διαχειριστούν μικρές και γνωστές καθυστερήσεις/χρόνους εκτέσεσης. Τέλος όσον αφορά την **τιμή** της κάθε επιλογής όταν κανείς αναζητά λύση στον χώρο της επιτάχυνσης λογισμικού, οι GPUs αποτελούν καλύτερη επιλογή τόσο όσο αφορά το κόστος του υλικού αλλά και όσο αφορά την κόστος ανάπτυξης και υλοποίησης της λύσης. Τα FPGAs από την άλλη μεριά απαιτούν την πρόσληψη εξειδικευμένων μηχανικών με γνώσεις σε πολλές και διαφορετικές περιοχές της τεχνολογίας. Η παρούσα διπλωματική αφορά την επιτάχυνση αλγορίθμων μηχανικής μάθησης, τομέας στον οποίο σχετική δουλειά που έχει γίνει στο παρελθόν (π.χ. K-Means [28]) δείχνει ότι Fpga's μπορούν νσ προσφέρουν αρκετά σε σχέση με τις GPU.

1.7 Ενσωμάτωση των FPGA στα DataCenter

Πώς είναι δυνατόν να ενσωματωθούν οι δύσκολες στο σχεδιασμό υπολογιστικές μονάδες fpga στις ταχέως μεταβαλλόμενες υποδομές των κέντρων δεδομένων και του cloud computing. Σε αυτή τη διπλωματική εργασία θα χρησιμοποιήσουμε ένα συνδυασμό υφιστάμενων εργαλείων για την επίτευξη αυτής της ενσωμάτωσης. Παρακάτω παρουσιάζουμε τα εργαλεία αυτά ονομαστικά, και αργότερα σε αυτό το κεφάλαιο αναλύουμε το καθένα ξεχωριστά.

- Εργαλεία Λογισμικού
  - Apache Spark
  - Xilinx SDSoC
  - Xilinx Vivado HLS
- Hardware εργαλεία.
  - Zynq IC
  - Zedboard Development Board
  - Pynq Development Board
  - ZC702 Development Board

1.7.1 Apache Spark

Επισκόπηση

Το Apache Saprk είναι ένα πρότζεκτ ανοικτού πηγαίου κώδικα που στοχεύει να καταστήσει ευκολότερη τη χρήση του cluster computing.Το Apache Spark παρέχει στους προγραμματιστές ένα API (application programming interface) για τη δημιουργία και τον χειρισμό μιας δομής δεδομένων που ονομάζεται RDD (Resilient Distributed Dataset). Ένα Resilient Distributed Dataset είναι ένα σύνολο στοιχείων που χωρίζονται και διανέμονται μεταξύ των μηχανών (κόμβων) ενός cluster, με τρόπο ανεκτικό σε σφάλματα. Κάθε RDD είναι μόνο ανάγνώσιμο ώστε να είναι ευκολότερη η διατήρηση της συνεκτικότητας και της συνέπειας, με αποτέλεσμα η κάθε λειτουργία σε ένα RDD να δημιουργεί ένα νέο RDD, αντί να χειρίζεται το ίδιο. Το Spark σχεδιάστηκε ως βελτιωμένη έκδοση του μοντέλου προγραμματισμού MapReduce.[8]

Τα RDDs διευκολύνουν την υλοποίηση τόσο των επαναληπτικών αλγορίθμων, που επισκέπτονται το σύνολο δεδομένων τους πολλές φορές σε ένα βρόχο,όσο και την στατιστική ανάλυση των δεδομένων. Ο χρόνος εκτέλεσης τέτοιων εφαρμογών (με χρήση και εφαρμογή του μοντέλου MapReduce, όπως συνηθίζεται στο Apache Hadoop) μπορεί να μειωθεί κατά αρκετές τάξεις μεγέθους. Μεταξύ της κατηγορίας των επαναληπτικών αλγορίθμων είναι οι αλγόριθμοι εκπαίδευσης για συστήματα μηχανικής μάθησης, οι οποίοι αποτέλεσαν την αρχική ώθηση για την ανάπτυξη του Apache Spark[8]. Το Apache Spark απαιτεί διαχειριστή cluster και ένα διανεμημένο σύστημα αποθήκευσης, προκειμένου να λειτουργήσει. Οι υποστηριζόμενοι διαχειριστές cluster είναι ο ανεξάρτητος διαχειριστής Spark, ο οποίος είναι ο προκαθορισμένος διαχειριστής μέσα στο Spark, το Hadoop YARN και ο Apache Mesos. Σχετικά με την αποθήκευση το Apache Spark μπορεί να διασυνδέεται με πολλαπλά κατανεμημένα συστήματα αρχείων όπως το Hadoop (HDFS), το MapR-FS, το Cassandra, το OpenStack Swift, το Amazon S3, το Kudu ή οποιοδήποτε custom σύστημα αρχείων. Τέλος, το Spark υποστηρίζει μια ψευδο-κατανεμημένη τοπική λειτουργία, που χρησιμοποιείται κυρίως για δοκιμαστικούς σκοπούς , στον οποίο δεν χρειάζεται να υπάρχει εξωτερικός διαχειριστής για το cluster αλλά ούτε και ένα κατανεμημένο σύστημα αρχείων.

Το προγραμματιστικό μοντέλο Map Reduce

Το μοντέλο προγραμματισμού MapReduce, το οποίο έχει σχεδιαστεί για την εκτέλεση παράλληλων εργασιών σε μεγάλους όγκους δεδομένων, αποτελείται από δύο βασικές λειτουργίες[9]

- Operation ***map()*** : αυτή η λειτουργία αντιστοιχίζει μια συνάρτηση σε κάθε τμήμα δεδομένων, δημιουργώντας ένα νέο σύνολο δεδομένων που περιέχει τις μετασχηματισμένες τιμές
- Operation ***reduce()***:αυτή η ενέργεια εκτελεί μια συσσώρευση στο νεοσυσταθέν σύνολο δεδομένων

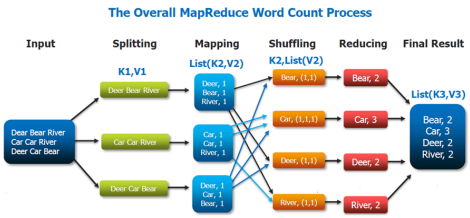
π.χ. σε περίπτωση που το σύνολο δεδομένων μας αποτελείται από τις ακόλουθες σειρά χαρακτήρων
Dear Bear River,Car Car River,Deer Car Bear

και ο σκοπός μας είναι να καθορίσουμε πόσες φορές κάθε λέξη εμφανίζεται τότε, το Map θα ήταν μια λειτουργία όπως αυτή:

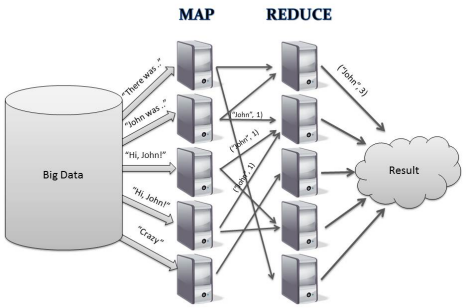
```
map_function(String phrase){
        array_of_words = tokenize( phrase )
        foreach word in array_of_words:
                        create_tuple( (key = word,value = 1) )
}
```

και το *reduce()* θα ήταν μια συνάρτηση όπως αυτή:

```
reduce_function(){
                for all touple having same key:
                        sum += touple.value
                result.append( (key,sum) )
}
```



(α΄) 1. το σύνολο δεδομένων διαχωρίζεται μεταξύ των κόμβων του cluster, η λειτουργία 2.το map δημιουργεί το νέο σύνολο δεδομένων που περιέχει τις τούπλες, 3. τα αποτελέσματα γίνονται reduce στο πρόγραμμα-οδηγό



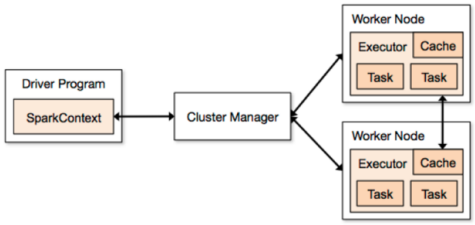
(β΄) ένα άλλο παράδειγμα του map-reduce, σε αυτή την περίπτωση μετράμε το πλήθος των λέξεων "John"

Σχήμα 1.4: Το προγραμματιστικό μοντέλο MapReduce

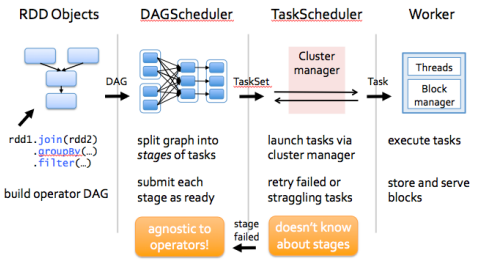
Ροη εκτέλεσης στο Apache Spark

Οποιαδήποτε εφαρμογή έχει υλοποιηθεί με χρήση του Apache Spark API αποτελείται από ένα πρόγραμμα *οδηγό* που τρέχει την κύρια λειτουργία του προγράμματος και είναι υπεύθυνο για την εκτέλεση διαφόρων παράλληλων λειτουργιών στο cluster. Κάθε μηχανή σε ένα cluster ονομάζεται *κόμβος*. Ένας κόμβος κατά το χρόνο εκτέλεσης ενεργεί ως host ενός *spark worker* και κάθε spark worker ενεργεί ως host ενός επιθυμητού αριθμού από *spark executors*. Το μηχάνημα στο οποίο τρέχει το πρόγραμμα οδήγησης ονομάζεται *master* ενώ τα μηχανήματα στα οποία τρέχουν οι spark workers καλούνται *slaves*. Όταν το πρόγραμμα οδήγησης που εκτελείται στον master υποβάλλει μια εργασία στους σκλάβους του cluster, η εργασία κατανέμεται αρχικά σε *stages* από έναν δρομολογητή που ονομάζεται DAGscheduler (Direct Acyclic Graph), υπάρχουν δύο βασικές κατηγορίες σταδίων, τα maps και τα reduces. Στη συνέχεια, κάθε στάδιο διαχωρίζεται σε πολλαπλά μικρότερα στάδια (*tasks*),συγκεκριμένα ένα task ανά τμήμα

δεδομένων στο cluster, έτσι ώστε μια εργασία να εκτελείται σε συγκεκριμένο τμήμα δεδομένων σε συγκεκριμένο εκτελεστή (spark executor). Τέλος, ένας χρονοπρογραμματιστής εργασιών εκτελεί τις εργασίες μέσω του cluster manager (π.χ. Νήματα ή Messos) στους spark executors[10].



(α′) Μια αφαιρετική αναπαράσταση της Αρχιτεκτονική Apache Spark . Το πρόγραμμα οδηγός εκτελείται στον master και επικοινωνεί με το Apache Spark και το cluster μέσω ενός αντικειμένου που ονομάζεται SparkContext.



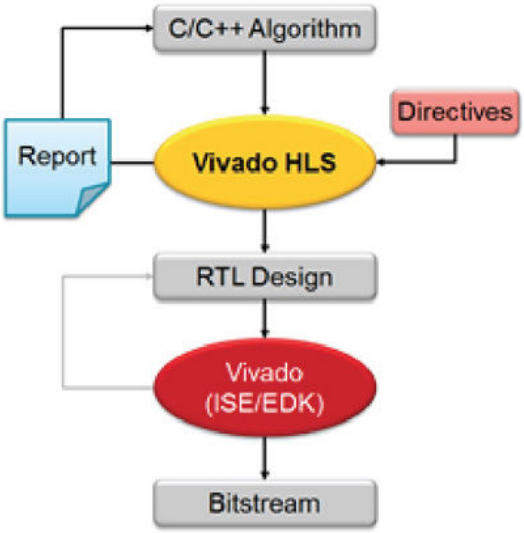
(β′) η ροή εκτέλεσης apache όπως περιγράφεται παραπάνω

1.7.2 Vivado HLS

Το Vivado HLS (High Level Synthesis) είναι ένα εργαλείο σχεδίασης που δημιουργήθηκε από την Xilinx και επιτρέπει το σχεδιασμό και την υλοποίηση application specific IPs , από γλώσσες υψηλού επιπέδου όπως C και C ++, επιτρέποντας έτσι στους κατασκευαστές επιταχυντών να δουλέψουν σε υψηλότερο επίπεδο αφαίρεσης χωρίς να θυσιάζουν τις υψηλές απαιτήσεις απόδοσης. Πρόκειται για μια προσπάθεια με στόχο να διευκολυνθούν οι προγραμματιστές λογισμικού, στην δημιουργία επιταχυντών υλικού, για τα τμήματα των αλγορίθμων που έχουν έντονες υπολογιστικές απαιτήσεις , και να μπορούν να τα ενσωματωσούν με ευκολία σε οποιαδήποτε συσκευή της εταιρείας Xilinx [11].

Η ροή σχεδιασμολυ φαίνεται στο σχήμα [12]. Ο σχεδιαστής γράφει πρώτα τον πηγαίο κώδικα που θέλει να επιταχύνει σε C ή C ++. Το HLS επιτρέπει στον σχεδιαστή να αξιολογήσει γρήγορα τη λειτουργική ορθότητα του αλγορίθμου, πραγματοποιώντας ένα προκαθορισμένο από το χρήστη δοκιμαστικό test σε κάθε βήμα του σχεδιασμού. Η ενσωμάτωση του testbench είναι ζωτικής σημασίας για την εξασφάλιση της ορθότητας της διαδικασίας σχεδιασμού και την επιτάχυνση της διαδικασίας σχεδίασης και υλοποίησης, καθώς είναι πολύ πιο εύκολη η αξιολόγηση της λειτουργικότητας ενός συστήματος γραμμένου σε γλώσσα υψιλού επιπέδου, παρά η εξακρίβωση της λειτουργικότητας μιας γλώσσας περιγραφής υλικού, αφού η δεύτερη χρειάζεται εξωτερικά εργαλεία προσομοίωσης που λειτουργούν σε πολύ χαμηλό επίπεδο (π.χ. Modelsim). Στη συνέχεια ακολουθεί η αυτόματη δημιουργία του αλγορίθμου σε επίπεδο RTL (Register Level Level), η διαδικασία αυτή ονομάζεται *synthesis*. Η σύνθεση RTL ελέγχεται από ένα σύνολο οδηγιών που ονομάζονται *pragmas*, οι οποίες ορίζονται από τον σχεδιαστή στον πηγαίο κώδικα είτε απευθείας μέσα στον κώδικα C / C ++ με τη μορφή οδηγιών μεταγλωττιστή, είτε σε ξεχωριστό αρχείο. Το HLS εκτελεί κάποιες προεπιλεγμένες βελτιστοποιήσεις, ωστόσο, προκειμένου να επιτευχθεί βέλτιστη επίδοση, ο σχεδιαστής

πρέπει να κάνει αποτελεσματική χρήση των οδηγιών *pragma*. Σε αυτό το βήμα ο σχεδιαστής εξετάζει τις αναφορές HLS και, αν χρειαστεί, επαναλαμβάνει τη διαδικασία είτε με την αναδιάρθρωση του αρχικού κώδικα είτε με τη χρήση διαφορετικών οδηγιών βελτιστοποίησης. Στο RTL εφαρμόζεται το testbench με στόχο την επιβεβαίωση της ορθής του λειτουργίας .
Τέλος, ένα κύκλωμα σε επίπεδο λογικών πυλών δημιουργείται από το RTL από το οποίο κατασκευάζεται ένα bitstream αρχείο που στοχεύει μια συγκεκριμένη συσκευή FPGA. Ο προγραμματιστής έχει τη δυνατότητα να συσκευάσει το σχέδιό του σε πολυάριθμες μορφές IP για περαιτέρω χρήση στο μέλλον. Ένα από τα κρίσιμα βήματα που



Σχήμα 1.6: ροή σχεδίασης HLS

προαναφέρθηκαν είναι εκείνο της αξιολόγησης των αναφορών που προκύπτουν από το εργαλείο της σχεδίασης, μετά τη σύνθεση. Η εφαρμογή μπορεί να βελτιώνεται συνεχώς, αξιολογώντας την ανατροφοδότηση από αυτές τις αναφορές, αναφορικά με τρεις κύριες μετρήσεις:

- ***Χρησιμοποίηση πόρων (Area)*** : Η μετρική αυτή αφορά τους πόρους που καταναλώνονται για την υλοποίηση του σχεδίου. Οι πιο συνηθισμένοι πόροι που αναφέρονται σε αυτήν τη αναφορά είναι η χρήση των διαθέσιμων LUT, η χρήση των DSP, η χρήση της BRAM και η χρήση των καταχωρητών.

- ***Χρόνος εκτέλεσης (Latency)***: Ο Χρόνος εκτέλεσης μιας συνάρτησης είναι ο χρόνος που παρέρχεται από τη στιγμή έναρξης της εκτέλεσης μέχρι την παραγωγή όλων των εξόδων. Σε έναν σχεδιασμό μιας εφαρμογής σε FPGA η καθυστέρηση εκτέλεσης μπορεί να μετρηθεί με ακρίβεια κύκλων του ρολογιού.

- ***Διάστημα επανενεργοποίησης (Initiation Interval II)***:Το διάστημα επανενεργοποίησης είναι οι κύκλοι που απαιτούνται από την επιτάχυμενη συνάρτηση-πυρήνα πάνω στο υλικού, προτού μπορέσει να αποδεχθεί νέα είσοδο.

1.7.3 Zynq - 7000

Περίληψη

Το Zynq-7000 [13] είναι ένα σύστημα SoC (System on Chip) που ενσωματώνει ένα σύστημα επεξεργασίας διπλού πυρήνα Cortex-A9 MPCore (PS) και προγραμματιζόμενη λογική Xilinx (PL) σε μία μόνο συσκευή. Ο Arm Cortex-A9 MPCore είναι η καρδιά του συστήματος επεξεργασίας και περιλαμβάνει μνήμη παω στο chip, εξωτερικές διεπαφές μνήμης και πλούσιο σετ περιφερειακών I / O. Το Zynq-7000 επιχειρεί να διευκολύνει και να κάνει ταχύτερη την διασύνδεση του λογισμικού που εκτελείται πάνω στο PS και των προσαρμοσμένων επιταχυντών που εκτελούνται στο PL. Επιπλέον, η ενσωμάτωση του PS με το PL παρέχει επίπεδα απόδοσης που δυο ξεχωριστά ολοκληρωμένα (για παράδειγμα, μια CPU και ένα FPGA) δεν μπορούν να συναγωνιστουν λόγω του περιορισμένου εύρους ζώνης I / O, και της μεγάλης κατανάλωσης ισχύος ισχύος που απαιτείτα απο δύο ξεχωριστά ολοκληρωμένα.

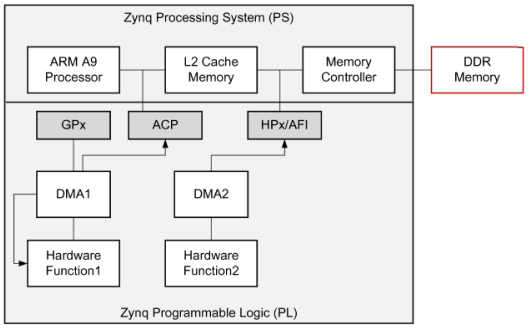
Διεπαφή PS-PL

Τα σημαντικοότερα στοιχεία της διεπαφής PS-PL είναι τα ακόλουθα[14]:

- Δυο θύρες I/O γενικού σκοπού (GP0-1)
  - Δυο 32-bit AXI master διεπαφές.
  - Δυο 32-bit AXI slave διεπαφές.
- Τέσσερις Θύρες υψηλής επίδοσης (High Performance Ports HP0-3)
  - 32-bit/64-bit διαμορφώσιμες, buffered AXI slave διεπαφές με πρόσβαση ,μέσω ενός συστήματος διασύνδεσης μνήμης (memory interconnect module), στην On Chip μνήμη (OCM) και την μνήμη DDR.
- Accelerator Coherence Port (ACP)
  - 64-bit AXI slave διεπαφή με δυνατότητα πρόσβασης στην cache της CPU χωρίς να παραβιάζεται η συνέπεια της μνήμης
- Clocks και Resets
  - Τέσσερις έξοδοι ρολογιών από το PS προς το PL με έλεγχο έναρξης/παύσης.
  - Τέσσερις έξοδοι επαναφοράς(reset) από το PS προς το PL.

Όπως φαίνεται στο απλοποιημένο διάγραμμα παρακάτω, το μπλοκ του συστήματος επεξεργασίας (PS) των συσκευών Zynq έχει τρία είδη θυρών που χρησιμοποιούνται για τη μεταφορά δεδομένων από τη μνήμη του επεξεργαστή του Zynq στην προγραμματιζόμενη λογική (PL) και πίσω. Είναι η επιταχυμένη θύρα συνεκτικότητας (ACP)που επιτρέπει στο υλικό να έχει άμεση πρόσβαση στην L2 Cache του επεξεργαστή με συνεκτικό τρόπο, οι θύρες υψηλής απόδοσης 0-3 (HP0-3), οι οποίες παρέχουν άμεση αποθήκευση στην DDR μνήμη ή τη μνήμη on-chip από το υλικό παρακάμπτοντας τη

χρήση της cache μνήμης του επεξεργαστή χρησημοποιώντας ασύγχρονη διασύνδεση FIFO (AFI) και οι θύρες IO γενικής χρήσης (GP0 / GP1) που επιτρέπουν στον επεξεργαστή την ανάγνωση / εγγραφή καταχωρητών υλικού.



Σχήμα 1.7: Αφαιρετική παρουσίαση της διεπαφής PS-PL στο Zynq

Θύρα	Ιδιότητες
ACP	Η λειτουργία επεξεργαστή και υλικού έχει πρόσβαση στην ίδια γρήγορη μνήμη προσωρινής αποθληκευσης (cache) ως κοινή μνήμη. Η συνοχή είναι εγγυημένη από μια μονάδα μονάδα υλικού που ονομάζεται cache Snoop unit.
HPx	Η μνήμη cache πρέπει να συγχρονιστεί με την DDR πριν από η συνάρτηση υλικού που βρισκεται στο PL μπορέσει να διαβάσει τα δεδομένα από την DDR με την χρήση αυτών των θυρών.
GPx	Ο επεξεργαστής γράφει / διαβάζει δεδομένα απευθείας από/προς την συνάρτηση υλικού στο PL. Δεν συνίσταται για μεγάλες μεταφορές δεδομένων.

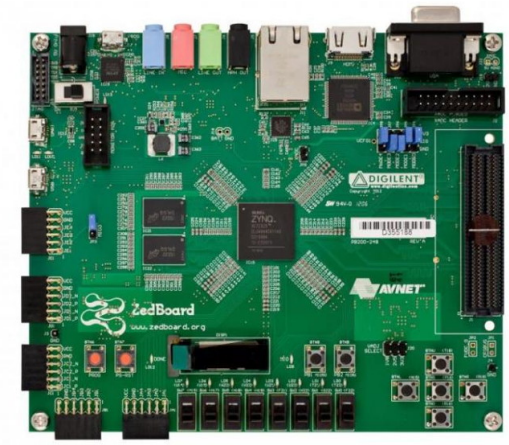
1.7.4 ZedBoard

Η πρώτη πλακέτα ανάπτυξης που χρησιμοποιήσαμε είναι το Zedboard [15], είναι ένα board πρωτοτυποποίησης για το Zynq-7000, το οποίο υποστηρίζεται πλήρως από τα εργαλεία της Xilinx.

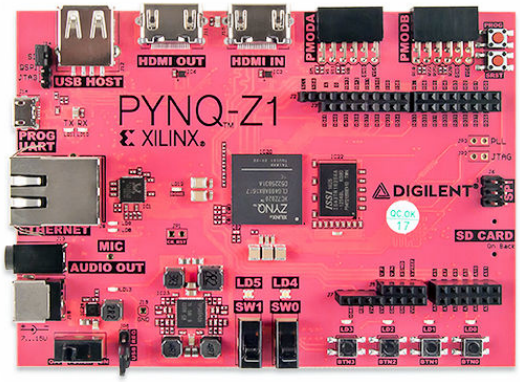
Στο Zedboard δοκιμάσαμε τον κώδικα γραμμένο σε C μαζί με τους επιταχυντές,όπως αυτά προέκυψαν από το πρόγραμμα ανάπτυξης SDSoC, για να πάρουμε τα πρώτα αποτελέσματα της επιτάχυνσης.

1.7.5 Pynq

Η δεύτερη πλακέτα ανάπτυξης που χρησιμοποιήσαμε είναι το Pynq,το οποίο είναι μια άλλη πλακέτα πρωτηποποίησης με βασικό chip το Zynq-7000. Σε αυτήν την πλακέτα συσκευάσαμε το IP του επιταχυντή με το κομμάτι του λογισμικού γραμμένο σε python αντί για C, καθώς η Xilinx παρέχει ένα image για αυτό την πλακέτα που περιέχει όλους τους απαραίτητους οδηγούς για τη διασύνδεση της Python με τα IP που βρίσκονται στο PL του Zynq.



Σχήμα 1.8: ZedBoard



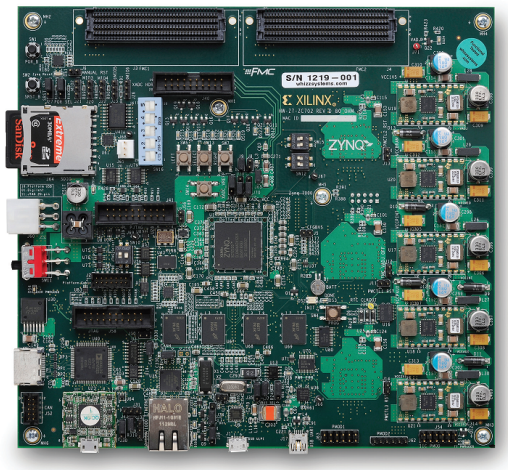
Σχήμα 1.9: Pynq

Χρησιμοποιήσαμε επίσης τέσσερα pynqs για να δημιουργήσουμε ένα μίνι πρωτότυπο cluster συντονιζόμενο από το apache-spark με στόχο να τρέξει την επιταχυνόμενη έκδοση του ALS..

1.7.6 Zc702

Τέλος χρησιμοποιήσαμε το board zc702 [16] για να μετρήσουμε την κατανάλωση ενέργειας της επιταχυνόμενης έκδοσης του ALS, διότι η συγκεκριμένη πλακέτα παραδίδεται με εξωτερικούς ελεγκτές που μπορούν να μετρήσουν την ισχύ σε πραγματικό χρόνο.

Αυτή η πλακέτα χρησιμοποιεί το ίδιο SoC (Zynq-7000) με το ZedBoard και το Pynq.



Σχήμα 1.10: zc702

2 Συστήματα Συστάσεων

2.1 Εισαγωγή

Τα συστήματα συστάσεων (recommendation systems) άλλαξαν τον τρόπο με τον οποίο επικοινωνούν οι ιστότοποι με τους χρήστες τους. Αντί, οι ιστότοποι , να παρέχουμε μια στατική εμπειρία, στην οποία οι χρήστες αναζητούν και δυνητικά αγοράζουν προϊόντα, τα συστήματά συστάσεων αυξάνουν την αλληλεπίδραση με στόχο την προσφορά μιας πλουσιότερης εμπειρίας. Τα συστήματα συστάσεων εντοπίζουν αυτόματα συστάσεις για τους χρήστες[17], βάσει των προηγούενων αγορών και αναζητήσεων τους καθώς επίσης και με βάση την συμπεριφοράς άλλων χρηστών. Τα συστήματα συστάσεων επιχειρούν να προτείνουν προϊόντα ή υπηρεσίες στους χρήστες ακολουθώντας μία από τις παρακάτω προσεγγίσεις[17]:

- Συνεργατικό Φιλτράρισμα (Collaborative Filtering - CF)
  - βασισμένο σε μοντέλο (model based)
  - βασισμένο σε μνήμη (memory based)
    * με επίκεντρο τον χρήστη(User-based)
    * με επίκεντρο το αντικείμενο (Item-based)[18]1
- Φιλτράρισμα Περιεχομένου (Content-Based Filtering)
- Υβριδικό.

2.2 Συνεργατικό Φιλτράρισμα

Το συνεργατικό φιλτράρισμα (CF) επιδιώκει την αναζήτηση πρότυπων στην προηγούμενη συμπεριφορά και δραστηριότητα του χρήστη για την παραγωγή νέων συστάσεων. Η συλλογή δεδομένων χρήσης είναι υποχρεωτική, όπως για παράδειγμα αξιολογήσεις χρηστών για ταινίες ή κλικ χρηστών σε αρθρα, που δείχνουν πόσο τους άρεσαν

τα αντίστοιχα αντικείμενα προς σύσταση. Η βασική ιδέα στην περίπτωση αυτή είναι ότι οι χρήστες που έχουν αξιολογήσει στοιχεία με συγκεκριμένο τρόπο κατά το παρελθόν, πιθανόν να δώσουν παρόμοιες αξιολογήσεις σε νέα στοιχεία στο μέλλον. Είναι σημαντικό να υπογραμμίσουμε ότι σε αντίθεση με το Φιλτράρισμα Περιεχομένου δεν χρειάζονται περαιτέρω πληροφορίες σχετικά με τα αντικείμενα (π.χ. τίτλος, περιγραφή, έτος, συγγραφέας, σκηνοθέτης κ.λπ.) ή τους χρήστες (π.χ. ηλικία, φύλο κλπ. ..). για τη δημιουργία συστάσεων. Υπάρχουν δυο βασικές προσεγγίσεις για το συνεργατικό φιλτράρισμα, η προσέγγιση που είναι βασισμένη στο μοντέλο και η προσέγγιση που είναι βασισμένη στην μνήμη. Στις μεθόδους που βασίζονται στη μνήμη, οι αξιολογήσεις των χρηστών για τα αντικείμενα χρησιμοποιούνται άμεσα για την παραγωγή νέων συστάσεων, ενώ οι προσεγγίσεις βασισμένες σε μοντέλα χρησιμοποιούν την ανατροφοδότηση (π.χ. αξιολογήσεις) για τη δημιουργία προγνωστικών μοντέλων στην βάση των οποίων γίνονται οι συστάσεις για τα άλλα αντικείμενα. Οι τεχνικές συνεργατικού φιλτραρίσματος τείνουν να παράγουν καλύτερα αποτελέσματα από τις τεχνικές που βασίζονται στο περιεχόμενο και δεν απαιτούν καμία γνώση για τα αντικείμενα. Παρ 'όλα αυτά υποφέρουν από το πρόβλημα της ψυχρής εκκίνησης, που σημαίνει ότι δεν μπορούν να παρέχουν συστάσεις σε ένα νέο χρήστη του οποίου τα γούστα είναι άγνωστα, είτε να προτείνουν νέα αντικείμενα στους χρήστες και επίσης υποφέρουν από μεροληψία δημοτικότητας, πράγμα που σημαίνει ότι τα δημοφιλέστερα αντικείμενα τείνουν να συνιστώνται πιο συχνά αφήνοντας πιο σπάνια αντικείμενα απαρατήρητα.

2.2.1 Βασισμένο σε Μοντέλο

Οι τεχνικές που βασίζονται στη μνήμη χωρίζονται σε CF που βασίζονται σε χρήστη και CF βασισμένο σε αντικείμενα. Το φιλτράρισμα βασισμένο σε χρήστες συγκεντρώνει χρήστες που έχουν παρόμοια προτίμηση για αντικείμενα και συνιστούν με βάση αυτό που τους αρέσει, ενώ το φιλτράρισμα με βάση τα αντικείμενα συγκεντρώνει αντικείμενα που έχουν προταθεί από παρόμοιους χρήστες και συνιστα ανάλογα.

2.2.2 Βασισμένο σε Μνήμη

Οι μέθοδοι που βασίζονται σε μοντέλα θεωρούνται γενικά πιο προχωρημένες μέθοδοι για τη δημιουργία συστημάτων συστάσεων.Συγκεκριμένα βοηθούν να ξεπεραστούν κάποια μειονεκτήματα των μεθόδων που βασίζονται στη μνήμη. Αυτή η προσέγγιση προσπαθεί να δημιουργήσει μοντέλα που αφορούν την αλληλεπίδραση χρήστη-στοιχείου, αυτά τα μοντέλα προέρχονται από αλγορίθμους μηχανικής μάθησης. Υπάρχουν πολλοί αλγόριθμοι που χρησιμοποιούνται για τη δημιουργία μοντέλου, οι περισσότεροι από τους οποίους είναι τα bayesian δίκτυα, ομαδοποίηση, ταξινόμηση, παλινδρόμηση, διατριβή επικεντρωνόμαστε στην επιτάχυνση του φιλτραρίσματος συνεργασίας που βασίζεται στο μοντέλο που κάνει χρήση της τεχνικής των εναλλασσόμενα ελάχιστων τετράγωνων (Alternating Least Squares - ALS).

2.2.3 Παραγοντοποίηση Πίνανα

Η μέθοδος παραγοντοποίησης μήτρας παίρνει ως είσοδο έναν δισδιάστατο πίνακα $R_{n\ timesm}$ όπου n και m είναι ο αριθμός των χρηστών και των αντικειμένων αντίστοιχα και προσπαθεί να την αποσυνθέσει σε δύο μήτρες $U_{n\ timesk}$ και $M_{k\ timesm}$, όπου k είναι

ο αριθμός των χαρακτηριστικών, έτσι ώστε $U_{n\ timesk} x M_{k\ timesm} = R_{n\ timesm}$.
Η διαίσθηση πίσω από αυτή την προσέγγιση είναι ότι κάθε χρήστης συνιστά ένα στοιχείο λαμβάνοντας υπόψη τις προτιμήσεις του για συγκεκριμένα χαρακτηριστικά αυτού του στοιχείου. Ας υποθέσουμε ότι μια ταινία εκφράζεται ώς ένα διάνυσμα $k = 5$ χαρακτηριστικών και ένας χρήστης προτείνει ταινίες με το ίδιο σύνολο κανόνων / χαρακτηριστικών $k = 5$. Το ακόλουθο παράδειγμα είναι ένα παράδειγμα μιας τέτοιας κατάστασης, ο πρώτος πίνακας

Πίνακας 2.1: Οι προτιμήσεις ενός χρήστη εκφρασμένες σε ένα χώρο διανυσμάτων 5 στοιχείων

χαρακτηριστικό	τιμή
είναι κωμωδία	0.8
δεν είναι δράμα	0.9
πρωταγωνιστεί ο μαρλο μπράντο	0.9
είναι επιστημονικής φαντασίας	0.8
διάρκεια μικρότερη από 2 ώρες	0.7

Πίνακας 2.2: μια ταινία εκφρασμένη σαν διάνυσμα στόν ίδιο χώρο 5 χαρακτηριστικών

χαρακτηριστικό	τιμή
είναι κωμωδία	0.8
δεν είναι δράμα	0.7
πρωταγωνιστεί ο μαρλο μπράντο	1
είναι επιστημονικής φαντασίας	0.2
διάρκεια μικρότερη από 2 ώρες	1

Βρίσκοντας το εσωτερικό γινόμενο για το διάνυσμα του συγκεκριμένου χρήστη και της συγκεκριμένης ταινίας, μπορούμε να βρούμε μια πιθανή βαθμολογία για για αυτό το ζευγάρι χρήστη ταινίας.

$$\begin{bmatrix} 0.8 \\ 0.9 \\ 0.9 \\ 0.8 \\ 0.7 \end{bmatrix} \times \begin{bmatrix} 0.8 & 0.7 & 1 & 0.2 & 1 \end{bmatrix} \quad = 3.03$$

Οι τεχνικές που προσεγγίζουν το πρόβλημα με το μοντέλο παραγοντοποίησης πίνακα, επιχειρούν να εξάγουν αυτόματα τα χαρακτηριστικά, και να δημιουργήσουν τα διανύσματα για κάθε αντικείμενο και κάθε χρήστη, κάνοντας συστάσεις βασισμένο στο εσωτερικό γινόμενο αυτών των διανυσμάτων.

2.3 Content-Based Filtering

Το φιλτράρισμα βάσει περιεχομένου[19] αναλύει τις τις επιλογές του χρήστη στο παρελθόν και συνιστά αντικείμενα που θεωρεί πως είναι παρόμοια με αυτά. Η ομοιότητα των στοιχείων προκύπτει με βάση το περιεχόμενο ή τις ιδιότητες του αντικειμένου

(π.χ. τίτλος, έτος, περιγραφή) και όχι τον τρόπο με τον οποίο τα χρησιμοποιούν. Για παράδειγμα, αν σε κάποιο χρήστη αρέσουν οι ταινίες "The Matrix" και "The Matrix Reloaded" τότε χρησιμοποιώντας τις λέξεις στον τίτλο, ο αλγιθμος μπορεί να προτείνει την ταινία "The Matrix Revolutions".Το φιλτράρισμα βάσει περιεχομένου αξίζει να χρησιμοποιηθεί όταν υπάρχουν διαθέσιμα μεγάλα ποσά δεδομένων που μπορούν να χρησιμοποιηθούν για την περιγραφή κάθε στοιχείου ως διάνυσμα χαρακτηριστικών (π.χ. τίτλος, έτος, περιγραφή και διάφορα άλλα μεταδεδομένα). Μια κοινή προσέγγιση παραδείγματος χάριν είναι η δημιουργία μιας «σακούλας λέξεων» από την περιγραφή κάθε στοιχείου. Στη συνέχεια, δημιουργείται ένα μοντέλο για κάθε γούστο χρήστη χρησιμοποιώντας αυτά τα διανύσματα χαρακτηριστικών. Μια ποικιλία τεχνικών ανάκτησης πληροφοριών (π.χ. tf-idf) και μηχανικής μάθησης (π.χ. Naive Bayes, μηχανήματα φορέα υποστήριξης, δέντρα αποφάσεων κ.λπ.) μπορεί να χρησιμοποιηθεί για τη δημιουργία ενός μοντέλου χρήστη με βάση το οποίο μπορούν να δημιουργηθούν οι αντίστοιχες συστάσεις . Ας υποθέσουμε ότι έχουμε κάποιους χρήστες που μας έδωσαν ρητή ανατροφοδότηση για μια σειρά αντικειμένων αξιολογώντας τα.Κάνουμε την παραδοχή ότι η βαθμολογία κυμαίνεται από 1 έως 5. Ο πιο εύκολος και τυπικός τρόπος για να αναπαραστήσει κανείς τις αξιολογήσειν είναι ένας δισδιάστατος πίνακας με την ακόλουθη μορφή.

	$item_1$	$item_2$	$item_3$	$item_4$	$item_5$	$item_6$
$user_1$	4	3			5	
$user_2$	5		4		4	
$user_3$	4		5	3	4	
$user_4$		3				5
$user_5$		4				4
$user_6$			2	4		5

αν γνωρίζουμε ότι η περιγραφή κάθε αντικειμένου είνια η ακόλουθη:

αντικείμενο	περιγραφή
$item_1$	Introduction to Recommender Systems
$item_2$	Machine learning Paradigms
$item_3$	Social Network-based Recommender Systems
$item_4$	Learning Spark
$item_5$	Recommender Systems Handbook
$item_6$	Recommender Systems and the Social Web

μπορούμε τώρα να περιγράψουμε με διάνυσμα το διαθέσιμη πληροφορία κάθε αντικειμένου, που σε αυτήν την περίπτωση είναι ο τίτλος τους.

	$item_1$	$item_2$	$item_3$	$item_4$	$item_5$	$item_6$
introduction	1					
recommender	1		1		1	1
systems	1		1		1	1
machine		1				
learning		1		1		
paradigms		1				
social			1			1
network-based			1			
spark				1		
handbook					1	
web						1

Μετά τη διαμόρφωση αυτής της αναπαράστασης κάθε στοιχείου είναι πολύ εύκολο να μετρήσουμε την ομοιότητα κάθε στοιχείου με όλα τα άλλα. Μπορούμε να χρησιμοποιήσουμε π.χ. την ευκλείδια απόσταση ή κάποια νόρμα γενικότερα. Σε αυτό το παράδειγμα θα χρησιμοποιηθεί ομοιότητα συνιμιτόνου η αλλιώς ομοιότητα εσωτερικού γινομένου. Αφού υπολογίσουμε την ομοιότητα για όλα τα πιθανά ζευγάρια βιβλίων, μπορούμε να διαμορφώσουμε τον πίνακα ομοιότητας.

	$item_1$	$item_2$	$item_3$	$item_4$	$item_5$	$item_6$
$item_1$	1	0	0.58	0	0.67	0.58
$item_2$	0	1	0	0.41	0	0
$item_3$	0.58	0	1	0	0.58	0.75
$item_4$	0	0.41	0	1	0	0
$item_5$	0.67	0	0.58	0	1	0.58
$item_6$	0.58	0	0.75	0	0.58	1

Τώρα πρέπει μόνο να βρούμε τα πιο όμοια αντικείμενα με εκείνα που έχει δωθελι καλή αξιολόγηση από κάθε χρήστη για να δημιουργήσουμε τις συστάσεις. Για παράδειγμα για τον $user_1$ έχουμε:

	$item_1$	$item_2$	$item_3$	$item_4$	$item_5$	$item_6$
$user_1$	4	3			5	

τώρα γνωρίζουμε ότι τα περισσότερο όμοια αντικείμενα με αυτά που έχει αξιολογήσει είναι τα:

- $item_1$ →$item_5$,$item_6$
- $item_2$ →$item_3$
- $item_5$ →$item_1$,$item_6$

Μια πιθανή αξιολόγηση του $user_1$ για το $item_6$ θα μπορούσε να είναι:

$$recommendation(item_6) =$$
$$= \frac{rat(item_1)sim(item_1, item_6) + rat(item_2)sim(item_2, item_6) + rat(item_5)sim(item_5, item_6)}{rat(item_1) + rat(item_2) + rat(item_5)}$$
$$= \frac{4 * 0.58 + 3 * 0 + 5 * 0.58}{0.58 + 0.58} = 4.5$$

Οι προσεγγίσεις που βασίζονται στο περιεχόμενο έχουν διαφορετικά πλεονεκτήματα και μειονεκτήματα σε σύγκριση με το συνεργατικό φιλτράρισμα. Συγκεκριμένα, δεν υποφέρουν από κρύα εκκίνηση, που σημαίνει ότι μπορούν να εκτελούν συστάσεις ακόμα και όταν υπάρχει ελάχιστη ανατροφοδότηση απο τον χρήστη. Επιπλέον, δεν υποφέρουν από την μεροληψία δημοτικότητας, που σημαίνει ότι μπορούν να προτείνουν σπάνια αντικείμενα ευκολότερα από τους cf αλγορίθμους. Από την άλλη πλευρά απαιτούν το περιεχόμενο του αντικειμένου να είναι αναγνώσιμο και να περιέχει κάποιου είδους νόημα, είναι πολύ εύκολο για έναν χρήστη να παγιδευτεί μέσα σε ένα συγκεκριμένο σύνολο αντικειμένων και τέλος είναι δύσκολο να συνδυαστούν μαζί τα χαρακτηριστικά του πολλών αντικειμένων.

2.4 Παραγοντοποίηση Πίνακα με χρήση εναλλασσόμενων ελάχιστων τετραγώνων (ALS)

Έστω ότι το $R = [r_{ij}]_{n_u\ timesn_m}$ υποδηλώνει την μήτρα χρηστών-ταινιών εισόδου, όπου κάθε στοιχείο r_{ij} αντιπροσωπεύει το βαθμό με τον οποίο αξιολόγησε ο χρήστης i στην ταινία j με κάθε τιμή να είναι είτε πραγματικός αριθμός είτε κενό, τα n_m και n_u υποδηλώνουν τον αριθμό των ταινιών και τον αριθμό των χρηστών αντίστοιχα. Στόχος μας είναι να γεμίσουμε τις ελλείπουσες-κενές τιμές του R με τιμές όσο το δυνατόν πιο κοντά στην πραγματικότητα, με βάση τις ήδη γνωστές τιμές[20][21].

Θα προσπαθήσουμε να μοντελοποιήσουμε τόσο τις ταινίες όσο και τους χρήστες με ένα διάνυσμα χαρακτηριστικών και κάθε βαθμολογία (γνωστή ή άγνωστη) ως το εσωτερικό γινόμενο του διανύσματος της αντίστοιχης ταινίας και του αντίστοιχου χρήστη. Έστω ότι $U = [u_i]$ είναι ο πίνακας που περιέχει τα διανύσματα χαρακτηριστικών των χρηστών όπου $u_i\ in\ mathbbR^{n_f}$ για $i = 1...n_u$, και έστω $M = [m_j]$ η μήτρα που περιέχει τα διανύσματα χαρακτηριστικών των ταινιών, όπου $m_j\ in\ mathbbR^{n_f}$ για όλα τα $j = 1...n_m$. Η διάσταση του χώρου χαρακτηριστικών είναι n_f,και πρακτικά είναι ο αριθμός των χαρακτηριστικών που ο αλγόριθμος θα πρέπει να μάθει για κάθε χρήστη και κάθε ταινία. Ο καθορισμός του καλύτερων δυνατου n_f καθώς και κάποιων άλλων παραμέτρων του μοντέλου, οι οποίες θα συζητηθούν αργότερα, μπορεί να επιτευχθεί με cross-validation ή άλλες δημοφιλείς τεχνικές που χρησιμοποιούνται στη μηχανική μάθηση[20][21].
Στην ιδανική περίπτωση, θα θέλαμε να επιτύχουμε $r_{ij} =< u_i, m_j >\ \ foralli, j$. Στην πράξη όμως προσπαθούμε να ελαχιστοποιήσουμε μια συνάρτηση απώλειας των U και M για να τα αποκτήσουμε τις καλυτερες δυνατές τιμές των διανυσμάτω χαρακτηριστικών. Σε αυτή την διπλωματική εξετάσαμε την ελαχιστοποίηση του μέσου τετραγωνικού σφάλματος (Mean-Squared-Error) αλλά ο σκοπός μας δεν είναι να συντονίσουμε τις παραμέτρους του μοντέλου όσο το δυνατόν καλύτερα για να ελαχιστοποιήσουμε το RMSE αλλά η επιτάχυνση του αλγορίθμου και η ελαχιστοποίηση της ενεργειακής κατανάλωσης. Η συνάρτηση απώλειας όπως απορρέει απο μια μόνο βαθμολογία έχει ως εξής.

$$L^2(r, u, m) = (r- < u, m >)^2 \tag{2.4.1}$$

Η συνάρτηση συνολικής απώλειας, λαμβάνοντας υπόψη το σύνολο των πινάκων

U, M, μπορεί να οριστεί συνεπώς ως η μέση απώλεια για όλες τις γνωστές βαθμολογίες.

$$L^{total}(R, U, M) = \frac{1}{n} \sum_{(i,j)\in I} L^2(r_{ij}, u_i, m_j) \tag{2.4.2}$$

όπου I είναι το σύνολο των δεικτών όλων των γνωστών αξιολογήσεων και n είναι ο αριθμός των στοιχείων μέσα στο I.
Σκοπός του αλγορίθμου μας είναι η ελαχιστοποίηση της συνάρτησης απώλειας, όπως φαίνεται παρακάτω

$$(U, M) = min_{(U,M)} L^{total}(R, U, M)$$

όπου $U\ in\ mathbbR^{n_u\ timesn_f}$ και $M\ \ mathbbR^{n_m\ timesn_f}$.
Έχουμε, λοιπόν,συνολικά $(n_u{+}n_m)\ timesn_f$ ελεύθερες παραμέτρους για να "μάθουμε". Επειδή οι γνωστές βαθμολογίες $n = \ lVertI\ rVert$ είναι πολύ λιγότερες από αυτές που πρέπει να μάθουμε, είναι πολύ εύκολο να γίνει overfit στα δεδομένα εισόδου. Το overfitting μπορεί να αποφευχθεί με την χρήση του όρου κανονικοποίησης Tckhonov [21] στη συνάρτηση συνολικού κόστους

$$L^{reg}(R, U, M) = L^{total}(R, U, M) + (\|U\Gamma_U\|^2 + \|M\Gamma_M\|^2) \tag{2.4.3}$$

όπου $Gamma_U$ και $Gamma_M$ είναι κατάλληλα επιλεγμένοι πίνακες Tikhonov Χρησιμοποιήσαμε πίνακες κανονικοποίησης Tikhonov που αναφέρονται στο [21], $Gamma_U = diag(n_{u_i})$ και $Gamma_M = diag(n_{m_j})$.

$$f(U, M) = \sum_{(i,j)\in I} (r_{i,j} - u_i^T m_j)^2 + (\sum_i n_{u_i}\|u_i\|^2 + \sum_j n_{m_j}\|m_j\|^2) \tag{2.4.4}$$

όπου n_{u_i} είναι ο αριθμός αξιολογήσεων που έχει κάνει ο χρήστης i και n_{m_j} ο αριθμός των αξιολογήσεων που έχει λάβει η ταινία j.

Ο αλγόριθμος που επιχειρήσαμε να επιταχύνουμε [21] μπορεί να συνοψιστεί στα παρακάτω βήματα.

1. Αρχικοποιήσαι τη μήτρα M, αναθέτοντας τη μέση βαθμολογία γαι κάθε ταινία ως πρώτη σειρά και μικρούς τυχαίους αριθμούς για τις υπόλοιπες καταχωρίσεις.

2. Κράτα σταθερό τον πίνακα M και ανανέωσε τις τιμές των U ελαχιστοποιώντας την αντικειμενική συνάρτηση (το άθροισμα των τετραγώνων τών σφαλμάτων)

3. Κράτα σταθερότον πίνακα U, ανανέωσε τις τιμές της μήτρας M με την ελαχιστοποίηση της αντικειμενικής συνάρτησης όπως στο προηγολυμενο βήμα.

4. Επαναλάβετε τα βήματα 2 και 3 μέχρι να ικανοποιηθεί μια συνθήκη τερματισμού.

Έστω ότι το I_i να δηλώσει το σύνολο των ταινιών j που ο χρήστης i έχει βαθμολογηθεί και I_j να δηλώσει τους χρήστες που έχουν βαθμολογήσει την ταινία j (Προφανώς ισχύει $mathbfcard(I_i) = n_{u_i} quad quad mathbfcard(I_j) = n_{u_j}$)

2.4.1 Διαίσθηση

Για κάθε γνωστή βαθμολογία r_{ij} του χρήστη i για το στοιχείο j, θέλουμε $u_i\, timesm_j = r_{ij}$, όπου u_i είναι το διάνυσμα χαρακτηριστικών του χρήστη i ή του ή αλλιώς η i 'στη σειρά του πίνακα U και m_j είναι το διάνυσμα χαρακτηριστικών της ταινίας j ή αλλιώς η j'στη στήλη του πίνακα M. Αν λοιπόν το $R(i, I_i)$ είναι το διάνυσμα γραμμών που περιέχει όλες τις γνωστές αξιολογήσεις του χρήστη i τότε θέλουμε

$$M_{I_i}^T u_i = R(i, I_i) \tag{2.4.5}$$

όπου $M_{I_i}^T$ είναι ο πίνακας που περιέχει τους διανύσματα χαρακτηριστικών των ταινιών που ο χρήστης i έχει αξιολογήσει.Το παραπάνω σύστημα συνήθως δεν έχει καμία ακριβή λύση, οπότε ο στόχος είναι να βρούμε το διάνυσμα u_i που ταιριάζει καλύτερα στις εξισώσεις και αυτό γίνεται πολλαπλασιάζοντας και τα δύο μέρη των εξισώσεων με τη ανάστροφο του $(M_{I_i}^T)_{n_{u_i}\ timesn_f}$ [22].

$$\begin{aligned} M_{I_i} M_{I_i}^T u_i &= M_{I_i} R(i, I_i) \\ \Rightarrow u_i &= K^{-1} L \\ where \quad K &= M_{I_i} M_{I_i}^T \\ and \quad L &= M_{I_i} R(i, I_i) \end{aligned} \tag{2.4.6}$$

2.4.2 Μαθηματική προσέγγιση

Τώρα θα δείξουμε πώς μπορεί να λυθεί το πρόβλημα για τον πίνακα U όταν δίνεται ο M (ακολουθείται η ίδια προσέγγιση όταν θέλουμε να βρούμε τον M και θεωρούμε σταθερό τον U). Η επίλυση του U σημαίνει την ενημέρωση κάθε γραμμής u_i του U επιλύοντας ένα πρόβλημα του γραμμικών ελάχιστων τετραγώνων που περιλαμβάνει τις γνωστές βαθμολογίες του χρήστη i και τις σειρές m_j του M που αντιστοιχούν στις ταινίες που ο i έχει αξιολογήσει.

$$\begin{aligned} &\frac{\partial f}{2\partial u_{ki}} = 0 \forall i, k \\ &\Rightarrow \sum_{j \in I_i} (u_i^T m_j - r_{ij}) m_{kj} + \lambda n_{ui} u_{ki} = 0 \forall i, k \\ &\Rightarrow \sum_{j \in I_i} m_{kj} m_j^T u_i + \lambda n_{ui} u_{ki} = \sum_{j \in I_i} m_{kj} r_{ij} \forall i, k \\ &\Rightarrow (M_{I_i} M_{I_i}^T + \lambda n_{u_i} E) u_i = M_{I_i} R(i, I_i) \forall i \\ &\Rightarrow u_i = A_i^{-1} V_i \forall i \end{aligned} \tag{2.4.7}$$

όπου $A_i = M_{I_i} M_{I_i}^T + \lambda n_{u_i} E, V_i = M_{I_i} R(i, I_i)$ και E είναι ο $n_f \times n_f$ μοναδιαίος πίνακας. Για ακόμα μια φορά M_{I_i} είναι ο υποπίνακας του M όπου έχουν επιλεχθεί οι στήλες $j \in I_i$ και $R(i, I_i)$ είναι η i'th γραμμή του R απο την οποία έχουν επιλεχθεί μόνο οι στήλες $j \in I_i$.
Παρομοίως , όταν θέλουμε να ανανεώσουμε τα στοιχεία του M κάθε m_j υπολογίζεται

κάνοντας χρήση των διανυσμάτων χαρακτηριστικών των χρηστών που έχουν αξιολογήσει την αντίστοιχη ταινία j και φυσικά των ίδιων των βαθμολογιών όπως φαίνεται στην συνέχεια:

$$m_j = A_j^{-1} V_j, \forall j \tag{2.4.8}$$

σε αυτήν την περίπτωση, $A_j = U_{I_j} U_{I_j}^T + \lambda n_{m_j} E$ and $V_j = U_{I_j} R(I_j, j)$. U_{I_j} είναι ο υποπίνακας U όπου μόνο οι γραμμές $i \in I_j$ έχουν επιλεχθεί και $R(I_j, j)$ είναι εα υποδιάνυσμα του R όπου μόνο οι γραμμές $i \in I_j$ της j'th στήλης έχουν επιλεχθεί.

3 Συγγραφή Αλγορίθμου

3.1 Τα σύνολα δεδομένων

Για να πραγματοποιήσουμε τις δοκιμές για αυτήν την διπλωματική και να παρουσιάσουμε τα αντίστοιχα αποτελέσματα χρησιμοποιήσαμε τα σύνολα δεδομένων Movielens [23], τα οποία συλλέχθηκαν από το *movielens.org*.Το MovieLens είναι ένα σύστημα συστάσεων στο διαδίκτυο και μια εικονική κοινότητα που συνιστά ταινίες προς παρακολούθηση για τους χρήστες του, με βάση τις προτιμήσεις τους, χρησιμοποιώντας συνεργατικό φιλτράρισμα μεταξύ των ταινιών και των μελών. Το MovieLens δημιουργήθηκε το 1997 από την GroupLens Research, ένα ερευνητικό εργαστήριο στο Τμήμα Πληροφορικής και Μηχανικών Υπολογιστών του Πανεπιστημίου της Μινεσότα, προκειμένου να συγκεντρώσει ερευνητικά δεδομένα για εξατομικευμένες συστάσεις [24]. Αυτά τα σύνολα δεδομένων θεωρούνται σταθερά και αξιόπιστα για ακαδημαϊκή έρευνα.

Το σύνολο *Movielens100k*[23] περιέχει 100000 βαθμολογίες από 943 χρήστες σε 1682 ταινίες.Κάθε χρήστης έχει βαθμολογήσει τουλάχιστον 20 ταινίες. Χρήστες και ταινίες είναι αριθμημένα συνεχόμενα ξεκινώντας από το ένα.Τα δεδομένα βρίσκονται σε τυχαία σειρά.
Παραθέτουμε μερικές γραμμές από το σύνολο δεδομένων *Movielens-100k*:

196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
...			

η πρώτη στήλη αποτελεί το id του χρήστη η δεύτερη στήλη αποτελέι το id της ταινίας η τρίτη την βαθμολογία και η τελική στήλη είναι ένα timestamp.

Movielens-1m[23] dataset , contains 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users who joined MovieLens in 2000.

- Τα IDs των χρηστών κινούνται από 1 εώς 6040 .
- Τα IDs των ταινιών κινούνται από 1 εώς 3952.
- Οι Βαθμολογίες έχουν γίνει σε κλίμακα 5 αστέρων (ακέραιοι αριθμοί μόνο).
- Κάθε χρήστης έχει βαθμολογήσει τουλάχιστον 20 ταινίες.

Παραθέτουμε μερικές γραμμές από το σύνολο δεδομένων *Movielens-1m*:

1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
...

Το φορμάτ αυτού του συνόλου δεδομένων είναι userID::movieID::rating::timestamp. Τα σύνολα δεδομένων Movielens [**?**] παρέχουν διάφορα δημογραφικά δεδομένα ώστε να μπορούν χρησιμοποιηθούν με πολλούς αλγορίθμους,ωστόσο εμείς δεν τα χρησιμοποιήσαμε την παρούσα διπλωματική.
Επειδή λείπουν κάποια IDsταινιών, δημιουργήσαμε ένα σκριπτ προ-επεξεργασίας σε python που αντιστοιχεί τα υπάρχοντα αναγνωριστικά (ID) χρηστών και ταινιών στο εύρος [0, συνολικός αριθμός χρηστών-1] και [0, συνολικός αριθμός ταινιών-1] αντίστοιχα, προκειμένου να γίνει ταχύτερη η εκπαίδευση του μοντέλου, το σκριπτ python δημιουργεί τα αντίστοιχα λεξικά έτσι ώστε τα εικονικά ids να μπορούν να αντιστοιχηθουν στα πραγματικά. Οι υπόλοιπες λειτουργίες της προεπεξεργασίας θα αναλυθούν αργότερα.

3.2 Ο Αλγόριθμος

Γράψαμε τη δική μας υλοποίηση του αλγορίθμου Alternating Least Squares σε γλώσσα C με βάση τις οδηγίες που δίνονται στο cite ALSpaper. Τα σύνολα δεδομένων που χρησιμοποιήσαμε για την αξιολόγηση είναι τα Movielens, τα οποία είναι πολύ συνηθισμένα στην έρευνα συστημάτων συστάσεων.

Παρουσιάζουμε τις μεταβλητές που εμφανίζονται συχνότερα μέσα στον κώδικα μας, για να κάνουμε την κατανόηση του κώδικα ευκολότερη στον αναγνώστη, οι μεταβλητές σε παρένθεση είναι οι αντίστοιχες τιμές της ανάλυσης που έχουμε κάνει στο Chapter 2.

Μεταβλητή	**Περιγραφή**
numFeatures	Ο αριθμός των χαρακτηριστικών με τα οποία θα αναπαραστήσουμε τις ταινίες και τους χρήστες (n_f).
numUsers	Ο αριθμός των χρηστών.(n_u)
numMovies	Ο αριθμός των ταινιών.(n_m)
numData	Το πλήθος των συνολικά γνωστών αξιολογήσεων, ισοδυναμεί με τον αριθμό των γραμμών που βρίσκονται στα σύνολα δεδομένων Movielens ($n = \|I\|$)
U,M	$U_{numUsers \times numFeatures}$ και $M_{numMovies \times numFeatures}$ είναι οι πίνακες που περιέχουν τις ελεύθερες μεταβλητές του μοντέλου, αυτές είναι οι μεταβλητές τις οποίες επιχειρεί να βελτιστοποιήσει ο αλγόριθμος.

Η *sparseEntry* είναι μια δομή που αναπαριστά μια γραμμή του συνόλου δεδομένου.Αρχικά ολόκληρο το σύνολο δεδομένων αναπαρίσταται ως ένας πίνακας από δομές sparseEntry μεγέθους *numData*.

```
struct sparse_entry{
        int    rowUser;  // userID
        int    colMovie; //movieID
        float  rating;   //rating
};
typedef struct sparse_entry sparseEntry;
```

Το *struct Info* είναι η δεύτερη και τελική αναπαράσταση του συνόλου δεδομένων. Δημιουργούμε δυο στιγμιότυπα πινάκων *info*, το ένα περιέχει το σύνολο της πληροφορίας για τους χρήστες ενώ το άλλο περιέχει το σύνολο της πληροφορίας για τις ταινίες.Το πεδίο *id* είναι το id του χρήστη η της ταινίας αντίστοιχα, *numRatings* είναι ο ακέραιος που αναπαριστά το πλήθος των ταινιών το οποίο ο συγκεκριμένος χρήστης έχει βαθμολογήσει, ενώ στη περίπτωση των ταινιών,απο πόσους χηστες έχει βαθμολογηθεί αυτή η ταινία. Στους δυο πίνακες *rating* και *rId* αποθηκεύουμε τις βαθμολογίες κάθε χρήστη και το *id* της ταινίας στην οποία απευθύνεται η βαθμολογία, ενώ στην περίπτωση των πληροφοριών των ταινιών αποτελούν τις βαθμολογίες για αυτήν την ταινία και το αντίστοιχο *id* του χρήστη που την πραγματοποίησε.

```
struct Info{
  // numRatings -> how many movies has this user rated/By how many users
    htis movie been rated
  // rating     -> array containing the ratings os this user/movie
  // rId        -> array containing the coresponding id of the movie/user
  // id         -> id of the user/movie
  // if for example rating[0] = 5 && rId[0] = 6 this means either the
    user rated movie 6 with 5 either
  // this movie has been rated with 5 by user 6
  int numRatings;
  float* rating;
  int* rId;
  int id;
};
typedef struct Info info;
```

3.2.1 Βασικές Συναρτήσεις

Η συνάρτηση **findNumOfEntries** μετρά το πλήθος των γραμμών του συνόλου δεδομένων. Υποθέτουμε ότι κάθε γραμμή αντιστοιχεί σε μια βαθμολογία, αυλτη η συνάρτηση λοιπόν στη πραγματικότητα τον συνολικό αριθμό γνωστών αξιολογήσεων στο σύνολο δεδομένων.

```
int findNumOfEntries(FILE* file){
  char    *line = NULL;
  size_t  len   = 0;
  ssize_t read;
  int     k     = 0;

  while(1){
    read = getline(&line,&len,file);
    if(read==-1) break;
    else if(line[0]=='#'||line[0]=='\0'||!strcmp(line,"")||!strcmp(line,"
     ")||line[0]=='\n'){
      continue;
    }
    k++;
  }

  free(line);
  return k;
}
```

Η συνάρτηση *getDataFromFile* διαβάζει τα δεδομένα και τα αποθηκεύει στον πίνακα *sparseEntry* array R.

```
void getDataFromFile(FILE *file,char* delimeter, sparseEntry* R){
        char    *line = NULL;
        size_t  len   = 0;
        ssize_t read;
        int     k     = 0;

        while(1){
                read = getline(&line,&len,file);
                if(read==-1) break;
                else if(line[0]=='#'||line[0]=='\0'||!strcmp(line,"")||!
   strcmp(line," ")||line[0]=='\n'){
                        continue;
                }
                R[k].rowUser = atoi(strtok(line,delimeter));
                R[k].colMovie = atoi(strtok(NULL,delimeter));
                R[k].rating = atof(strtok(NULL,delimeter));
                k++;
        }
        free(line);
}
```

Η συνάρτηση **updateU** είναι υπεύθυνη για την ανανέωση των διανυσμάτων γραμμών του U. Όπως αναφέραμε και στην θεωρία του ALS chapter 2 ο κανόνας ανανέωσης κάθε διανύσματος χρήστη u_i απαιτεί την επίλυση του συστήματος $(M_{I_i}M_{I_i}^T + \lambda n_{u_i}E)u_i = M_{I_i}R(i,I_i)$. Η ακόλουθη συνάρτηση υπολογίζει το γινόμενο του πίνακα $M_{I_i}M_{I_i}^T$ και του $M_{I_i}R(i,I_i)$ αντίστοιχα, και στην συνέχεια εκτελεί την κανονικοποίηση.

Τέλος καλεί την συνάρτηση που λύνει το σύστημα με χρήση της τεχνικής παραγοντοποίησης Cholesky και της τεχνικής backward και forward αντικατάστασης.

```
void updateU(float** U,float** M,info userInf[], int numUsers, float* A,
    float* V,float lamda,int numFeatures){
  int i,j,k,indx,u;
  double temp;


  for(u=0;u<numUsers;u++){            //for each user
    printf("\r");
    printf("updating  user  %d/%d ",u+1,numUsers);
    int*   sel      = userInf[u].rId;
    float* ratings  = userInf[u].rating;
    int    nratings = userInf[u].numRatings;
    int    id       = userInf[u].id;

    for(i=0;i<numFeatures;i++){    // calculate  A = M_{I_i} M_{I_i}^T
      for(j=0;j<numFeatures;j++){
        if(i<=j){
          temp = 0;
          for(k=0;k<nratings;k++){
            indx = sel[k];
            temp += M[indx][i]*M[indx][j];
          }
          A[i*numFeatures+j] = temp;
        }
      }
    }

    for(i=0;i<numFeatures;i++){ // calculate V = M_{I_i} R(i,I_i)
      V[i]=0;
      for(j=0;j<nratings;j++){
        V[i]+=M[sel[j]][i]*ratings[j];
      }
    }

    //Regularization
    for(j=0;j<numFeatures;j++)A[j*numFeatures+j]+=lamda*nratings;

    CholeskySolver(A,V,U[id],numFeatures); //Linear System Solver
  }
  printf("\n");
}
```

Λειτουργία **updateM** εκτελεί ακριβώς την ίδια λειτουργία με το **updateU** αλλά για την μήτρα M. Θα μπορούσαμε εύκολα να ενσωματώσουμε και τις δύο λειτουργίες σε μία, αλλά για λόγους σαφήνειας, αποφασίσαμε να τις χωρίσουμε.

```
void updateM(float** U,float** M,info movieInf[], int numMovies, float*
    A,float* V,float lamda,int numFeatures){
  int i,j,k,indx,m;
  double temp;

  for(m=0;m<numMovies;m++){
    printf("\r");
```

```
    printf("updating movie %d/%d",m+1,numMovies);
    int*   sel      = movieInf[m].rId;
    float* ratings  = movieInf[m].rating;
    int    nratings = movieInf[m].numRatings;
    int    id       = movieInf[m].id;
    int i,j,k,indx;

    for(i=0;i<numFeatures;i++){     //Calculate A = U_Ij U_Ij^T
      for(j=0;j<numFeatures;j++){
        if(i<=j){
          temp = 0;
          for(k=0;k<nratings;k++){
            indx = sel[k];
            temp += U[indx][i]*U[indx][j];
          }
          A[i*numFeatures+j] = temp;
        }
      }
    }

    for(i=0;i<numFeatures;i++){ //Calculate V_j = U_Ij U_Ij^T
      V[i]=0;
      for(j=0;j<nratings;j++){
        V[i]+=U[sel[j]][i]*ratings[j];
      }
    }

    //Regularize
    for(j=0;j<numFeatures;j++)A[j*numFeatures+j]+=lamda*nratings;

    CholeskySolver(A,V,M[id],numFeatures); //Linear System Solver
    printf("\r");
  }
  printf("\n");
}
```

Η συνάρτηση **CholeskySolver** παραγοντοποιεί τον θετικό συμμετρικό πίνακα A σε δύο συμμετρικές μήτρες έτσι ώστε $A = LL^T$, χρησιμοποιούμε την προς τα εμπρός και προς τα πίσω υποκατάσταση (backward-forward substitution) για να λύσει το σύστημα $AX = B$ ως εξής:

$$
\begin{aligned}
& AX = B && \xrightarrow{\text{cholesky}} \\
& LL^T X = B && \text{assuming that } L^T X = Y (1) \Rightarrow \\
& LY = B && \xrightarrow{\text{after solving Y}} \\
& (1) \Rightarrow L^T X = Y &&
\end{aligned}
$$

```
void CholeskySolver(float* A,float* B,float* X,int numFeatures){
        int i,j,k;
        float L[numFeatures*numFeatures];
        float Y[numFeatures];
```

```
        for (i = 0; i < numFeatures; i++)     // Cholesky Decomposition
                for (j = 0; j < (i+1); j++) {
                        float s = 0;
                        float value,temp;
                        for (k = 0; k < j; k++)
                                s += L[i*numFeatures+k] * L[j*numFeatures
    +k];
                        temp = i<=j?A[i*numFeatures+j]:A[j*numFeatures+i
    ];
                        value = (i == j)?sqrtf(A[i*numFeatures+i] - s)
    :(1.0 / L[j*numFeatures+j] * (temp - s));
                        L[i*numFeatures+j] = value;
                }

        for(i=0;i<numFeatures;i++){   //Forward Substitution
                Y[i] = B[i];
                for(j=0;j<i;j++){
                        Y[i] -= L[i*numFeatures+j]*Y[j];
                }
                Y[i]/=L[i*numFeatures+i];
        }

        for(i=numFeatures-1;i>=0;i--){   //Backward Substitution
                X[i] = Y[i];
                for(j=i+1;j<numFeatures;j++){
                        X[i]-=X[j]*L[j*numFeatures+i];
                }
                X[i]/=L[i*numFeatures+i];
        }
}
```

3.2.2 Συνάρτηση Main

Στην συνάρτηση main ταξινομούμε πρώτα τον πίνακα *SparseEntry* με βάση τα userIDs για να δημιουργήσουμε το *Info struct* για τους χρήστες που ονομάζεται *userInf*. Επίσης, βρίσκουμε τον αριθμό των μεμονωμένων χρηστών που περιέχονται στο σύνολο δεδομένων και το αποθηκεύουμε στο *numUsers*.

```
  qsort(R,nData,sizeof(sparseEntry),compareByUser);

  for(i=1;i<nData;i++){
    if(R[i].rowUser!=R[i-1].rowUser)
      numUsers++;
  }
```

Στην συνέχεια,κατασκευάζουμε τον πίνακα τύπου **Info** που ονομάζεται *userInf* και συμπληρώνουμε τα στοιχεία του με χρήση του πίνακα R ο οποίος είναι ταξινομημένος με βάση τα αναγνωριστικά των χρηστών. Η μεταβλητή c αναπαριστά τον αριθμό των αξιολογήσεων του συγκεκριμένου χρήστη. Για να πετύχουμε γρήγορη πρόσβαση στα δεδομένα έχουμε κάνει την παραδοχή *userInf [x] .id* $= x$ ώστε η πρόσβαση σε ένα συγκεκριμένο χρήστη να μπορεί να γίνει σε O(1). Η μόνη περίπτωση όπου αυτή η παραδοχή δεν είναι ικανοποιήσιμη είναι στην περίπτωση που ένα αναγνωριστικό χρήστη λείπει, αν για παράδειγμα τα αναγνωριστικά των χρηστών είναι τα [1,2,3,5,6], μπορούμε

να είμαστε βέβαιοι ότι αυτή η υπόθεση είναι ικανοποιήσιμη επειδή προ-επεξεργαζόμαστε το σύνολο δεδομένων μας με ένα script python που μετατοπίζει τα IDs έτσι ώστε κανένα από αυτά να μην λείπει. Στο συγκεκριμένο παράδειγμα τα αναγνωριστικά των χρηστών θα γίνονταν [1,2,3,4 (ex 5), 5 (ex 6)].

```
info* userInf  = (info*)malloc(numUsers*sizeof(info));

  int c = 0;
  int id = 0;
  for(i=0;i<=nData;i++){

    if(i!=nData && R[i].rowUser==id){
      c++;
    }
    else{
      userInf[id].id = id;
      userInf[id].numRatings = c;
      userInf[id].rating     = (float*)malloc(c*sizeof(float));
      userInf[id].rId        = (int*)malloc(c*sizeof(int));

      for(j=0;j<c;j++){
        userInf[id].rId[j]      = R[i-c+j].colMovie;
        userInf[id].rating[j]   = R[i-c+j].rating;
      }
      id++;
      c=1;
    }
  }
```

Στη συνέχεια, πραγματοποιούμε την ίδια μορφοποίηση για τις ταινίες. Για ακόμα μια φορά ταξινομούμε τον πίνακα *SparseEntry* με βάση τα iDs των ταινιών ώστε να δημιουργήσουμε τον πίνακα από *Info struct* για τις ταινίες, που ονομάζεται *movieInf*. Επίσης, βρίσκουμε τον αριθμό των ξεχωριστών ταινιών που περιέχονται στο σύνολο δεδομένων και το αποθηκεύουμε στην *numMovies*.

```
  qsort(R,nData,sizeof(sparseEntry),compareByMovie);

  for(i=1;i<nData;i++){
    if(R[i].colMovie!=R[i-1].colMovie)
      numMovies++;
  }
```

Για ακόμα μια φορά κατασκευάζουμε τον πίνακα τύπου **Info** που ονομάζεται *movieInf* και γεμίζουμε τα στοιχεία του χρησιμοποιώντας τον πίνακα R το οποίο τώρα ταξινομούμε με βάση τα movie-ids. Στην μεταβλητή c αποθηκεύουμε το πλήθος των αξιολογήσεων που έχουν αποδοθεί στην συγκεκριμένη ταινία. Για γρήγορη πρόσβαση έχουμε κάνει την παραδοχή $movieInf[x].id = x$ ωστε να μπορούμε να έχουμε πρόσβαση σε μια συγκεκριμένη ταινία σε O(1). Ο μόνος τρόπος που αυτή η παραδοχή να μην ικανοποιήται είναι στην περίπτωση που ένα αναγνωριστικο ταινίας (id) είναι απών από το σύνολο δεδομένων, ωστόσο όπως αναφέρθηκε προηγουμένως έχουμε επεξεργαστεί τα σύνολα δεδομένων μας με ένα Python που μετατοπίζει τα αναγνωριστικά με τέτοιο τρόπο ώστε να μην λείπει κανένα από αυτά.

```
info* movieInf = (info*)malloc(numMovies*sizeof(info));
```

```
  c=0;
  id = 0;

  for(i=0;i<=nData;i++){

    if(i!=nData && R[i].colMovie==id){
      c++;
    }
    else{
      movieInf[id].id = id;
      movieInf[id].numRatings = c;
      movieInf[id].rating     = (float*)malloc(c*sizeof(float));
      movieInf[id].rId        = (int*)malloc(c*sizeof(int));

      for(j=0;j<c;j++){
        movieInf[id].rId[j]      = R[i-c+j].rowUser;
        movieInf[id].rating[j]   = R[i-c+j].rating;
      }
      id++;
      c=1;
    }
  }
```

Τέλος, το τμήμα εκπαίδευσης του αλγορίθμου. Σε κάθε επανάληψη ενημερώνουμε εναλλακτικά το M μετά το U. Σε κάθε επανάληψη υπολογίζουμε το RMSE στο σετ εκπαίδευσης.

```
  float* A       = (float*)malloc(numFeatures*numFeatures*sizeof(float));
  float* V       = (float*)malloc(numFeatures*sizeof(float));

  calculateRMSE(nData,userInf,numUsers,M,U,numFeatures);

  for(i=0;i<iterations;i++){
    printf("#-#-#-#- ITERATION %d -#-#-#-#\n",i);
    updateU(U,M,userInf,numUsers,A,V,lamda,numFeatures);
    updateM(U,M,movieInf,numMovies,A,V,lamda,numFeatures);
    calculateRMSE(nData,userInf,numUsers,M,U,numFeatures);
  }
exportRes(nData,userInf,numUsers,M,U,numFeatures);
```

3.3 Απεικόνιση χρονικού προφιλ

Αφού δημιουργήσαμε τη δική μας υλοποίηση του αλγορίθμου συστάσεων ALS, σε γλώσσα C, όπως περιγράφεται στο **Κεφάλαιο 2** και αναφέρθηκε στον hyperref [ο Αλγόριθμος] Κεφάλαιο 3 επιχειρήσαμε να καταγράψουμε το χρονικό προφίλ του αλγορίθμου μας. Η εύρεση του χρονικού προφιλ του αλγορίθμου είναι ζωτικής σημασίας κατά την εκτέλεση της ανάλυσης πριν από την επιτάχυνση, προκειμένου να αναγνωριστούν τα πλέον απαιτητικά τμήματα σε υπολογιστική ένταση. Τα τμήματα αυτά αξιολογούνται ως υποψήφια προς επιτάχυνση. Χρησιμοποιήσαμε το εργαλείο Gprof για τη δημιουργία του χρονικού προφίλ.Στους παρακάτω πίνακες παρουσιάζονται τα αποτελέσματα της διαδικασίας..

Dataset	numFeatures	iterations
Movielens1m (1 million Ratings)	80	10

time %	cumulative seconds	self seconds	calls	self s/call	total s/call	function name
49.80	203.92	203.92	10	20.39	21.49	updateM
42.55	378.18	174.25	10	17.43	19.21	updateU
7.02	406.92	28.74	97460	0.00	0.00	CholeskySolver
0.66	409.63	2.71	11	0.25	0.25	calculateRMSE
0.02	409.70	0.07	1	0.07	0.07	exportR
0.01	409.75	0.05				compareByUser
0.01	409.79	0.04				compareByMovie
0.00	409.81	0.02				main
0.00	409.81	0.00	1	0.00	0.00	checkDataset
0.00	409.81	0.00	1	0.00	0.00	fileExists
0.00	409.81	0.00	1	0.00	0.00	findNumOfEntries
0.00	409.81	0.00	1	0.00	0.00	getDataFromFile
0.00	409.81	0.00	1	0.00	0.00	initM
0.00	409.81	0.00	1	0.00	0.00	initU

function index	% time	self	children	times called	function name	
[1]	100.0	0.02	409.70		main	[1]
		203.92	10.93	10/10	updateM	[2]
		174.25	17.81	10/10	updateU	[3]
		2.71	0.00	11/11	calculateRMSE	[5]
		0.07	0.00	1/1	exportR	[6]
		0.00	0.00	1/1	fileExists	[10]
		0.00	0.00	1/1	findNumOfEntries	[11]
		0.00	0.00	1/1	getDataFromFile	[12]
		0.00	0.00	1/1	checkDataset	[9]
		0.00	0.00	1/1	initU	[14]
		0.00	0.00	1/1	initM	[13]
		203.92	10.93	10/10	main	[1]
[2]	52.4	203.92	10.93	10	updateM	[2]
		10.93	0.00	37060/97460	CholeskySolver	[4]
		174.25	17.81	10/10	main	[1]
[3]	46.9	174.25	17.81	10	updateU	[3]
		17.81	0.00	60400/97460	CholeskySolver	[4]
		10.93	0.00	37060/97460	updateM	[2]
		17.81	0.00	60400/97460	updateU	[3]
[4]	7.0	28.74	0.00	97460	CholeskySolver	[4]
		2.71	0.00	11/11	main	[1]
[5]	0.7	2.71	0.00	11	calculateRMSE	[5]
		0.07	0.00	1/1	main	[1]
[6]	0.0	0.07	0.00	1	exportR	[6]
[7]	0.0	0.05	0.00		compareByUser	[7]
[8]	0.0	0.04	0.00		compareByMovie	[8]
		0.00	0.00	1/1	main	[1]
[9]	0.0	0.00	0.00	1	checkDataset	[9]
		0.00	0.00	1/1	main	[1]
[10]	0.0	0.00	0.00	1	fileExists	[10]
		0.00	0.00	1/1	main	[1]
[11]	0.0	0.00	0.00	1	findNumOfEntries	[11]
		0.00	0.00	1/1	main	[1]
[12]	0.0	0.00	0.00	1	getDataFromFile	[12]
		0.00	0.00	1/1	main	[1]
[13]	0.0	0.00	0.00	1	initM	[13]
		0.00	0.00	1/1	main	[1]
[14]	0.0	0.00	0.00	1	initU	[14]

Από τους πίνακες απεικόνισης του προφίλ μπορούμε να διαπιστώσουμε ότι οι συναρτήσεις *updateU* (42.55 %) και *updateM* (49.80 %) καταναλώνουν περίπου το **92.35 %** του συνολικού χρόνου εκτέλεσης, το υπόλοιπο 7.02% του χρόνου εκτέλεσης αναλώ-

νεται στην συνάρτηση CholeskySolver και το υπόλοιπο 0,63% του χρόνου εκτέλεσης καταναλώνεται κατά την ανάγνωση των δεδομένων και τον υπολογισμό του μέσου τετραγωνικού σφάλματος. Αυτά τα αποτελέσματα προέκυψαν από μια εκτέλεση στην οποία υπολογίζαμε τον μέσο τετραγωνικό σφάλμα σε κάθε επανάληψη, κάτι που είναι περιττό και το κάναμε μόνο για να παρατηρήσουμε την μείωση του RMSE σε κάθε επανάληψη. Ωστόσο, μπορούμε να δούμε ότι αυτό δεν μας εμπόδισε να βγάλουμε κάποια προφανή αποτελέσματα. The above analysis indicates that the most likely candidates for acceleration are the operations performed by *updateU* and *updateM* which are (as mentioned before) the following:

updateU	$A_j = U_{I_j} U_{I_j}^T + \lambda n_{m_j} E$	$V_j = U_{I_j} R(I_j, j)$
updateM	$A_i = M_{I_i} M_{I_i}^T + \lambda n_{u_i} E$	$V_i = M_{I_i} R(i, I_i)$

4

Υλοποίηση Επιταχυντή

4.1 Εισαγωγή

Οι παρακάτω υλοποιήσεις επιταχυντή έγιναν με την χρήση του εργαλείου της Xilinx, SDSoC , το οποίο καλεί για την δημιουργία του RTL και την εξαγωγή του bitstream το Vivado HLS. Η πορεία σχεδιασμού που ακολουθήσαμε είναι αυτή που αναφέραμε στο πρώτο κεφάλαιο. Συγκεκριμένα μετά από κάθε υλοποίηση κάναμε ανάλυση των αναφορών του εργαλείου σχετικά με τις μετρικές ταχύτητας και κατανάλωσης χώρου και διαστήματος επανενεργοποίησης με στόχο να τις χρησιμοποιήσουμε ως ανατροφοδότηση για να βελτιστοποιήσουμε τις αντίστοιχες μετρικές στον πυρήνα. Για την βελτιστοποίηση των παραπάνω μετρικών, χρησιμοποιήσαμε τις *pragma* οδηγίες υψηλού επιπέδου που εφαρμόζονται σε *συναρτήσεις*, *βρόγχους* και *Πίνακες*.Στον παρακάτω πίνακα παρουσιάζουμε τις πιο κοινές οδηγίες που χρησιμοποιούνται στο στάδιο της βελτιστοποίησης.

Οδηγία	Περιγραφή
PIPELINE	Επιχειρέι να μειώσει το διάστημα έναρξης του σώματος ενός βρόχου,προκειμένου να είναι δυνατή η λήψη νέων εισόδων πριν ολοκληρωθεί η επεξεργασία των προηγούμενων.
DATAFLOW	Επιχειρέι την επίτευξη διοχέτευσης σε επίπεδο εργασιών.Αυτή η οδηγία δεν χρησιμοποιείται για την διοχέτευση της εκτέλεσης εντολών ,αλλά στην διοχέτευση ολόκληρων υπολογιστικών εργασιών,όπου μια εργασία μπορεί να είναι είτε μια συνάρτηση είτε ένας βρόχος.
INLINE	Αντικαθιστά μια κλήση συνάρτησης με το σώμα της συνάρτησηε, ιδιαίτερα χρήσιμη για την αποφύγεί του χρόνου κλήσης μικρών συναρτήσεων..
UNROLL	Αναδιπλώνει το σώμα ενός βρόχου με στόχο την ταυτόχρονη εκτελέση περισσότερων από ένα σωμάτων βρόχου μέσα σε ένα παράθυρο επανάληψης.Για αυτή την οδηγία, είναι σημαντικό να αποφεύγεται η εξάρτηση δεδομένων μεταξύ διαδοχικών βρόχων.
ARRAY_PARTITION	Επιτρέπει στον προγραμματιστή να διαιρέσει τα δεδομένα σε διαφορετικά Block-Rams, έτσι ώστε ο επιταχυντής να μπορεί να διαβάσει πολλαπλά δεδομένα ταυτόχρονα και να μην περιρίζεται από τις πόρτες ενός bram. Οι συστοιχίες που αποθηκεύονται σε ένα Block-Ram δημιουργούν bottlenecks μνήμης, καθώς το Block-Ram έχει το πολύ δύο θύρες..
ARRAY_MAP	Συνδυάζει πολλαπλούς πίνακες σε ένα ενιαίο,μεγαλύτερο, προκειμένου να μειωθεί η χρήση των Block Ram.
ARRAY_RESHAPE	Συγχωνεύει κελιά πινάκων σε επίπεδο bit, αυξάνοντας το μήκος της λέξης της μνήμης. Ιδιαίτερα χρήσιμο για την μεταφορά ταυτόχρονα πολλαπλών δεδομένων, χωρίς τη χρήση πολλαπλών Block-Rams.

Στο προηγούμενο κεφάλαιο καταλήξαμε στο συμπέρασμα ότι πρέπει να επιταχύνουμε τέσσερις πολλαπλασιασμούς πινάκων. Ωστόσο, δεν χρειαζόμαστε πραγματικά τέσσερις διαφορετικούς πυρήνες υλικού, διότι αυτές οι τέσσερις πολλαπλασιασμοί μπορούν να ομαδοποιηθούν, έτσι ώστε να χρειαστεί μόνο να δημιουργήσουμε δύο επιταχυντές πολλαπλασιασμού που θα χρησιμοποιηθούν δύο φορές. Η πρώτη ομάδα πολλαπλασιασμών πινάκων που μπορεί να υπολογιστεί από τον ίδιο επιταχυντή είναι η ακόλουθη

$$A_i = M_{I_i} M_{I_i}^T$$
$$A_j = U_{I_j} U_{I_j}^T$$

Για αυτή την ομάδα εξισώσεων χρειαζόμαστε ένα επιταχυντή που μπορεί να εκτελεί γρήγορο και ενεργειακά αποδοτικό πολλαπλασιασμό πινάκων με διαστάσεις $(n_f \; timesn_{u_i}/n_{m_j}) \times (n_{u_i}/n_{m_j} \; timesn_f)$. Μπορούμε ήδη να διαπιστώσουμε ότι το πρώτο πρόβλημα που προκύπτει είναι ότι το (n_{u_i} και το n_{m_j}, ο αριθμός των ταινιών που έχει βαθμολογήσει ο

χρήστης i και ο αριθμός των αξιολογήσεων που έχουν δωθεί για την ταινία j αντίστοιχα είναι άγνωστοι κατά τον χρόνο της μεταγλώττισης και μπορούν να υπολογιστούν μόνο κατά τον χρόνο εκτέλεσης.

Η δεύτερη ομάδα πολλαπλασιασμών πινάκων που μπορούν να υπολογιστούν στον ίδιο επιταχυντή:

$$V_i = M_{I_i} R(i, I_i)$$
$$V_j = U_{I_j} R(I_j, j)$$

Για αυτή την ομάδα εξισώσεων χρειαζόμαστε έναν επιταχυντή που μπορεί να εκτελεί γρήγορο και ενεργειακά αποδοτικό πολλαπλασιασμό μεταξύ δύο πινάκων με διαστάσεις $(n_f\ timesn_{u_i}/n_{m_j})$ $(n_f\ n_{u_i}/n_{m_j})$. Για άλλη μια φορά το (n_{u_i} και το n_{m_j} είναι άγνωστα κατά το χρόνο μεταγλώττισης και μπορούν να υπολογιστούν κατά τον χρόνο εκτέλεσης.

4.2 Προκλήσεις

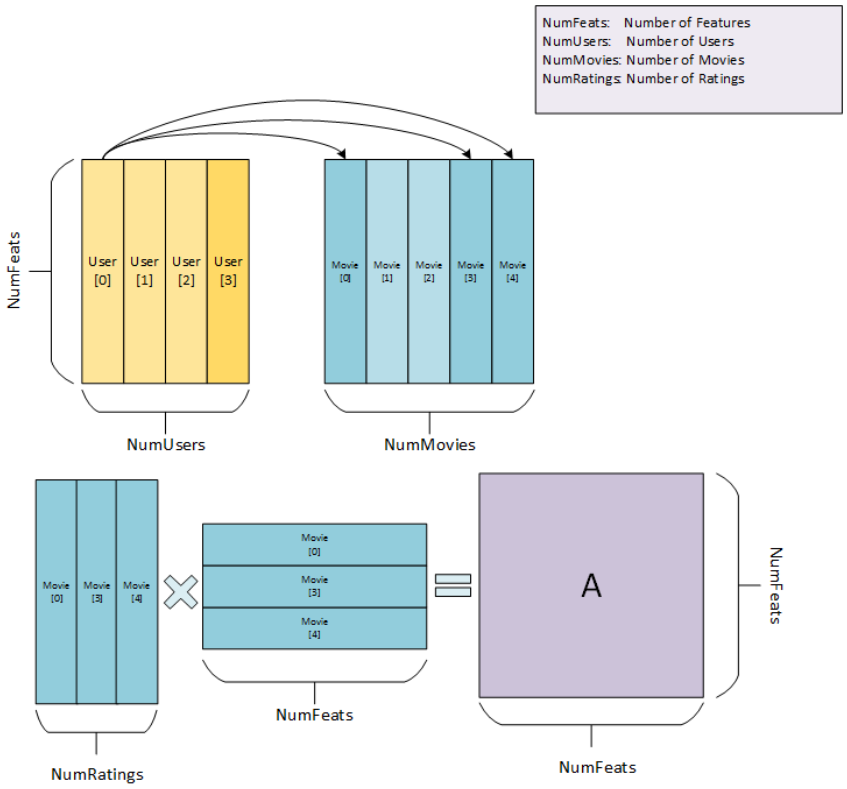
Υπάρχουν τρεις κύριες προκλήσεις που πρέπει να αντιμετωπίσουμε, σε αυτή την περίπτωση, προκειμένου να δημιουργήσουμε γρήγορους επιταχυντές.

- Οι πίνακες μπορεί να είναι απεριόριστα μεγάλοι.
- Το μέγεθός τους είναι άγνωστο την στιγμή της μεταγλώττισης.
- Οι πίνακες είναι διασκορπισμένοι μέσα στην μνήμη DDR.

Η πρώτη πρόκληση στην πραγματικότητα σημαίνει ότι ο επιταχυντής πρέπει να είναι σε θέση να πραγματοποιήσει πολλαπλασιασμό μεταξύ μεγάλων πινάκων επειδή μια από τις διαστάσεις των πινάκων, η οποία ισούται με τον πλήθος των βαθμολογήσεων, μπορεί να είναι τεράστια σε περίπτωση που για παράδειγμα μια ταινία είναι δημοφιλής και έχει βαθμολογηθεί από χιλιάδες χρήστες.
Η δεύτερη πρόκληση είναι ότι παρόλο που μπορούμε να κρατήσουμε σταθερό τον αριθμό των χαρακτηριστικών (n_f) που είναι η μία διάσταση των πινάκων δεν μπορούμε να κρατήσουμε σταθερή τη δεύτερη διάσταση επειδή είναι άγνωστη κατά τη διάρκεια της μεταγλώττισης και εκτός αυτού αλλάζει συνεχώς κατά τον χρόνο εκτέλεσης για κάθε διαφορετική ταινία και κάθε διαφορετικό χρήστη.
Τέλος, όπως φαίνεται στην παρακάτω εικόνα, οι γραμμές των πινάκων δεν είναι συνεχόμενες στη μνήμη, γεγονός που αφήνει ανοιχτές πολλές αποφάσεις για εξερεύνηση σχετικά με την διεπαφή του επιταχυντής με τον επεξεργαστή, τη μνήμη DDR και την κρυφή μνήμη.

4.3 Βασικές Παραδοχές Σχετικά με τον Πολλαπλασιασμό πινάκων

Ακολουθούν οι γενικές παραδοχές που ακολουθήσαμε κατά την αναζήτηση ενός πυρήνα που ταιριάζει καλύτερα στις ανάγκες μας.
Γνωρίζουμε ότι τα δεδομένα μας δίδονται σε αραιή μορφή, όμως δεν είμαστε υποχρεωμένοι από τον αλγόριθμο να εφαρμόσουμε αραιούς πολλαπλασιασμούς πινάκων, επιπλέον



Σχήμα 4.1: Απεικόνιση δημιουργίας πίνακα A για τον user[0].Δεν μπορούμε να γνωρίζουμε πόσες ταινίες (NumRatings = 3 σε αυτήν την περίπτωση) και ποιες ταινίες έχει βαθμολογήσει ένας χρήστης (οι ταινίες 0,3 και 4 εδώ) πριν τον χρόνο εκτέλεσης.

διαπιστώσαμε ότι τα εργαλεία σύνθεσης υψηλού επιπέδου μπορεί να μην λειτουργούν πολύ καλά όσο η πυκνότητα της μήτρας μειώνεται [25], ως αποτέλεσμα χρησιμοποιήσαμε μετασχηματισμούς από πηγαίο σε πηγαίο κώδικα για να δουλέψουμε μόνο με πυκνούς πίνακες.
Λαμβάνοντας υπόψη ότι χρησιμοποιήσαμε λογισμικό και υλικό της Xilinx για την εκτέλεση του High Level Synthesis, η βασική ιδέα για τον επιταχυντή μας προέρχεται από την τεκμηρίωση της Xilinx [26] σχετικά με τα εργαλεία HLS και το ολοκληρωμένο Zynq. Αυτός είναι ένας βασικός αλγόριθμος μήτρας-πολλαπλασιασμού.

```
1  void mm_parallel_dot_product(int in_a[A_ROWS][A_COLS],
2                  int in_b[A_COLS][B_COLS],
3                    int out_c[A_ROWS][B_COLS])
4  {
5    int sum_mult;
6    a_row_loop: for (int i = 0; i < A_ROWS; i++){
7      b_col_loop: for (int j = 0; j < B_COLS; j++){
8        sum_mult = 0;
9        a_col_loop: for (int k = 0; k < A_COLS; k++){
10         sum_mult += in_a[i][k] * in_b[k][j];
```

```
11        }
12        out_c[i][j] = sum_mult;
13      }
14    }
15 }
```

Ουσιαστικά οι λειτουργίες μέσα στο *a_ col_ loop* που πρέπει να εκτελεστούν πάνω στο υλικό είναι οι εξής:

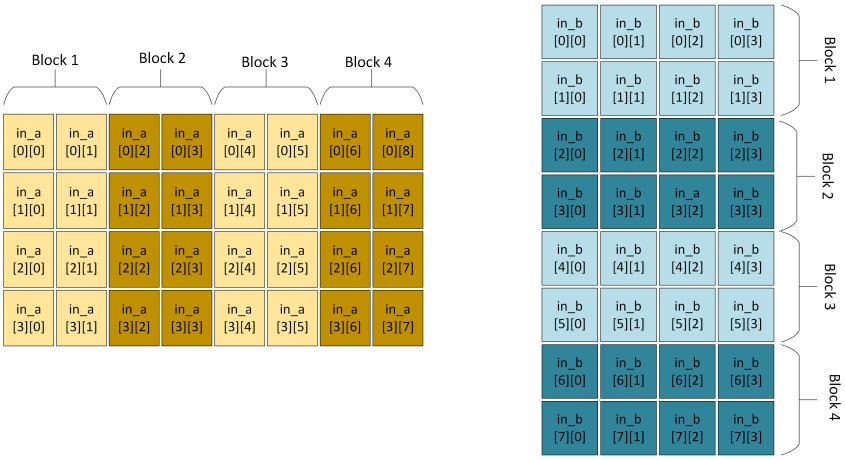
```
1 read in_a[0][0]
2 read in_b[0][0]
3 mul  temp,in_a[0][0],in_b[0][0]
4 add  sum_mult,sum_mult,temp
5
6 read in_a[0][1]
7 read in_b[1][0]
8 mul  temp,in_a[i][0],in_b[0][j]
9 add  sum_mult,sum_mult,temp
10
11 .
12 .
13 .
14
15 read in_a[0[A_COLS-1]
16 read in_b[A_COLS-1][0]
17 mul  temp,in_a[i][0],in_b[0][j]
18 add  sum_mult,sum_mult,temp
```

Ενώ μια CPU θα εκτελούσε κάθε μία από τις παραπάνω λειτουργίες διαδοχικά, τα εργαλεία σύνθεσης υψηλού επιπέδου της Xilinx θα επιχειρήσουν να τις χρονοδρομολογήσουν όσο συντομότερα και περισσότερο παράλληλα επιτρέπεται από τους *λογικούς* και *φυσικούς-υλικούς* περιορισμούς. 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 επιτύχει παράλληλη εκτέλεση. Ο σκοπός μας είναι να ρυθμίσουμε σωστά το FPGA χρησιμοποιώντας εντολές υψηλού επιπέδου 'προκειμένου να ξεπεραστούν οι φυσικοί περιορισμοί.
Ως πρώτο βήμα μπορούμε να παρατηρήσουμε ότι οι εντολές *read* είναι απολύτως λογικά ανεξάρτητες, επομένως δεν υπάρχει λόγος κάθε εντολή *read* να περιμένουν οποιαδήποτε προηγούμενη *read* ή αριθμητική λειτουργία για να ολοκληρωθεί. Στην πραγματικότητα όλες οι ανάγνωσεις μπορούν να εκτελεστούν ταυτόχρονα στην αρχή του *a_ col_ loop*. Συνεπώς, για να είναι δυνατή η ανάγνωση παράλληλα όλων των απαιτούμενων δεδομένων σε κάθε εκτέλεση βρόχου, πρέπει να εξαναγκάσουμε την αποθήκευση των στοιχείων *in_ a* και *in_ b* του πίνακα σε μια αποτελεσματική μορφή. Εάν αποθηκεύαμε *in_ a* και *in_ b* στην ίδια bram τότε θα μπορούσαμε να φέρουμε μόνο δύο στοιχεία

κάθε φορά επειδή τα brams του zynq έχουν διπλή θύρα. Αυτό θα είχε ώς αποτέλεσμα ενώ μεταφέρονται τα *in_ a[i][k]*, *in_ b[k][j]* να βρίσκονται σε αναμονή για μεταφορά τα *in_ a[i][k-1]*, *in_ b [k-1][j]* πράγμα που θα πάγωνε την εκτέλεση.

Ωστόσο, αυτή η συμφόρηση μνήμης μπορεί να ξεπεραστεί, αν παρατηρήσουμε τον τρόπο με τον οποίο ο αλγόριθμος διασχίζει τα δεδομένα. Εύκολα συμπεραίνουμε ότι μπορούμε να διαχωρίσουμε (αποθηκεύσουμε σε διαφορετικά brams) τα στοιχεία του *in_ a* και του *in_ b* που ανήκουν στον άξονα στον οποίο τα διατρέχει ο αλγόριθμος, στην συγκεκριμένη περίπτωση την 2η και 1η -διαστάση αντιστοίχως. Υποθέτοντας $A_ROWS = 4$ και $A_COLS = 8$ τότε οι *in_ a* και *in_ b* θα τοποθετηθούν σε brams όπως φαίνεται στην εικόνα.

Επειδή τα Brams του Zynq είναι διπλής θύρας, που σημαίνει ότι μπορούμε να έχουμε



Σχήμα 4.2: Αυτός είναι ο τρόπος με τον οποίο οι πίνακες προς πολλαπλασιασμό αποθηκεύονται σε Bram μέσα στην προγραμματιστική λογική. Τα Brams του Zynq είναι διπλής θύρας, και συνεπώς μπορούμε να έχουμε έναν συντελεστή διαχωρισμού "2" αντί για συντελεστή "1"

ταυτόχρονα πρόσβαση σε δύο στοιχεία, αντί να χρησιμοποιήσουμε τον *complete* διαχωρισμό σε κάθε διάσταση, δηλαδή να αποθηκεύσουμε όλα τα στοιχεία των στηλών ή των σειρών σε ξεχωριστούς καταχωρητές ή brams , μπορούμε να χρησιμοποιήσουμε την διαμέριση τύπου *block*. Η διαμέριση τύπου *block* δημιουργεί μικρότερες συστοιχίες από διαδοχικά μπλοκ της αρχικής συστοιχίας.Αυτό διαχωρίζει αποτελεσματικά τη συστοιχία σε N ίσα τμήματα, όπου N είναι ο ακέραιος που ορίζεται στην επιλογή *factor*. Εδώ *factor* = *A_ COLS / 2*. Αν για παράδειγμα _a[4][8] και _b[8][4], που σημαίνει ότι $A\_COLS = 8$, οι ακόλουθες εντολές υψηλού επιπέδου θα παράγουν το αποτέλεσμα που είναι ορατό στο σχήμα.

- #pragma HLS PARTITION block factor=A_COLS/2 VARIABLE=in_a DIM=2
- #pragma HLS PARTITION block factor=A_COLS/2 VARIABLE=in_b DIM=1

```
1 void mm_parallel_dot_product(int in_a[A_ROWS][A_COLS],
```

```
                int in_b[A_COLS][B_COLS],
                  int out_c[A_ROWS][B_COLS])
{
#pragma HLS PARTITION block factor=A_COLS/2 VARIABLE=in_a
    DIM=2
#pragma HLS PARTITION block factor=A_COLS/2 VARIABLE=in_b
    DIM=1

  int sum_mult;
  a_row_loop: for (int i = 0; i < A_ROWS; i++){
    b_col_loop: for (int j = 0; j < B_COLS; j++){
      sum_mult = 0;
      a_col_loop: for (int k = 0; k < A_COLS; k++){
        sum_mult += in_a[i][k] * in_b[k][j];
      }
      out_c[i][j] = sum_mult;
    }
  }
}
```

Ακολουθούν οι πολλαπλασιασμοί. Οι πολλαπλασιασμοί είναι λογικά περιορισμένοι μόνο από τις λειτουργίες *read*, αλλά αφού έχουμε καταφέρει να εκτελέσουμε όλες τις λειτουργίες ανάγνωσης στην αρχή του βρόχου μπορούμε τώρα να εκτελέσουμε όλους τους πολλαπλασιασμούς ταυτόχρονα. Μπορούμε να το επιτύχουμε είτε με την πλήρη αναδίπλωση του *a_ col_ loop* είτε μέσω pipetining του *b_ col_ loop*. Η δεύτερη επιλογή αναγκάζει την πλήρη αναδίπλωση κάθε εσωτερικού βρόχου, και συνεπώς είναι μια καλή επιλογή. Αυτή η λειτουργία μπορεί να εκτελεστεί από την εντολή υψηλού επιπέδου:

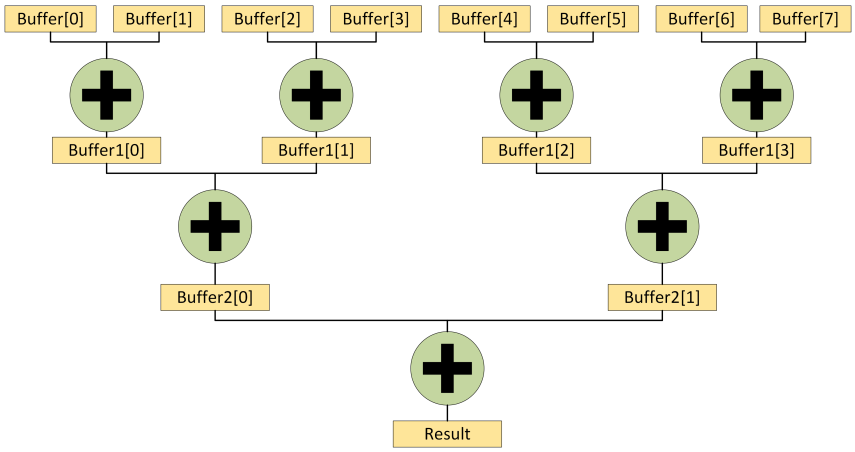
- #pragma HLS PIPELINE II=1

που βρίσκεται στην αρχή του *b_ col_ loop*. Τώρα, όλοι οι πολλαπλασιασμοί (εφ 'όσον έχουμε αρκετά DSP48 για να τις εκτελέσουμε) μπορούν να πραγματοποιηθούν ταυτόχρονα ακριβώς μετά την ολοκλήρωση των αναγνώσεων.

```
void mm_parallel_dot_product(int in_a[A_ROWS][A_COLS],
                int in_b[A_COLS][B_COLS],
                  int out_c[A_ROWS][B_COLS])
{
#pragma HLS PARTITION block factor=A_COLS/2 VARIABLE=in_a
    DIM=2
#pragma HLS PARTITION block factor=A_COLS/2 VARIABLE=in_b
    DIM=1

  int sum_mult;
  a_row_loop: for (int i = 0; i < A_ROWS; i++){
    b_col_loop: for (int j = 0; j < B_COLS; j++){
    #pragma HLS PIPELINE II=1
```

```
12        sum_mult = 0;
13        a_col_loop: for (int k = 0; k < A_COLS; k++){
14          sum_mult += in_a[i][k] * in_b[k][j];
15        }
16        out_c[i][j] = sum_mult;
17      }
18    }
19  }
```

Ο τελευταίος περιορισμός εμφανίζεται στο βήμα της συσσώρευσης των αποτελεσμάτων των πολλαπλασιασμών. 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 σε σειρά με μια πιο πολύπλοκη δομή αθροιστών η οποία μπορεί να εκτελέσει τους πολλαπλασιασμούς με χρονική πολυπλοκότητα $O(log(n))$. Αντί να συσσωρεύουμε σειριακά κάθε αποτέλεσμα των πολλαπλασιαστών 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.



Σχήμα 4.3: Λογαριθμικός αθροιστής. 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 δυνατό περισσότερη παραλληλοποίηση των υπολογισμών.

Στον παρακάτω κώδικα επιδεικνύουμε πώς είναι εφικτό να δημιουργήσουμε έναν λογαριθμικό αθροιστή με χρήση "High Level Synthesis" Πρωτίστος υπολογίζουμε

πόσα επίπεδα θα έχει ο λογαριθμικός αθροιστής. Συγκεκριμένα στο παράδειγμα μας $A\_COLS = 8$ συνεπώς χρειαζόμαστε $\log(8) = 3$ επίπεδα. Στην συνέχεια, για να έχουμε την δυνατότητα να γράψουμε σε κάθε επίπεδο καταχωρητών τα αποτελέσματα παράλληλα θα πρέπει αυτοί να μην βρίσκονται στο ίδιο Bram, αφού σε αυτήν την περίπτωση θα μπορούσαμε να πραγματοποιήσουμε μόνο δύο εγγραφές ταυτόχρονα, συνεπώς όπως δείξαμε και στις προηγούμενες περιπτώσεις θα πρέπει να διαμερίσουμε, πλήρως στην συγκεκριμένη περίπτωση, τους πίνακες που περιέχουν τους καταχωρητές. Οι αντίστοιχες HLS εντολές για να γίνει αυτό, είναι οι ακόλουθες:

- #pragma HLS array_partition variable=Buffer complete
- #pragma HLS array_partition variable=Buffer1 complete
- #pragma HLS array_partition variable=Buffer2 complete

Τώρα που μπορούμε να εκτελέσουμε παράλληλες λειτουργίες ανάγνωσης και εγγραφής σε κάθε στοιχείο σε όλα τα επίπεδα, πρέπει μόνο να χρονοδρομολογήσουμε τις παράλληλες προσθέσεις αναδιπλώνοντας κάθε βρόχο συσσώρευσης με την οδηγία HLS:

- #pragma HLS UNROLL

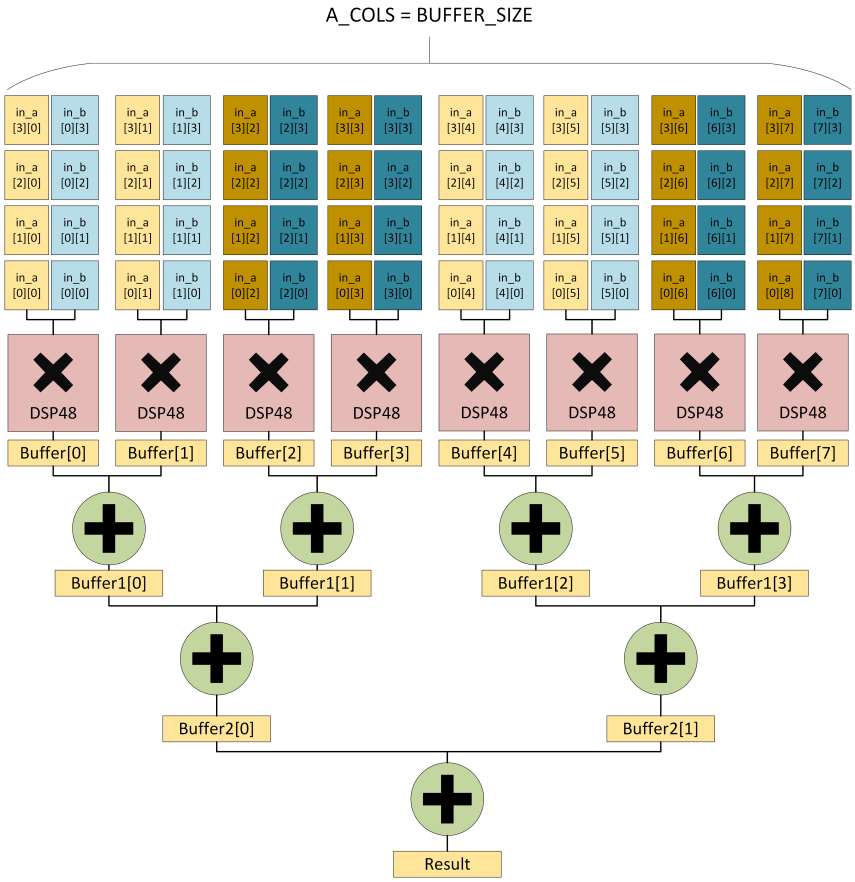
```
1  float log_add(float Buffer[A_COLS]){
2  #pragma HLS INLINE
3  int r;
4  float Buffer1[A_COLS/2];
5  float Buffer2[A_COLS/4];
6  float result;
7  int a;
8  #pragma HLS array_partition variable=Buffer1 complete
9  #pragma HLS array_partition variable=Buffer2 complete
10
11   //level2
12   a = 0;
13   for(r=0;r<A_COLS/2;r++){
14   #pragma HLS UNROLL
15       Buffer1[r] = Buffer[a]+Buffer[a+1];
16         a+=2;
17   }
18
19   //level3
20   a = 0;
21   for(r=0;r<A_COLS/4;r++){
22   #pragma HLS UNROLL
23         Buffer2[r] = Buffer1[a]+Buffer1[a+1];
24           a+=2;
25   }
```

```
26
27   float result = Buffer2[0] + Buffer2[1];
28   return x;
29 }
30
31 void mm_parallel_dot_product(int in_a[A_ROWS][A_COLS],
32                  int in_b[A_COLS][B_COLS],
33                    int out_c[A_ROWS][B_COLS])
34 {
35   int sum_mult;
36   float Buffer[A_COLS];
37 #pragma HLS PARTITION VARIABLE = Buffer complete
38 #pragma HLS PARTITION block factor=A_COLS/2 VARIABLE=in_a
     DIM=2
39 #pragma HLS PARTITION block factor=A_COLS/2 VARIABLE=in_b
     DIM=1
40
41
42   a_row_loop: for (int i = 0; i < A_ROWS; i++){
43     b_col_loop: for (int j = 0; j < B_COLS; j++){
44     #pragma HLS PIPELINE II=1
45       a_col_loop: for (int k = 0; k < A_COLS; k++){
46          //level1
47          Buffer[k] = in_a[i][k] * in_b[k][j];
48       }
49       out_c[i][j] = log_add(Buffer);
50     }
51   }
52 }
```

Αυτός είναι ο κύριος πυρήνας που χρησιμοποιήσαμε για την επιτάχυνση, οι πραγματικές τιμές που χρησιμοποιήσαμε είναι **A_ROWS = A_NFEATS = 80, A_COLS = BUFFER_SIZE = 20**. Μέχρι τώρα έχουμε δει μόνο την επιτάχυνση των δύο μικρών πολλαπλασιασμών μήτρας στα επόμενα κεφάλαια συζητούμε πώς χρησιμοποιήσαμε αυτό ως βάση για να εκτελέσουμε μαζικό πολλαπλασιασμό των πινάκων από τις αραιές σειρές στη μνήμη DDR.

4.4 Διεπαφή CPU και FPGA

Προκειμένου να πετύχουμε την μεγαλύτερη δυνατή ταχύτητα μεταφοράς των δεδομένων ανάμεσα σε CPU και FPGA αποφασίσαμε να χρησιμοποιήσουμε τους ταχύτερους διεπαφές που διαθέτει το Zynq. Αναλυτικότερα αποφασίσαμε να χρησιμοποιήσουμε την διεπαφή ACP η με την οποία τα δεδομένα μεταφέρονται κατευθείαν απο την cache των CPU. Στην μεριά του Programming Logic χρησιμοποιήσαμε AXI Simple DMAs (AXIDMA_SIMPLE) οι οποίοι είναι κατα πολύ ταχύτεροι απο τους AXI Scatter Gather DMAs (AXIDMA_SG) ωστόσο εισάγουν δυο περιορισμούς.



Σχήμα 4.4: Αφαιρετική παρουσίαση της διάταξης πολλαπλασιασμού πινάκων που δημιουργήσαμε στην προγραμματιστική λογική.

Πρώτον δουλεύουν απαραίτητα με συνεχείς φυσικές διευθύνσεις μνήμης , σε αντίθεση με του AXIDMA_SG οι οποιοι μπορουν να δουλέψουν σε επίπεδο εικονικής μνήμης κάνοντας μόνοι τους την συλλογή των δεδομένων από την φυσική μνήμη , και δεύτερον μπορούν να χρησιμοποιηθούν μόνο για την μεταφορά μεγέθους μέχρι 8Mb. Τέλος για την επικοινωνία μεταξύ των DMAs και του επιταχυντή μας αποφασίσαμε να χρησιμοποιήσουμε το πρωτόκολλο AXI_STREAM.
Στα κεφάλαια που ακολουθούν παρουσιάζουμε τους τρεις πυρήνες που κατασκευάσαμε οι οποίοι πέτυχαν τα καλύτερα αποτελέσματα σε σχέση με την εκτέλεση σε λογισμικό χωρίς επιτάχυνση , αλλά και σε σχέση με τους υπόλοιπους πυρήνες που δοκιμάσαμε.

4.5 Πρώτη έκδοση πυρήνα (Version 1)

Υπενθυμίζουμε ότι στο υλικό έχουμε υλοποιήσει πολλαπλασιασμούς πινάκων μεγέθους $BUFFER\_SIZE \times NFEATS = 20 \times 80$. Στην πρώτη απόπειρα για επιτάχυνση με σχετικά καλά αποτελέσματα έχουμε κάνει την παραδοχή ότι ο αριθμός των χαρακτηριστικών είναι σταθερός και ίσος με $80 (NFEATS = 80)$, πράγμα που σημαίνει οτι

σε περίπτωση που ο χρήστης θέλει να να εκτελέσει τον αλγόριθμο για διαφορετικό πλήθος χαρακτηριστικών θα πρέπει να κάνει επαναδημιουργία του bitstream που προγραμματίζει το FPGA , διαδικασία σχετικά χρονοβόρα. Αυτή η παραδοχή ωστόσο είναι αρχικά απαραίτητη διότι για να γίνουν unroll και pipeline οι λούπες που αναφέραμε σε προηγούμενο κεφάλαιο θα πρέπει να είναι εκ των προτέρων γνωστό το πλήθος των επαναλήψεων που πρόκειται να εκτελεστούν στο υλικό.
Για να αντιμετωπίσουμε το γεγονός ότι τα διανύσματα βρίσκονται διασκορπισμένα στην μνήμη DDR καθώς και το ότι δεν είμαστε σε θέση να γνωρίζουμε εκ των προτέρων το πλήθος τον βαθμολογιών ενός χρήστη ή μιας ταινίας, φτιάξαμε έναν driver του επιταχυντή οποίος εκτελείτε απο την CPU και επιτελεί τις εξής βασικές λειτουργίες

- Συγκεντρώνει σε έναν τοπικό buffer ($Sbuffer_{NFEATS \times BUFFER_SIZE} = Sbuffer_{80\times20}$) τα διανύσματα χαρακτηριστικών των ταινιών που έχει βαθμολογήσει ο τρέχων χρήστης ή τα διανύσματα χαρακτηριστικών των χρηστών που έχουν βαθμολογήσει την τρέχουσα ταινία. Φυσικά ο Sbuffer χωράει τα διανύσματα μόνο $BUFFER_SIZE = 20$ ταινιών διότι θα πρέπει να σταλεί στο επιταχυντή υλικού, οποίος δέχεται πίνακες συγκεκριμένης διάστασης. Ο οδηγός φροντίζει σε κάθε επανάληψη να γεμίσει τον Sbuffer με την επόμενη εικοσάδα διανυσμάτων.

- συγκεντρώνει σε ένα δεύτερο τοπικό buffer ($Rbuffer_{\times1}$) τις αντίστοιχες βαθμολογίες που έχουν αποδοθεί στις ταινίες πάλι ανά εικοσάδες για τους ίδιους ακριβώς λόγους.

- σε περίπτωση που το πλήθος των βαθμολογιών δεν διαιρείται ακριβώς με το BUFFER_SIZE ($nratingsmodBUFFER\_SIZE \neq 0$) , αναλαμβάνει να προσθέσει το αντίστοιχο padding με μηδενικά στους τελευταίους Sbuffer και Rbuffer.

Τέλος όσον αφορά την διεπαφή των DMAs με τον πυρήνα μας έχουμε χρησιμοποιήσει το πρωτόκολλο AXI-STREAM, την διεπαφή του οποίου την έχουμε προσαρτήσει κατευθείαν πάνω στον πυρήνα μας.

Πρώτα παρουσιάζουμε το πρόγραμμα οδηγό το οποίο εκτελείται στην CPU.

Αρχικά υπολογίζουμε το πλήθος των buffers στους οποίος θα κατανέμουμε τα δεδομένα μας.

```
void bufferizeAndSend(float* RBUFFER,float* SBUFFER ,float* M,int* sel,
    float* ratings,float* MM_HW,float* V,int nratings)
{
  //calculate the number of buffers we need
  int numBuffers     = nratings/BUFFER_SIZE+1;
  int lastBufferSize = nratings%BUFFER_SIZE;
  int i,j,b,buffIndx,rbuffIndx,mov,k1=0;
```

Στην συνέχεια αρχικοποιούμε τους πίνακες στους οποίους θα αποθηκευτεί το τελικό αποτέλεσμα στο 0 . Πάνω σε αυτούς ο υλικός επιταχυντής θα κάνει σταδιακά accumulate το αποτέλεσμα , συνεπώς είναι αναγκαίο να είναι αρχικοποιημένοι με μηδενικά.

```
  //initialize results to zero
  for(i=0;i<NFEATS*NFEATS;i++){
    MM_HW[i]=0;
  }
```

```
  for(i=0;i<NFEATS;i++){
    V[i]=0;
  }
```

ξεκινάμε επαναληπτικά το γέμισμα των buffers με το διάνυσμα της ταινίας mov η οποία είναι μια απο της ταινίες που έχει βαθμολογήσει ο χρήστης. Ο πίνακας sel[nratings], περιέχει ακριβώς τις ταινίες που ο χρήστης έχει βαθμολογήσει.

```
  for(b=0;b<numBuffers;b++){
    if(b!=numBuffers-1){
      buffIndx=0;
      for(i=0;i<BUFFER_SIZE;i++){
        mov = sel[k1];
        RBUFFER[i]  = ratings[k1];
        for(j=0;j<NFEATS;j++){
          SBUFFER[buffIndx] = M[mov*NFEATS+j];
          buffIndx++;
        }
        k1++;
      }
    }
```

Οι τελευταίοι buffers θα πρέπει γεμίσουμε με τα απαραίτητα μηδενικά για τους λόγους που προαναφέραμε.

```
    else if(lastBufferSize!=0){
      buffIndx = 0;
      rbuffIndx= 0;
      for(i=0;i<lastBufferSize;i++){
        mov = sel[k1];
        RBUFFER[rbuffIndx] = ratings[k1];
        rbuffIndx++;
        for(j=0;j<NFEATS;j++){
          SBUFFER[buffIndx] = M[mov*NFEATS+j];
          buffIndx++;
        }
        k1++;
      }
      for(i=buffIndx;i<BUFFER_SIZE*NFEATS;i++){SBUFFER[i]=0;}
      for(i=rbuffIndx;i<BUFFER_SIZE;i++){RBUFFER[i]=0;}
    }
```

στο τέλος καλούμε με τους Buffers που κατασκευάσαμε την συνάρτηση που έχει υλοποιηθεί πάνω στο υλικό και εκτελεί τους δυο πολλαπλασιασμούς που χρήζουν επιτάχυνση.

```
    topLevelHW(SBUFFER,RBUFFER,MM_HW,V,MM_HW,V);
  }
}
```

Η συνάρτηση που ακολουθεί αποτελεί ουσιαστικά τον επιταχυντή που κατασκευάσαμε. Το Vivado SDSoC θα φροντίσει μέσω των οδηγιών που θα δώσουμε να φτιάξει το bitstream για την παρακάτω συνάρτηση , να προγραμματίσει το FPGA καθώς επίσης και την δημιουργία της κατάλληλης διεπαφής μεταξύ CPU και FPGA , πάλι σύμφωνα με τις δικές μας οδηγίες.

Αρχικά ορίζουμε στην κεφαλίδα της συνάρτησης όλα τα πεδία που είναι απαραίτητα για να φτιαχτεί η διεπαφή AXI_STREAM, και δηλώνουμε ρητά ότι επιθυμούμε η μεταφορά όλων των πινάκων να γίνει με το συγκεκριμένο πρωτόκολλο (axis).

```
void topLevelHW(ap_axiu<32,1,1,1> SBUFFER[BUFFER_SIZE*NFEATS],
                ap_axiu<32,1,1,1> RBUFFER[BUFFER_SIZE]  ,
          ap_axiu<32,1,1,1> A_HWin[NFEATS*NFEATS],
          ap_axiu<32,1,1,1> V_HWin[NFEATS],
          ap_axiu<32,1,1,1> A_HWout[NFEATS*NFEATS],
          ap_axiu<32,1,1,1> V_HWout[NFEATS])
{

  #pragma HLS interface axis port  =SBUFFER
  #pragma HLS interface axis  port =RBUFFER
  #pragma HLS interface axis port  =A_HWin
  #pragma HLS interface axis port  =V_HWin
  #pragma HLS interface axis  port =A_HWout
  #pragma HLS interface axis port  =V_HWout
  #pragma HLS interface ap_ctrl_none port=return
```

Στην συνέχεια ξεκινάμε την ανάγνωση των δεδομένων. Τα σημαντικό πεδίο εδώ είναι το πεδίο .data το οποίο είναι αυτό που περιέχει και τα δεδομένα μας. Τα δεδομλενα που περιέχονται στους Sbuffer και Rbuffer αποθηκεύονται σε τοπικά brams τα: _buffer1 , _buffer2 , _rbuff , _A και _V . Στην συνέχεια δίνουμε οδηγίες για το partitioning των brams , για να αποφύγουμε τα memory bottlenecks , όπως ακριβώς εξηγήσαμε σε προηγούμενο κεφάλαιο.

```
  volatile ap_axiu<32,1,1,1> tmp1, tmp2,tmp3,tmp4;
       union {
            int ival;
            float floatval;
       } conv1, conv2,conv3,conv4;
  float temp;
  int i,j,k;
  float _buffer1[BUFFER_SIZE][NFEATS];
  float _buffer2[NFEATS][BUFFER_SIZE];
  float _rbuff[BUFFER_SIZE];
  float _A[NFEATS][NFEATS];
  float _V[NFEATS];

  #pragma HLS array_partition variable=_buffer2 block factor=BUFFER_SIZE
    /2 dim=2
  #pragma HLS array_partition variable=_buffer1 block factor=BUFFER_SIZE
    /2 dim=1
  #pragma HLS array_partition variable=_rbuff   block factor=BUFFER_SIZE
    /2 dim=1

  // READ FEATURE VECTORS
  COPY_IN1: for(i=0,k=0;i<BUFFER_SIZE;i++){
    COPY_IN2: for(j=0;j<NFEATS;j++,k++){
      #pragma HLS PIPELINE
      conv1.ival  = SBUFFER[k].data;
      temp        = conv1.floatval;
      tmp1.keep   = SBUFFER[k].keep;
      tmp1.strb   = SBUFFER[k].strb;
```

```
      tmp1.user    = SBUFFER[k].user;
      tmp1.last    = SBUFFER[k].last;
      tmp1.id      = SBUFFER[k].id;
      tmp1.dest    = SBUFFER[k].dest;

      _buffer1[i][j] = temp;
      _buffer2[j][i] = temp;
      }
    }

  // READ RATINGS
  COPY_IN3: for(i=0;i<BUFFER_SIZE;i++){
    #pragma HLS PIPELINE
    conv2.ival  = RBUFFER[i].data;
    _rbuff[i] = conv2.floatval;
    tmp2.keep    = RBUFFER[i].keep;
    tmp2.strb    = RBUFFER[i].strb;
    tmp2.user    = RBUFFER[i].user;
    tmp2.last    = RBUFFER[i].last;
    tmp2.id      = RBUFFER[i].id;
    tmp2.dest    = RBUFFER[i].dest;
    }
```

Εδώ ουσιαστικά μεταφέρουμε στο υλικό τους μέχρι τώρα υπολογισμούς που έχουν προκύψει από τους προηγούμενους buffers, στην πρώτη δηλαδη επανάληψη-κλήση που ακόμα δεν έχει γίνει κανένας πολλαπλασιασμός το υλικό θα λάβει μόνο μηδενικά ενώ στις υπόλοιπες κλήσεις το υλικό θα λάβει το αποτέλεσμα που έχει προκύψει μέχρι τότε από τους πολλαπλασιασμούς των buffers.

```
  // READ ACCUMMULATORS
  COPY_IN4: for(i=0,k=0;i<NFEATS;i++){
    COPY_IN5: for(j=0;j<NFEATS;j++,k++){
      #pragma HLS PIPELINE
      conv3.ival  = A_HWin[k].data;
      _A[i][j]    = conv3.floatval;
      tmp3.keep    = A_HWin[k].keep;
      tmp3.strb    = A_HWin[k].strb;
      tmp3.user    = A_HWin[k].user;
      tmp3.last    = A_HWin[k].last;
      tmp3.id      = A_HWin[k].id;
      tmp3.dest    = A_HWin[k].dest;
    }
  }
  COPY_IN6: for(i=0;i<NFEATS;i++){
    #pragma HLS PIPELINE
    conv4.ival  = V_HWin[i].data;
    _V[i] = conv4.floatval;
    tmp4.keep    = V_HWin[i].keep;
    tmp4.strb    = V_HWin[i].strb;
    tmp4.user    = V_HWin[i].user;
    tmp4.last    = V_HWin[i].last;
    tmp4.id      = V_HWin[i].id;
    tmp4.dest    = V_HWin[i].dest;
  }
```

Αφού έχει ολοκληρωθεί το διάβασμα τόσο των buffer όσο και των πινάκων που περιέχουν τα μέχρι τώρα αποτελέσματα (accumulators) μπορούμε να καλέσουμε τους δυο πολλαπλασιαστές.

```
// MULTIPLY
mul_hw1(_buffer1,_buffer2,_A);
mul_hw2(_buffer1,_rbuff,_V);
```

Αφού ολοκληρωθούν οι πολλαπλασιασμοί στέλνουμε πίσω τα αποτελέσματα στους DMAs πάλι με χρήση του πρωτοκόλλου AXI_STREAM, οι οποίοι με την σειρά τους θα τα στείλουν πίσω στην cache με μέσω της θύρας ACP.

```
//STREAM OUT MM_HWout
  for(int i = 0, k = 0; i < NFEATS; i++) {
      for(int j = 0; j < NFEATS; j++, k++) {
    #pragma HLS PIPELINE II=1
          conv1.floatval = _A[i][j];
          A_HWout[k].data  = conv1.ival;
          A_HWout[k].keep  = 15;
          A_HWout[k].strb  = 1;
          A_HWout[k].user  = 0;
          A_HWout[k].last  = (k == (NFEATS*NFEATS-1)) ? 1 : 0;
          A_HWout[k].id    = 0;
          A_HWout[k].dest  = 0;
      }
  }

//STREAM OUT V_HWout
  for(int i = 0 ; i < NFEATS; i++) {
  #pragma HLS PIPELINE II=1
    conv1.floatval = _V[i];
    V_HWout[i].data  = conv1.ival;
    V_HWout[i].keep  = 15;
    V_HWout[i].strb  = 1;
    V_HWout[i].user  = 0;
    V_HWout[i].last  = (i == (NFEATS-1)) ? 1 : 0;
    V_HWout[i].id    = 0;
    V_HWout[i].dest  = 0;
}

}
```

Οι πολλαπλασιασμοί εκτελούνται όπως ακριβώς αναφέραμε σε προηγούμενο κεφάλαιο. Πλέον είναι ακόμα περισσότερο ξεκάθαρος ο λόγος που χρησιμοποιούμε σταθερές τιμές για τον αριθμό των χαρακτηριστικών (NFEATS). Ακόμα αφού γνωρίζουμε ότι το αποτέλεσμα είναι ένας συμμετρικός πίνακας, μπορούμε να εκτελέσουμε τους πολλαπλασιασμούς από την διαγώνιο και πάνω πράγμα που το πετυχαίνουμε με την χρήση ενός πολυπλέκτη [if(j>=i)]. Τέλος αξίζει να παρατηρήσουμε ότι στον λογαριθμικό αθροιστή που έχουμε κατασκευάσει, εκτός από τα αποτελέσματα των διαδοχικών πολλαπλασιασμών περνάμε και την τιμή που έχουν μέχρι τώρα οι πίνακες συσσώρευσης.

```
void mul_hw1(float _buffer1[BUFFER_SIZE][NFEATS],float _buffer2[NFEATS][
    BUFFER_SIZE],float _A[NFEATS][NFEATS]){
```

```
int i,j,r;
float result[BUFFER_SIZE];
#pragma HLS array_partition variable=result  complete
MM1: for(i=0;i<NFEATS;i++){
  MM2: for(j=0;j<NFEATS;j++){
      #pragma HLS PIPELINE
      if(j>=i){
      float old_value = _A[i][j];
            MM3: for(r=0;r<BUFFER_SIZE;r++){
                  result[r] = _buffer2[i][r]*_buffer1[r][j];
               }
               _A[i][j] = log_add(result,old_value);
      }
      }
    }
}
```

Όμοια με τον προηγούμενο.

```
void mul_hw2(float _buffer1[BUFFER_SIZE][NFEATS],float _rbuff[BUFFER_SIZE
    ],float _V[NFEATS]){
int i,j;
float result[BUFFER_SIZE];
#pragma HLS array_partition variable=result  complete

  V1: for(i=0;i<NFEATS;i++){
     float old_value=_V[i];
     #pragma HLS PIPELINE
     V2: for(j=0;j<BUFFER_SIZE;j++){
         result[j]=_buffer1[j][i]*_rbuff[j];
       }
       _V[i] = log_add(result,old_value);
     }
}
```

Ο λογαριθμικός αθροιστής έχει υλοποιηθεί ακριβώς όπως τον περιγράψαμε.

```
float log_add(float result[BUFFER_SIZE],float value){
#pragma HLS INLINE
int r;
float result1[BUFFER_SIZE/2],result2[BUFFER_SIZE/4],result3[BUFFER_SIZE
    /8+1];
#pragma HLS array_partition variable=result1 complete
#pragma HLS array_partition variable=result2 complete
#pragma HLS array_partition variable=result3 complete

int a = 0;
  MM4: for(r=0;r<10;r++){
     result1[r] = result[a]+result[a+1];
         a+=2;
       }
       a=0;
   result2[0] = result1[0] + result1[1];
   result2[1] = result1[2] + result1[3];
   result2[2] = result1[4] + result1[5];
```

```
  result2[3] = result1[6] + result1[7];
  result2[4] = result1[8] + result1[9];

  result3[0] = result2[0] + result2[1];
  result3[1] = result2[2] + result2[3];
  result3[2] = result2[4] + value;

  float x = result3[0]+result3[1]+result2[4];
  return x;
}
```

Στο αρχείο των κεφαλίδων με τις παρακάτω HLS οδηγίες, ρυθμίσαμε την διεπαφή , η οποία αποτελείτε τρία πράγμα.

- Προσδιορισμός της πόρτας μεταξύ CPU και FPGA (system port: HP,ACP, κλπ).
- Προσδιορισμός των DMAs που θέλουμε να χρησιμοποιήσουμε.
- Προσδιορισμός της διεπαφής-πρωτοκόλλου για την μεταφορά των δεδομένων από τους DMAs στον πυρήνα.

Επιλογή διεπαφής CPU-FPGA(sysport)

- #pragma SDS data sys_port (SBUFFER: ACP, RBUFFER: ACP, A_HWin: ACP, V_HWin: ACP, A_HWout: ACP, V_HWout: ACP)

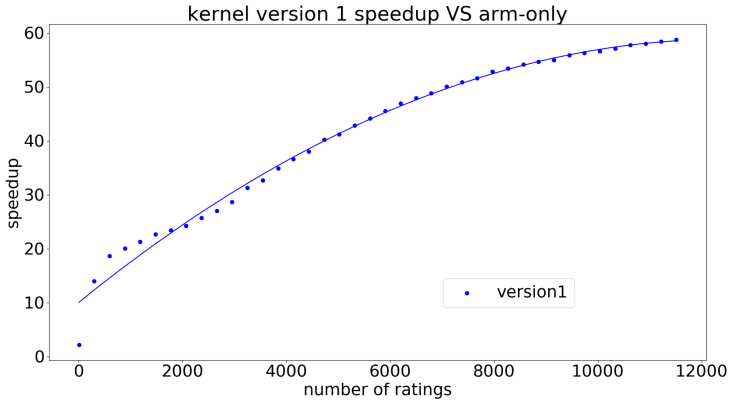
Επιλογή DMAs

Αρχικά δηλώνουμε ότι οι παρακάτω πίνακες βρίσκονται σε συνεχόμενη φυσική μνήμη , πράγμα που καταφέρνουμε χρησιμοποιώντας τις βιβλιοθήκες που παρέχει η xilinx για το SDSoC , ώστε να μπορέσουμε να κάνουμε χρήση των AXIDMA_SIMPLE, και στην συνέχεια δηλώνουμε ρητά πλέον ότι θέλουμε τους απλούς αλλά και παράλληλα ταχύτερους DMAs για την μεταφορά των δεδομένων:

- #pragma SDS data mem_attribute (Sbuffer: PHYSICAL_CONTIGUOUS | CACHEABLE ,RBUFFER: PHYSICAL_CONTIGUOUS | CACHEABLE ,A_HWin: PHYSICAL_CONTIGUOUS | CACHEABLE, V_HWin: PHYSICAL_CONTIGUOUS | CACHEABLE, A_HWout: PHYSICAL_CONTIGUOUS | CACHEABLE, V_HWout: PHYSICAL_CONTIGUOUS | CACHEABLE)
- #pragma SDS data data_mover(Sbuffer: AXIDMA_SIMPLE ,RBUFFER: AXIDMA_SIMPLE,A_HWin: AXIDMA_SIMPLE, V_HWin: AXIDMA_SIMPLE , A_HWout: AXIDMA_SIMPLE ,V_HWout: AXIDMA_SIMPLE)

Επιλογή πρωτοκόλλου επικοινωνίας DMA-Πυρήνα

- έχει οριστεί απευθείας στων κώδικα παραπάνω ως AXI_STREAM.



Σχήμα 4.5: Η επιτάχυνση που πέτυχε ο υλικός επιταχυντής (πρώτος πυρήνας) σε σχέση με την εκτέλεση μόνο σε λογισμικό(arm cortex A9). Για να είναι "δίκαια" τα αποτελέσματα στον χρόνο που μετρήσαμε για τον επιταχυντή έχουμε προσθέσει και τον χρόνο εκτέλεσης του οδηγού. Οι κουκίδες αποτελούν τις μετρήσεις που πήραμε ενώ η καμπύλη προέκυψε μετά από παλινδρόμηση σε πολυώνυμο δεύτερου βαθμού

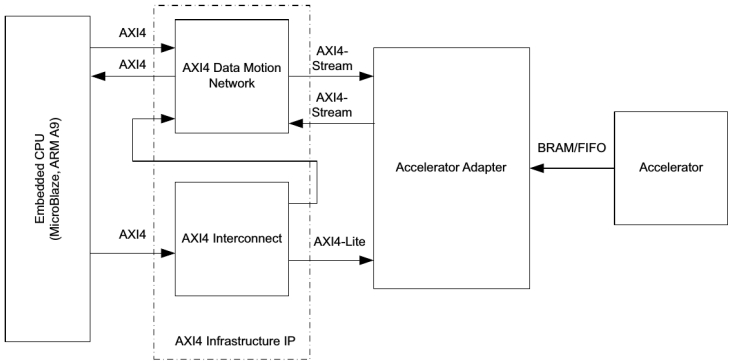
Παρακάτω παρουσιάζουμε τους πόρους που κατέλαβε ο πυρήνας μας πάνω στο υλικό. Πλέον γίνεται φανερός ο λόγος που χρησιμοποιήσαμε BUFFER_SIZE = 20, αφού με αυτήν την επιλογή χρησιμοποιούμε σχεδόν όλα όλους τους πολλαπλασιαστές (Χρησιμοποίηση DSP48 = 89 % έχουμε αφήσει σκόπιμα ένα 10% αχρησιμοποίητο διότι σε διαφορετική περίπτωση το SDSoC δυσκολευόταν να κατασκευάσει ένα κρίσιμο μονοπάτι το οποίο να μπορεί να εκτελεστεί στη συχνότητα που επιλέξαμε).

	Bram_18k	DSP48E	FF	LUT
Σύνολο	57	196	24637	34997
Διαθέσιμα	280	220	106400	53200
(%)Χρησιμοποίηση	20	89	23	65

Τέλος φτιάξαμε ένα benchmark στο οποίο εκτελούντα υπολογισμοί με επιτάχυνση και δίχως επιτάχυνση με την τιμή της μεταβλητής nratings να μεταβάλλεται από 1 μέχρι περίπου 12000. Στην εικόνα παρουσιάζεται πόσες φορές ταχύτερα εκτελείτε ο υπολογισμός σε σχέση με το την υλοποίηση στο λογισμικό. Για nratings κοντά στο 12000 η επιτάχυνση αγγίζει το 60x.

4.6 Δεύτερη Έκδοση Πυρήνα (Version 2) - Χρήση του IP AXI4-Stream Accelerator Adapter

Αν και η επιτάχυνση που καταφέραμε να πετύχουμε με τον πρώτο πυρήνα ήταν ικανοποιητική αποφασίσουμε να ελέγξουμε αν μπορούμε να κάνουμε χρήση κάποιου έτοιμου IP (Intellectual Property Block) που παρέχει η Xilinx. Συγκεκριμένα ψάξαμε για IPs τα οποία να σχετίζονται με κάποιον τρόπο με την δουλειά που έχουμε κάνει μέχρι τώρα καθώς επίσης και να είναι κατάλληλα για το σκοπό που τα θέλουμε δηλαδή



Σχήμα 4.6: Παρέχει μια AXI4-Stream διεπαφή για την μεταφορά των πινάκων προς την AXI υποδομή και μια AXI-LITE για την μεταφορά των αριθμών ανάμεσα στην CPU και το FPGA , ενώ παρέχει BPAM/FIFO προς τον επιταχυντή.

την χρήση του σε μια ετερογενής αρχιτεκτονική με ενσωματωμένη CPU. Το καταλληλότερο IP που βρήκαμε είναι το AXI4-Stream Accelerator Adapter[27].

Ο AXI4-Stream Accelerator Adapter παρέχει διεπαφές AXI4-Stream προς την υποδομή AXI ανάμεσα στο υλικό και την CPU , και διεπαφή BRAM/FIFO προς τον πυρήνα επιτάχυνσης που κατασκευάσαμε. Αυτό το 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 , πράγμα που μας φάνηκε εξαιρετικά χρήσιμο στην συνέχεια που προσπαθήσαμε να τοποθετήσουμε το πυρήνα μας σε περιβάλλον Python.

Η βελτίωση σε επίπεδο συστήματος του accelerator adapter οφείλεται στο γεγονός ότι κάνει εξαιρετικά εύκολο το pipelining σε επίπεδο task. Παρέχει δηλαδή την δυνατότητα μέσω ενός απλού ap_ctrl να ενορχηστρωθούν πολλαπλοί πυρήνες ωστε να εκτελέσουν μια συνολική εργασία με χρήση διοχέτευσης. Μπορεί στην δική μας περίπτωση να μην είχαμε πολλαπλούς πυρήνες ώστε να εκμεταλευτούμε την διοχέτευση ωστόσο όπως θα φανεί και αργότερα και μόνο ο βέλτιστος αυτόματος χειρισμός του AXI4-STREAM από το IP είναι αρκετός για να αυξήσει την επίδοση του επιταχυντή.Το μόνο που χρειάζεται να κάνουμε στο περιβάλλον του SDSoC για να προστεθεί ο Accelerator Adapter είναι να προσθέσουμε την εντολή σύνθεσης υψηλού επιπέδου.

- #pragma SDS data access_pattern(SBUFFER:SEQUENTIAL, RBUFFER:SEQUENTIAL, MM_HWin:SEQUENTIAL, V_HWin:SEQUENTIAL, MM_HWout:SEQUENTIAL, V_HWout:SEQUENTIAL)

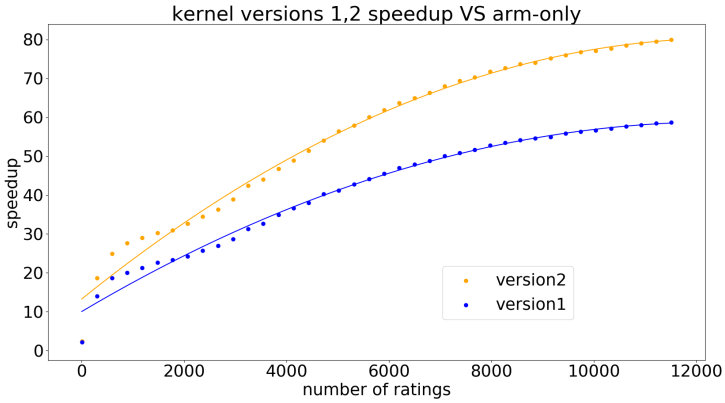
και φυσικά αφου πλέον ο πυρήνας μας επικοινωνεί με το accelerator adapter με διεπαφή FIFO θα μπορούμε να αφαιρέσουμε απο τον κώδικα μας τον χειρισμό του AXI-STREAM , διασφαλίζοντας ωστόσο πάντα ότι η ανάγνωση των δεδομένων γίνεται σειριακά.

```
void topLevelHW(float SBUFFER[BUFFER_SIZE*NFEATS], float RBUFFER[
    BUFFER_SIZE] ,float MM_HWin[NFEATS*NFEATS], float V_HWin[NFEATS], float
    MM_HWout[NFEATS*NFEATS], float V_HWout[NFEATS]){
  float _buffer1[BUFFER_SIZE][NFEATS];
  float _buffer2[NFEATS][BUFFER_SIZE];
  float _rbuff[BUFFER_SIZE];
  float MUL_OUT[NFEATS][NFEATS];
  float V_OUT[NFEATS];
#pragma HLS array_partition variable=_buffer2 block factor=BUFFER_SIZE/2
    dim=2
#pragma HLS array_partition variable=_buffer1 block factor=BUFFER_SIZE/2
    dim=1
#pragma HLS array_partition variable=_rbuff   block factor=BUFFER_SIZE/2
    dim=1

  readSData(SBUFFER, _buffer1, _buffer2);
  readRData(RBUFFER, _rbuff);
  mul_hw1(_buffer1, _buffer2,MUL_OUT);
  mul_hw2(_buffer1, _rbuff,V_OUT);
}
```

Στην λογική των πολλαπλασιασμών δεν έχει αλλάξει κάτι για αυτό τον λόγο αποφύγαμε να επαναλάβουμε τον κώδικα των πολλαπλασιασμών. Ωστόσο όσον αφορά το διάβασμα παρατηρούμε ότι πλέον δεν χρειάζεται κανένας χειρισμός από την πλευράς του πρωτοκόλλου AXI4-STREAM ή των σημάτων που αυτό περιλαμβάνει. Φυσικά είναι ακόμα απαραίτητο η ανάγνωση των δεδομένων εως ότου αυτά αποθηκευτούν σε BRAMs να γίνεται σειριακά , αφού η διεπαφή FIFO τα αποστέλλει με τέτοιο τρόπο στο πυρήνα.

```
void readSData(float SBUFFER[BUFFER_SIZE*NFEATS], float _buffer1[
    BUFFER_SIZE][NFEATS], float _buffer2[NFEATS][BUFFER_SIZE]){
  float temp;
  int i,j;
COPY_IN1:for(i=0;i<BUFFER_SIZE;i++){
COPY_IN2:for(j=0;j<NFEATS;j++){
#pragma HLS PIPELINE REWIND
      temp = SBUFFER[i*NFEATS+j]; //sequential consumption in order to
    stream
      _buffer1[i][j] = temp;
      _buffer2[j][i] = temp;
    }
    }
}

void readRData(float RBUFFER[BUFFER_SIZE], float _rbuff[BUFFER_SIZE]){
  int i;
COPY_IN3:for(i=0;i<BUFFER_SIZE;i++){
#pragma HLS PIPELINE REWIND
      _rbuff[i] = RBUFFER[i];
    }
}
```

Παρατηρούμε ότι η χρησιμοποίηση των πόρων δεν έχει μεταβληθεί ιδιαίτερα σε σχέση με τον προηγούμενο πυρήνα



Σχήμα 4.7: Speedups εκδόσεων πυρήνα 1 και 2 σε σχέση με arm-only εκτέλεση.Παρατηρούμε ότι με χρήση του Accelerator-Adapter βελτιώθηκε αρκετά η απόδοση.Οι κουκίδες αποτελούν τις μετρήσεις που πήραμε ενώ η καμπύλη προέκυψε μετά από παλινδρόμηση σε πολυώνυμο δεύτερου βαθμού

	Bram_18k	DSP48E	FF	LUT
Σύνολο	57	200	25296	30866
Διαθέσιμα	280	220	106400	53200
(%)Χρησιμοποίηση	20	90	23	58

4.7 Τρίτη Έκδοση Πυρήνα (Version 3) - Accelerator Adapter - Μεταβλητό Παράθυρο

Οι παραπάνω υλοποιήσεις μπορεί υπερέχουν κατά πολύ μιας απλής υλοποίησης του αλγορίθμου σε επεξεργαστή, ωστόσο η υπάρχουν ακόμα περιθώρια για περαιτέρω βελτίωση. Σε όσες υλοποιήσεις είδαμε μέχρι τώρα αποστέλλουμε στο FPGA τα features σταθερού πλήθους ταινιών ή χρηστών και ίσου με BUFFER_SIZE (το οποίο ορίστικε ίσο με 20) εκτελούσαμε τους υπολογισμούς, επιστρέφαμε το αποτέλεσμα πίσω στην CPU και στην συνέχεια κάναμε το ίδιο για τα επόμενα διανύσματα με features. Οι υλοποιήσεις αυτές ωστόσο περιέχουν κάποια αχρείαστα I/O overheads.
Πιο συγκεκριμένα οι παραπάνω υλοποιήσεις απαιτούν συνεχή αποστολή και λήψη του πίνακα που συσσωρεύει το αποτέλεσμα χωρίς να υπάρχει πραγματικός λόγος να γίνει αυτό εφόσον η CPU δεν κάνει καμία χρήση του συσσωρευτή. Ιδανικά θα θέλαμε να στέλνουμε στο FPGA τα διανύσματα των features με μονόδρομη επικοινωνία ανάμεσα από την CPU προς το FPGA , το οποίο με την σειρά του να κρατάει τοπικά σε BRAMs τον συσσωρευτή και στο τέλος των υπολογισμών , όταν πλέον ο συσσωρευτής θα περιέχει το τελικό αποτέλεσμα να επιστρέφεται στην CPU.
Θα μπορούσαμε να υποθέσουμε ότι αυτό είναι εφικτό με τις παραπάνω υλοποιήσεις , με την έννοια ότι αν στείλουμε τον συσσωρευτή μόνο στην πρώτη επικοινωνια CPU-FPGA, τότε αν δεν σταλθεί στις υπόλοιπες επικοινωνίες υποθέτουμε ότι δεν υπάρχει λόγος να έχει αλλάξει το περιεχόμενο των BRAM του FPGA , και συνεπώς μπορούμε

απο αυτά να διαβάσουμε τις τιμές που έχουν υπολογιστεί μέχρι εκείνη την στιγμή και να τις χρησιμοποιήσουμε σε συνδυασμό με τις νέες τιμές συνεχίζοντας τους υπολογισμούς. Ωστόσο κάτι τέτοιο θα ήταν παρακινδυνευμένο διότι δεν μπορούμε να γνωρίζουμε με απόλυτη βεβαιότητα την εσωτερική αρχιτεκτονική του FPGA, καθώς και το πώς συμπεριφέρονται τα BRAMs του FPGA όταν αυτό είναι idle, όταν δηλαδή δουλεύει ο επεξεργαστής και ότι το hardware.
Απο την άλλη δεν μπορούμε να στείλουμε όλα τα διανύσματα χαρακτηριστικών στο FPGA εν μια ριπή διότι θα ήταν αδύνατο να χωρέσουν στα BRAMs, ιδιαίτερα όταν έχουμε να κάνουυμε με μεγάλα datasets.
Η λύση στο παραπάνω πρόβλημα είναι να έχουμε όλα τα δεδομένα έτοιμα στην CPU, στην συνέχεια περνώντας τον έλεγχο στο FPGA κατασκευάζουμε έναν συσσωρευτή αρχικοποιημένο με μηδενικά και μέσα από το υλικό διαβάζουμε επαναληπτικά διανύσματα χαρακτηριστικών πλήθους BUFFER_SIZE εως ότου αυτά εξαντληθούν, τέλος στέλνουμε το τελικό αποτέλεσμα που είναι αποθηκευμένο στον συσσωρευτή πίσω στον επεξεργαστή. Με αυτό τον τρόπο ο έλεγχος δεν χρειάζεται να επιστρέψει πίσω στην CPU παρά μόνο μετά το πέρας των υπολογισμών.

```
void bufferizeAndSend(float* M,int* sel,float* ratings,float* MM_HW,float
    * V,int nratings,float* Sbuffer){

  int i,j,b,buffIndx=0,rbuffIndx,mov,k1=0;

  for(i=0;i<nratings;i++)
  {
    mov = sel[i];
    for(j=0;j<NFEATS;j++)
    {
      Sbuffer[buffIndx] = M[mov*NFEATS+j];
      buffIndx++;
    }
  }

  topLevelHW(Sbuffer,ratings,nratings,MM_HW,V);
}
```

Ο νέος οδηγός που κατασκευάστηκε στο λογισμικό είναι πολύ απλούστερος αφού όπως αναφέραμε, πλέον ο χειρισμός των διανυσμάτων και η τμηματοποίησή τους σε buffers γίνεται στην μεριά του υλικού επιταχυντή. Το μόνο που χρειάζεται να γίνει στην μεριά του λογισμικού είναι η συγκέντρωση όλων των διανυσμάτων χαρακτηριστικών σε συνεχόμενες φυσικές θέσεις μνήμης (Sbuffer) που έχουν γίνει allocate κάνοντας χρήση των αντίστοιχων βιβλιοθηκών (sds_alloc αντί για malloc).
Παρατηρούμε ότι αφού πλέον ο χειρισμός γίνετε στο υλικό, είναι αναγκαίο να αποστείλουμε σε αυτό το πλήθος (nratings) των βαθμολογήσεων της ταινίας ή του χρήστη αντίστοιχα.

Ο κώδικας του υλικού πλέον είναι ο εξής.

```
void topLevelHW(float Sbuffer[NMAXRAT*NFEATS],float ratings[NMAXRAT],int
    nratings ,float Aout[NFEATS*NFEATS],float Vout[NFEATS]){
  int b,i,j,k,r,c;
  float _buffer1[BUFFER_SIZE][NFEATS];
  float _buffer2[NFEATS][BUFFER_SIZE];
  float _rbuff[BUFFER_SIZE];
  float _A[NFEATS][NFEATS];
  float _V[NFEATS];
  float temp;
  int streampos;
#pragma HLS array_partition variable=_buffer2 block factor=BUFFER_SIZE/2
    dim=2
#pragma HLS array_partition variable=_buffer1 block factor=BUFFER_SIZE/2
    dim=1
#pragma HLS array_partition variable=_rbuff   block factor=BUFFER_SIZE
    dim=1
  //INITIALIZE
initAout:for(i=0;i<NFEATS;i++){
initAin :for(j=0;j<NFEATS;j++){
#pragma HLS PIPELINE
      _A[i][j] = 0;
    }
  }
init_V:for(i=0;i<NFEATS;i++){
#pragma HLS PIPELINE
    _V[i] = 0;
  }

  //START-CALC
  int iterations = nratings/BUFFER_SIZE+1;
  streampos=0;
  for(i=0;i<iterations;i++){
#pragma HLS loop_tripcount max = NMAXRAT/BUFFER_SIZE/2
    readSData(Sbuffer,_buffer1,_buffer2,streampos,nratings);
    readRData(ratings,_rbuff,streampos,nratings);
    mul_hw1(_buffer1,_buffer2,_A);
    mul_hw2(_buffer1,_rbuff,_V);
    streampos += BUFFER_SIZE;
  }
  //END-CALC

  //STREAM OUT
cpoutAout:for(i=0;i<NFEATS;i++){
cpoutAin:   for(j=0;j<NFEATS;j++){
#pragma HLS PIPELINE
      Aout[i*NFEATS+j] = _A[i][j];
    }
  }
cpoutV: for(i=0;i<NFEATS;i++){
#pragma HLS PIPELINE
    Vout[i] = _V[i];
  }
}
```

Παρατηρούμε ότι με βάση το πλήθος των αξιολογήσεων (nratings) και το μέγεθος του Buffer (BUFFER_SIZE) υπολογίζεται πόσες επαναλήψεις θα εκτελέσει το υλικό

(iterations = nratings/BUFFER_SIZE+1), πόσες φορές δηλαδή θα ζητήσει δεδομένα από τους DMAs και πόσες φορές θα εκτελέσει τους αντίστοιχους υπολογισμούς. Φυσικά μετά το πέρας κάθε επανάληψης απαιτείται να ενημερωθεί ο δείκτης(streampos) στον οποίο αποθηκεύουμε μέχρι ποιο σημείο των δεδομένων που βρίσκονται στον DMA έχουμε διαβάσει. Παρατηρούμε οτι τώρα υπάρχουν δύο τοπικοί συσσωρευτές αποτελεσμάτων (_A[NFEATS][NFEATS],_V[NFEATS]) οι οποίοι μεταφέρονται πίσω στην CPU μετάτο πέρας των υπολογισμών. Ο τρόπος που διεκπεραιώνονται οι υπολογισμοί δεν έχει αλλάξει ωστόσο έχει αλλάξει ο τρόπος που γίνεται η ανάγνωση των δεδομένων, όπως φαίνεται παρακάτω.

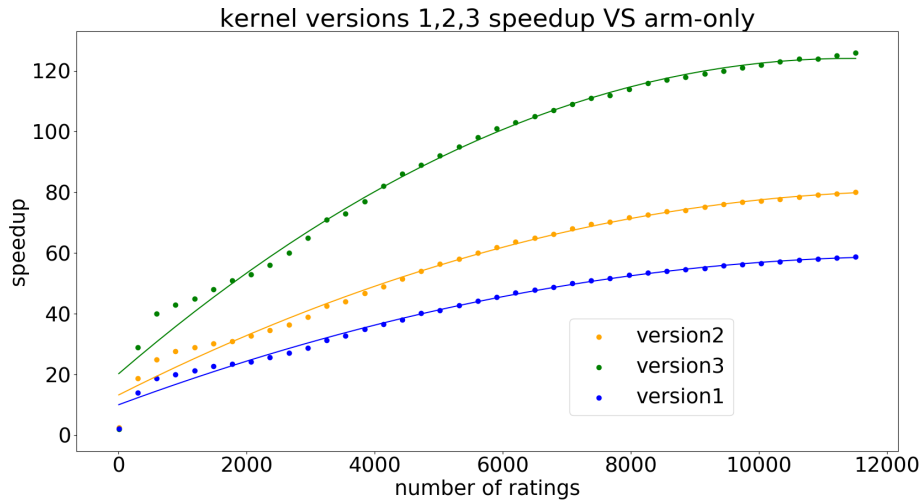
```
void readSData(float Sbuffer[NMAXRAT*NFEATS],float _buffer1[BUFFER_SIZE][
    NFEATS],float _buffer2[NFEATS][BUFFER_SIZE],int streampos,int nratings
    ){
  float temp;
  int i,j,row;
COPY_IN1:for(i=0;i<BUFFER_SIZE;i++){
   row = streampos*NFEATS;
COPY_IN2:for(j=0;j<NFEATS;j++){
#pragma HLS PIPELINE REWIND
     temp = streampos<nratings?Sbuffer[row+j]:0; //sequential consumption
     in order to stream
     _buffer1[i][j] = temp;
     _buffer2[j][i] = temp;
   }
   streampos++;
   }
}
```

Η παραπάνω συνάρτηση διαβάζει εικοσάδες (BUFFER_SIZE=20) διανυσμάτων απο τους DMAs και τα αποθηκεύει σε τοπικές BRAMs. To indexing στον πίνακα Sbuffer είναι απόλυτο , για αυτόν τον λόγο όπως προαναφέραμε χρειαζόμαστε έναν δείκτη να μας κρατά ενήμερους για την θέση μέχρι την οποία έχουμε διαβάσει.

```
void readRData(float ratings[NMAXRAT],float _rbuff[BUFFER_SIZE],int
    streampos,int nratings){
  int i;
COPY_IN3:for(i=0;i<BUFFER_SIZE;i++){
#pragma HLS PIPELINE REWIND
     _rbuff[i] = (streampos+i)<nratings?ratings[streampos+i]:0;
   }
}
```

Παρόμοια είναι η κατάσταση και με τις βαθμολογίες μόνο που σε αυτήν την περίπτωση, ο αντί για διανύσματα χαρακτηριστικών έχουμε βαθμωτές τιμές και συνεπώς τα η δομή είναι λίγο απλούστερη. Είναι σημαντικό να τονίσουμε ότι πλέον για τις βαθμολογίες δεν χρειαζόμαστε ενδιάμεσους buffers , αλλά αποστέλλουμε απευθείας ολόκληρο το διάνυσμα με τις βαθμολογίες το οποίο γίνεται ανάγνωση τμηματικά.

Σε αυτό το σημείο γεννιέται η απορία πώς μπορεί το υλικό να γνωρίζει κάθε φορά πόσες ταινίες έχει βαθμολογήσει ένας χρήστης ή αντίστοιχα από πόσους χρήστες έχει βαθμολογηθεί μια ταινία ώστε να μεταφερθεί ο αντίστοιχος όγκος δεδομένων από τους



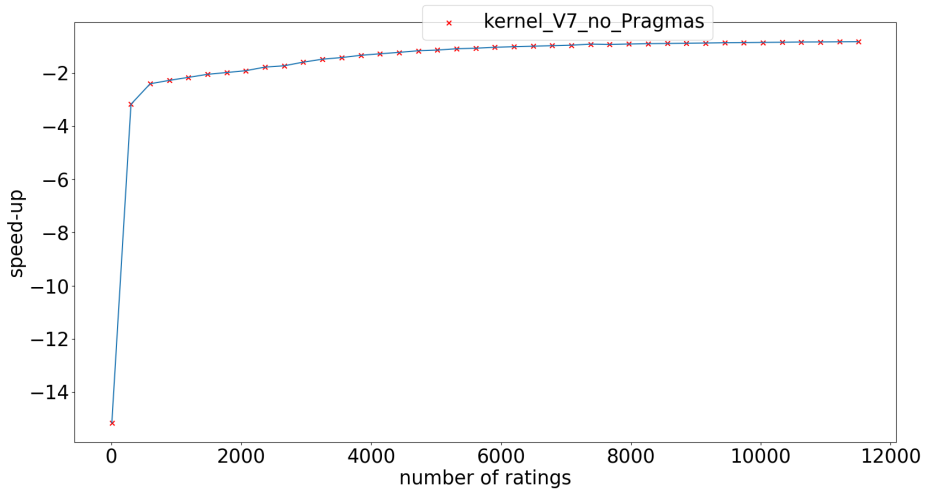
Σχήμα 4.8: Είναι εμφανές ότι ο τελευταίος πυρήνας (Version 3) που είναι απαλλαγμένος από τα κόστη μεταφοράς κατά την διάρκεια των υπολογισμών, πετυχαίνει καλύτερη επίδοση από τους δυο προηγούμενους, και κλιμακώνει πολύ καλά καθώς το μέγεθος των δεδομένων αυξάνει, όσο δηλαδή αυξάνεται η τιμή της μεταβλητής nratings. Οι κουκίδες αποτελούν τις μετρήσεις που πήραμε ενώ η καμπύλη προέκυψε μετά από παλινδρόμηση σε πολυώνυμο δευτέρου βαθμού.

DMAs.Διότι μπορεί εμείς στον πυρήνα που κατασκευάσαμε να καταναλώνουμε τα δεδομένα με βάση την μεταβλητή nratings, ωστόσο οι DMAs δημιουργούνται αυτόματα από το HLS εργαλείο SDSOC, και ο μόνος τρόπος που διαθέτουμε για να τους παραμετροποιήσουμε είναι οι HLS εντολές. Πράγματι το SDSOC μας δίνει την δυνατότητα να μεταφέρουμε μεταβλητό αριθμό δεδομένων σε κάθε κλήση , με την προυπόθεση ότι κάνουμε χρήση κάποιου streaming πρωτοκόλλου, πράγμα που εμείς κάνουμε. Η hls εντολή που χρησιμοποιήθηκε είναι η παρακάτω:

```
#pragma SDS data copy(Sbuffer[0:nratings*NFEATS],ratings[0:nratings])
```

παράμετρος της οποίας είναι η μεταβλητή nratings. Η παραπάνω εντολή τοποθετήθηκε πρίν την κεφαλίδα της συνάρτησης πυρήνα, στο σημείο δηλαδή που έχουν τοποθετηθεί και οι υπόλοιπες εντολές που καθορίζουν την διεπαφή υλικού και λογισμικού.

Όσον αφορά την επίδοση το παρακάτω γράφημα παρουσιάζει πόσες φορές γρηγορότερα εκτελούν οι πυρήνες που έχουμε κατασκευάσει μέχρι τώρα σε σχέση με την εκτέλεση μόνο σε λογισμικό τους υπολογισμούς , για διάφορα πλήθη βαθμολογιών (για διάφορες δηλαδή τιμές της μεταβλητής nratings).



Σχήμα 4.9: Η επίδοση του πυρήνα χωρίς καμία HLS εντολή

Παρατηρούμε ότι αυτές οι αλλαγές δεν έχουν επιφέρει κάποια αισθητή διαφορά στην χρησιμοποίηση των πόρων αφού αφορούν την διεπαφή παρά τον τρόπο υπολογισμού.

	Bram_18k	DSP48E	FF	LUT
Σύνολο	58	204	28603	35037
Διαθέσιμα	280	220	106400	53200
(%)Χρησιμοποίηση	20	92	26	65

4.8 Επίδοση χωρίς HLS εντολές

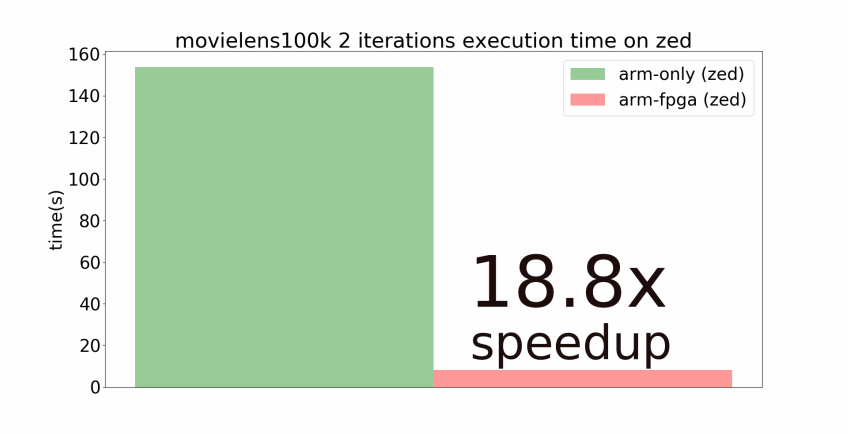
Έχει ενδιαφέρον να δούμε τι επιδόσεις πετυχαίνει ο τελευταίος πυρήνας, ο οποίος είναι και ο ταχύτερος, σε σχέση με το λογισμικό, στην περίπτωση που αφαιρέσουμε όλες τις HLS εντολές που έχουμε προσθέσει σε αυτόν και αφορούν τόσο την διεπαφή όσο και το κομμάτι των υπολογισμών.

Παρατηρούμε ότι για λίγα δεδομένα (σχετικά μικρό nratings) το hardware είναι εώς και 16 φορές πιο αργό από την CPU, συμπέρασμα απόλυτα αναμενόμενο αφού πλέον το υλικό εκτελεί σειριακά εντολές σαν μια απλή CPU και επιπροσθέτως έχει επιπλέον καθυστέρηση της μεταφοράς των δεδομένων από και προς την CPU. Αυτή η καθυστέρηση γίνεται ιδιαίτερα εμφανής όταν ο αριθμός των δεδομένων είναι μικρός και συνεπώς ο χρόνος μεταφοράς τους είναι κατά πολύ μεγαλύτερος από τον χρόνο της εκτέλεσης των υπολογισμών. Ωστόσο καθώς ο αριθμός των δεδομένων αυξάνεται και συνεπώς ο χρόνος εκτέλεσης υπολογισμών υπερκαλύπτει αυτόν της μεταφοράς, παρατηρούμε ότι ο συνολικός χρόνος εκτέλεσης στο υλικό γίνεται περίπου ίσος με τον χρόνο εκτέλεσης στην CPU.

4.9 Επίδοση στα datasets

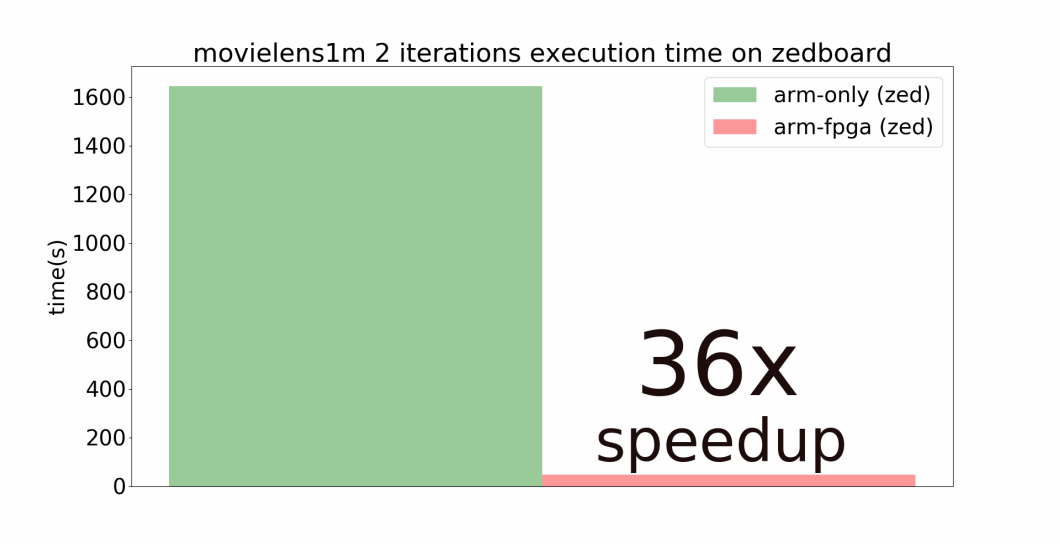
Χρησιμοποιώντας το ”καλύτερο” από τους πυρήνες που κατασκευάσαμε μπορούμε πλέον να τον ενσωματώσουμε πάνω στον αλγόριθμό μας και να πάρουμε τις αντίστοιχες μετρήσεις πάνω στα datasets που έχουμε προαναφέρει.Παρακάτω φαίνεται η διάρκεια εκτέλεσης του αλγορίθμου τόσο αν αυτός τρέξει μόνο στον arm που βρίσκεται πάνω στο zedboard αλλά και αυ τρέξει με την χρήση του επιταχυντή.

συσκευή	σύνολο δεδομένων	επαναλήψεις	χρόνος χωρίς επιτάχυνση(s)	χρόνος με επιτάχυνση(s)
zed	movielens100k	2	153.734	8.279
zed	movielens1m	2	1644.347	46.058
zc702	movielens 100k	5	384.462	19.760
zc702	movielens1m	1	858.684	25.881
i7-7500U	movielens1m	2	76.914	-

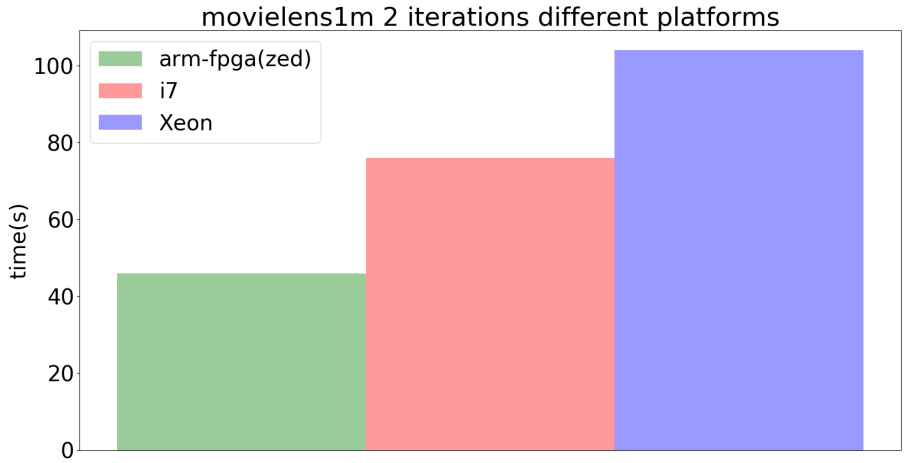


Σχήμα 4.10: Σε αυτό το διάγραμμα παρουσιάζεται η μέση διάρκεια εκτέλεσης δυο επαναλήψεων του αλγορίθμου πάνω στο dataset movielens100k. Παρατηρούμε ότι η εκτέλεση με την βοήθεια του υλικού (κόκκινο) είναι περίπου 18.8 φορές ταχύτερη σε σχέση με την εκτέλεση αποκλειστικά πάνω στον arm CPU

Για το δεύτερο dataset αποφασίσαμε να προσθέσουμε στην σύγκριση και έναν επεξεργαστή τελευταίας γενιάς (όσον αφορά το 2017 που γράφεται το παρών κείμενο) Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz cache size : 4096 KB καθώς επίσης και έναν Xeon επεξεργαστή Xeon CPU E5-2650 v2 @ 2.60GHz.Παρακάτω φαίνεται η διάρκεια εκτέλεσης του αλγορίθμου αν αυτός τρέξει στον i7 , στον Xeon και στο zedboard με χρήση επιταχυντή.



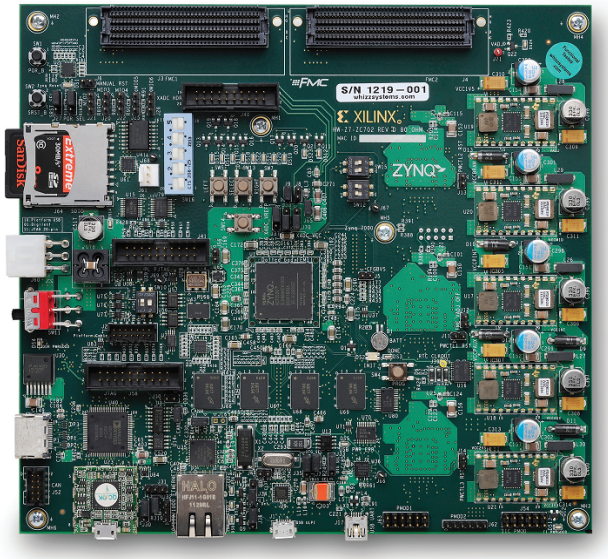
Σχήμα 4.11: Σε αυτό το διάγραμμα παρουσιάζεται η διάρκεια εκτέλεσης δυο επαναλήψεων του αλγορίθμου πάνω στο dataset movielens1m. Παρατηρούμε ότι η εκτέλεση με την βοήθεια του υλικού (κόκκινο) είναι περίπου 36 φορές ταχύτερη σε σχέση με την εκτέλεση αποκλειστικά πάνω στον arm CPU.



Σχήμα 4.12: Σε αυτό το διάγραμμα παρουσιάζεται η διάρκεια εκτέλεσης δυο επαναλήψεων του αλγορίθμου πάνω στο dataset movielens1m στις τρεις διαφορετικές πλατφόρμες.

4.10 Ενεργειακή επίδοση

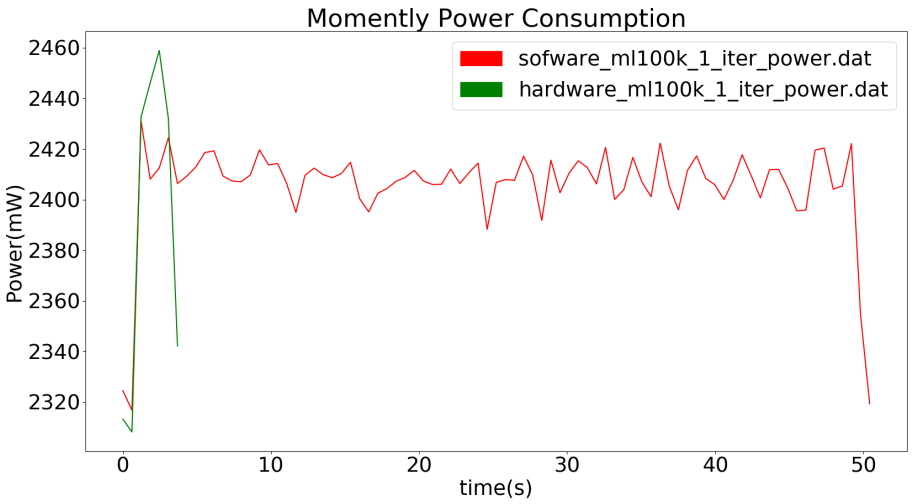
Εκτός από το όφελος που έχουμε σε ταχύτητα εκτέλεσης , ένας ακόμα λόγος που αποφασίσαμε να κάνουμε χρήση των FPGA είναι και η εξοικονόμηση ενέργειας. Για τον λόγο αυτό μετρήσαμε την κατανάλωση ενέργειας του αλγορίθμου σε δυο περιπτώσεις. Είναι σημαντικό να τονίσουμε ότι η κατανάλωση ενέργειας ενδέχεται να έχει πάρα πολλές διακυμάνσεις οι οποίες μπορεί να εξαρτώνται από τα περιφερειακά του επεξεργαστή, το dataset αλλα και πολλούς άλλους παράγοντες. Γνωρίζοντας τα παραπάνω τονίζουμε ότι τα παρακάτω αποτελέσματα είναι ενδεικτικά. Πιο συγκεκριμένα για να μετρήσουμε την ενεργειακή κατανάλωση του αλγορίθμου χρησιμοποιήσαμε την πλακέτα ZC702 η οποία διανέμεται με μικροελεγχτή μέτρησης ενέργειας.



Σχήμα 4.13: Η πλακέτα Zc702 χρησιμοποιεί το ίδιο ολοκληρωμένο με το ZedBoard ωστόσο περιέχει περιφερειακό μικροελεγκτή καταγραφής ισχύος

Η πλακέτα zc702 χρησιμοποιεί το ίδιο ολοκληρωμένο εττερογενούς αρχιτεκτονικής με την πλακέτα Zedboard, συνεπώς μπορούμε να υποθέσουμε οτι η κατανάλωση ενέργειας στις δυο πλακέτες είναι παρόμοια. Επίσης εκτελέσαμε μια μόνο επανάληψη σε software και μια σε hardware. Στο παρακάτω διάγραμμα παρουσιάζονται τα αποτελέσματα για μια εκτέλεση του movielens100k.

Στο διάγραμμα παρουσιάζεται η στιγμιαία ισχύς που καταναλώνεται
καθώς εκτελούνται οι διάφοροι υπολογισμοί. Η κατανάλωση ισχύος και στις δυο περιπτώσεις αυξάνεται κατακόρυφα καθώς ξεκινούν οι υπολογισμοί (κοντά στην χρονική στιγμή 0) και μειώνεται σχεδόν κατακόρυφα μετά το πέρας των υπολογισμών στις χρονικές στιγμές 5 για την επιταχυμένη εκτέλεση(πράσινο) και 50 για την μη επιταχυμένη έκδοση.
Παρατηρούμε οτι παρά το γεγονος οτι η επιταχυμένη έκδοση έχει στιγμιαία υψηλότερη



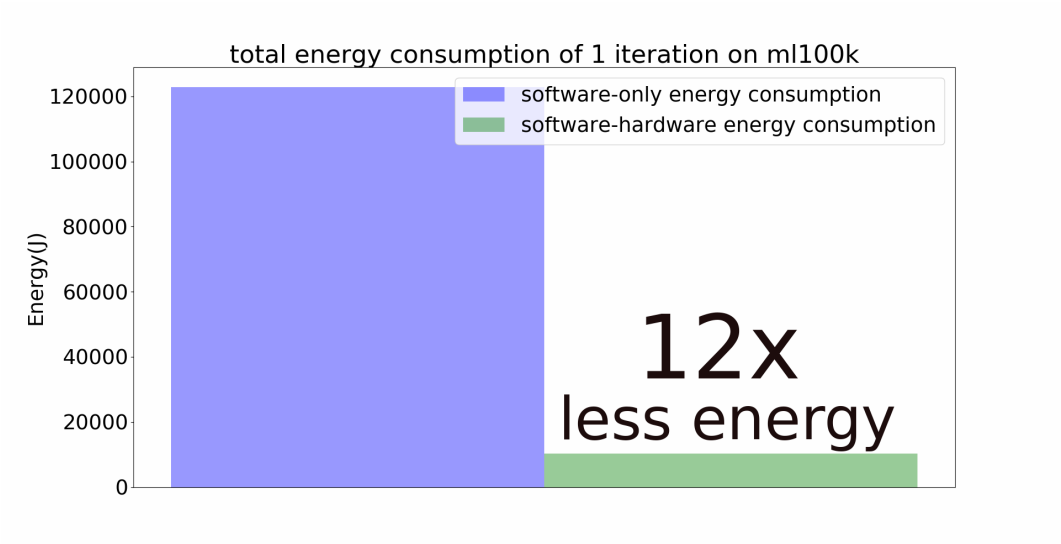
Σχήμα 4.14: Στιγμιαία κατανάλωση ισχύος κατά την εκτέλεση μιας επανάληψης του αλγορίθμου με είσοδο το dataset movielens 100k. Με πράσινο χρώμα φαίνεται η κατανάλωση της επιταχυμένης έκδοσης ενώ με κόκκινο η κατανάλωση της εκτέλεσης πάνω στην arm-CPU

κατανάλωση επιτυγχάνει συνολικά μικρότερη κατανάλωση ενέργειας.Το γεγονός αυτό σε συνδυασμό με το γεγονός ότι και οι 2 εκδόσεις εκτελούν το ίδιο πλήθος υπολογισμών υποδηλώνει ότι με την χρήση FPGA πετυχαίνουμε σημαντική βελτίωση της μετρικής "επίδοση ανα watt" (Peformance per Watt). Με ολοκλήρωση των καμπυλών του παραπάνω διαγράμματος μπορούμε να υπολογίσουμε στην συνολική ενέργεια που καταναλώθηκε στις δυο αυτές εκτελέσεις, η οποία παρουσιάζεται στο παρακάτω διάγραμμα.

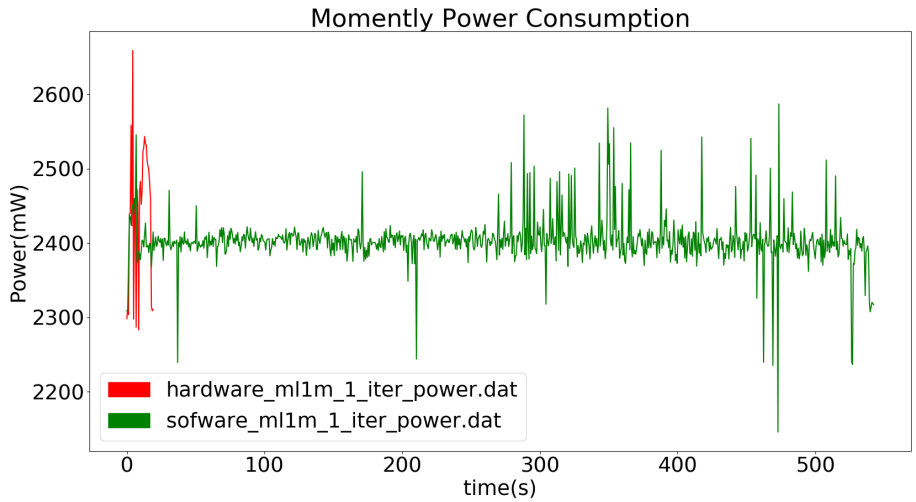
Η μη επιταχυμένη έκδοση κατανάλωσε συνολική ενέργεια περίπου ίση με 122774J ενώ η επιταχυμένη έκδοση κατανάλωσε συνολική ενέργεια 10291J, πράγμα που σημαίνει ότι η επιταχυμένη έκδοση εκτέλεσε σου ίδιους υπολογισμούς χρησιμοποιώντας 12 φορές λιγότερη ενέργεια σε σχέση με την απλή arm CPU.

Και στο dataset movielens_1m μπορούμε να κάνουμε τις ίδιες παρατηρήσεις την αύξηση δηλαδή της μετρικής "επίδοση ανά Watt". Η συνολική ενέργεια που καταναλώθηκε για την εκτέλεση μιας επανάληψης σε αυτό το dataset φαίνεται στο παρακάτω διάγραμμα.

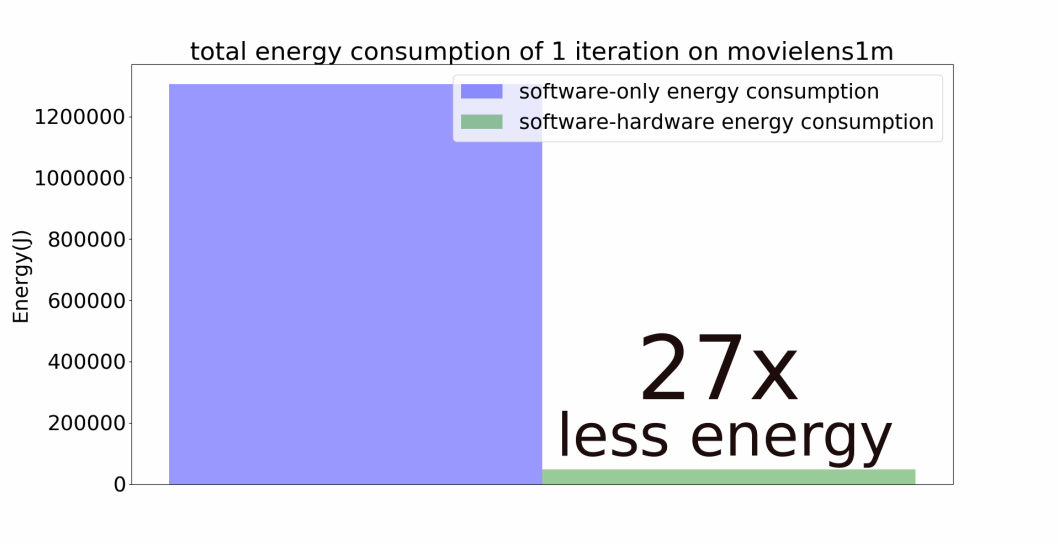
Η μη επιταχυμένη έκδοση κατανάλωσε συνολική ενέργεια περίπου ίση με 1304843J ενώ η μη επιταχυμένη έκδοση κατανάλωσε συνολική ενέργεια 48000J, πράγμα που σημαίνει ότι η επιταχυμένη έκδοση εκτέλεσε σου ίδιους υπολογισμούς χρησιμοποιώντας 27 φορές λιγότερη ενέργεια σε σχέση με την απλή arm CPU.



Σχήμα 4.15: Η συνολική ενέργεια που καταναλώθηκε κατα την εκτέλεση μιας επανάληψης του αλγορίθμου με είσοδο το dataset movielens100k, όπως αυτή προέκυψε από την ολοκλήρωση του διαγράμματος της στιγμιαίας ισχύος



Σχήμα 4.16: Στιγμιαία κατανάλωση ισχύος κατά την εκτέλεση μιας επανάληψης του αλγορίθμου με είσοδο το dataset movielens_1m. Με κόκκινο χρώμα φαίνεται η κατανάλωση της επιταχυμένης έκδοσης ενώ με πράσινο η κατανάλωση της εκτέλεσης πάνω στην arm-CPU



Σχήμα 4.17: Η συνολική ενέργεια που καταναλώθηκε κατα την εκτέλεση μιας επανάληψης του αλγορίθμου με είσοδο το dataset movielens1m, όπως αυτή προέκυψε από την ολοκλήρωση του διαγράμματος της στιγμιαίας ισχύος

4.11 Κανονικοποίηση

Στις διαδοχικές υλοποιήσεις πυρήνα που παρουσιάσαμε παραπάνω σημειώνουμε ότι η κανονικοποίηση όπως την αναλύσαμε στο δεύτερο κεφάλαιο, και όπως φαίνεται στις παρακάτω εξισώσεις, έγινε εξολοκλήρου στο λογισμικό. Συγκεκριμένα, την στιγμή που ο επιταχυντής υλικού επέστρεφε τα αποτελέσματα των υπολογισμών που έγιναν πάνω στο υλικό, το λογισμικό πρόσθετε στην διαγώνιο του πίνακα A το πλήθος των βαθμολογιών του αντίστοιχου χρήστη ή της αντίστοιχης ταινίας. Στην δική μας υλοποίηση οπου $NFEATS = 80 \Rightarrow A_(80 \times 80)$ η κανονικοποίηση ισοδυναμεί με ογδόντα προσθέσεις. Πειραματικά δοκιμάσαμε να συγκρίνουμε αν έχουμε ώφελος να συμπεριλάβουμε την κανονικοποίηση στον πυρήνα. Οι λόγοι που μια τέτοια ενέργεια μπορεί να μην είναι συμφέρουσα είναι ότι χρειαζόμαστε ένα μηχανισμό,π.χ. εναν πολυπλέκτη, ο οποίος θα κρίνει αν ένα στοιχείο είναι στην διαγώνιο η όχι, πράγμα που ενδέχεται να μειώσει τον ρυθμό εκτέλεσης των πολλαπλασιασμών και να αυξήσει την κατανάλωση ενέργειας.

$$A_i = M_{I_i} M_{I_i}^T + \lambda n_{u_i} E A_j = U_{I_j} U_{I_j}^T + \lambda n_{m_j} E$$

Επειδή έχουμε ουσιαστικά μία υλοποίηση κινούμενου παραθύρου, θα πρέπει να είμαστε σίγουροι ότι η κανονικοποίηση θα γίνει μετά την μεταφορά του τελευταίου παραθύρου δεδομένων, δηλαδή θα προστεθούν στην διαγώνιο οι τιμές μόνο στο τελικό αποτέλεσμα και όχι σε κάθε ενδιάμεση επανάληψη. Για την επίτευξη του παραπάνω χρησιμοποιούμε την μεταβλητή αληθείας $regularize$ η οποία γίνεται αληθής μόνο όταν ο επιταχυντής πρόκειται να εκτελέσει την τελευταία επανάληψη.

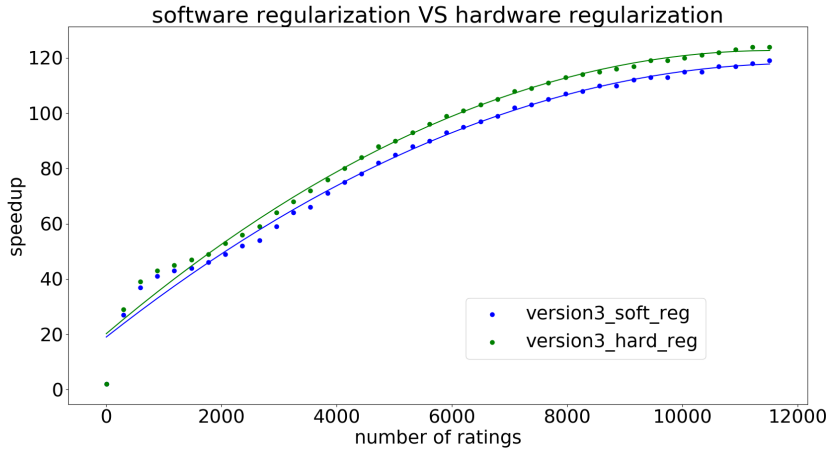
```
for(i=0;i<iterations;i++){
    #pragma HLS loop_tripcount max = NMAXRAT/BUFFER_SIZE/2
    regularize = (i==iterations-1)?1:0;
```

```
    readSData(Sbuffer,_buffer1,_buffer2,_buffer3,streampos,nratings);
    readRData(Rbuffer,_rbuff,streampos,nratings);
    mul_hw1(regularize,lamda,nratings,_buffer1,_buffer2,_A);
    mul_hw2(_buffer3,_rbuff,_V);
    streampos += BUFFER_SIZE;
  }
```

Το μ΄ονο που χρειάζεται να προσθέσουμε στο υπολογιστικό κομμάτι του κώδικα είναι ένας επιλογέας, που ενεργοποιείται μόνο στην τελευταία επανάληψη ($regularize == True$) και μόνο αν επεξεργαζόμαστε στοιχείο της διαγωνίου ($i == j$).

```
_A1: for(i=0;i<NFEATS;i++){
_A2: for(j=0;j<NFEATS;j++){
  #pragma HLS PIPELINE
if(i<=j){
      float result2 = _A[i][j];
          _A3: for(r=0;r<BUFFER_SIZE;r++){
        result2 += _buffer2[i][r]*_buffer1[r][j];
          }
      _A[i][j] = (regularize && i==j)?result2+lamda*nratings:result2;
    }
    }
    }
```

Ο τρόπος που έχουν προστεθεί οι δομές επιλογής μας εξασφαλίζει ότι δεν θα υπάρξη καθυστέρηση στην εκτέλεση μέχρι να επιλυθεί η συνθήκη διότι τα εργαλεία και το υλικό της xilinx υποστηρίζουν if-conversion.



Σχήμα 4.18: Παρατηρούμε ότι έχουμε χρονικό ώφελος να προσθέσουμε ακόμα και αυτόν το μικρό πλήθος πράξεων που αποτελούν την κανονικοποίηση, στον επιταχυντή.

4.12 Συμπερασματικές παρατηρήσεις

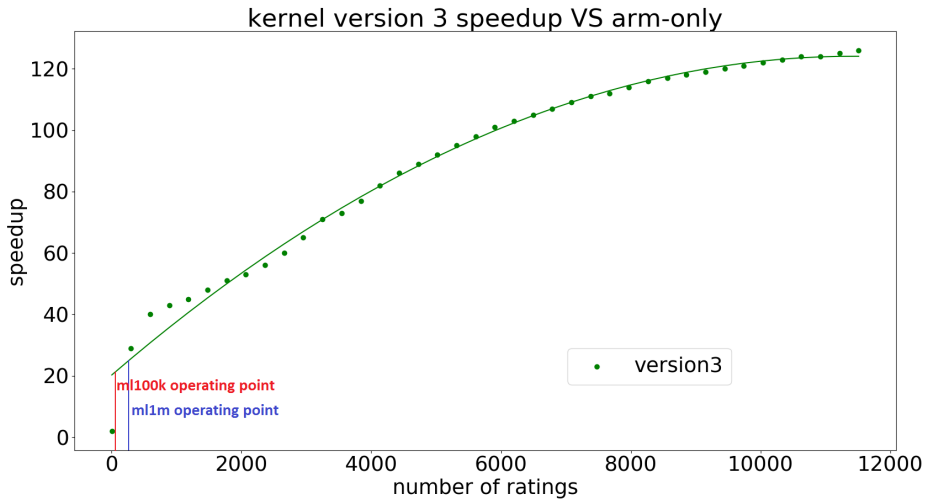
Στα δυο dataset που χρησιμοποιήσαμε για να πάρουμε τις τελευταίες μετρήσεις είναι σημαντικό να τονιστεί δεν παρατηρούμε τις πλήρεις δυνατότητες της επιταχυμένης έκδοσης. Συγκεντρωτικά είδαμε τα παρακάτω

Σύνολο Δεδομένων	Χρονική Βελτίωση	Ενεργειακή Εξοικονόμηση
Movielens100k	x19	x12
Movielens1m	x36	x27

Ωστόσο στα παραπάνω dataset ο μέσος αριθμός βαθμολογιών ανα ταινία/χρήστη είναι

Σύνολο Δεδομένων	Μέσος όρος βαθμολογιών ανά ταινία/χρηστη
Movielens100k	76.2
Movielens1m	205.2

πράγμα που σημαίνει οτι το σημείο λειτουργίας του πυρήνα είναι περίπου αυτό που φαίνεται παρακάτω:



Σχήμα 4.19: Το σημείο λειτουργίας του πυρήνα για τα dataset ml100k και ml1m. Είναι εμφανές ότι ο πυρήνας μπορεί να πετύχει πολύ καλύτερες επιδόσεις έναντι του απλού software για μεγαλύτερα datasets.

5 Ενσωμάτωση με Python

5.1 Εισαγωγή

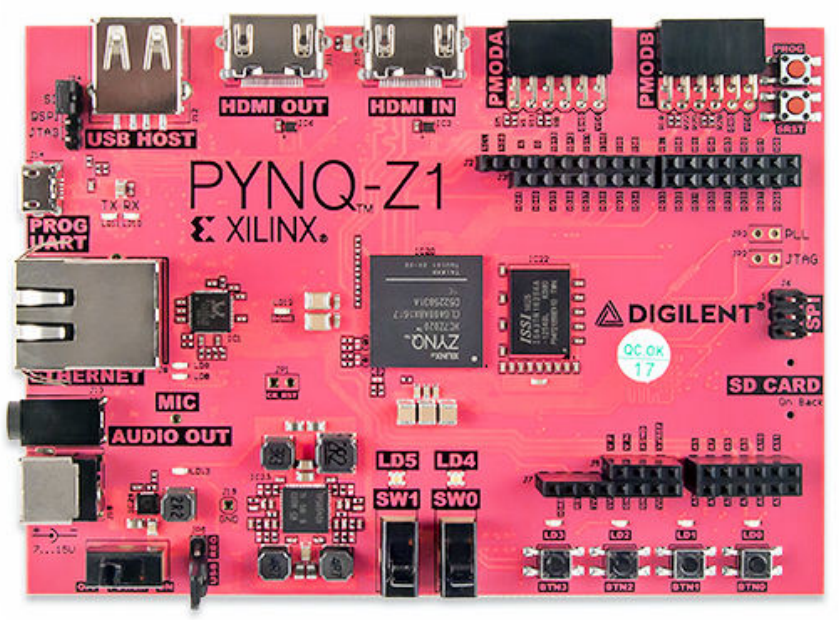
Το επόμενο βήμα μετά την επιτυχή επιτάχυνση του υπολογιστικά έντονου μέρος του αλγορίθμου Alternating Least Squares στο περιβάλλον του SDSoC με χρήση C, είναι η ενσωμάτωση του σε κώδικα Python. Οι περισσότερες εφαρμογές που αφορούν τεχνητή νοημοσύνη σήμερα είναι γραμμένες σε γλώσσες υψηλότερου επιπέδου απο την C , όπως η Python και η Java, διότι παρέχουν πολύ ισχυρότερες εντολές και βιβλιοθήκες σε σχέση με μια γλώσσα που είναι πολύ κοντά στον υλικό όπως η C. Εδώ λοιπόν δημιουργείται ένα μεγάλο ερώτημα, πώς θα γίνει η ενσωμάτωση ενός υλικού επιταχυντή, του οποίου η διεπαφή απαιτεί γνώσεις των φυσικών διευθύνσεων των DMAs αλλά και πολλές άλλες λεπτομέρειες σχετικά με το υλικό πάνω στο οποίο αυτός τρέχει, σε μια γλώσσα που φαινομενικά είναι απομακρυσμένη από τον υλικό πάνω στο οποίο τρέχει, όπως η Python. Μέχρι τώρα τις λεπτομέρειες αυτές τις απέκρυπτε το περιβάλλον SDSoC, στο οποίο ορίζοντας την πλακέτα πάνω στην οποία θα εργαζόμασταν αναλάμβανε να κατασκευάσει διεπαφές υλικού και λογισμικού με βάση τον χώρο διευθύνσεων που έχει δωθεί σε αυτές από τον κατασκευαστή (Xilinx) πάνω στην συγκεκριμένη πλακέτα, ωστόσο το SDSoC είναι ένα περιβάλλον κατασκευασμένο να παρέχει τέτοια επίπεδα αφαιρετικότητας μόνο για συγγραφή κώδικα σε C ή C++.

Μπορεί λοιπόν 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 Z1 είναι κατάλληλη για επιτάχυνση εφαρμογών σε Python διότι παρέχεται για αυτήν ένα image ανοιχτού κώδικα,στο οποίο περιέχει ένα API χαμηλού επιπέδου, μέσω του οποίου ο κώδικας Python που προορίζεται να τρέξει στον arm CPU μπορεί να αλληλεπιδράσει με τον πυρήνα fpga που ζει στο υλικό. Το Pynq δεν υπάρχει σαν Project για περισσότερο απο ένα χρόνο κατά την διάρκεια συγγραφής αυτής της διπλωματικής(2017) και συνεπώς χρειάστηκαν να γίνουν αλλαγές στο API που παρέχεται στο GitHub ώστε να ταιριάξει στις ανάγκες μας. Είναι σημαντικό να

τονίσουμε ότι παρόλο που η Xilinx παρέχει το API που προαναφέραμε, αυτό βρίσκεται σε αρκετά πρώιμο στάδιο συνεπώς η ενσωμάτωση του πυρήνα στην python χρειάστικε να γίνει κάνοντας reverse engineering στο SDSoC ώστε να ανακαλύψουμε πώς επικοινωνεί η C με τον πυρήνα και να κάνουμε τις αντίστοιχες ενέργειες κάνοντας χρήση του Python API πάνω στο Pynq.

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

- συγγραφή σε Python τα κομμάτια του αλγορίθμου που προορίζονται για τρέξιμο σε CPU.
- απομόνωση του IP του πυρήνα απο το περιβάλλον του SDSoC και δημιουργία νέου bitstream συμβατού με το PYNQ-Z1
- Δημιουργία οδηγών python, που χρησιμοποιούν το xilinx API, και αλληλεπιδρούν με τον πυρήνα, και ενσωμάτωση τους στον κώδικα που γράψαμε στο πρώτο βήμα.



Σχήμα 5.1: Η πλακέτα Pynq Z1 χρησιμοποιεί το ίδιο ολοκληρωμένο με το ZedBoard

5.2 Xilinx's Python API στο Pynq - Z1

Μέσα στο image που εγκαταστήσαμε στο Pynq υπάρχουν όλα τα αρχεία που συνθέτουν το API μέσω του οποίου θα αλληλεπιδράσουμε με τον πυρήνα μας. Τα σημαντικότερα απο αυτά είναι:

- libdma.so

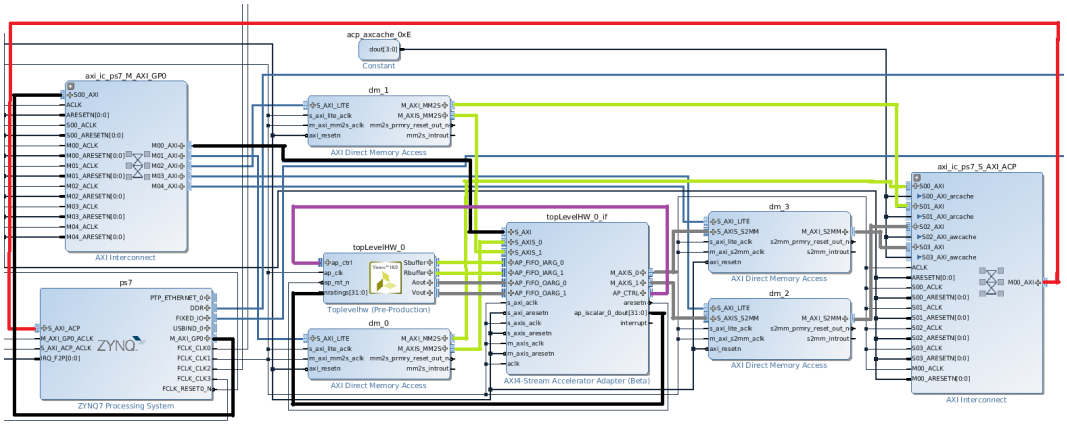
- libsds_lib.so
- xlnk.py
- dma.py
- mmio.py
- pl.py

Τα αρχεία libdma.so και libsds_lib.so αποτελούν precompiled βιβλιοθήκες γραμμένες σε C. Το libdma.so είναι ένα low level C API το οποίο περιέχει precompiled βιβλιοθήκες για τον χειρισμό των DMAs που έχουμε κατασκευάσει στον πυρήνα μας ενώ το libsds_lib.so περιέχει γενικότερου σκοπού compiled συναρτήσεις όπως είναι για παράδειγμα συναρτήσεις δέσμευσης/αποδέσμευσης συνεχόμενης φυσικής μνήμης (cma_alloc) κλπ. Το xlnk.py αποτελεί ουσιαστικά έναν Python wraper του libsds_lib.so, αντικαθιστά δηλαδή τις κλήσεις των C συναρτήσεων που βρίσκονται εντός του libsds_lib.so με κλήσεις Python συναρτήσεων, πράγμα που επιτυγχάνει μέσω της Python βιβλιοθήκης cffi η οποία επιτρέπει στην Python να καλέσει ή να εκτελέσει οποιοδήτοτε κώδικα σε C είτε αυτός είναι σε source μορφή είτε σε είναι σε binary. Το dma.py αποτελεί έναν wraper για το libdma.so, ουσιαστικά περιέχει ένα Python API για την επικοινωνία του Python κώδικα με τους DMAs πάνω στο FPGA διαμέσω ενός DMA Python Object του οποίου οι μέθοδοι καλούν τις συναρτήσεις που περιέχονται στα .so αρχεία με χρήση της βιβλιοθήκης cffi. Το αντικείμενο DMA περιέχει όλες τις απαραίτητες μεθόδους για την κατασκευή,καταστροφή και μεταφορά δεδομένων διαμέσω DMAs υλικού. Το mmio.py περιέχει Python συναρτησεις που κάνουν Wrap τις C συναρτήσεις του libsds_lib.so τις οποίες αφορούν memory mapped I/O. Υπενθυμίζουμε οτι η CPU του Zynq βλέπει το FPGA σε ένα Memory Maped χώρο διευθύνσεων συνεπώς είναι απαραίτητες οι αντίστοιχες βιβλιοθήκες για τέτοιου είδους επικοινωνία. Τέλος το αρχείο pl.py είναι διαθέτει τις απαραίτητες συναρτήσεις για την επικοινωνία της CPU με το device /dev/xdevcfg , το οποίο ουσιαστικά αποτελεί το FPGA, όπως π.χ. την εγγραφη του Bitstream.

5.3 Απομόνωση του πυρήνα

Όπως αναφέραμε το πρώτο βήμα της ενσωμάτωσης είναι να απομονώσουμε τον πυρήνα απο το SDSoC project. Επειδή το SDSoC καλεί το vivado για την δημιουργία του IP, αρκεί να βρούμε το αντίστοιχο vivado project που περιέχεται μέσα στο SDSoC project. Μέσα στο Vivado Project περιέχεται ο πυρήνας που κατασκευάσαμε, ωστόσο ο συγκεκριμένος έχει δημιουργηθεί για την πλακέτα Zedboard, συνεπώς το μόνο που μένει από εμάς 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.

Στο tcl αρχείο που παράγεται απο το vivado μπορούμε να δούμε τις ακριβείς φυσικές διευθύνσεις στις οποίες έχουν χαρτογραφηθεί οι DMAs και ο FIFO ACCELERATOR ADAPTER στην ακόλουθη μορφή.



Σχήμα 5.2: Το block design που προέκυψε στο vivado μετά την μετατροπή. Παρατηρούμε την γραμμή ACP(κόκκινο) μέσω της οποίας μεταφέρονται οι πίνακες σε ένα AXI INTERCONNECT, τις εισόδους του πυρήνα με πράσινο χρώμα οι πίνακες, οι γραμμές δηλαδή των DMA εισόδου και με **μαύρο** η βαθμωτή είσοδος (nratings), ενώ με γκρί χρώμα φαίνονται οι γραμμές των εξόδων. Τέλος με μώβ χρώμα παρουσιάζονται οι γραμμές των σημάτων ελέγχου του πυρήνα.

- 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 0x40410000 [get_bd_addr_spaces ps7/Data] [get_bd_addr_segs dm_1/S_AXI_LITE/Reg] SEG_dm_1_Reg
- create_bd_addr_seg -range 0x00010000 -offset 0x40420000 [get_bd_addr_spaces ps7/Data] [get_bd_addr_segs dm_2/S_AXI_LITE/Reg] SEG_dm_2_Reg
- create_bd_addr_seg -range 0x00010000 -offset 0x40430000 [get_bd_addr_spaces ps7/Data] [get_bd_addr_segs dm_3/S_AXI_LITE/Reg] SEG_dm_3_Reg
- create_bd_addr_seg -range 0x00010000 -offset 0x43C00000 [get_bd_addr_spaces ps7/Data] [get_bd_addr_segs topLevelHW_0_if/S_AXI/Reg] SEG_topLevelHW_0_if_Reg

Τα τέσσερα πρώτα στοιχεία της λίστας αφορούν τους DMAs, με συμβολικά ονόματα dm_0,dm_1,dm_2,dm_3 αντίστοιχα, ενώ το τελευταίο αφορά τον FIFO ACCELERATOR ADAPTER. Συνεπώς σε αυτό το στάδιο έχουμε καταφέρει να απομονώσουμε σε δυο αρχεία,ALS.bit και ALS.tcl,τον πυρήνα και όλες τις απαραίτητες πληροφορίες για αυτόν αντίστοιχα.

5.4 Συγγραφή των οδηγών του υλικού

Μετά την απομόνωση του πυρήνα στα .bit και .tcl αρχεία, δημιουργήσαμε τους αντίστοιχους οδηγούς του , κάνοντας χρήση του Python API που προαναφέραμε, με στόχο

την επικοινωνία υλικού και λογισμικού. Ο πρώτος οδηγός (createHardwareInterface) είναι υπεύθυνος για το αρχικές ρυθμίσεις και όπως το κατέβασμα του bitstream στο FPGA και, και την δημιουργία των απαραίτητων αντικειμένων.

```
def createHardwareInterface():
        ol = Overlay("ALS2_0.bit")
        ol.download()
```

Στις πρώτες δύο γραμμές κατασκευάζουμε ένα αντικείμενο τύπου Overlay στου οποίου τον constructor δίνουμε σαν όρισμα το αρχείο του bitstream, και στην συνέχεια καλούμε την μέθοδο download η οποία γράφει στο device xdevcfg (το οποίο είναι το FPGA) το bitstream. Σε αυτό το σημείο το API αναζητά ομώνυμο αρχείο με κατάληξη .tcl, από το οποίο αντλεί πληροφορίες που θα μας φανούν χρήσιμες αργότερα.

```
        DMA_TO_DEV   = 0
        DMA_FROM_DEV = 1
```

Στην συνέχεια δηλώνουμε δυο σταθερές κατεύθυνσης, σε όλο το API το 0 αντιστοιχεί σε μεταφορά δεδομένων από το FPGA προς την CPU ενώ το 1 σε μεταφορά δεδομένων απο την CPU στο FPGA.

```
    #DMA OBJECT CREATION
        #DMA0 ----> TRANSFERS SBUFFER
        #DMA1 ----> TRANSFERS RBUFFER
        #DMA2 ----> TRANSFERS AOUT
        #DMA3 ----> TRANSFERS VOUT

        #-------PHYSICAL ADD OF THE DMAs-------------------------
        ADDR_DMA0_BASE = int(PL.ip_dict["SEG_dm_0_Reg"][0],16)
        ADDR_DMA1_BASE = int(PL.ip_dict["SEG_dm_1_Reg"][0],16)
        ADDR_DMA2_BASE = int(PL.ip_dict["SEG_dm_2_Reg"][0],16)
        ADDR_DMA3_BASE = int(PL.ip_dict["SEG_dm_3_Reg"][0],16)
        ADDR_MMIO      = int(PL.ip_dict["SEG_topLevelHW_0_if_Reg"][0],16)
        RANGE_MMIO     = int(PL.ip_dict["SEG_topLevelHW_0_if_Reg"][1],16)
```

Οποιαδήποτε διεπαφή CPU - FPGA είναι memory maped πάνω σε ένα εύρος φυσικών διευθύνσεων, κάθε εύρος φυσικών διευθύνσεων που αντιστοιχεί σε μια διεπαφή περιγράφεται στο TCL αρχείο από την φυσική διεύθυνση που ξεκινά (offset) και από το εύρος (range) των διευθύνσεων. Για παράδειγμα ένας DMA μπορεί να αντιστοιχεί στην διεύθυνση 0x60007000 και να έχει offset 64kb. Παραπάνω αποθηκεύουμε στις μεταβλητές
ADDR_DMA0,1,2,3_BASE το offset των τριών DMAs, ενώ στις μεταβλητές ADDR_MMIO και RANGE_MMIO αποθηκεύουμε το offset και το range αντίστοιχα του Memory Maped I/O που χρησιμοποιούμε για την αποστολή της βαθμωτής τιμής (nratings) από την CPU στο FPGA. Όλες οι πληροφορίες διαβάζονται από το TCL αρχείο. Από το PL αρχείο του API χρησιμοποιούμε το λεξικό ip_dict και συγκεκριμένα την τιμή με κλειδί το συμβολικό όνομα του της διεπαφής π.χ. SEG_dm_0_Reg ,η οποία είναι ένας πίνακας με πρώτο στοιχείο το offset και δεύτερο το range της συγκεκριμένης διεπαφής.

```
        #-------CREATE DMA OBJECTS-------------------------------
        # direction = 0 send to PL ,  direction = 1 receive from PL

        SbufferDMA = DMA(ADDR_DMA0_BASE, direction = DMA_TO_DEV)
        RbufferDMA = DMA(ADDR_DMA1_BASE, direction = DMA_TO_DEV)
```

```
6         VoutDMA     = DMA(ADDR_DMA2_BASE, direction = DMA_FROM_DEV)
7         AoutDMA     = DMA(ADDR_DMA3_BASE, direction = DMA_FROM_DEV)
8         mmio        = MMIO(ADDR_MMIO, RANGE_MMIO)
```

Στην συνέχεια δημιουργούμε τα αντικείμενα τύπου DMA τα οποία θα χρησιμοποιήσουμε για την επικοινωνια με τους DMAs, των οποίων ο κατασκευαστής έχει σαν ορίσματα το offset και την κατεύθυνση αποστολής. Είναι σημαντικό να σημειωθεί οτι είναι στο χέρι του προγραμματιστή να μην ξεπεράσει το range καθενός interface, αν κάτι τέτοιο συμβεί τότε αναλαμβάνει το λειτουργικό και τερματίζει την εφαρμογή με σήμα παραβίασης μνήμης SIGSEV. Επίσης σημαντικό είναι να τονίσουμε οτι τα παραπάνω αντικείμενα δεν είναι τίποτα άλλο πέρα απο προγραμματιστικές λογικές οντότητες που κάνουν ευκολότερη την διεπαφή μας με τους πραγματικούς DMA που βρίσκονται πάνω στο υλικό του FPGA.

```
1         #----------BUFFER CREATION----------
2
3         SbufferDMA.create_buf(4*g.NMAXRAT*g.NFEATS,1)
4         RbufferDMA.create_buf(4*g.NMAXRAT,1)
5         AoutDMA   .create_buf(4*g.NFEATS*g.NFEATS,1)
6         VoutDMA   .create_buf(4*g.NFEATS,1)
7
8         #------------------------------------------------
9         # ----------get DMAs Buffer Address
10        Sbuffer = SbufferDMA.get_buf(data_type = 'float')
11        Rbuffer = RbufferDMA.get_buf(data_type = 'float')
12        Aout    = AoutDMA   .get_buf(data_type = 'float')
13        Vout    = VoutDMA   .get_buf(data_type = 'float')
14
15        dmas          = {'SbufferDMA':SbufferDMA ,'RbufferDMA':RbufferDMA ,
    'AoutDMA': AoutDMA, 'VoutDMA': VoutDMA , 'MMIO': mmio }
16        dmaBuffers  = {'Sbuffer'   :Sbuffer     ,'Rbuffer'  :Rbuffer     ,'
    Aout'    :Aout     ,'Vout'    :Vout}
17
18        return dmas
```

Έπειτα για κάθε DMA αντικείμενο δημιουργούμε αντίστοιχου μεγέθους φυσικούς buffers. Υπενθυμίζουμε ότι χρησιμοποιούμε απλούς DMAs (simple DMAs) και όχι Scatter Gather συνεπώς χρειαζόμαστε οι δεδομένα αποθηκευμένα σε συνεχόμενες θέσεις μνήμης, αυτό σημαίνει ότι δεν οι DMAs δεν μπορούν να αντλήσουν τα δεδομένα απο κάποιο native Python datatype όπως οι λίστες αλλα ούτε και απο κάποιο datatype βιβλιοθήκης όπως τα numpy arrays , αφού σε καμία απο τις δυο περιπτώσεις δεν έχουμε έλεγχο του τρόπου αποθήκευσης των δεδομένων, ωστόσο θα χειριστούμε τα δεδομένα με την βιβλιοθήκη Numpy κάνοντας τις αντίστοιχες μετατροπές αργότερα.Τέλος συνάρτηση επιστρέφει ένα λεξικό με τα DMA αντικείμενα.

```
1 def topLevelHW(nratings,dmas):
2         SbufferDMA = dmas["SbufferDMA"]
3         RbufferDMA = dmas["RbufferDMA"]
4         AoutDMA    = dmas["AoutDMA"]
5         VoutDMA    = dmas["VoutDMA"]
6         mmio       = dmas["MMIO"]
```

Σε αυτήν την συνάρτηση-οδηγό κάνουμε όλη την διάδραση με τον πυρήνα. Αρχικά τραβάμε από το λεξικό που κατασκευάσαμε στην συνάρτηση createHardwareInterface

τα DMA αντικείμενα που περιέχουν τους πίνακες συνεχόμενης φυσικής μνήμης, τους οποίους γεμίζουμε στο software με δεδομένα.

```
        DMA_TO_DEV    = 0
        DMA_FROM_DEV = 1

        CMD_REG              = 0x0028
        CTR_REG              = 0x0000
        ISCALAR0_DATA        = 0x0080
        ISCALAR0_STATUS      = 0x0180
        IARG0_STATUS         = 0x0100
        IARG1_STATUS         = 0x0104
        OARG0_STATUS         = 0x0140
        OARG1_STATUS         = 0x0144
```

Διαβάζοντας το manual του FIFO Accelerator Adapter IP, βρήκαμε σε τις αντίστοιχες διευθύνσεις που χρειαζόμαστε για να γράψουμε στους καταχωρητές εισόδου-εξόδου του, να ελέγξουμε την κατάσταση των θυρών του, για να τον προγραμματίσουμε γενικότερα. Οι κυριότεροι καταχωρητές είναι οι

- CMD_REG (Command Register) : Ο καταχωρητής στον οποίο στέλνουμε εντολές για την λειτουργία του FIFO ACC ADAPTER, όπως έναρξη εκτέλεσης (execute), ανανέωση τιμών εξόδου, ανανέωση τιμών εισόδου κλπ, οι δυο τελευταίες εντολές χρειάζονται διότι το συγκεκριμένο IP περιέχει σειριακούς multibuffers τους οποίους έχει κανείς την δυνατότητα να γεμίσει με ένα data transfer overhead και στην συνέχεια με τις κατάλληλες εντολές, μέσω του CMD_REG να τους ταίσει στον πυρήνα είτε με διοχέτευση είτε ακολουθιακά, είτε να τον ξαναταΐσει με τα ίδια δεδομένα χωρίς να χρειαστεί να τα στείλουμε απο την CPU. Αν δηλαδή θέλουμε σε κάποια επανάληψη να ταίσουμε τον πυρήνα με τα ίδια δεδομένα, αντί να τα ξαναστείλουμε μπορούμε να μην κάνουμε ανανέωση της εξόδου του FIFO accelerator adapter και στην συνέχεια να στείλουμε εντολή execute.

- ISCALARX_DATA : Ο καταχωρητής στον οποίο αποστέλουμε τα βαθμωτά δεδομένα, υπάρχουν επτά τέτοιοι καταχωρητές συνολικά (X=0..6), ωστόσο εμείς θέλοντας να αποστείλουμε μόνο ένα βαθμωτό δεδομένο (nratings) χρησιμοποιούμε μόνο τον ISCALAR0_DATA.

- ISCALARX_STATUS : έλεγχος της κατάστασης του του αντίστοιχου βαθμωτού καταχωρητή , χρήσιμο για debuging.

- IARGX_STATUS,OARGX_STATUS : καταχωρητές ελέγχου κατάστασης των μή βαθμωτών εισόδων (πίνακες),επίσης χρήσιμοι για debuging.

```
    #transfer scalar
        mmio.write(ISCALAR0_DATA,int(nratings))
        #update output
        mmio.write(CMD_REG,0x00010003)
        #send command execute CMD_REG
        mmio.write(CMD_REG,0x00020000)
        #update input
        mmio.write(CMD_REG,0x103)
```

Στο παραπάνω κομμάτι κώδικα μεταφέρουμε την βαθμωτή είσοδο, στέλνουμε εντολή ανανέωσης εξόδου διότι επιθυμούμε να σταλούν στον πυρήνα οι νέες τιμές και όχι οι παλαιές, και στο τέλος δίνουμε εντολή έναρξης εκτέλεσης. Το ότι δίνουμε εντολή έναρξης εκτέλεσης χωρίς να έχουμε στείλει ακόμα τα δεδομένα των πινάκων δεν είναι πρόβλημα , διότι ο πυρήνας θα διαβάσει την βαθμωτή τιμή και στην συνέχεια αφού το πρωτόκολλο με το οποίο επικοινωνεί με τον FIFO Accelerator Adapter είναι τύπου FIFO Streaming θα κολλήσει περιμένοντας να διαβάσει την πρώτη τιμή. Με αυτόν τον τρόπο είμαστε επίσης σίγουροι ότι η βαθμωτή τιμή από την οποία εξαρτάται το πλήθος των επαναλήψεων μέσα στον πυρήν αλλά και το πόσα στοιχεία θα ζητήσει να διαβάσει, θα αναγνωσθεί από τον πυρήνα πριν από τα υπόλοιπα δεδομένα.

```
        #transfer(NUM_OF_BYTES_TO_TRANSFER, DIRECTION)
        SbufferDMA.transfer(4*nratings*g.NFEATS,DMA_TO_DEV)
        RbufferDMA.transfer(4*nratings          ,DMA_TO_DEV)

        SbufferDMA.wait()
        RbufferDMA.wait()

        AoutDMA.transfer(4*g.NFEATS*g.NFEATS,DMA_FROM_DEV)
        VoutDMA.transfer(4*g.NFEATS          ,DMA_FROM_DEV)

        VoutDMA.wait()
        AoutDMA.wait()
```

Το μόνο που απομένει είναι να κάνουμε χρήση της συνάρτησης "transfer(size in bytes)" των DMA αντικειμένων, η οποία αποστέλει δεδομένα από τους buffers που κάναμε allocate στην "createHardwareInterface" μεγέθους όσα τα bytes που της δίνουμε σαν όρισμα. Υπενθυμίζουμε ότι οι buffers αυτοί έχουν γεμίσει με δεδομένα από το software κομμάτι της υλοποίησής μας όπως θα φανεί και στην συνέχεια. Τέλος είναι σημαντικό να τονίσουμε ότι άν μεταφέρουμε λιγότερα δεδομένα με τους DMAs τότε ο πυρήνας θα κολλήσει αναμένοντας ανάγνωση δεδομένων από την FIFO διεπαφή του Adapter. Αφού ολοκληρωθεί η μεταφορά των δεδομένων από την CPU προς το FPGA, γίνεται η αντίστροφη διαδικασία , συλλέγονται δηλαδή τα αποτελέσματα από το FPGA πίσω στην CPU κάνοντας πάλι χρήση της συνάρτησης "transfer" των DMA αντικειμένων αλλά αυτήν την φορά με αντίθετη κατεύθυνση, φυσικά τα αποτελέσματα συλλέγονται στους buffers των αντίστοιχων DMAs.

5.5 Συγγραφή του υπόλοιπου Software

Παρακάτω παραθέτουμε περιληπτικά το κομμάτι του software όπως αυτό γράφτηκε σε Python.

```
def updateU(U,M,lamda,numUsers,userInf,dmas):
        ffi = cffi.FFI()
        printProgressBar(0, numUsers, prefix = 'UPDATING USERS:', suffix
    = 'Complete', length = 50)
```

Αρχικά δημιουργούμε ένα αντικείμενο τύπου cffi για να έχουμε πρόσβαση στα κομμάτια που API που έχουν γραφτεί σε C.

```
        for i in range(numUsers):
```

```
                printProgressBar(i + 1, numUsers, prefix = ' UPDATING
    USERS:', suffix = 'Complete', length = 50)
                sel           = userInf[i]['rId']
                ratings       = userInf[i]['ratings']
                numRatings    = userInf[i]['numRatings']
                iD            = userInf[i]['iD']
```

Δεν παραθέτουμε τον κώδικα προεπεξεργασίας των δεδομένων αφού τον έχουμε παρουσιάσει ήδη στην C έκδοση. Αρκεί σε αυτό το σημείο να γίνει κατανοητό οτι τα δεομένα είναι ήδη επεξεργασμένα και τοποθετημένα στο αντίστοιχο λεξικό. Σε αυτό το σημείο αντλούμε από το λεξικό τα δεδομένα για τον χρήστη i.

```
                hardwareKernel(M, sel , ratings , numRatings , dmas)

                Aout  = dmas["AoutDMA"].get_buf(data_type="float")
                Vout  = dmas["VoutDMA"].get_buf(data_type="float")

                V    = ffi.buffer(Vout, g.NFEATS* 4)
                V2   = np.frombuffer(V, dtype = np.float32)

                Ac    = ffi.buffer(Aout, (g.NFEATS*g.NFEATS) * 4)
                Ac2   = np.frombuffer(Ac, dtype = np.float32)
                Ac3   = np.copy(np.reshape(Ac2, (g.NFEATS, g.NFEATS)))

                Ac4 = np.add(Ac3, Ac3.T) - np.diag(np.diag(Ac3))

                Ac4[np.diag_indices_from(Ac4)] += lamda*numRatings

                U[iD] = scipy.linalg.cho_solve(scipy.linalg.cho_factor(
    Ac4, lower=True) ,V2)
```

Αφού γίνει η εξαγωγή των δεδομένων αποστέλλονται στην συνάρτηση "hardwareKernel" η οποία γεμίζει τους buffers όπως θα δούμε και στην συνέχεια και καλεί τον πυρήνα, έπειτα δεν μένει παρά να μετατρέψουμε τα δεδομένα σε numpy arrays για την τελική επεξεργασία. Η τελική επεξεργασία αποτελείτε από δυο στάδια την κανονικοποίηση των δεδομένων του χρήστη ή της ταινίας, την πρόσθεση δηλαδή ενός ποσοστού του πλήθους τον συνολικών βαθμολογιών του στόν αντίστοιχο πίνακα και την επίλυση του συστήματος με χρήση cholesky.Οι buffers που επιστρέφει το hardware είναι μονοδιάστατοι, για τον λόγο αυτό είναι αναγκαίο να τους μετασχηματίσουμε σε δισδιάστατους όπου χρειάζεται ώστε να μπορέσουμε να εκτελέσουμε τους υπολογισμούς κάνοντας χρήση των ταχύτατων βιβλιοθηκών numpy και scipy. Τέλος σημειώνουμε ότι ο πίνακας A που επιστρέφει το υλικό είναι άνω τριγωνικός, αφου για εξοικονόμηση υπολογισμών εκτελέσει στο υλικό το ελάχιστο δυνατό πλήθος υπολογισμών, ωστόσο οι numpy βιβλιοθήκες απαιτούν εισόδους με πλήρεις πίνακες, για αυτόν τον λόγο στον πίνακα Ac4 αντιγράφουμε τον Ac3 και τον ανάστροφό του.

Στην παρακάτω συνάρτηση αντιγράφουμε τα δεδομένα στους buffers των DMAs που κατασκευάσαμε στην συνάρτηση (createHardwareInterface).

```
def bufferizeAndSend(M, sel , ratings , nratings , dmas):
        buffIndx = 0
        Sbuffer = dmas["SbufferDMA"].get_buf(data_type="float")
        Rbuffer = dmas["RbufferDMA"].get_buf(data_type="float")
```

```
5
6           ffi = cffi.FFI()
7           s = ffi.from_buffer(M[sel,:].flatten().astype(np.float32))
8           s_buf = ffi.cast("float *", s)
9           Sbuffer[0:nratings*g.NFEATS] = s_buf[0:nratings*g.NFEATS]
10
11          r = ffi.from_buffer(ratings)
12          r_buf = ffi.cast("float *", r)
13          Rbuffer[0:nratings] = r_buf[0:nratings]
14
15
16          hardware.topLevelHW(nratings,dmas)
17          Aout = dmas["AoutDMA"].get_buf(data_type="float")
```

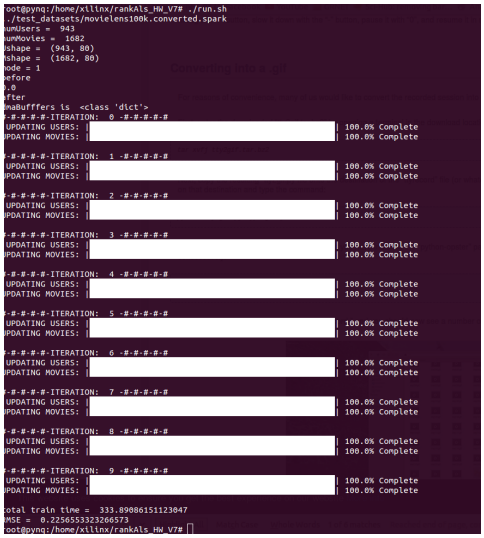
Οι buffers στους οποίους δείχνουν οι DMAs είναι έχουν μετατραπεί σε αντικείμενα Python, πράγμα που αναφέρεται στον κώδικα του API στο git hub. Αυτό σημαίνει ότι μπορούμε σε αυτά να γράψουμε κατευθείαν απο άλλα Python αντικείμενα, στην συγκεκριμένη περίπτωση τον πίνακα M. Ωστόσο παρατηρήσαμε ότι μετατρέψουμε τον πίνακα M πρώτα σε ωμά C data bytes και έπειτα τα αντιγράψουμε στους buffers, τότε επιτυγχάνουμε πολύ καλύτερη επίδοση. Αντίστοιχα πράττουμε και για τα υπόλοιπα δεδομένα(πίνακας ratings[]) .

```
1 def hardwareKernel(M, sel, ratings, numRatings, dmas):
2           bufferizeAndSend(M, sel, ratings, numRatings, dmas)
3           Aout  = dmas["AoutDMA"].get_buf(data_type="float")
4           Vout  = dmas["VoutDMA"].get_buf(data_type="float")
```

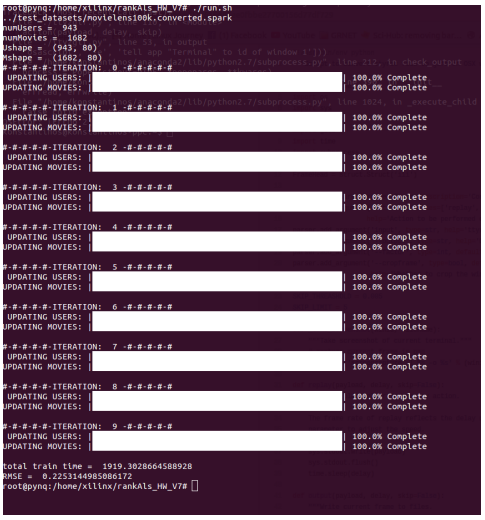
5.6 Επίδοση στο Pynq

Αφού πλέον έχει γίνει η ενσωμάτωση του πυρήνα στην Python μπορούμε πλέον να πάρουμε τις απαραίτητες μετρήσεις, ώστε να δούμε αν διατηρούμε ακόμα τα πλεονεκτήματα που πετύχαμε στην υλοποίηση με C. Παρακάτω παρουσιάζονται τα αποτελέσματα, όσον αφορά τον χρόνο εκτέλεσης, για δέκα επαναλήψεις του ALS με είσοδο το dataset movielens100k και πέντε επαναλήψεις με είσοδο το dataset movielens1m.

Παρατηρούμε ότι στα σύνολα δεδομένων movielens100k και movielens1m πετύχαμε επιτάχυνση 6×και 14×, ενώ σε αντίστοιχη περίπτωση στην C παρατηρήσαμε ότι οι επιταχύνσεις ήταν 18×και 35×φορές αντίστοιχα. Φυσικά ο πυρήνας είναι ακριβώς ίδιος και στις 2 περιπτώσεις συνεπώς θα πρέπει να αναζητήσουμε τα αίτια της μειωμένης επίδοσης στο software κομμάτι της υλοποίησης. Η κύρια διαφορά ανάμεσα στο λογισμικό των δυο υλοποιήσεων είναι είναι ότι στην C υλοποίηση χρησιμποποιώντας τις κατάλληλες βιβλιοθήκες,π.χ. sds_alloc(num_bytes), είχαμε την δυνατότητα να δεσμεύσουμε από την αρχή της εκτέλεσης συνεχόμενες θέσεις φυσικής μνήμης και να εκτελέσουμε με αυτές όλους τους υπολογισμούς ταυτόχρονα σε software και hardware, ενώ στην python αναγκαζόμαστε συνεχώς να γράφουμε απο numpy-arrays σε φυσικές θέσεις μνήμης (με cffi) και από φυσικές θέσεις μνήμης σε numpy ώστε να είμαστε σε θέση να εκτελούμε γρήγορους υπολογισμούς στο λογισμικό με την χρήση της numpy_lib. Συγκεκριμένα μετρήσαμε τον συνολικό χρόνο που απαιτείται για την εγγραφή και ανάγνωση των συνεχόμενων θέσεων μνήμης και τα αποτελέσματα φαίνονται στον παρακάτω πίνακα.



(α′) Δέκα επαναλήψεις ALS πάνω στο Pynq με την βοήθεια του hardware kernel, με είσοδο το dataset movielens100k



(β′) Δέκα επαναλήψεις ALS πάνω στο Pynq χωρίς χρήση του επιταχυντή, με είσοδο το dataset movielens100k

dataset	iterations	total train time accel.	I/O physical cont. buffers	percent
ml_100k	10	322.85s	199.5s	61.7%
ml_1m	5	701.56s	441.68s	62.9%

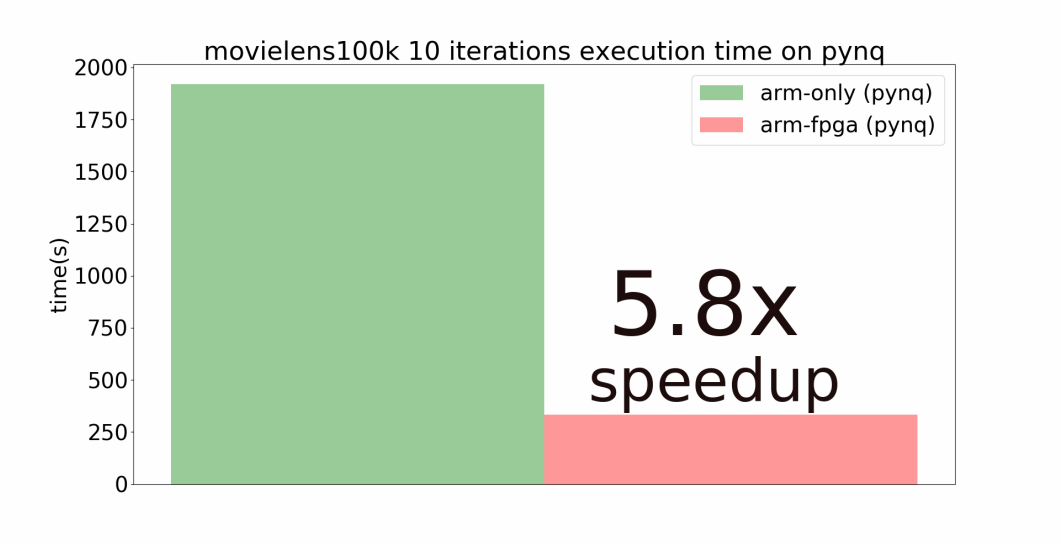
Αν αφαιρέσουμε αυτούς τους χρόνους από τον συνολικό χρόνο εκπαίδευσης προκείπτει ο παραπάνω πίνακας γίνεται:

dataset	iterations	train time accel. - I/O buffer time	software only	est.speedup
ml_100k	10	322.85s - 199.5s = 123.35	1919s	15.55×
ml_1m	5	701.56s - 441.68s = 259.88	9721s	37.4×

Παρατηρούμε ότι πάνω από το 60% αναλώνεται στην εγγραφή και ανάγνωση των buffers συνεχούς φυσικής μνήμης , ενώ με ο αντίστοιχος χρόνος σε C ήταν μόλις 12%. Μάλιστα γίνεται φανερό από τα δυο dataset ότι όσο μεγαλώνουν οι συγκεκριμένοι buffers τόσο περισσότερο χρόνο απαιτεί η εγγραφή και η ανάγνωσή τους. Παρατηρούμε ότι η εκτιμόμενη επιτάχυνση, δίχως την εγγραφή/ανάγνωση των buffer ταυτίζεται με την επιτάχυνση που είχαμε στην C υλοποίηση, πράγμα που σημαίνει ότι έχουμε εντοπίσει ακριβώς το πρόβλημα της μειωμένης επίδοσης.

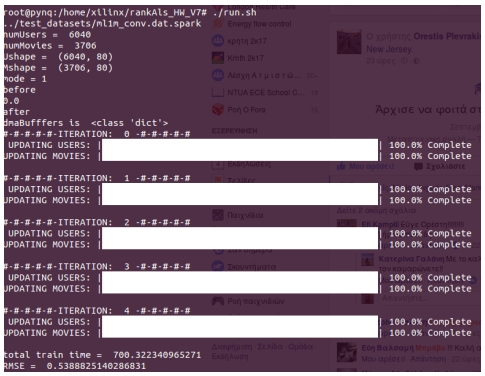
5.7 Συμπερασματικές Παρατηρήσεις

Σε αυτό το κεφάλαιο καταφέραμε να ενσωματώσουμε τον πυρήνα επιτάχυνσης στο περιβάλλον της γλώσσας Python χρησιμοποιώντας το τις βιβλιοθήκες που παρέχει η Xilinx στο image του Pynq. Η παραπάνω διαδικασία ήταν αρκετά πιο χρονοβόρα από την αντίστοιχη σε C , διότι στην δεύτερη περίπτωση το περιβάλλον SDSoC αναλάμβανε εξολοκλήρου την σύνδεση software - hardware ενώ στην πρώτη περίπτωση κάναμε χρήση βιβλιοθηκών χαμηλού επιπέδου για να πετύχουμε την αντίστοιχη λειτουργικότητα. Το αποτέλεσμα ήταν να πετύχαμε μειωμένη επιτάχυνση σε σχέση με την C υλοποίηση, διότι στην Python δεν μπορούσαμε να έχουμε την ίδια ταχύτητα εγγραφής

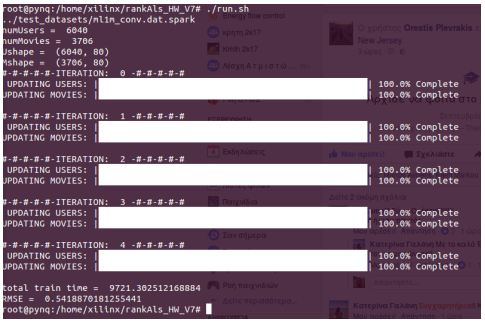


Σχήμα 5.4: Παραπάνω φαίνεται ο χρόνος εκτέλεσης 10 επαναλήψεων πάνω στο Pynq, με είσοδο το dataset movielens100k ,κανονικοποιημένος ως προς την software έκδοση.

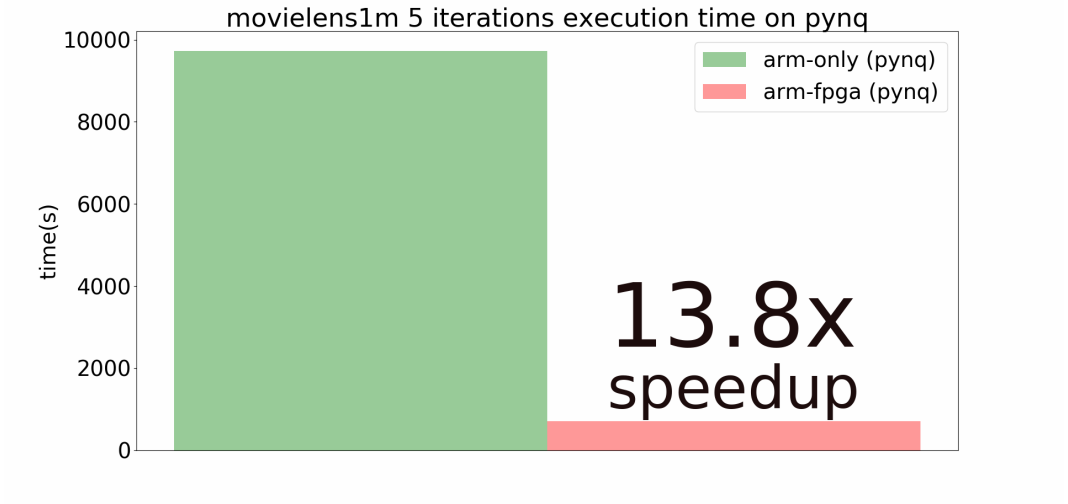
και ανάγνωσης. Συγκεκριμένα τα αποτελέσματα του υλικού στη C δεν τα μεταφέραμε καθόλου σε νέους πίνακες αλλά τα επεξεργαζόμασταν κατευθείαν πάνω στις δομές που μας επέστρεφε το υλικό, ενώ τα αποτελέσματα του υλικού στην python τα μεταφέραμε με χρονοβόρα αντιγραφή σε numpy arrays για να είμαστε σε θέση να εκτελέσουμε γρήγορους τους υπόλοιπους software υπολογισμούς.



(α΄) Πέντε επαναλήψεις ALS πάνω στο Pynq με την βοήθεια του hardware kernel, με είσοδο το dataset movielens1m



(β΄) Πέντε επαναλήψεις ALS πάνω στο Pynq χωρίς χρήση του επιταχυντή, με είσοδο το dataset movielens1m



Σχήμα 5.6: Παραπάνω φαίνεται ο χρόνος εκτέλεσης 5 επαναλήψεων πάνω στο Pynq, με είσοδο το dataset movielens1m ,κανονικοποιημένος ως προς την software εκδοση.

6 Ενσωμάτωση στο Spark

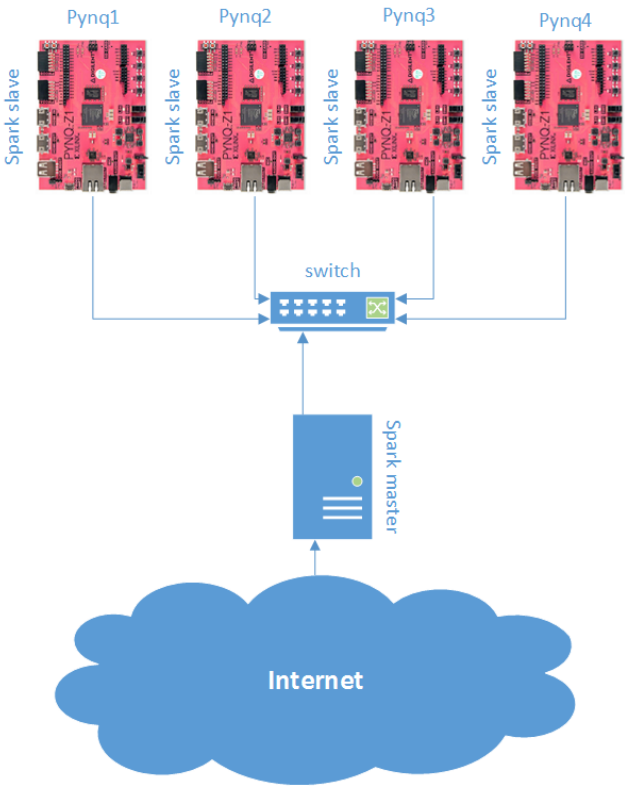
6.1 Εισαγωγή

Μετά την ενσωμάτωση του επιταχυντή σε περιβάλλον python μένει να επιχειρήσουμε την ενσωμάτωσή της python έκδοσης σε περιβάλλον spark. Για τον σκοπό αυτό δημιουργήθηκε στο εργαστήριο το cluster που φαίνεται στην εικόνα. Σκοπός αυτού του κεφαλαίου είναι να συγκρίνουμε μια κλασσική λύση ενός Xeon επεξεργαστή με μια λύση ετερογενούς αρχιτεκτονικής.

Σχήμα 6.1: Το cluster που στήθηκε στο εργαστήριο με στόχο να δοκιμαστεί σε αυτό η εφαρμογή των αλγορίθμων σε spark. Αποτελείτε απο 4 pynq και ένα switch.

6.2 Περιγραφή του Cluster και Spark Configuration

Το cluster αποτελείται από τέσσερα pynq και ένα Dell Desktop συνδεδεμένα πάνω σε ένα ethernet switch. Δικτυακά το Desktop χρησιμοποιείται για να έχουμε πρόσβαση στο cluster,;έχει δηλαδή public IP, ενώ τα pynq δεν έχουν Public IP και η πρόσβαση σε αυτά μπορεί να πραγρατοποιηθεί μόνο μέσω του Desktop. Όσον αφορά το Spark το Desktop λειτουργεί σαν Spark master πάνω στον οποίο τρέχει ο Driver της εφαρμογής, ενώ τα Pynqs έχουν τον ρόλο των Spark Slaves. Ο αλγόριθμος που θα περιγραφεί στην συνέχεια δημιουργήθηκε ώστε το training να γίνεται εξολοκλήρου στους slaves και συνεπώς τα χαρακτηριστικά του Desktop δεν έχουν κάποια σημασία στις μετρήσεις που θα παρουσιάσουμε.



Σχήμα 6.2: Η τοπολογία του cluster.

Στο script spark_env.sh ορίζουμε τις παρακάτω μεταβλητές περιβάλλοντος του spark, τα βασικά χαρακτηριστικά των οποίων είναι ότι στον driver δώσαμε όση μνήμη όσο και στους executors, 505mb, ορίσαμε έναν worker για κάθε slave και έναν executor για κάθε worker.

Listing 6.1: spark_env.sh configuration στον Master

```
#!/usr/bin/env bash

export PYSPARK\_PYTHON=python3.4
```

```
export PYTHONPATH=\$PYTHONPATH:/home/xilinx/  \
                spark-2.1.1/python: \
                /python/lib/py4j-0.10.4-src.zip
export PYTHONHASHSEED=0
export SPARK\_DRIVER\_MEMORY=505m
export SPARK\_EXECUTOR\_INSTANCES=1
export SPARK\_EXECUTOR\_CORES=1
export SPARK\_EXECUTOR\_MEMORY=505m
export SPARK\_MASTER\_HOST=192.168.1.203
export SPARK\_LOCAL\_IP=192.168.1.203
export SPARK\_WORKER\_MEMORY=505m
export SPARK\_WORKER\_CORES=1
export SPARK\_DAEMON\_MEMORY=505m
export SPARK\_WORKER\_INSTANCES=1
export SPARK\_WORKER\_TYPE=0
```

Και το configuration spark-defaults.conf στο οποίο ορίζουμε τις υπόλοιπες ρυθμίσεις του cluster, public IP του master, καθορισμός του object serializer των αντικειμένων με τον οποίο θα αποστέλλονται τα αντικείμενα μεταξύ των slaves και του master.

Listing 6.2: spark-defaults.conf στον Master

```
spark.master                     spark://192.168.1.203:7077
spark.eventLog.enabled           true
spark.eventLog.dir  /home/xilinx/spark-2.1.1/work
spark.serializer    org.apache.spark.serializer.KryoSerializer
spark.driver.memory              505m
spark.executor.instances         1
spark.executor.memory            505m
spark.executor.cores             1
spark.history.fs.logDirectory /home/xilinx/spark-2.1.1/work
```

6.3 Ο Αλγόριθμος

Ό αλγόριθμος που κατασκευάστηκε στο προηγούμενο κεφάλαιο απαιτεί κάποιες μικρές αλλαγές για να προσαρμοστεί στο περιβάλλον του Spark. Γνωρίζουμε ότι κάθε διάνυσμα του πίνακα U μπορεί να ανανεωθεί ανεξάρτητα απο τα υπόλοιπα διανύσματα με σταθερό τον πίνακα M και αντίστοιχα κάθε διάνυσμα ταινίας του πίνακα M μπορεί να ανανεωθεί ανεξάρτητα από τα υπόλοιπα διανύσματα του πίνακα M, συνεπώς η καλύτερη δυνατή προσέγγιση είναι να όταν ανανεώνουμε τον πίνακα U , να διαμοιράσουμε τα διανύσματα του στους spark slaves του U και να κάνουμε broadcast τον πίνακα M, αντίστοιχα για να ανανεώσουμε τα διανύσματα του πίνακα M μπορούμε να τα διαμοιράσουμε στους slaves μαζί με τον ολόκληρο τον πίνακα U. Η στρατηγική αυτή φαίνεται στο παρακάτω κομμάτι κώδικα.

```
1 Ub        = sc.broadcast(U)
```

```
Mb          = sc.broadcast(M)
NFEATSb     = sc.broadcast(g.NFEATS)
lamdab      = sc.broadcast(lamda)

for i in range(iterations):
    U = userInfRDD.mapPartitions(lambda userInfPartition : modules.
    updateU(Mb.value,lamdab.value,userInfPartition,modeb.value,NFEATSb.
    value,NMAXRATb.value)).collect()
    Ub = sc.broadcast(np.array(U))
    M = movieInfRDD.mapPartitions(lambda movieInfPartition : modules.
    updateM(Ub.value,lamdab.value,movieInfPartition,modeb.value,NFEATSb.
    value,NMAXRATb.value)).collect()
    Mb = sc.broadcast(np.array(M))
```

Αρχικά γνωστοποιούμε (broadcast) σε όλους τους spark slaves τις U,M το πλήθος των χαρακτηριστικών (NFEATS) και τον παράγοντα κανονικοποίησης (lamda), οι μεταβλητές NFEATS και lamda θα διαβαστούν μόνο απο τους slaves συνεπώς δεν θα χρειαστεί να τις αποστείλουμε ξανά , ενω οι μεταβλητές U,M θα πρέπει να γνωστοποιούνται εκ νέσου στους slaves κατά την διάρκεια της εκπαίδευσης. Όσον αφορά την εκπαίδευση αρχικά διαμοιράζουμε την δομή userInf σε τέσσερα partitions (ένα σε κάθε slave) , και για κάθε partition καλούμε στον ανσιστοιχο spark slave την συνάρτηση updateU. Έπειτα συλλέγουμε (collect()) το αποτελέσμα,δηλαδή τον ανανεωμένο πίνακα U, πίσω στον master και τον γνωστοποιούμε εκ νέου στους slaves διότι η επερχόμενη ανανέωση του πίνακα M απαιτεί γνώση του ανανεωμένου πίνακα U. Η διαδικασία επαναλαμβάνεται για την ανανέωση του πίνακα M.

```
def updateU(M,lamda,userInf,mode,NFEATS,NMAXRAT):
    result = []
    ffi = cffi.FFI()
    dmas = hardware.createHardwareInterface(NFEATS,NMAXRAT)
    for user in userInf:
        sel             = user['rId']
        ratings         = user['ratings']
        numRatings      = user['numRatings']
        iD              = user['iD']
        print("Uid: ",iD)


        hardwareKernel(M,sel,ratings,numRatings,dmas,NFEATS,ttU)
        Aout   = dmas["AoutDMA"].get\_buf(data\_type="float")
        Vout   = dmas["VoutDMA"].get\_buf(data\_type="float")

        V     = ffi.buffer(Vout, NFEATS* 4)
        V2    = np.frombuffer(V, dtype = np.float32)

        Ac     = ffi.buffer(Aout, (NFEATS*NFEATS) * 4)
        Ac2    = np.frombuffer(Ac, dtype = np.float32)
        Ac3    = np.copy(np.reshape(Ac2, (NFEATS,NFEATS)))
        Ac4 = np.add(Ac3,Ac3.T) - np.diag(np.diag(Ac3))
        Ac4[np.diag\_indices\_from(Ac4)] += lamda*numRatings

        result.append( scipy.linalg.cho\_solve(scipy.linalg.cho\_factor(
    Ac4,lower=True),V2) )
    return result
```

Παραπάνω παρουσιάζουμε την συνάρτηση updateU (αντίστοιχα και η updateM), όπως τροποποιήθηκε για να εκτελεστεί στους slaves του Apache Spark. Αφού αρχικοποιήσουμε ένα αντικείμενο FFI ωστε να μπορούμε να επικοινωνούμε με το API, και κατεβάσουμε το bitstream στον συγκεκριμένο slave, ξεκινάει η ανανέωση κάθε διανύσματος χρήστη που εμπεριέχεται μέσα στο partition του userinf που επεξεργάζεται ο συγεκκριμένος slave. Αρχικά γίνεται κλήση του πυρήνα και στην συνέχεια μετατρέπουμε τα αποτελέσαμτα, τα οποία μας τα δίνει ο πυρήνας σε ffi buffer με συνεχόμενες θέσεις φυσικής μνήμης, σε numpy arrays ώστε να είμαστε σε θέση να χρησιμοποιήσουμε την numpy library για τους υπόλοιπους υπολογισμούς. Κάθε ανανεωμένο διάνυσμα προστίθεται στην λίστα result η οποία και επιστρέφεται απο τον slave στον driver μετά το πέρας της εκτέλεσης.

6.4 Επίδοση

Μετά την κατασκευή του cluster, την εγκαταστση του Spark σε αυτό και την συγγραφή του κώδικα εκτελέσαμε πάνω στο cluster τον αλγόριθμο με είσοδο το σύνολο δεδομένων Movielens 1m για μια επανάληψη.Επειδή ωστόσο σκοπός μας ήταν να δούμε κατά πόσο τέτοια λύση είναι συγκρίσιμη με ένα πραγματικό datacenter, δοκιμάσαμε να τρέξουμε τον ίδιο κώδικα σε ένα server με επεξεργαστη, Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz. Συγκεκριμένα στήσαμε στον server την ίδια ακριβώς έκδοση του apache spark με αυτήν που έχουμε στήσει και στο pynq cluster, και αποδόσαμε ώς πόρους 4 πυρήνες του Xeon επεξεργαστή με 1gb μνήμης στον καθένα, κάθε πυρήνας αντιστοιχεί σε έναν spark-slave.Συνολικά οι χρόνοι που χρειάστηκαν για την εκτέλεση μιας επανάληψης στο Pynq Cluster ήταν 77 δευτερόλεπτα ενώ στο Xeon cluster χρειάστηκαν μόλις 15 δευτερόλεπτα, πράγμα που σημαίνει ότι η κλασσική λύση εκτέλεσε τον αλγόριθμο 5 φορές ταχύτερα απο το cluster που κατασκευάσαμε.

Αυτή η διαφορά στην επίδοση μας οδήγησε στην καταγραφή του χρονικού προφίλ της εκτέλεσης για τις δυο διαφορετικές αρχιτεκτονικές, από την μια 4 Xeon Cores και απο την άλλη 4 arm υποβοηθούμενοι από επιταχυντές FPGA.

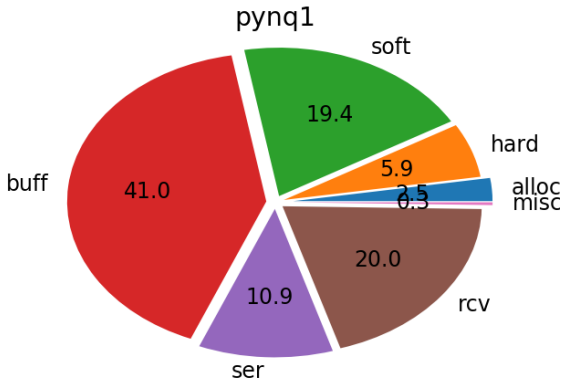
Αρχικά παρουσιάζουμε το προφίλ εκτέλεσης του Pynq Cluster. Στον παρακάτω πίνακα φαίνεται για κάθε ένα από τα τέσσερα pynq και για κάθε task (updateU() και updateM()) πώς διαμοιράστηκε ο χρόνος εκτέλεσης.

slave	op	total(s)	ser(s)	alloc(s)	hard(s)	soft(s)	buff(s)	rcv(s)
1	U	42	5	0.72	2.15	6.96	16.12	9
1	M	16	1	0.67	1.12	3.18	7.47	2
2	U	28	7	0.72	1.12	3.50	8.06	7.6
2	M	17	2	1.4	1.46	3.15	7.64	1.4
3	U	41	6	2.21	2.01	6.58	15.27	8.93
3	M	18	1	1.2	1.48	3.15	8.09	2.02
4	U	27	5	0.67	1.05	3.83	7.97	7.48
4	M	16	1	0.46	1.46	3.14	7.66	1.22

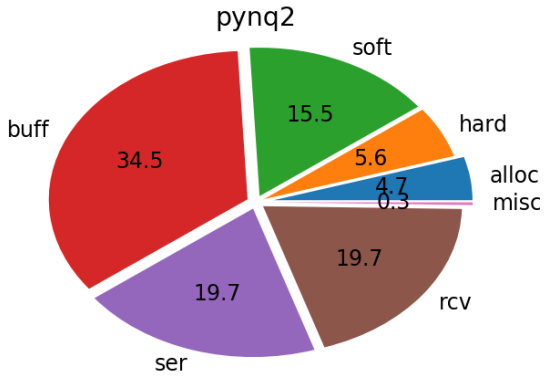
Όπου

- op (operation) : U ή M αν ο συγκεκριμένος χρόνος αναφέρεται σε εκτέλεση της συνάρτηση updateU ή της updateM αντίστοιχα.

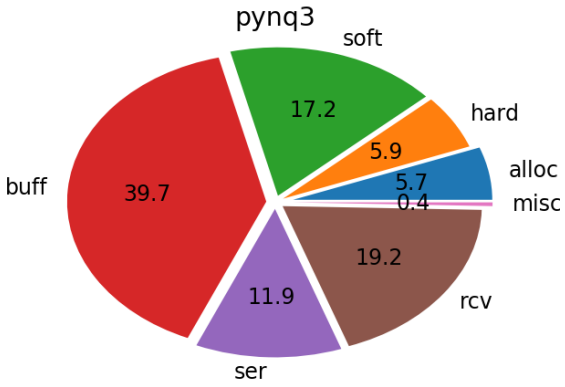
- total: συνολικός χρόνος εκτέλεσης στον slave.
- ser (serialize/deserialize): χρόνος που καταναλώθηκε στη σειριοποίηση και αποσειριοποίηση των δεδομένων.
- alloc (allocation): χρόνος που καταναλώθηκε για την δέσμευση μνήμης.
- hard (hardware calculations): χρόνος που καταναλώθηκε στον υλικό επιταχυντή.
- soft (software calculations): χρόνος που καταναλώθηκε στους software υπολογισμούς πάνω στον arm (π.χ. cholesky).
- buff (buffers): χρόνος που καταναλώθηκε για γραφή και ανάγνωση των δεδομένων στους φυσικά συνεχείς buffers.
- rcv (receive): χρόνος που καταναλώθηκε μέχρι ο spark slave να λάβει τα δεδομένα.
- misc : λοιποί χρόνοι



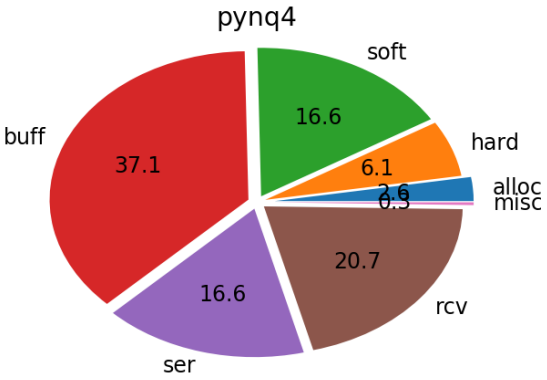
(α′) ποσοστό ανά εργασία για το pynq1



(β′) ποσοστό ανά εργασία για το pynq2



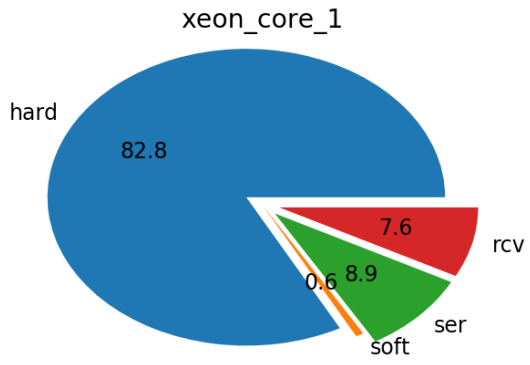
(γ′) ποσοστό ανά εργασία για το pynq3



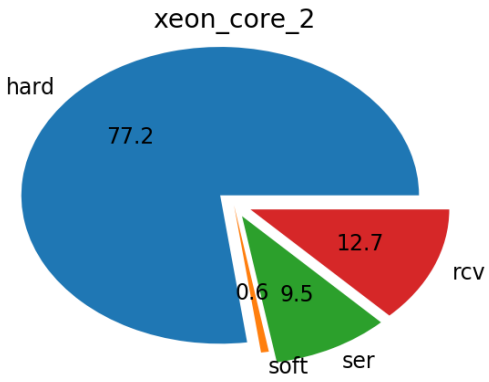
(δ′) ποσοστό ανά εργασία για το pynq4

Ενώ αντίστοιχα το ίδιο προφίλ για την εκτέλεση στους Xeon Cores φαίνεται παρακάτω.

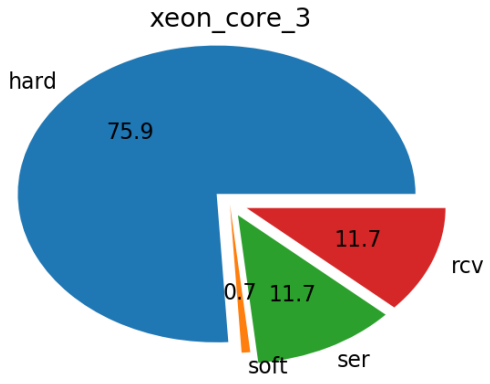
slave	op	total(s)	ser(s)	alloc(s)	hard(s)	soft(s)	buff(s)	rcv(s)
1	U	4.72	0.7	0.0	3.5	0.02	0	0.6
1	M	3.13	0.1	0.0	3	0.03	0	0
2	U	5.23	0.8	0.0	3.2	0.03	0	1.2
2	M	4.23	0.1	0.0	4.1	0.03	0	0
3	U	3.63	0.9	0.0	1.9	0.03	0	0.8
3	M	2.73	0.1	0.0	2.4	0.03	0	0.2
4	U	7.33	1	0.0	4.7	0.03	0	1.6
4	M	3.33	0.1	0.0	3.2	0.03	0	0



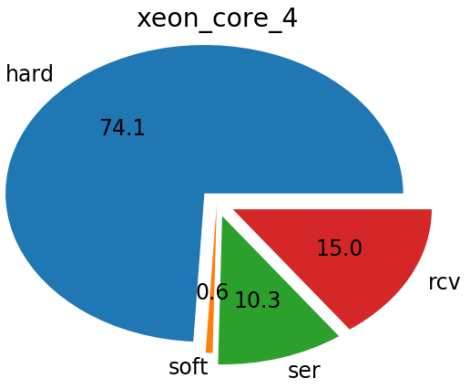
(α′) ποσοστό ανά εργασία για τον Xeon Core 1



(β′) ποσοστό ανά εργασία για το Xeon Core 2



(γ′) ποσοστό ανά εργασία για το Xeon Core 3



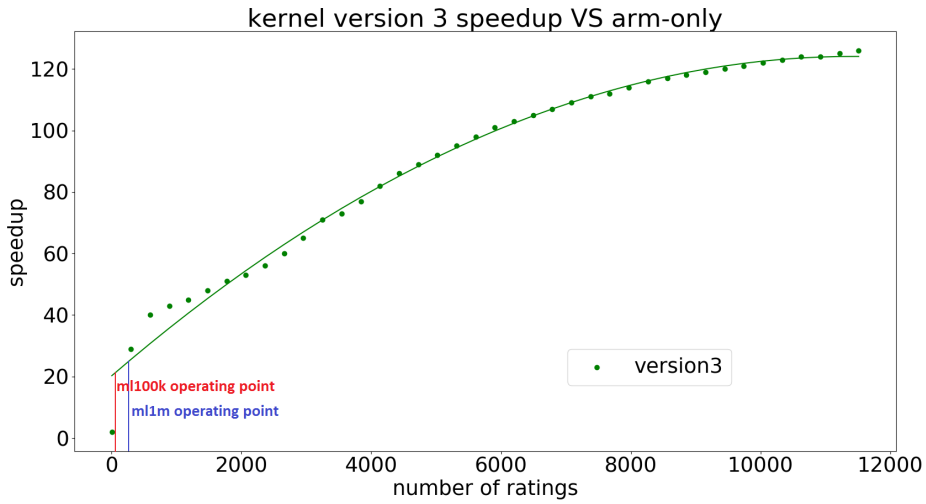
(δ′) ποσοστό ανά εργασία για το Xeon Core 4

6.5 Παρατηρήσεις

Δυστυχώς το Spynq δεν κατάφερε να ξεπεράσει σε επίδοση την κλασική λύση ενός Xeon επεξεργαστή ωστόσο αυτό σίγουρα δεν σημαίνει ότι τα FPGA δεν είναι κατάλληλα για ενσωμάτωση σε ένα κέντρο δεδομένων.Συγκεκριμένα υπάρχουν τρεις

συγκεκριμένοι λόγοι που στο συγκεκριμένο πείραμα παρατηρήθηκε μειωμένη επίδοση, τους οποίους και θα αναλύσουμε παρακάτω.

1. PCI over Ethernet.Στον συγκεκριμένο αλγόριθμο κατά την διάρκεια κάθε επανάληψης απαιτείται να διαμοιραστούν δεδομένα ανάμεσα στους slaves (οι πίνακες U και M). Στον Xeon οι τέσσερις apache-workers (Xeon Cores) βρίσκονται στο ίδιο chip και επίσης διασυνδέονται με την μνήμη με PCI bus παράγοντες που συντελούν ο απαιτούμενος χρόνος του διαμοιρασμού των δεδομένων να είναι σχεδόν μηδενικός. Από την άλλη στο Spynq τα δεδομένα διαμοιράζονται μεταξύ των Spark-workers μέσω του δικτύου με ethrnet γεγονός που εισάγει τεράστιες καθυστερήσεις. Συγκεκριμένα για τον Xeon μετρήσαμε ένα μέσο χρόνο κάτω από μισό δευτερόλεπτο για τον broadcast των πινάκων U και M ενώ για το Spynq ο αντίστοιχος μέσος χρόνος ανήλθε στα 5 δεύτερα δηλαδή 10 φορές περισσότερος σε σχέση με τον Xeon και φυσικά το αναμενόμενο για περισσότερα δεδομένα είναι αυτό ο παράγοντας να αυξηθεί.
2. Immature Api. Παρατηρούμε ότι στο Spynq το 40% του χρόνου, δηλαδή περίπου 10 δευτερόλεπτα, καταναλώνεται στο γέμισμα των buffer συνεχούς φυσικής μνήμης, Αυτή η καθυστέρηση οφείλεται στο γεγονός ότι οι συγκεκριμένοι buffers δεν είναι συμβατοί με τις numpy βιβλιοθήκες και συνεπώς για να μεταβούμε από τους hardware υπολογισμούς στους software και αντίστροφα απαιτούνται σημαντικές καθυστερήσεις. Η έκδοση που έτρεξε στον Xeon δεν είχε καμία τέτοια επιβάρυνση αφού εκτελέσθηκε εξολοκλήρου σε software.
3. Arm Poor Performance. Τέλος είναι εμφανές ότι ο Arm επεξεργαστής στο Pynq προσθέτει τεράστιες καθυστερήσεις στα software κομμάτια του αλγορίθμου και του spark. Συγκεκριμένα κάθε pynq spark-worker χρειάστηκε 3.5sec για να κάνει εκτελέσει τις λειτουργίες σειριοποίησης και αποσειριοποίησης του spark ενώ ο αντίστοιχος χρόνος για τους xeon spark-slaves ήταν κάτω από μισό δευτερόλεπτο 7 φορές δηλαδή ταχύτερες σε σχέση με τον arm και όπως και στο PCI over Ethernet πρόβλημα δεν είμαστε σε θέση να γνωρίζουμε πώς κλιμακώνεται αυτός ο παράγοντας για περισσότερα δεδομένα. Αντίστοιχα για τις υπόλοιπες software λειτουργίες όπως η παραγοντοποίηση cholesky και η επίλυσή του ο Xeon χρειάστηκε κατα μέσο όρο μόλις 0.03 δευτερόλεπτα σε αντίθεση με τους arm που χρειάστηκαν κατά μέσο όρο 4.2 δευτερόλεπτα δηλαδή υπήρξαν περίπου 140 φορές πιο αργοί.
4. Memory Deficit Όσον αφορά τους εκεί παρατηρούμε ότι τα οι Pynq workers είναι περίπου κατα 2.2 φορές ταχύτεροι. Το συγκεκριμένο όφελος είναι τεράστιο αν αναλογιστούμε ότι περίπου το 80% του χρόνου εκτέλεσης των Xeon καταναλίσκεται σε υπολογισμούς που έχουν γίνει accelerate. Ωστόσο σε αυτό το σημείο θα πρέπει να υπενθυμίσουμε ότι στην χαρακτηριστική του πυρήνα βρισκόμαστε σε πολύ χαμηλό σημείο λειτουργίας αφού το συγκεκριμένο dataset έχει κατά μέσο όρο 205 ratings ανά ταινία η χρήστη, πράγμα που σημαίνει ότι όταν τα διαμοιράζουμε σε 4 workers αυτός ο μέσος όρος μειώνεται σε 50 και συνεπώς η διαφορά στην επίδοση μειώνεται ακόμα περισσότερο.Σε ένα τόσο μικρό dataset δεν κάνουμε πλήρη αξιοποίηση των δυνατοτήτων του πηρήνα, ωστόσο τα Pynqs έχοντας μόλις 512Mb RAM δεν μας αφήνουν περιθώριο να δοκιμάσουμε σύνολα δεδομένων που πραγματικά εντάσσονται στην κατηγορία big data.



Σχήμα 6.5: Το σημείο λειτουργίας του πυρήνα για τα dataset ml100k και ml1m. Είναι εμφανές ότι ο πυρήνας μπορεί να πετύχει πολύ καλύτερες επιδόσεις έναντι του απλού software για μεγαλύτερα datasets. Ο κάθετος άξονας έχει της επιταχύνσεις σε σχέση με τον Arm ωστόσο η καμπύλη μπορεί να κλιμακωθεί και για τον Xeon.

7 Σύνοψη

7.1 Περίληψη

Σίγουρα η εκθετική αύξηση των δεδομένων σε συνδυασμό με την όλο και μεγαλύτερη ανάγκη για επεξεργασία τους με στόχο την παροχή καλύτερων υπηρεσιών μας αναγκάζει να αναζητήσουμε νέες μεθόδους επεξεργασίας τους, που θα οδηγήσουν τόσο σε μεγαλύτερες ταχύτητες επεξεργασίας όσο και σε μείωση της συνολικής ανάγκης σε ενέργεια. Σε αυτή την διπλωματική και στο γενικότερο πλαίσιο που αυτή εντάσσεται προτείνουμε σαν εναλλακτική στην CPU only επεξεργασία και το CPU-GPU coprocesing, την ιδέα του custom-computing, την ενσωμάτωση δηλαδή των hardware επιταχυντών σχεδιασμένων πάνω σε FPGA και την χρήση τους σαν επεξεργαστικά βοηθήματα για τα υπολογιστικά έντονα κομμάτια των αλγορίθμων. Στόχος αυτής της διπλωματικής ήταν να αναδείξει και μια μεθοδολογία μέσω της οποίας μπορούν τα FPGA να ενταχθούν στα κέντρα δεδομένων.

Ως αλγόριθμο για να αναδείξουμε την μεθοδολογία και να κάνουμε συνολική αξιολόγηση της ιδέας χρησιμοποιήσαμε έναν αλγόριθμο συστάσεων μηχανικής μάθησης ο οποίος εντάσσεται στην κατηγορία του συνεργατικού φιλτραρίσματος. Ο αλγόριθμος μοντελοποιεί χρήστες και αντικείμενα ώς διανύσματα χαρακτιρηστίκων και χρησιμοποιεί το εσωτερικό τους γινόμενο για να κάνει προβλέψεις, τα διανύσματα αυτά ωστόσο προκύπτουν απο μια απαιτητική υπολογιστικά διαδικασία σπασίματος ενός αραιού πίνακα σε δυο πίνακες μικρότερων διαστάσεων. Η παραγοντοποίηση του αραιού πίνακα αποτελείται απο δυο θεμελιώδεις λειτουργείς, πολλάπλασιασμούς πινάκων και επίλυση συστημάτων. Μετά απο αποτύπωση του χρονικού προφίλ της διαδικασίας αποφασίσαμε να επιταχύνομε στο FPGA την κομμάτι των πολλαπλασιασμών και να αφήσουμε την επίλυση των συστημάτων σε software επεξεργασία εφόσον οι πόροι του υλικού ήταν περιορισμένοι.

Αρχικά κάναμε συγγραφή του αλγορίθμου σε C ώστε να κάνουμε χρήση του περιβάλλοντος HLS SDSoC της Xilinx, το οποίο δίνει την δυνατότητα αυτοματοποιημένης διασύνδεσης του υλικού με την CPU στην πλακέτα Zedboard. Αφού κάναμε εξερεύνηση των διαφόρων HLS εντολών αλλά και των διαφόρων τρόπων διεπαφής του υλικού με την CPU καταλήξαμε σε έναν πυρήνα επιτάχυνσης ικανό να πετύχει τεράστια χρονικά και ενεργειακά οφέλη σε σχέση με την softare-only επεξεργασία του arm πάνω στο zedboard.

Στην συνέχεια επιχειρήσαμε να μεταφέρουμε την υλοποίηση μας σε μια περισσότερο δημοφιλή γλώσσα για τέτοιου είδους tasks. Για τον σκοπό αυτό επιλέξαμε την Python αφού διαθέταμε την αναπτυξιακή πλακέτα Pynq για την οποία παρέχεται ένα linux image με βιβλιοθήκες χαμηλού επιπέδου, οι οποίες επιτρέπουν την διεπαφή του υλικού με την CPU. Πετύχαμε τον παραπάνω στόχο απομονώνοντας τον πυρήνα που δημιούργησε το περιβάλλον SDSoC και κάνοντας reverse engineering στο low level API που χρησιμοποιεί το SDSoc ώστε να πετύχουμε τα ίδια αποτελέσματα στην Python. Όταν ολοκληρώθηκε η ενσωμάτωση με Python παρατηρήσαμε μειωμένη επιτάχυνση σε σχέση με την C υλοποίηση αφού έπρεπε να μετατρέπουμε συνεχώς τα δεδομένα ανάμεσα σε δύο διαφορετικές μορφές ώστε να πετυχαίνουμε γρήγορή επεξεργασία τόσο στον PL-space όσο και στον PS-space.

Τέλος κατασκευάσαμε ένα cluster αποτελούμενο από τέσσερα Pynq με στοχο να δείξουμε ότι είναι εφικτή όχι μόνο η επιτάχυνση αλγορίθμων αλλά και η παραλληλοποίηση αυτή της επιτάχυνσης. Πάνω στο cluster εγκαταστησαμε το Apache Spark και καταφέραμε να τρέξουμε παράλληλα τον αλοριθμό με τρόπο τέτοιο ώστε κάθε apache-worker να έχει τον δικό του επιταχυντή. Ωστόσο η παραπάνω απόπειρα δεν κατάφερε να αναδείξει χρονικά πλεονεκτήματα έναντι σε έναν κλασικό Xeon επεξεργαστή για τέσσερις πολύ συγκεκριμένους λόγους τους αφορούν κυρίως την αρχιτεκτονική που χρησιμοποιήσαμε και όχι την γενικότερη ιδέα.

7.2 Μελλοντική δουλειά

Η παρούσα διπλωματική ανοίγει πολλές προοπτικές για μελλοντική έρευνα και εργασία. Πρώτος και κύριος στόχος θα πρέπει να είναι η εστίαση στα μειονεκτήματα της αρχιτεκτονικής που παρουσιάσαμε στο τελευταίο κεφάλαιο.Συγκεκριμένα προτείνουμε να γίνει χρήση και ενσωμάτωση της τεχνολογίας που παρουσιάσαμε σε μια απο της νέες πλατφόρμες της intel η της Amazon, με την πρώτη να διασυνδέει Xeon επεξεργαστές με FPGA cards με QPi και την δεύτερη να διασυνδέει επεξεργαστές με FPGA μέσω PCIe, με στόχο την ταχύτατη εκτέλεση τόσο των software όσο και των hardware κομματιών ενός αλγορίθμου.Προτείνουμε επίσης την επέκταση των δυνατοτήτων του Apache Spark, ώστε να διευκολύνει το εγχείρημα αυτής της διπλωματικής, με δυνατότητες όπως deserialization των tasks απευθείας σε συνεχόμενες θέσεις μνήμης κλπ. Σαν τελικό στόχο προτείνουμε την δημιουργία μιας βιβλιοθήκης η οποία θα περιέχει hardware accelerated tasks που είναι εξαιρετικά δημοφιλή στα κέντρα δεδομένων καθώς επίσης και η δημιουργία ενός υψηλού επιπέδου API που θα αποκρύπτει την πολυπλοκότητα από τον προγραμματιστή και θα κάνει εύκολη την ενσωμάτωση των

accelerators μέσα στον κώδικα.

Βιβλιογραφία

[1] FPGA Architecture for the Challenge
http://www.eecg.toronto.edu/ vaughn/challenge/fpga_arch.html

[2] Clive Maxfield, Programmable Logic DesignLine, "Xilinx unveil revolutionary 65nm FPGA architecture: the Virtex-5 family.

[3] FPGA or CPU? fpgacenter.com http://fpgacenter.com/fpga/fpga_or_cpu.php

[4] eetimes: Xilinx aims at DSP applications
http://www.eetimes.com/document.asp?doc_id=1303129

[5] Recent Trends in FPGA Architectures and Applications Philip H.W. Leong Dept. of Computer Science and Engineering The Chinese University of Hong Kong, Hong Kong

[6] A. Rajaraman, "More data usually beats better algorithms;' Datawocky Blog, 2008. http://anand.typepad.com/datawocky/2008/03/more-data-usual.html

[7] Xeon+FPGA Platform for the Data Center ISCA/CARL 2015 PK Gupta, Director of Cloud Platform Technology, DCG/CPG

[8] official spark: https://spark.apache.org/

[9] MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean and Sanjay Ghemawat Google, Inc.

[10] Cloudera:
https://www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_spark_apps.html

[11] xilinx official: https://www.xilinx.com/products/design-tools.html

[12] Vivado Design Suite User Guide 902 High-Level Synthesis

[13] xilinx official: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[14] Zynq-7000 All Programmable SoC Data Sheet Product Specification

[15] ZedBoard (ZynqTM Evaluation and Development) Hardware User's Guide

[16] Zynq-7000 All Programmable SoC ZC702 Evaluation Kit Quick Start Guide

[17] Encyclopedia of Machine Learning Chapter No: 00338 Page Proof Page 1 22-4-2010 Prem Mellville, Vikas Sindhwani Machine Learning, IBM T. J. Watson Research Center,

[18] Item-Based Collaborative Filtering RecommendatAlgorithms Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl GroupLens Research Group/Army HPC Research Center Department of Computer Science and Engineering University of Minnesota

[19] Using Content-Based Filtering for Recommendation1 Robin van Meteren1 and Maarten van Someren2 University of Amsterdam

[20] Yehuda Koren, Yahoo Research Robert Bell and Chris Volinsky MATRIX FACTORIZATION TECHNIQUES FOR RECOMMENDER SYSTEMS Labs—Research

[21] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber and Rong Pan *Large-scale Parallel Collaborative Filtering for the Netflix Prize.*

[22] Jenny A. Baglivo *Mathematica Laboratories for Mathematical Statistics, Emphasizing Simulation and Computer Intensive Methods*

[23] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

[24] Ojeda-Zapata, Julio (1997-09-15). "New Site Personalizes Movie Reviews". St. Paul Pioneer Press. p. 3E.

[25] Sam Skalicky, Christopher Wood,Rochester Institute of Technology,Marcin Łukowiak, Matthew Ryan Rochester, NY High Level Synthesis: Where Are We? A Case Study on Matrix Multiplication

[26] Building Zynq Accelerators with Vivado High Level Synthesis Stephen Neuendorffer and Fernando Martinez-Vallina FPGA Feb 11, 2013

[27] AXI4-Stream Accelerator Adapter v2.1 LogiCORE IP Product Guide

[28] PERFORMANCE COMPARISON OF FPGA, GPU AND CPU IN IMAGE PROCESSING Shuichi Asano, Tsutomu Maruyama and Yoshiki Yamaguchi Systems and Information Engineering, University of Tsukuba 1-1-1 Ten-ou-dai Tsukuba Ibaraki 305-8573 JAPAN