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

Σχεδιασμός και Βελτιστοποίηση επιταχυντή βασισμένου σε FPGA για τον αλγόριθμο Πολλαπλασιασμού Αραιού Πίνακα με Διάνυσμα

Διπλωματικη Εργασια

ΤΑΣΟΥ ΙΩΑΝΝΑ

Επιβλέπων: Γκούμας Γεώργιος
Αναπληρωτής Καθηγητής

Αθήνα, Απρίλιος 2023

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογιας Πληροφορικής και Υπολογιστικων Συστηματων

Σχεδιασμός και Βελτιστοποίηση επιταχυντή βασισμένου σε FPGA για τον αλγόριθμο Πολλαπλασιασμού Αραιού Πίνακα με Διάνυσμα

Διπλωματικη Εργασια

ΤΑΣΟΥ ΙΩΑΝΝΑ

Επιβλέπων: Γκούμας Γεώργιος
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 10 Απριλίου 2023.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Γκούμας Γεώργιος	Κοζύρης Νεκτάριος	Πνευματικάτος Διονύσιος
Αναπληρωτής Καθηγητής	Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Απρίλιος 2023

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογιας Πληροφορικής και Υπολογιστικων Συστηματων

(Υπογραφή)

.....
ΤΑΣΟΥ ΙΩΑΝΝΑ
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογοστών Ε.Μ.Π.

Περίληψη

Ο Πολλαπλασιασμός Αραιού Πίνακα με Διάνυσμα (SpMV) αποτελεί έναν ευρέως χρησιμοποιούμενο υπολογιστικό πυρήνα σε πολλές επιστημονικές εφαρμογές. Με την αύξηση του μεγέθους και της πολυπλοκότητας των δεδομένων, ο SpMV αποτελεί πλέον ένα υπολογιστικά βαρύ πρόβλημα. Τα Field Programmable Gate Arrays (FPGAs) έχουν κερδίσει αρκετό έδαφος ως μία εναλλακτική των κλασσικών CPUs και GPUs εξαιτίας της υψηλής παραλληλοποίησης των υπολογισμών, της δυνατότητας επαναπρογραμματισμού τους και της ενεργειακής τους αποδοτικότητας.

Στόχος της διπλωματικής εργασίας είναι η μελέτη της υλοποίησης του Πολλαπλασιασμού Αραιού Πίνακα με Διάνυσμα σε FPGA, η αξιολόγηση της επίδοσης της και της ενεργειακής της αποδοτικότητας συγκριτικά με υλοποιήσεις σε CPU και GPU, και η σύγκριση με την Vitis Sparse Library για τον SpMV στο Xilinx Alveo U280 FPGA. Στα πλαίσια της εργασίας θα εξετάσουμε διαφορετικές σχεδιαστικές επιλογές για την υλοποίηση του SpMV, με χρήση της παράλληλης επεξεργασίας και την βελτιστοποίηση της κατανομής της μνήμης. Παράλληλα, θα εξετάσουμε τα πλεονεκτήματα και τα μειονεκτήματα της υλοποίησης του SpMV σε FPGA όσον αφορά την αξιοποίηση των πόρων, την επίδοση, την κλιμακωσιμότητα και την ενεργειακή κατανάλωση.

Λέξεις Κλειδιά

Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών (FPGA), Πολλαπλασιασμός Αραιού Πίνακα με Διάνυσμα (SpMV), Εργαλεία Σύνθεσης Υψηλού Επιπέδου (HLS), Υπολογιστική Υψηλών Επιδόσεων (HPC), ZCU102, Alveo U280

Abstract

Sparse matrix-vector multiplication (SpMV) is a fundamental operation in many scientific and engineering applications. As the size of matrices and vectors used in these applications increases, SpMV becomes a computationally intensive task. Field Programmable Gate Arrays (FPGAs) have gained popularity as an alternative to traditional CPUs and GPUs due to their high parallelism, reconfigurability, and energy efficiency.

This diploma thesis aims to explore the implementation of SpMV on an FPGA and evaluate its performance compared to CPU- and GPU-based implementations, as well as a different FPGA-based implementation. This thesis will investigate different design choices for implementing SpMV on an FPGA, including the use of parallel processing and memory optimization techniques. Additionally, this thesis will explore the challenges and trade-offs involved in designing SpMV on an FPGA, such as resource utilization, performance, scalability and power consumption.

Keywords

Field Programmable Gate Array (FPGA), Sparse Matrix-Vector Multiplication (SpMV), High Level Synthesis Tools (HLS), High Performance Computing (HPC), ZCU102, Alveo U280

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον καθηγητή κ. Γεώργιο Γκούμα για την επίβλεψη αυτής της διπλωματικής εργασίας και για την ευκαιρία που μου έδωσε να την εκπονήσω στο Εργαστήριο Υπολογιστικών Συστημάτων. Επίσης θα ήθελα να ευχαριστήσω ιδιαίτερα τον Παναγιώτη Μπάκο για την καθοδήγησή του και την βοήθεια που μου προσέφερε κατά την διάρκεια της διπλωματικής. Τέλος θα ήθελα να ευχαριστήσω τους γονείς μου για την καθοδήγηση και την ηθική συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια.

Αθήνα, Απρίλιος 2023

Τάσου Ιωάννα

Περιεχόμενα

Κατάλογος Εικόνων

Κεφάλαιο 1

Εισαγωγή

Ο Πολλαπλασιασμός Αραιού Πίνακα με Διάνυσμα (SpMV) αποτελεί έναν ευρέως χρησιμοποιούμενο υπολογιστικό πυρήνα σε πολλές επιστημονικές εφαρμογές όπως αριθμητική ανάλυση, ανάλυση δεδομένων, θεωρία γράφων και επίλυση διαφορικών εξισώσεων. Με την αύξηση του μεγέθους και της πολυπλοκότητας των δεδομένων, ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα απαιτεί υπολογιστική δύναμη η οποία ξεπρνά τις δυνατότητες των παραδοσιακών CPU συστημάτων.

Παλαιότερα, ο προγραμματισμός των FPGAs αποτελούσε μία χρονοβόρα και δύσκολη διαδικασία καθώς απαιτούσε τον σχεδιασμό σε επίπεδο μεταφοράς καταχωρητών Register Transfer Level (RTL) και άρα τη συγγραφή κώδικα χαμηλού επιπέδου γλώσσας περιγραφής υλικού. Αυτό καθιστούσε επίπονο τον προγραμματισμό και δύσκολη την μεταφορά του κώδικα σε διαφορετικά FPGA.

Τα τελευταία χρόνια, όμως, τα Field Programmable Gate Arrays (FPGAs) έχουν κερδίσει αρκετό έδαφος στην επιτάχυνση εφαρμογών μεγάλης έντασης δεδομένων λόγω της ανάπτυξης των εργαλείων High Level Synthesis (HLS). Έτσι προσαρμόζοντας την αρχιτεκτονική τους στις ανάγκες της εκάστοτε εφαρμογής και λόγω των δυνατοτήτων τους στην παράλληλη επεξεργασία και στον επαναπρογραμματισμό τους, τα FPGA παρέχουν μια αποδοτική εναλλακτική σε σχέση με τα παραδοσιακά επεξεργαστικά συστήματα που βασίζονται σε CPU.

1.1 Αντικείμενο της διπλωματικής

Στα πλαίσια αυτής της διπλωματικής θα υλοποιήσουμε τον αλγόριθμο πολλαπλασιασμού αραιού πίνακα με διάνυσμα (Sparse Matrix-Vector Multiplication - SPMV) στο FPGA Alveo™ U280 Data Center accelerator card της Xilinx, έχοντας ως βάση μια αντίστοιχη υλοποίηση του προβλήματος αυτού στο Zynq Ultrascale+ MP-Soc(Multi-processor System-on-chip) ZCU102 της Xilinx. Επιχειρούμε, δηλαδή, την μεταφορά (migration) του κώδικα μεταξύ δύο FPGA αρκετά διαφορετικής τεχνολογίας, διατηρώντας όσα χαρακτηριστικά του κώδικα ενισχύουν την απόδοση όπως η παραλληλοποίηση των υπολογισμών και η αναπαράσταση των δεδομένων. Όλα αυτά σε ένα πρόβλημα το οποίο αποτελεί βασικό υπολογιστικό πυρήνα σε ένα ευρύ πεδίο επιστημονικών εφαρμογών αλλά ταυτόχρονα αποτελεί ένα βαρύ υπολογιστικά αλγόριθμό λόγω του πλήθους των διαφορετικών προσβάσεων στην μνήμη. Τέλος, αξιολογούμε την επίδοση της υλοποιησής μας συγκρίνοντας την με μία υλοποίηση σε CPU, μία σε GPU, καθώς και με την βιβλιοθήκη της Xilinx για τον SPMV.

1.2 Οργάνωση του τόμου

Η διπλωματική οργανώνεται ως εξής:

Το **Κεφάλαιο 2** αποτελεί το θεωρητικό υπόβαθρο της διπλωματικής και περιλαμβάνει αρχικά την περιγραφή της αρχιτεκτονικής και του τρόπου λειτουργίας ενος FPGA καθώς και την σύγκριση των FPGA των δύο υλοποιήσεων. Στην συνέχεια αναλύεται ο αλγόριθμος SPMV και ο τρόπος διαχείρισης και αποθήκευσης του αραιού πίνακα. Τέλος παρουσιάζεται η αρχική υλοποίηση του ZCU102 FPGA στην οποία βασιστήκαμε.

Στο **Κεφάλαιο 3** παρουσιάζονται όλα τα στάδια της υλοποίησης που προτείνουμε με επιμέρους αποτελέσματα. Στο **Κεφάλαιο 4** εκτελείται η σύγκριση της υλοποιησής μας με μία υλοποίηση σε CPU, μία σε GPU, καθώς και με την βιβλιοθήκη της Xilinx για τον SPMV και η αξιολόγηση της απόδοσης και της ενεργειακής της επίδοσης.

Το **Κεφάλαιο 5** περιλαμβάνει ιδέες για βελτίωση και για μελλοντικές επεκτάσεις της υλοποίησης ενώ το **Κεφάλαιο 6** αποτελεί τον επίλογο με τα συμπεράσματα της εργασίας.

Μέρος I

Θεωρητικό Μέρος

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

Στο κεφάλαιο αυτό παρουσιάζονται αρχικά κάποια βασικά στοιχεία που αφορούν την αρχιτεκτονική καθώς και τον προγραμματισμό των FPGA και στην συνέχεια αναλύεται το πρόβλημα του πολλαπλασιασμού αραιού πίνακα με διάνυσμα (SPMV).

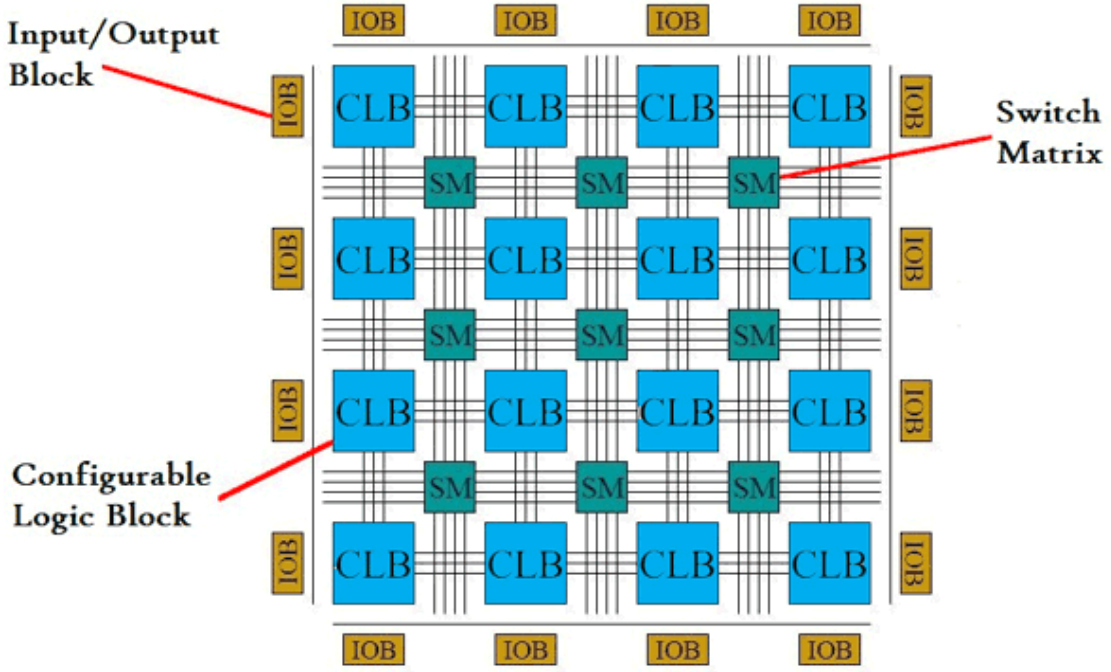
2.1 Field Programmable Gate Arrays (FPGA)

2.1.1 Εισαγωγή στα FPGA

Το FPGA ή Field Programmable Gate Array ή Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών είναι τύπος ολοκληρωμένου κυκλώματος γενικής χρήσης το οποίο είναι σχεδιασμένο έτσι ώστε να μπορεί να προγραμματιστεί μετά την κατασκευή του. Αυτό το χαρακτηριστικό τα διαφοροποιεί από τα Application Specific Integrated Circuits (ASIC), τα οποία είναι κατασκευασμένα για εκτέλεση προκαθορισμένων εφαρμογών. Ο προγραμματισμός των FPGA γίνεται με την χρήση γλωσσών περιγραφής υλικού (Hardware Description Languages (HDL)) παρόμοιων με αυτές που χρησιμοποιούνται στα Application-Specific Integrated Circuits(ASIC).

Τα FPGA αποτελούν διατάξεις ημιαγωγών που κατασκευάζονται γύρω από έναν πίνακα από μπλοκ προγραμματιζόμενης λογικής Configurable Logic Blocks (CLBs), τα οποία επικοινωνούν μεταξύ τους μέσω προγραμματιζόμενων διασυνδέσεων Programmable Interconnects. Κατά τον προγραμματισμό του FPGA, ο οποίος γίνεται πάντοτε ενώ αυτό είναι τοποθετημένο στο τυπωμένο κύκλωμα, ενεργοποιούνται οι επιθυμητές λειτουργίες και διασυνδέονται μεταξύ τους έτσι ώστε το FPGA να συμπεριφέρεται ως ολοκληρωμένο κύκλωμα με συγκεκριμένη λειτουργία.

Όλα τα FPGA αποτελούνται από τα εξής βασικά δομικά στοιχεία: CLBs, Clock Circuitry, I/O Blocks, Programmable Interconnects. Παλαιότερα τα FPGA επιλέγονταν για λιγότερο απαιτητικές εφαρμογές, τα σύγχρονα όμως FPGA περιέχουν και εξειδικευμένα block που επιτελούν λειτουργίες μνήμης ή πιο απαιτητικούς υπολογισμούς αυξάνοντας την ταχύτητα και την επίδοση τους, καθιστώντας τα δελεαστική πρόταση για οποιαδήποτε αρχιτεκτονική. Πιο αναλυτικά τα βασικά δομικά στοιχεία ενός σύγχρονου FPGA είναι τα εξής:



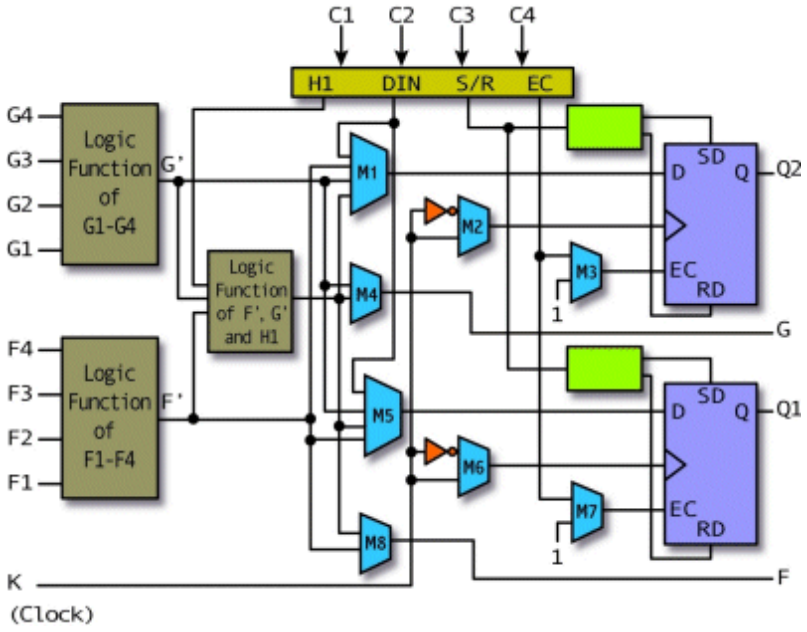
Εικόνα 2.1: *Δομή ενός FPGA [1]*

- **Configurable Logic Blocks (CLBs)**

  Περιέχουν την λογική του FPGA. Μπορούν να προγραμματιστούν για να εκτελούν πολύπλοκες πράξεις συνδυαστικής λογικής ή και να λειτουργήσουν ως απλές λογικές πύλες. Κάθε block περιέχει PAM για την δημιουργία συναρτήσεων συνδυαστικής λογικής, γνωστών και ως lookup tables (LUTs) ή πίνακες αλήθειας της έκαστοτε συνάρτησης που δημιουργείται. Ακόμη αποτελείται από flip-flops ή ακόμη και πιο σύνθετα στοιχεία μνήμης και από πολυπλέκτες για την εσωτερική και εξωτερική δρομολόγηση της λογικής του CLB.

- **Clock Circuitry**

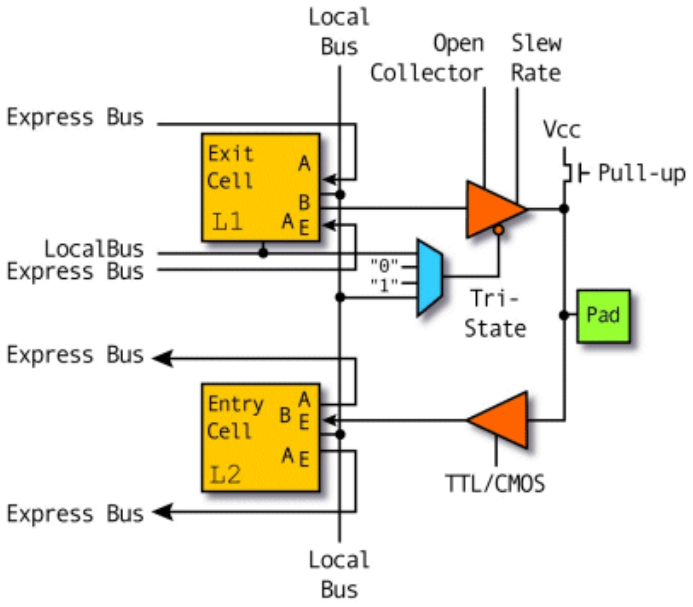
  Στο chip του FPGA υπάρχουν είδικά I/O Blocks με clock buffers υψηλής ακρίβειας, γνωστά ως clock drivers, που μεταφέρουν τα σήματα του ρολογιού στο chip καθιστώντας εφικτό τον συγχρονισμό του FPGA.



Εικόνα 2.2: *Δομή ενός Configurable Logic Block [2]*

- **Configurable I/O Blocks**

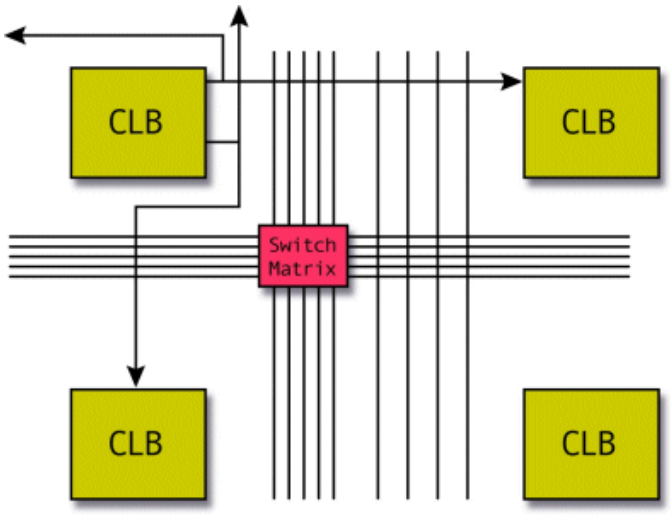
  Η επικοινωνία του FPGA με το εξωτερικό του περιβάλλον γίνεται μέσω προγραμματιζόμενων I/O Blocks.



Εικόνα 2.3: *Δομή ενός I/O Block [2]*

- **Programmable Interconnects**

  Μέσω αυτών επιτυγχάνεται η επικοινωνία των CLBs με τα I/O Blocks αλλά και μεταξύ τους και έτσι ανάλογα με τον τρόπο διασύνδεσης τους υλοποιούνται πιο περίπλοκες αρχιτεκτονικές και είναι εφικτός ο προγραμματισμός του FPGA για διαφορετικές εφαρμογές.



Εικόνα 2.4: *Διασύνδεση του FPGA [2]*

- **Digital Signal Processors (DSP)**

  Αποτελεί μία Αριθμητική και Λογική Μονάδα ALU που περιέχει αθροιστές, πολλαπλασιαστές και αφαιρέτες. Μπορεί να εκτελέσει απαιτητικές υπολογιστικά λειτουργίες αυξάνοντας έτσι την επίδοση του FPGA.

- **BlockRAM/UltraRAM**

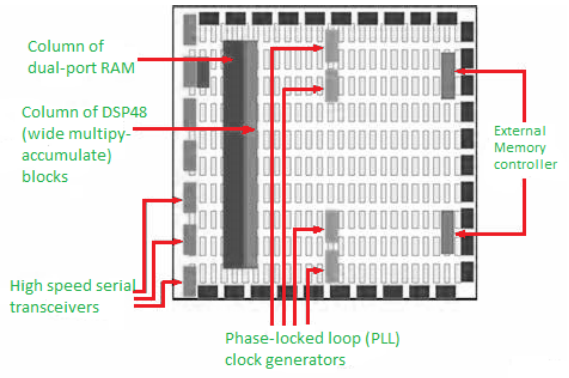
  Πρόκειται για μία μνήμη RAM dual port, δύο θυρών, η οποία είναι ενσωματωμένη στο chip και επιτρέπει την αποθήκευση σχετικά μεγάλων όγκων δεδομένων στα οποία απαιτείται γρηγορότερη πρόσβαση από την εφαρμογή. Οι δύο θύρες επιτρέπουν την ταυτόχρονη πρόσβαση σε δύο διαφορετικές περιοχές της BRAM στον ίδιο κύκλο του ρολογιού.

- **High Bandwidth Memory (HBM)**

  Αρκετά σύγχρονα FPGA περιέχουν High Bandwidth Memories οι οποίες επιτρέπουν ταχύτερη μεταφορά δεδομένων με χαμηλότερη ενεργειακή κατανάλωση. Για την δημιουργία της στοιβάζονται αρκετά layers DDR4 μνημών επιτυγχάνοντας έτσι την μείωση του χρόνου μεταφοράς των δεδομένων.

- **Θύρες I/O υψηλής ταχύτητας**

  Αποτελούν διασυνδέσεις που επιτυγχάνουν την ταχύτερη μεταφορά δεδομένων μεταξύ του chip και του εξωτερικού κόσμου.



Εικόνα 2.5: *Αρχιτεκτονική ενός σύγχρονου FPGA*

2.1.2 Σύγκριση embedded και non-embedded FPGA αρχιτεκτονικών

Τα FPGA χωρίζονται σε δύο βασικές κατηγορίες, ανάλογα με τον τρόπο διασύνδεσης τους με τον επεξεργαστή που μετεφέρει τα δεδομένα: embedded και non-embedded. Μεταξύ των δύο αυτών τεχνολογιών υπάρχουν οι εξής διαφορές:

- **Αρχιτεκτονική**. Τα non-embedded FPGAs ή αλλιώς accelerator FPGAs έχουν την κλασσική αρχιτεκτονική κατά την οποία επεξεργαστής και FPGA λειτουργούν ανεξάρτητα σε διαφορετικά chip και επικοινωνούν για την μεταφορά δεδομένων από και προς το FPGA καθώς και για τον προγραμματισμό του. Τα embedded FPGA ή αλλιώς System on a chip (SoC) αναπτύχθηκαν με στόχο την απλοποίηση αυτής της επικοινωνίας. Στο chip του FPGA (Programmable Logic (PL)) προστέθηκε επεξεργαστής καθως και περιφερειακές συσκευές(Processing System (PS)) καθιστώντας το ανεξάρτητο σε λειτουργία.

- **Επίδοση και Ενέργεια**. Τα embedded FPGAs είναι πιο μικρά για να χωράνε στο chip και καταναλώνουν λιγότερη ενέργεια. Τα accelerator FPGA είναι μεγαλύτερα σε μέγεθος, καταναλώνουν περισσότερη ενέργεια αλλά έχουν περισσότερους πόρους το οποίο οδηγεί σε μεγαλύτερες υπολογιστικές δυνατότητες και επίδοση.

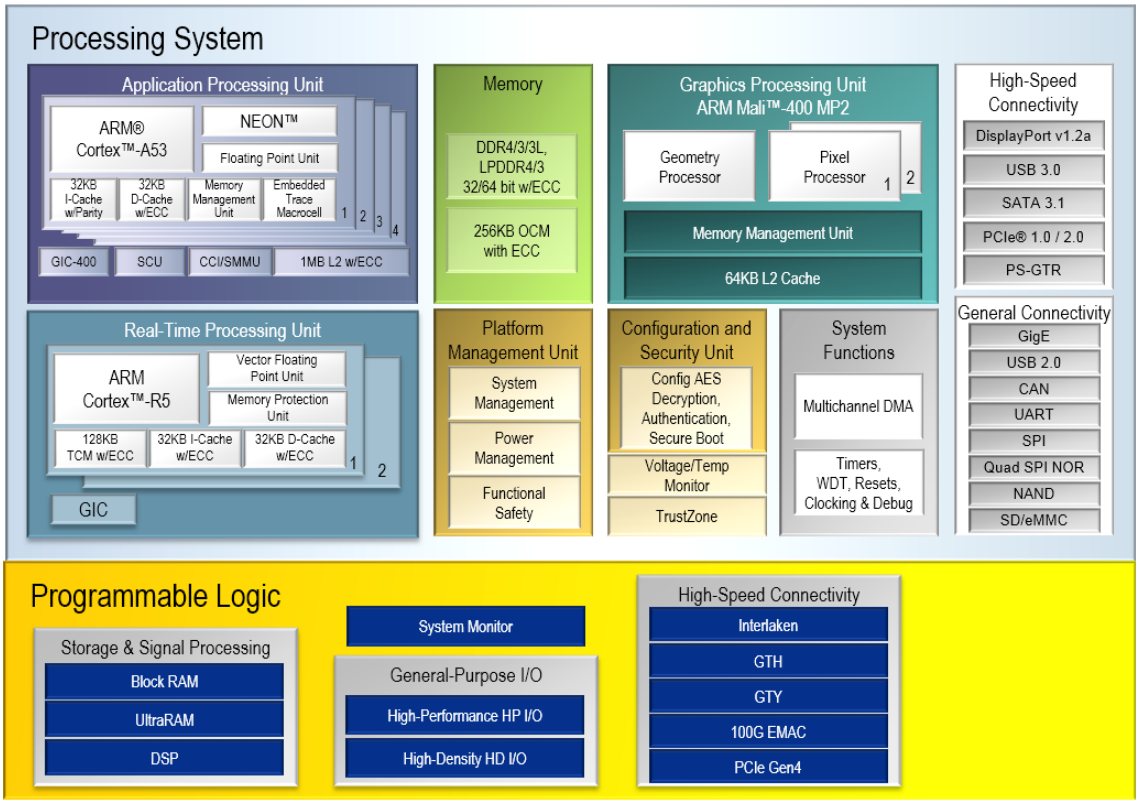
- **Προγραμματισμός**. Για τον προγραμματισμό των embedded FPGAs συνήθως απαιτούνται πιο εξειδεικευμένα εργαλεία και τεχνικές ανάλογα με την SoC πλατφόρμα. Αντίθετα, τα accelerator FPGA είναι πιο ευέλικτα στον προγραμματισμό και χρησιμοποιούν πιο συνηθισμένα εργαλεία και αρά καθστούν πιο εύκολη την μεταφορά του κώδικα και σε άλλα accelerator FPGA.

Συνολικά, η επιλογή μεταξύ ενσωματωμένου(embedded) FPGA και μη ενσωματωμένου(non-embedded) FPGA εξαρτάται από τη συγκεκριμένη εφαρμογή και τις απαιτήσεις της. Τα ενσωματωμένα FPGA είναι κατάλληλα για εφαρμογές όπου είναι σημαντική η χαμηλή κατανάλωση ισχύος και το μικρό μέγεθος, και όπου το FPGA χρειάζεται να επικοινωνεί αρκετά με άλλα εξαρτήματα στο σύστημα. Τα μη ενσωματωμένα FPGA μπορεί να είναι μια καλύτερη επιλογή για εφαρμογές όπου απαιτείται μεγαλύτερη ευελιξία, προσαρμογή και ισχύς, και όπου το FPGA μπορεί να λειτουργήσει ως αυτόνομη συσκευή.

2.1.3 Συγκριση των ZCU102 και Alveo U280

Κύριο θέμα αυτής της διπλωματικής είναι η μεταφορά της υλοποίησης από το ZCU102 Xilinx FPGA στην Alveo U280 Data Center Accelerator Card. Αξίζει, λοιπόν, να παρουσιάσουμε αναλυτικά την αρχιτεκτονική των δύο αυτών FPGA και να εντοπίσουμε τις διαφορές τους.

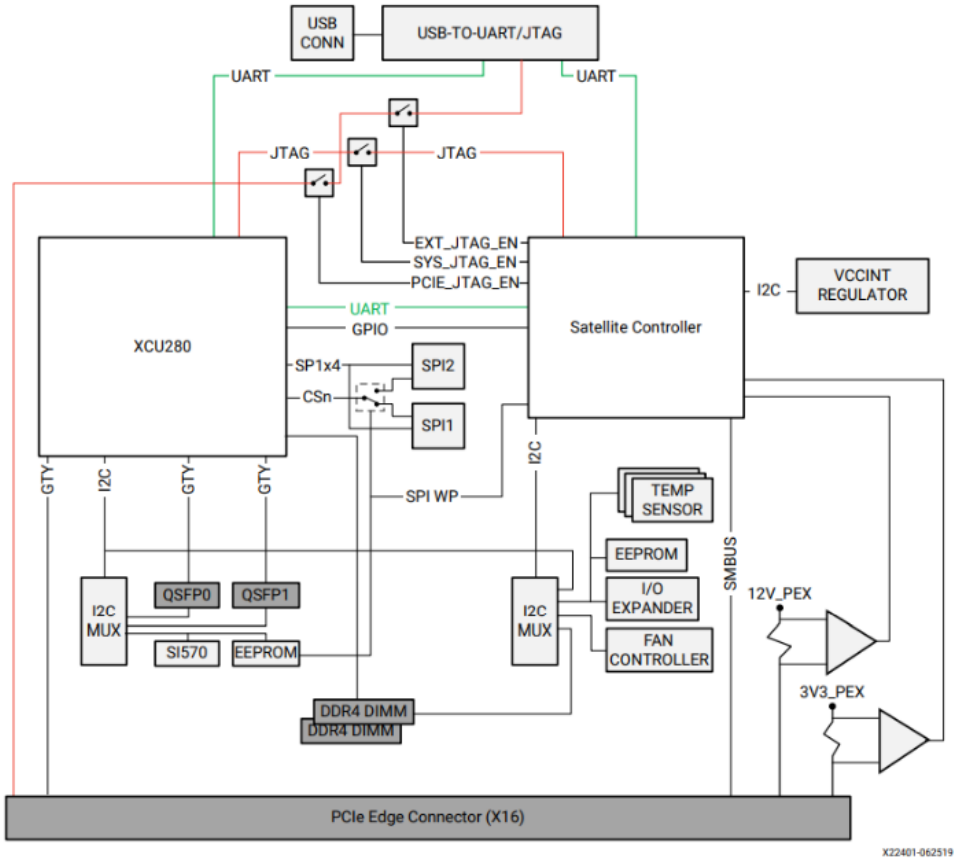
- **Zynq Ultrascale+ MP-Soc(Multi-processor System-on-chip) ZCU102**
  Η υλοποίηση στην οποία βασίστηκε αυτή η διπλωματική αφορούσε το Zynq Ultrascale+ MP-Soc(Multi-processor System-on-chip) ZCU102 της Xilinx, το οποίο αποτελεί ένα FPGA embedded αρχιτεκτονικής. Στο PS συνδυάζει έναν τετρα-πύρηνο ARM Cortex-A53 επεξεργαστή με έναν δι-πύρηνο Cortex-R5 επεξεργαστή. Προσφέρει 4GB για το σύστημα επεξεργασίας (PS) και 4GB για την προγραμματίσιμη λογική(PL) και Gigabit Ethernet, USB 3.0, DisplayPort, and HDMI ως επιλογές συνδεσιμότητας. Στην προγραμματίσιμη λογική το FPGA που χρησιμοποιείται είναι το ZU9EG της Xilinx και περιλαμβάνονται 32.1Mb BlockRAM, 274080 LUTs, 2520 DSPs, 548160 Flip-Flops. Υποστηρίζει διάφορα frameworks και προγραμματιστικά εργαλεία όπως Vivado Design Suite, Petalinux και SDSoC.



Εικόνα 2.6: *Το Block Diagram του ZCU102*

- **Alveo U280 Data Center Accelerator Card**
  Σε αντίθεση με το ZCU102, η Alveo U280 δεν περιέχει embedded επεξεργαστή αλλά το host κομμάτι του κώδικα που περιλαμβάνει το διάβασμα των αρχείων, το "πακετάρισμα" των δεδομένων και τον έλεγχο της ορθότητας των αποτελεσμάτων εκτελείται σε

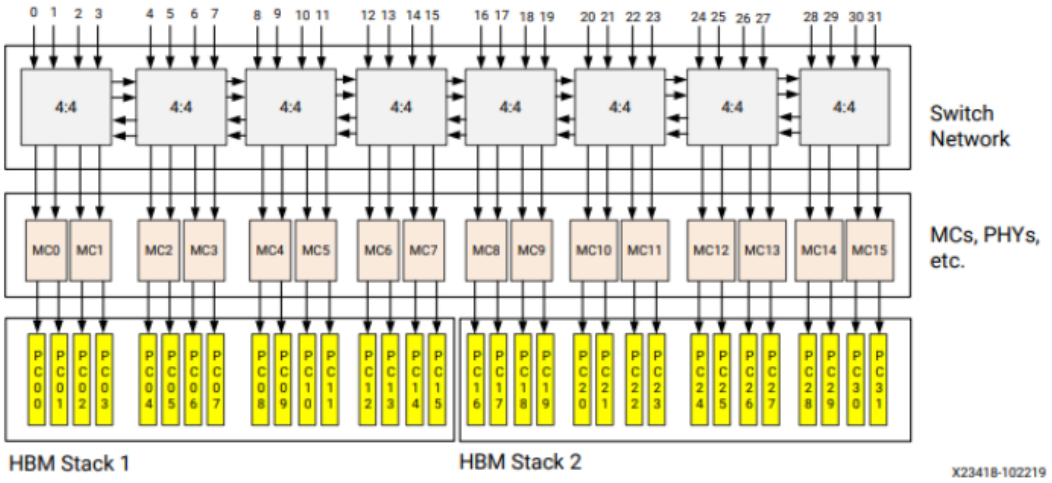
επεξεργαστή εκτός του FPGA. Όπως βλέπουμε και στο παρακάτω διάγραμμα η Alveo U280 accelerator card εμπεριέχει το 6 nm UltraScale+™ XCU280 FPGA. Έκτος αυτού περιέχει 2 ανεξάρτητες dual-bank DDR4 μνήμες των 16GB, δηλαδή 32GB DDR4 global μνήμης. Υποστηρίζει διάφορα frameworks και προγραμματιστικά εργαλεία όπως Vivado Design Suite, Xilinx Vitis and OpenCL. Ένα βασικό πλεονέκτημα αυτής της



Εικόνα 2.7: *Το Block Diagram του Alveo U280 [3]*

συσκευής ότι περιλαμβάνει δύο High Bandwidth Memory (HBM) stacks των 4GB. Είναι ένα είδος μνήμης σχεδιασμένο να προσφέρει υψηλό εύρος ζωνης(bandwidth) και χαμηλή ενεργειακή κατανάλωση. Η μνήμη HBM (High Bandwidth Memory) χρησιμοποιεί στοιβαγμένα "memory dies", τα οποία ενωμένα μέσω through-silicon vias (TSVs) παρέχουν μια υψηλής ευρυζωνικότητας διασύνδεση μεταξύ της μνήμης και του επεξεργαστή. Αυτή η αρχιτεκτονική επιτρέπει πολύ υψηλότερες ταχύτητες μεταφοράς δεδομένων, μεγάλη χωρητικότητα, μικρό latency και χαμηλότερη κατανάλωση ενέργειας σε σύγκριση με τις παραδοσιακές τεχνολογίες μνήμης.

Το Xilinx UltraScale+ FPGA της Alveo U280 Accelerator Card είναι διαχωρισμένο σε πολλαπλές SLRs (Super Logic Regions).Η αρχιτεκτονική SLR επιτρέπει τη βελτιωμένη απόδοση και την ευκολότερη υλοποίηση σχεδίων, απομονώνοντας διαφορετικά τμήματα του FPGA και παρέχοντας ένα αποκλειστικό δίκτυο διασύνδεσης (interconnect network) για την επικοινωνία μεταξύ των SLR. Κάθε SLR στο U280 περιλαμβάνει πολλούς



Εικόνα 2.8: *Το High Level Diagram των δύο HBM Stacks του Alveo U280 [3]*

πόρους, όπως προγραμματιζόμενη λογική, DSPs, BRAM, DDR4 μνήμες και επιπλέον η SLR 0 συνδέεται σε 2 High Bandwidth Memory Stacks. Καθίσταται, έτσι, δυνατή η ανάθεση των διάφορων υπολογιστικών διεργασιών σε διαφορετικές περιοχές του FPGA ανάλογα με τους πόρους που απαιτούνται για αυτές. Στην Εικόνα 2.9 βλέπουμε τον διαχωρισμό των πόρων του FPGA ανά τις τρεις SLR.

Area	SLR0	SLR1	SLR2
General Information			
SLR Description	Shared by dynamic and static region resources.	Shared by dynamic and static region resources.	Shared by dynamic and static region resources.
Dynamic Region Pblock Name	pfm_top_i_dynamic_region_pblock_dynamic_SLR0	pfm_top_i_dynamic_region_pblock_dynamic_SLR1	pfm_top_i_dynamic_region_pblock_dynamic_SLR2
Global Memory Resources Available in Dynamic Region			
System Port Name and Description	DDR[0]; (16 GB DDR4)	DDR[1]; (16 GB DDR4)	PLRAM[4:5]; (2 x up to 4 MB SRAM) Default of 128 KB
	HBM[0:31]; (32 x 256 MB HBM2 PC)	PLRAM[2:3]; (2 x up to 4 MB SRAM) Default of 128 KB	
	PLRAM[0:1]; (2x up to 4 MB SRAM) Default of 128 KB		
Approximate Available Fabric Resources in Dynamic Region			
CLB LUT	355K	340K	370K
CLB Register	725K	675K	734K
Block RAM Tile	490	490	510
UltraRAM	320	320	320
DSP	2733	2877	2880

Εικόνα 2.9: *Κατανομή των Πόρων του FPGA στις SLRs [3]*

2.1.4 Προγραμματισμός FPGA

Ο προγραμματισμός FPGA αν και έχει κάποιες ομοιότητες με τον κλασσικό προγραμματισμό σε CPU παρουσιάζει αρκετές διαφορές. Η βασική διαφορά έγκειται στο ότι στο FPGA επαναδιατάσσεται το κύκλωμα, δημιουργώντας υπολογιστικές μονάδες, με βάση τις ανάγκες της εφαρμογής ενώ στον κλασσικό προγραμματισμό η αρχιτεκτονική του υλικού είναι σταθερή και το μόνο που μπορεί να επηρεάσει την επίδοση μιας εφαρμογής είναι αλλαγές στον κώδικα αυτής.

Η βασική παραλληλοποίηση που εφαρμόζεται σε όλα τα προγράμματα σε έναν κλασσικό επεξεργαστή είναι ουσιαστικά το Pipelining. Κάθε εντολή του προγράμματος χωρίζεται σε μικρότερα τμήματα τα οποία εκετελούνται απο διαφορετικές υπολογιστικές μονάδες. Σε ένα κλασσικό pipelining υπάρχουν τα εξής στάδια εκτέλεσης μιας εντολής: IF(Instruction Fetch), ID(Instruction Decode), EXE(Execution), MEM(Memory Operations), WR(Write back).

	Clock Number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

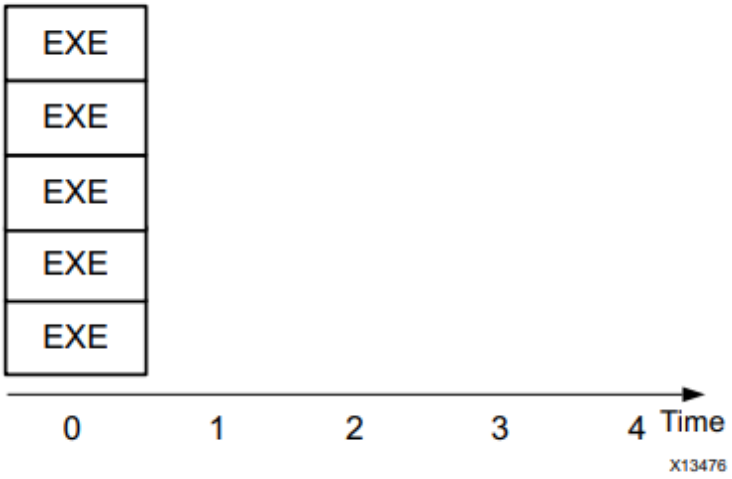
Εικόνα 2.10: *Pipelining σε κλασσικό επεξεργαστή*

Όπως βλέπουμε και στην εικόνα ένα απλό Pipelining σε έναν απλό επεξεργαστή μειώνει τους κύκλους εκτέλεσης από 25 (για 5 εντολές αν εκτελούνταν σειριακά) σε 9. Ανάλογα με τους πόρους του κάθε επεξεργαστή είναι δυνατόν να επεκταθεί επιπλέον η παραλληλοποίηση των εργασιών αλλά ταυτόχρονα πρέπει να λαμβάνουμε υπόψιν μας και τις εξαρτήσεις μεταξύ των εντολών που εισάγουν καθυστερήσεις.

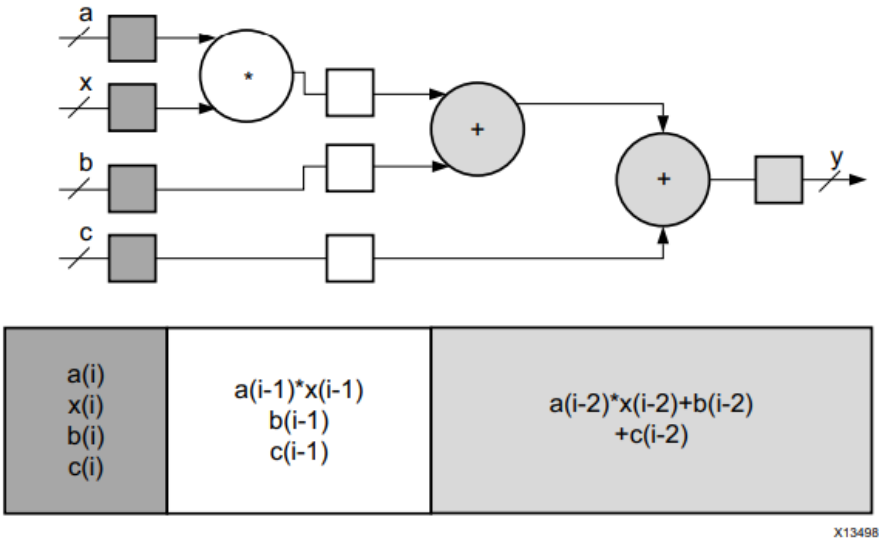
Αντίθετα, το FPGA προγραμματίζεται συγκερκιμένα για την εκτέλεση της εκάστοτε εφαρμογής. Οπότε δημιουργούνται ανεξάρτητα λογικά blocks τα οποία μπορούν να εκτελούν ταυτόχρονα τμήματα του κώδικα εφόσον αυτά είναι ανεξάρτητα μεταξύ τους. Οπότε σε ένα FPGA δεν ιφίσταται η διάσπαση της κάθε εντολής όπως αναλύσαμε παραπάνω αλλά οι διάφορες διεργασίες εκτελούνται παράλληλα σε ανεξάρτητα κυκλώματα.

Βέβαια και στα FPGAs εφαρμόζεται pipelining αλλά γίνεται διασπώντας τις πράξεις σε μικρότερα τμήματα. Στην παρακάτω εικόνα βλέπουμε το πως διασπάται η εκτέλεση της εντολής $y = ax + b + c$ σε έναν πολλαπλασιασμό και δύο προσθέσεις. Έτσι όταν η παραπάνω πράξη εκτελείται επαναληπτικά αφού έχει εκτελεστεί ο πολλαπλασιασμός μίας επανάληψης i, μετά εκτελείται ο πολλαπλασιασμός της i+1 επανάληψης όσο εκτελείται το πρώτο άθροισμα της i επανάληψης κ.ο.κ.

Η ίδια λογική ακολουθείται και σε πιο υψηλό επίπεδα του σχεδίου. Ο μεταγλωτιστής αναγνωρίζει εξαρτήσεις μεταξύ συναρτήσεων και με βάση αυτές δημιουργεί μια ροή δεδομένων (dataflow), μεταξύ των συναρτήσεων.



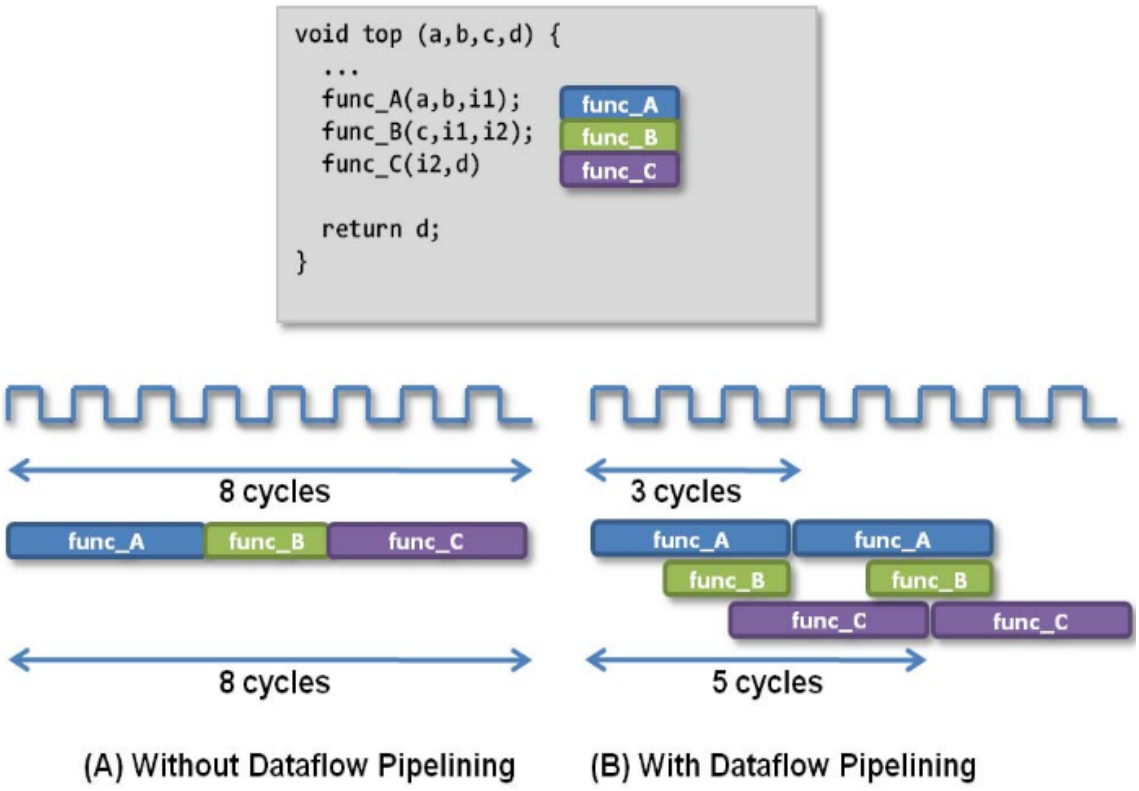
Εικόνα 2.11: *Παραλληλοποίηση σε FPGA [4]*



Εικόνα 2.12: *Pipelining σε FPGA [4]*

Τέλος, σε έναν κλασσικό επεξεργαστή τα δεδομένα βρίσκονται αποθηκευμένα σε αργές μνήμες DDR και, ανάλογα με τις προσβάσεις που κάνει το κάθε πρόγραμμα και την σειρά που τις έχει τοποθετήσει ο προγραμματιστής, είναι δυνατή η εκμετάλλευση γρήγορων μνημών (cache). Σε ένα FPGA ο προγραμματιστής καθορίζει κατά το build του κώδικα την κατανομή των δεδομένων στις μνήμες του FPGA και πρέπει να διαχειριστεί σωστά την χωρητικότητα και τις δυνατότητες τους.

Με όλους τους παραπάνω τρόπους το FPGA επιτυγχάνει πολύ μεγαλύτερη παραλληλοποίηση των πράξεων και έτσι μπορεί προγραμματιζόμενο σωστά να καταφέρει αρκετά υψηλότερες επιδόσεις από έναν κλασσικό επεξεργαστή σε συγκεκριμένες εφαρμογές. Έτσι προγραμματιζόμενο σε συχνότητα ρολογιού πολύ χαμηλότερη από έναν κλασσικό επεξεργαστή (έστω 2GHz ο επεξεργαστής, 500MHz ένα μέσο FPGA) μειώνει την ενεργειακή κατανάλωση αυξάνωντας την απόδοση.



Εικόνα 2.13: *Dataflow σϵ FPGA [4]*

2.1.5 High Level Synthesis Tools

Ο προγραμματισμός FPGA μπορεί να γίνει με τη χρήση είτε του σχεδιασμού σε επίπεδο μεταφοράς καταχωρητών Register Transfer Level (RTL) είτε της υψηλού επιπέδου σύνθεσης High Level Synthesis (HLS). Ο σχεδιασμός RTL περιλαμβάνει τη συγγραφή κώδικα χαμηλού επιπέδου γλώσσας περιγραφής υλικού Hardware Description Language(HDL) που περιγράφει την υλικολογική του σχεδίαση. Οι σχεδιασμοί RTL απαιτούν βαθιά κατανόηση του υλικού FPGA, συμπεριλαμβανομένων των πόρων, της αρχιτεκτονικής και των χρονικών περιορισμών της συγκεκριμένης συσκευής. Ο HLS σχεδιασμός, από την άλλη, περιλαμβάνει τη συγγραφή κώδικα υψηλού επιπέδου που περιγράφει τη λειτουργικότητα του σχεδίου. Το εργαλείο HLS δημιουργεί αυτόματα κώδικα RTL που υλοποιεί το σχέδιο στο FPGA, μεταφράζοντας κώδικα που έχει γραφτεί σε υψηλού επιπέδου γλώσσα (C,C++). Αυτό επιτρέπει πιο αφηρημένες μεθόδους σχεδιασμού, με τα εργαλεία να αντιμετωπίζουν μεγάλο μέρος των χαμηλού επιπέδου λεπτομερειών της υλοποίησης υλικού.

Η RTL τεχνική προσφέρει μεγαλύτερο έλεγχο του σχεδιασμού υλικού, καθώς ο σχεδιαστής μπορεί να βελτιστοποιήσει άμεσα το κύκλωμα για να ανταποκριθεί στους συγκεκριμένους χρονισμούς και περιορισμούς πόρων του FPGA. Επίτυγχάνεται λεπτομερής έλεγχος του ρολογιού και συγχρονισμός του σχεδιασμού, που μπορεί να είναι κρίσιμο σε συστήματα υψηλής απόδοσης. Ακόμη, αποτελεί μία καθιερωμένη μεθοδολογία σχεδιασμού, με πολλούς πόρους και παραδείγματα διαθέσιμα. Απαιτεί, όμως, βαθιά κατανόηση του υλικού FPGA και της γλώσσας HDL και είναι ένα είδος χρονοβόρου και δύσκολου προγραμματισμού.

Αντίθετα ο HLS προγραμματισμός προσφέρει μια πιο αφηρημένη μεθοδολογία σχεδίασης,

η οποία μπορεί να είναι πιο εύκολη στην κατανόηση και αποσφαλμάτωση και είναι ιδανική για πολυπλοκους σχεδιασμούς. Επιτυγχάνεται έτσι γρηγορότερος προγραμματισμός και αυτόματη βελτιστοποίηση για τη συσκευή FPGA στόχο, η οποία μπορεί να οδηγήσει σε ένα πιο αποδοτικό σχεδιασμό. Ταυτόχρονα, βέβαια, ο σχεδιαστής έχει λιγότερο έλεγχο στον σχεδιασμό του υλικού και αυτό μπορεί να καθιστά δύσκολη την εκπλήρωση συγκεκριμένου χρονισμού και την σωστή κατανομή των πόρων.

Τόσο στην αρχική όσο και στην δική μας υλοποίηση έχει χρησιμοποιηθεί σαν HLS περιβάλλον ανάπτυξης ο Vivado HLS Compiler της Xilinx. Η διαφορά μεταξύ των δύο υλοποιήσεων αφορά τα εργαλεία που χρησιμοποιήθηκαν για την επικοινωνία του επεξεργαστή με το FPGA. Στην αρχική υλοποίηση λόγω του embedded FPGA χρησιμοποιήθηκε το περιβάλλον SDSoC της Xilinx με βάση το οποίο η εφαρμογή εκετελείται στο PS τμήμα του SoC και το υπολογιστικά βαρύ τμήμα της εφαρμογής εκετλείται στο PL. Αντίθετα, ο προγραμματισμός του Alveo U280 και κατέπέκταση η επικοινωνία μεταξύ του επεξεργαστή(host) και του FPGA(accelerator card) επιτυγχάνεται στο Vitis 2020 περιβάλλον της Xilinx και η μεταφορά των δεδομένων επιτυγχάνεται με OpenCL συναρτήσεις.

Παραθέτουμε λοιπόν τα συγκεκριμένα directives που χρησιμοποιήθηκαν και για τις δύο υλοποιήσεις:

Vivado HLS Compiler

- **Array Partition**: Χωρίζει έναν πίνακα σε μικρότερους υποπίνακες, επιτρέποντας έτσι την ταυτόχρονη πρόσβαση σε διαφορετικά σημεία του πίνακα και την μεταφορά του μέσω περισσότερων θυρών.
- **Dataflow**: Αναλύθηκε προηγουμένως.
- **Unroll**: Τοποθετείται εντός ενός επαναληπτικού βρόγχου και προκαλεί 'unroll', 'ξετυλιγμα' του βρόγχου δηλαδή την διάσπαση των επαναλήψεων σε μικρότερα σύνολα, το μέγεθος των οποίων καθορίζεται από το unroll factor. Τα μικρότερα υποσύνολα εντολών μπορούν να εκτελούνται παράλληλα, αξιοποιώντας καλύτερα τους πόρους του FPGA και αυξάνοντας την επίδοση της εφαρμογής.
- **Pipeline**: Αναλύθηκε προηγουμένως.
- **Inline**: Ανεβάζει ένα επίπεδο τα υπολογιστικά τμήματα της εφαρμογής στην ιεραρχία του RTL σχεδίου.
- **Stream**: Δημιουργεί FIFO ουρές δεδομένων, στα οποία η πρόσβαση είναι σειριακή και γίνεται μόνο μία φορά.

SDSoC

- **Zero Copy**: Ορίζεται το μέγεθος των δεδομένων που μεταφέρεται από τον επεξεργαστή (PS) στο hardware (PL) διαμέσου μίας AXI master bus διεπαφής. Το μέγεθος καθορίζεται είτε από σταθερά είτε από μεταβλητή της hardware συνάρτησης.
- **Data Access Patern**: Καθορίζεται το μοτίβο πρόσβασης στα δεδομένα.

- **Mem Attribute**: Μέσω αυτής της εντολής ορίζεται ο τρόπος αποθήκευσης των δεδομένων ενός πίνακα στην φυσική μνήμη (συνεχόμενη ή όχι) για να καθοριστεί βέλτιστα ο τρόπος μεταφοράς τους από τον μεταγλωτιστή.

OpenCL/Vitis

- **Platform, Platform::get**: Ορίζει την πλατφόρμα στην οποία θα εκτελεστεί η εφαρμογή, βρίσκει τις διαθέσιμες πλατφόρμες.

- **Device, getDevices**: Ορίζει την συσκευή στην οποία θα εκτελεστεί η εφαρμογή, βρίσκει τις διαθέσιμες συσκευές στις οποίες μπορεί να εκτελεστεί η εφαρμογή.

- **Context**: Ένα OpenCL context είναι ένα αντικείμενο το οποίο περιέχει πληροφορίες για την κατάσταση και τους πόρους μίας η περισσότερων OpenCL συσκευών, παρέχωντας έτσι την δυνατότητα για καλύτερη διαχείρηση της μνήμης και για εκτέλεση πυρήνων σε OpenCL συσκευές.

- **CommandQueue**: Αποτελεί ένα OpenCL αντικείμενο το οποίο δημιουργεί μία ουρά από εντολές που πρέπει να εκτελεστούν από την συσκευή. Ουσιαστικά φροντίζει για τον συγχρονισμό και την σωστή σειρά εκτέλεσης των διάφορων εντολών όπως οι μεταφορές δεδομένων στην μνήμη και η εκτέλεση πυρήνων.

- **Kernel**: Ορίζει και δημιουργεί με βάση κώδικα που υπάρχει σε ξεχωριστό αρχείο έναν πυρήνα ο οποίος θα εκτελεστεί σε μία OpenCL συσκευή, συγκεκριμένα στην περίπτωση μας σε ένα FPGA.

- **Program**. Το πρόγραμμα αποτελεί ένα σύνολο πυρήνων και εντολών που θα εκτελεστούν σε μία συσκευή μετά τον προγραμματισμό της.

- **Program::Binaries**: Φορτώνει στο πρόγραμμα το bitstream (xclbin αρχείο) που έχει δημιουργηθεί κατα την διαδικασία του build και ουσιαστικά περιέχει όλη την πληροφορία για τον προγραμματισμό του FPGA.

- **Buffer**: Χρησιμοποιείται για την αποθήκευση μεγάλου όγκου δεδομένων που απαιτούνται για την εκτέλεση των πυρήνων.

- **setArg**: Προσδιορίζει τα ορίσματα-εισόδους της kernel συνάρτησης.

- **enqueueMigrateMemObjects**: Με αυτή την εντολή γίνεται η μεταφορά των δεδομένων από και προς την μνήμη της συσκευής που έχει οριστεί για αυτά. Η μεταφορά γίνεται πριν και μετά την εκτέλεση των πυρήνων.

- **enqueueTask**: Εκκινεί τους πυρήνες που θα εκτελεστούν στην συσκευή που προγραμματίστηκε. Αποτελεί ουσιαστικά την αρχή του υπολογιστικού τμήματος που ανατίθεται στο FPGA για επιτάχυνση.

- **finish**: Σηματοδοτεί το τέλος της χρήσης της συσκευής.

2.2 Πολλαπλασιασμός Αραιού Πίνακα με Διάνυσμα SpMV

Ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα αποτελεί βασικό υπολογιστικό πυρήνα για πληθώρα εφαρμογών. Απαντάται σε τομείς όπως η επίλυση Διαφορικών Εξισώσεων, η Θεωρία Δικτύων, η Όραση Υπολογιστών και το Machine Learning. Ακόμη, αραιοί πίνακες χρησιμοποιούνται για την αναπαράσταση γράφων.

2.2.1 Ορισμός και Αναπαράσταση Αραιών Πινάκων

Ως αραιός ορίζεται ένας πίνακας που περιέχει μεγάλο ποσοστό μηδενικών στοιχείων. Αντίθετα, πυκνός ονομάζεται ένας πίνακας του οποίου η πλειοψηφία των στοιχείων του είναι διαφορετικά του μηδενός. Για παράδειγμα ένας πίνακας με 20 μη-μηδενικά στοιχεία σε σύνολο 100 στοιχείων θεωρείται αραιός και λέμε ότι έχει αραιότητα(sparsity) 80% και πυκνότητα (density) 20%.

Όπως αναφέραμε ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα αποτελεί σημείο κλειδί για την εκτέλεση πολλών εφαρμογών. Ταύτοχρονα, αποτελεί μια αρκετά βαριά υπολογιστική εφαρμογή καθώς κάθε στοιχείο του πίνακα χρησιμοποιείται μόνο μία φορά, πράγμα το οποίο απαιτεί μεγάλο αριθμό προσβάσεων στην μνήμη αναλογικά με τις πράξεις που εκτελούνται ανά πρόσβαση. Για αυτό τον λόγο θεωρείται ιδιάιτερα σημαντική η αύξηση της επίδοσης του SpMV. Λόγω της φύσης του προβλήματος (απλός υπολογιστικός πυρήνας-μεγάλος όγκος δεδομένων), ο κύριος τρόπος για την βελτίωση της επίδοσης του SpMV είναι η πιο αποδοτική αναπαράταση του αραιού πίνακα και η αποθήκευση μόνο των μη-μηδενικών στοιχείων με τον βέλτιστο τρόπο.

$$A = \begin{bmatrix} 11 & 12 & 0 & 2 \\ 0 & 6 & 0 & 8 \\ 0 & 0 & 0 & 3 \\ 13 & 0 & 14 & 16 \end{bmatrix}$$

Εικόνα 2.14: *Αραιός Πίνακας σε πλήρη μαθηματική αναπαράσταση*

Στην προσπάθεια αυτή έχουν αναπτυχθεί οι εξής μορφές αποθήκευσης ενός αραιού πίνακα:

- **Coordinate list (COO) format**

  Αποτελεί την πιο απλή μορφή αναπαράστασης. Για κάθε μη-μηδενικό στοιχείο του πίνακα αποθηκεύεται η τιμή του, η γραμμή στην οποία βρίσκεται και η στήλη του. Δημιουργούνται έτσι 3 πίνακες,values, col_ind, row_ind με μέγεθος ίσο με τα μη-μηδενικά στοιχεία του πίνακα A. Αυτή η μορφή αποθήκευσης είναι η πιο εύκολη σε κατανόηση και κατασκευή αλλά ταυτόχρονα δεν είναι η πιο αποδοτική, ιδίως για μεγαλύτερους πίνακες, και έχει περιθώρια βελτίωσης.

$$values = [\ 11\ 12\ 2\ 6\ 8\ 3\ 13\ 14\ 16\ ]$$

$$col\_ind = [\ 0\ 1\ 3\ 1\ 3\ 3\ 0\ 2\ 3\ ]$$

$$row\_ind = [\ 0\ 0\ 0\ 1\ 1\ 2\ 3\ 3\ 3\ ]$$

Εικόνα 2.15: *COO Αναπαράσταση του πίνακα A*

- **Compressed sparse row (CSR) format**

  Είναι η πιο διαδεδομένη μορφή αναπαράστασης αραιών πινάκων και αποτελεί την βάση για πολλές παραλλαγές. Οι πίνακες values και col_ind είναι ακριβώς ίδιοι με της COO αναπαράστασης. Διαφέρει από την COO αναπαράσταση καθώς αντί για το row index κάθε μη-μηδενικού στοιχείου αποθηκεύει έναν δείκτη για κάθε αλλαγή γραμμής. Ο δείκτης αυτός είναι ουσιαστικά το index του πρώτου στοιχείου της κάθε γραμμής στους πίνακες values και col_ind και έχει μέγεθος ίσο με το πλήθος των γραμμών του πίνακα αυξημένο κατα 1(το τελευταίο κελί είναι ίσο με τον αριθμό των μη-μηδενικών στοιχείων του πίνακα).

$$values = [\ 11\ 12\ 2\ 6\ 8\ 3\ 13\ 14\ 16\ ]$$

$$col\_ind = [\ 0\ 1\ 3\ 1\ 3\ 3\ 0\ 2\ 3\ ]$$

$$row\_ptr = [\ 0\ 3\ 5\ 6\ 9\ ]$$

Εικόνα 2.16: *CSR Αναπαράσταση του πίνακα A*

Αυτή η δομή αναπαράστασης συμβάλλει στην μείωση του απαιτούμενου χώρου αφού οι γραμμές ενός πίνακα συνήθως είναι πολύ λιγότερες από τα μη-μηδενικά του στοιχεία. Επιπλέον διευκολύνει την πρόσβαση ανά γραμμές και την παραλληλοποίηση των υπολογισμών.

- **Compressed sparse column (CSC) format**

  Έχει ακριβώς την ίδια λογική με το CSR format αλλά αντίστροφα από αυτήν αποθηκεύει τη γραμμή κάθε μη-μηδενικού στοιχείου και διατηρεί πίνακα δεικτών για την αλλαγή στήλης.

$$values = [\ 11\ 12\ 2\ 6\ 8\ 3\ 13\ 14\ 16\ ]$$

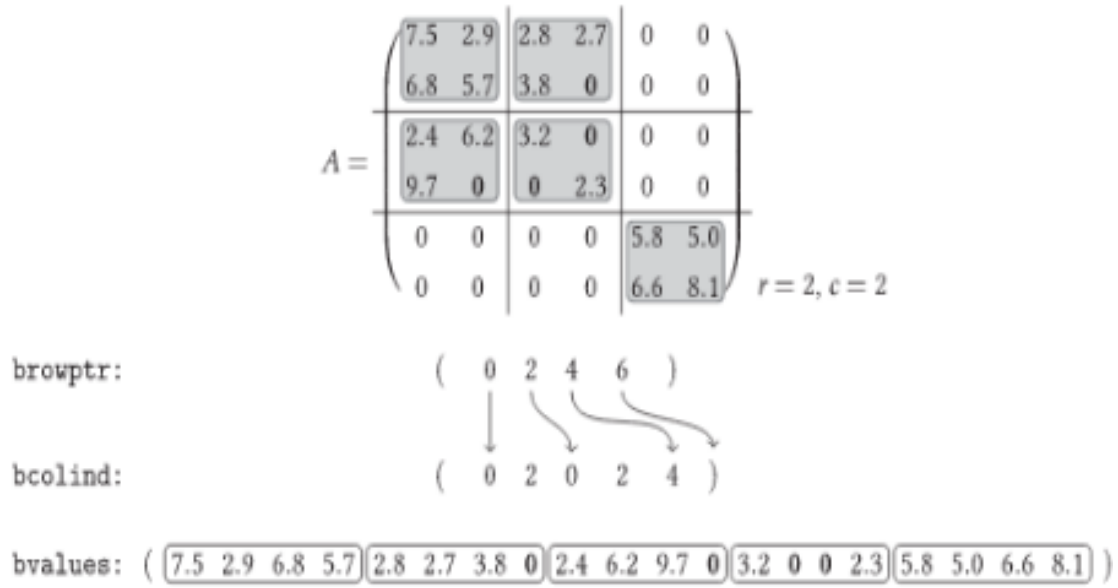
$$col\_ptr = [\ 0\ 2\ 4\ 5\ 9\ ]$$

$$row\_ind = [\ 0\ 0\ 0\ 1\ 1\ 2\ 3\ 3\ 3\ ]$$

Εικόνα 2.17: *CSC Αναπαράσταση του πίνακα A*

- **Blocked Compressed Sparse row (BCSR) format**

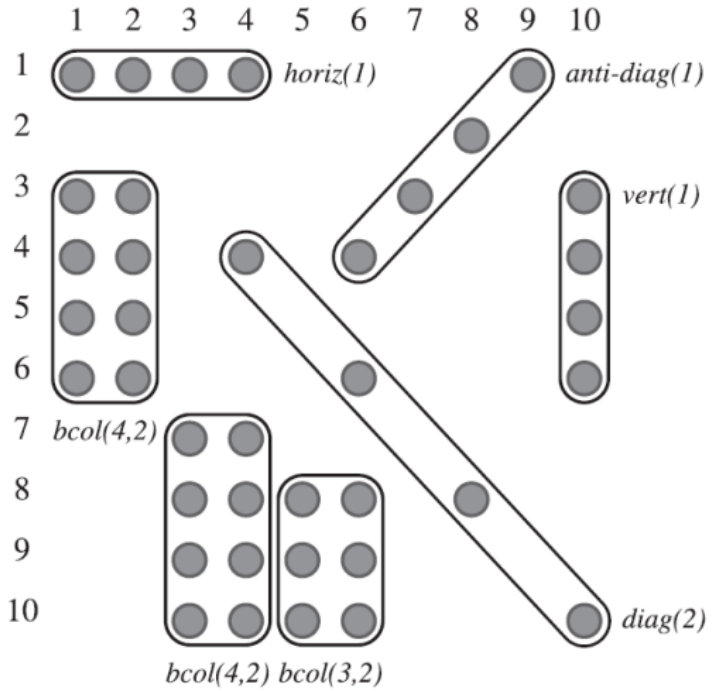
  Πολλές φορές τα μη-μηδενικά στοιχεία πινάκων πραγματικών εφαρμογών βρίσκονται συγκεντρωμένα ανά περιοχές του πίνακα. Αυτό καθιστά δυνατή την διάσπαση του πίνακα σε μη-μηδενικά block και οδηγεί σε περαιτέρω μείωση του χώρου αποθήκευσης. Προτάθηκε έτσι η BCSR αναπαράσταση η οποία αποθηκέυει block διαστάσεων r x c. Το μέγεθος των block είναι συγκεκριμένο οπότε για κάθε block αποθηκεύεται column index του πάνω αριστερά στοιχείου του και σε έναν brow_ptr πίνακα αποθηκεύεται το bcol index του block στο οποίο γίνεται αλλαγή γραμμής. Επίσης απαιτείται και zero-padding δηλαδή αποθήκευση επιπλέον μηδενικών τιμών για την συμπλήρωση των blocks .



Εικόνα 2.18: *BCSR Αναπαράσταση*

Σε αυτο το λογικό πλαίσιο έχουν δημιουργηθεί διάφορες αναπαραστάσεις οι οποίες είναι πιο προσαρμοσμένες στην εκάστοτε κατανομή των μη-μηδενικών στοιχείων στον πίνακα. Ένα τέτοιο παράδειγμα αποτελεί η 1D-VBL (1 Dimensional Variable Block Length), η οποία είναι ουσιαστικά μια παραλλαγή της BCSR με c=1 και r μεταβλητό. Προστίθεται βέβαια εκτός από τους πίνακες της κλασσικής BCSR ένας πίνακας με το μέγεθος του κάθε block, αλλά πλέον δεν υπάρχει ανάγκη αποθήκευσης μηδενικών στοιχείων. Επίσης υπάρχουν αναπαραστάσεις οι οποίες αποθηκεύουν συνεχή διαγώνια block τα οποία απαντώνται για παράδειγμα σε finite element και finite difference πίνακες.

Σε μία προσπάθεια συνένωσης όλων των παραπάνω μεθόδων έχει προταθεί η Compressed Sparse eXtended (CSX) αναπαράσταση η οποία εντοπίζει διάφορετικών ειδών ¨σχηματισμούς' μη-μηδενικών στοιχειών. Μειώνει έτσι τον χώρο αποθήκευσης καθώς είναι προσαρμόσιμη στις ιδιαιτερότητες του εκάστοτε πίνακα.



Εικόνα 2.19: *CSX Αναπαράσταση [5]*

Αναπαράσταση που χρησιμοποιήθηκε στα πλαίσια αυτής της διπλωματικής

Η αναπαράσταση που έχει χρησιμοποιηθεί και στις δύο υλοποιήσεις χρησιμοποιεί ως βάση της την CSR αναπαράσταση. Τα FPGA χειρίζονται καλύτερα πληροφορία η οποία είναι κωδικοποιημένη σε επίπεδο bit. Με βάση λοιπόν αυτήν την λογική στην αρχική υλοποίηση έχει αφαιρεθεί τελείως ο row_ptr πίνακας και αντ΄ αυτού η αλλαγή γραμμής σηματοδοτείται από την τιμή του τελευταίου bit του κάθε column index. Χρησιμοποιείται δηλαδή ένας πίνακας που περιέχει όλες τις τιμές των στοιχείων και ένας πίνακας που περιέχει το column index του κάθε στοιχείου ενωμένο με ένα bit(rowbit), που είναι 1 αν υπάρχει αλλαγή γραμμής σε εκείνο το στοιχείο και 0 αν δεν υπάρχει.

2.2.2 Υπολογιστικός Πυρήνας SpMV

Ο υπολογιστικός πυρήνας του SpMV αποτελεί το σημείο στο οποίο εκτελείται ουσιαστικά η πράξη του πολλαπλασιασμού και είναι το υπολογιστικά βαρύ τμήμα της εφαρμογής που μεταφέρεται στο hardware για επιτάχυνση. Η σωστή και αποδοτική του υλοποίηση, λοιπόν, είναι καίρια για την επίδοση της εφαρμογής. Λόγω της αναπαράστασης που επιλέχθηκε ο πυρήνας βασίστηκε αρχικά σε πυρήνα απλής CSR αναπαράστασης και στην συνέχεια προσαρμόστηκε στην λογική του rowbit.

Παρατηρούμε ότι το μόνο διάνυσμα στο οποίο εκτελούνται επαναλαμβανόμενες και τυχαίες προσβάσεις είναι το διάνυσμα x. Στους υπόλοιπους πίνακες σε όλα τα στοιχεία γίνεται μόνο μία πρόσβαση. Καθίσταται έτσι εφικτή η παραλληλοποίηση του αλγορίθμου ανά γραμμές.

Algorithm: SpMV Computation using the CSR scheme

1: ***Procedure SpMV*** $(A :: in, x :: in, y :: out)$
A*: *input *Matrix in CSR format*
x*: *input *vector*
y*: *output *vector*
2: ***for*** $i = 0$ *to* $N - 1$ ***do***
3: $for\ k = rowPtr\ [i] to\ k <\ rowPtr\ [i + 1] do$
4: $y[i] = y[i] + Val[k] * Vec[Col[k]]$
5: ***end for***
6: ***end for***
7: ***end Procedure***

Εικόνα 2.20: *Υπολογιστικός Πυρήνας SpMV βασισμένος σε CSR format [6]*

2.3 Vitis SpMV Library

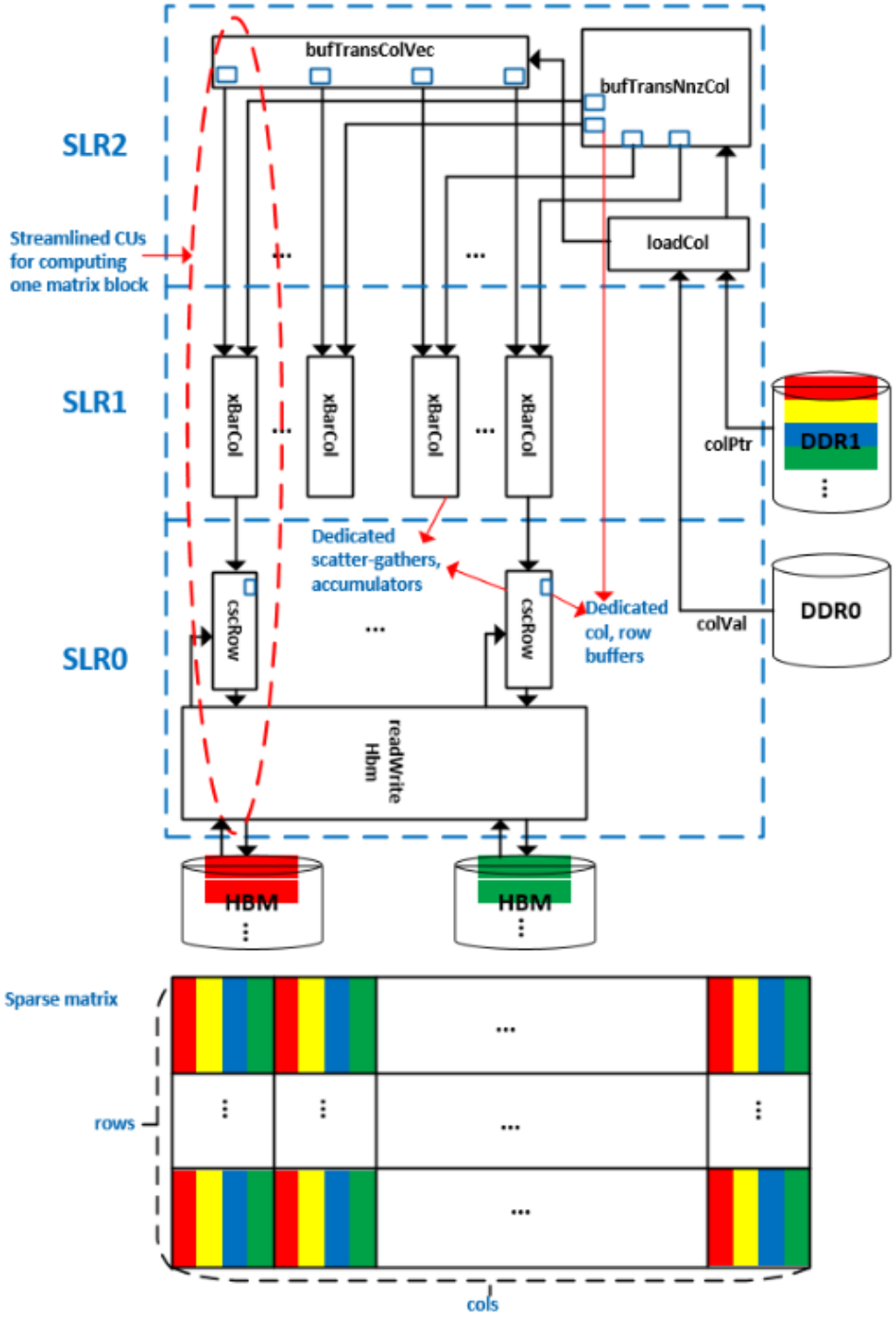
Στις μετρήσεις μας θα χρησιμοποιήσουμε σαν σημείο αναφοράς την Vitis SpMV Library, η οποία αποτελεί ουσιαστικά μια βιβλιοθήκη η οποία αναπτύχθηκε από την Xilinx και υλοποιεί τον SpMV.

Όπως φαίνεται στο παρακάτω σχήμα, ο επιταχυντής CSCMV που υλοποιήθηκε στο FPGA Alveo U280 αποτελείται από μια ομάδα CUs (compute units, τα instances των Vitis Kernels) που συνδέονται μέσω AXI STREAMs. Σε αυτόν τον σχεδιασμό, 16 (από τα 32) κανάλια HBM χρησιμοποιούνται για την αποθήκευση των τιμών και των δεικτών γραμμής των μη-μηδενικών στοιχείων ενός αραιού πίνακα. Κάθε κανάλι HBM οδηγεί ένα ειδικό μονοπάτι υπολογισμού που περιλαμβάνει τα xBarCol και cscRow για την εκτέλεση της λειτουργίας SpMV για το τμήμα των δεδομένων αραιών πινάκων που είναι αποθηκευμένα σε αυτό το κανάλι HBM. Συνολικά, εκτελούνται ταυτόχρονα 16 λειτουργίες SpMV για διαφορετικά τμήματα των δεδομένων του αραιού πίνακα. Χάρη στην αποθήκευση του αραιού πίνακα σε μορφή CSC, το πυκνό διάνυσμα εισόδου έχει υψηλό βαθμό επαναχρησιμοποίησης. Αυτή η επαναχρησιμοποίηση που αντιμετωπίζεται στις CUs bufTransColVec και bufTransNnzCol και η χαμηλή επιβάρυνση πρόσβασης στη μνήμη της συσκευής που αντιμετωπίζεται στη CU loadCol παρέχουν επαρκή ρυθμό μετάδοσης δεδομένων που επιτρέπει στα 16 παράλληλα μονοπάτια υπολογισμού να λειτουργούν στα 300MHz για την επίτευξη της υψηλότερης απόδοσης. Τα κυριότερα σημεία αυτής της αρχιτεκτονικής:

- Χρήση των AXI STREAMS για τη σύνδεση μεγάλου αριθμού CUs (37 CUs σε αυτή τη σχεδίαση) για την υλοποίηση μαζικού παραλληλισμού στο υλικό

- Αξιοποίηση των διαφορετικών μνημών της συσκευής για τη μείωση της επιβάρυνσης πρόσβασης στη μνήμη και την ικανοποίηση των απαιτήσεων απόδοσης δεδομένων των μονοπατιών υπολογισμού

- Ελαχιστοποίηση της επικοινωνίας μεταξύ των SLR (Super Logic Region) για την επίτευξη υψηλότερου ρυθμού ρολογιού



Εικόνα 2.21: *Αρχιτεκτονική της Vitis SpMV Βιβλιοθήκης [7]*

Μέρος II

Πρακτικό Μέρος

Κεφάλαιο 3

Υλοποίηση

Στο κεφάλαιο αυτό παρουσιάζονται αναλυτικά τα βήματα που ακολουθήθηκαν για την υλοποίηση του πολλαπλασιασμού αραιού πίνακα με διάνυσμα(SpMV) στο Alveo U280 Data Center Accelerator Card. Αρχικά, περιγράφουμε την δομή και τα βασικά χαρακτηριστικά της αρχικής υλοποίησης στο ZCU102, στην οποία βασιστήκαμε. Στην συνέχεια παραθέτουμε τις αλλαγές που έγιναν στον κώδικα για την μεταφορά του στο Alveo U280 και μετά όλες τις δοκιμές που έγιναν με στόχο την βελτίωση της απόδοσης. Στο τέλος, παρουσιάζουμε το project Optima SpMV, το οποίο αποτέλεσε μία βάση για την τελική μορφή της υλοποίησης μας.

3.1 Αρχική Υλοποίηση

Βάση της παρούσας διπλωματικής αποτέλεσε η υλοποίηση μιας προηγούμενης διπλωματικής του εργαστηρίου[8]. Σε αυτήν την διπλωματική υλοποιούνταν το ίδιο πρόβλημα (SpMV) σε ένα embedded FPGA, το ZCU102 της Xilinx. Αξίζει, λοιπόν, αρχικά να αναλύσουμε τα βασικά στοιχεία αυτής της αρχικής υλοποίησης.

Παράμετροι του build

Η λειτουργία του προγράμματος καθορίζονταν από τις εξής παραμέτρους:

- **Number of Compute Units(CU)**. Ο κώδικας χωρίζει το πρόβλημα σε μικρότερα υπολογιστικά τμήματα Compute Units επιτυγχάνοντας την βέλτιστη αξιοποίηση των πόρων του FPGA και την παραλληλοποίηση του προγράμματος. Το πρόγραμμα προσέφερε την δυνατότητα για 1,2,4,8,10,12,14,16 Compute Units, δηλαδή για διάσπαση των υπολογισμών σε αυτούς τους αριθμούς ανεξάρτητων τμήματων που εκτελούνταν ταυτόχρονα σε ξεχωριστές περιοχές του FPGA προγραμματισμένες για αυτό τον σκοπό.

- **Vectorization Factor(VF)**. Αποτελεί το παράγοντα¨ξετυλίγματος' (unroll) των βρόγχων που εκτελούν τους υπολογισμούς για τον πολλαπλασιασμό. Μπορεί να πάρει τιμές 1,2,4,8,16 και δηλώνει ουσιαστικά πόσες πράξεις ενός επαναληπτικού βρόγχου εκτελούνται ταυτόχρονα μειώνοντας τον χρόνο εκτέλεσης.

- **Float or Double Precision(DOUBLE)**. Δυνατότητα καθορισμού της ακρίβειας αποθήκευσης των τιμών του A και του x και κατ΄ επέκταση των υπολογισμών και των

αποτελεσμάτων στο y διάνυσμα. Οταν η είσοδος DOUBLE είναι ίση με 1 χρησιμοποιούνται 64 bit για την αναπράσταση των δεδομένων ενώ όταν είναι 0 32 bit.

Μορφή Αναπαράστασης του SpMV

Όπως αναλύσαμε και στην εισαγωγή, βάση της αναπαράστασης αποτελεί μία παραλλαγή του CSR (Compressed Sparse Row Format). Στο κλασσικό CSR στέλνονται στο FPGA 3 πίνακες, ένας με τα values του πίνακα A(values), ένας με τα column indices(col_ind) και ένας με row pointers που δείχνουν την αρχή της κάθε γραμμής(row_ptr). Στην ZCU102 υλοποίηση έχει αφαιρεθεί τελείως ο row_ptr πίνακας και αντ΄ αυτού η αλλαγή γραμμής σηματοδοτείται από την τιμή του τελευταίου bit του κάθε column index. Χρησιμοποιείται δηλαδή ένας πίνακας που περιέχει όλες τις τιμές των στοιχείων και ένας πίνακας που περιέχει το column index του κάθε στοιχείου ενωμένο με ένα bit(rowbit) που είναι 1 αν υπάρχει αλλαγή γραμμής σε εκείνο το στοιχείο και 0 αν δεν υπάρχει.

Κατανομή μνήμης και διαχωρισμός των δεδομένων

Κατά την εκτέλεση του υπολογιστικού πυρήνα του SpMV κάθε στοιχείο του πίνακα A χρησιμοποιείται μόνο μία φορά (κάθε γραμμή του πίνακα A αντιστοιχεί στον υπολογισμό ενός στοιχείου του αποτελέσματος y) ενώ το διάνυσμα x απαιτείται για τον υπολογισμό όλων των στοιχείων του y. Συνεπώς επιλέχθηκε η αποθήκευση του x στην BRAM του ZCU 102 με στόχο την επιτάχυνση των τυχαίων προσβάσεων στην μνήμη. Τα values και τα column indices του A καθώς και το αποτέλεσμα του πολλαπλασιασμού y λόγω των σειριακών προσβάσεων στέλνονται στο FPGA σε streams, δημιουργούν δηλαδή FIFO ουρές που αδειάζουν/γεμίζουν κατά την εκτέλεση του υπολογιστικού πυρήνα.

Για την πλήρη εκμετάλλευση των πόρων του FPGA και την μέγιστη παραλληλοποίηση των υπολογισμών χωρίζεται ο πίνακας A ανά γραμμές με εξισορρόπηση της κατανομής των μημηδενικών στοιχείων. Δημιουργούνται έτσι Compute Units τα οποία αποτελούν ανεξάρτητα τμήματα κώδικα που εκτελούνται ταυτόχρονα σε ξεχωριστές περιοχές του FPGA προγραμματισμένες για αυτό τον σκοπό.

Η BRAM του ZCU 102 έχει χωρητικότητα 32Mb και κάθε Compute Unit απαιτεί την αποθήκευση ξεχωριστού αντιγράφου του διανύσματος x. Έτσι, λοιπόν, με την αύξηση των Compute Units δεν επαρκεί η διαθέσιμη BRAM για την αποθήκευση όλων των x buffers. Για αυτό υπήρχε ανάγκη περαιτέρου χωρισμού των υπολογισμών. Εξαιτίας της φύσης του SpMV κάθε στήλη του A πολλαπλασιάζεται με το ίδιο στοιχείο(γραμμή) του x. Έτσι, διασπάμε το x διάνυσμα οριζόντια σε μικρότερα υποδιανύσματα που χωράνε στην BRAM και για καθένα από αυτά τα υποδιανύσματα μεταφέρουμε στον πυρήνα μόνο το τμήμα του πίνακα A που πολλαπλασιάζεται με αυτά. Ο πίνακας A χωρίζεται δηλαδή και ¨κάθετα' σε blocks τα οποία δεν εκτελούνται παράλληλα όπως τα Copute Units αλλά σειριακά το ένα μετά το άλλο για να επαρκεί κάθε φορά η διαθέσιμη BRAM. Χωρίζουμε, έτσι, τον πίνακα A κάθετα σε blocks καθένα από τα οποία χωρίζεται στην συνέχεια οριζόντια σε Compute Units.

Υπολογιστικός πυρήνας

Ο υπολογιστικός πυρήνας αποτελεί μια παραλλαγή του CSR υπολογιστικού πυρήνα βασισμένη στην παραπάνω αναπαράσταση. Ο υπολογισμός εκτελείται ανά στοιχείο πλέον και όχι ανά γραμμές και εσωτερικά γίνεται παραλληλοποίηση των πράξεων ανά Vectorization Factor στοιχεία.

```
#pragma HLS INLINE
ValueType sum = 0;
ValueType term[VectFactor];
#pragma HLS ARRAY_PARTITION variable=term complete dim=1
BoolType row_end[VectFactor];
#pragma HLS ARRAY_PARTITION variable=row_end complete dim=1
for (IndexType i=0; i<nr_nzeros; i+=VectFactor) {
    #pragma HLS PIPELINE
    ValueType value;
    CompressedIndexType col;
    CompressedIndexType realcol;
    for (IndexType j=0; j<VectFactor; j++) {
        #pragma HLS dependence variable=term inter false
        #pragma HLS dependence variable=row_end inter false
        #pragma HLS unroll
        value   = values_fifo.read();
        col     = col_fifo.read();
        realcol = col.range(COMPRESSED_INDEX_TYPE_BIT_WIDTH-2,0);
        term[j] = value * x[realcol];
        row_end[j] = col.range(COMPRESSED_INDEX_TYPE_BIT_WIDTH-1,COMPRESSED_INDEX_TYPE_BIT_WIDTH-1);
    }

    ValueType sum_tmp = 0;
    for (IndexType j=0; j<VectFactor; j++) {
        #pragma HLS dependence variable=term inter false
        #pragma HLS unroll
        sum_tmp += term[j];
    }
    sum += sum_tmp;
    if(row_end[VectFactor-1]==1){
        results_fifo << sum;
        sum = 0;
    }
}
```

Εικόνα 3.1: *Αρχικός Υπολογιστικός Πυρήνας*

Προ-επεξεργασία των δεδομένων και Πακετάρισμα

Εξαιτίας της τεχνολογίας του SDSoC ZCU102 στην αρχική υλοποίηση απαιτούνταν μεγάλη προεπεξεργασία και ειδικό πακετάρισμα των δεδομένων πριν την αποστολή τους στο FPGA. Πιο αναλυτικά, οι θύρες υψηλής απόδοσης που χρησιμοποιούνται για την μεταφορά των δεδομένων από το επεξεργαστικό σύστημα στην προγραμματίσιμη λογική έχουν εύρος ζώνης (BUS_BIT_WIDTH) 128 bits. Τα values του πίνακα A και το διάνυσμα x αποθηκεύονται ως float ή doubles και άρα απαιτούν 32 και 64 bits αντίστοιχα ενώ τα column indices απαιτούν μόνο 15 bits λόγω του χωρισμού του πίνακα σε μικρότερα blocks. Στα 15 αυτά bits προστίθεται στο τέλος ένα bit ενδεικτικό για την αλλαγή γραμμής (1 αν υπάρχει αλλαγή γραμμής αλλιώς 0).

Ορίζεται, λοιπόν, ως RATIO_v το πλήθος των values που μπορούν να μεταφερθούν ταυτόχρονα που ισούται με τον λόγο του BUS_BIT_WIDTH διά το πλήθος των bits που

απαιτούνται για την αναπαράσταση των A και x values. Ομοίως ορίζεται ως RATIO_col_ind το πλήθος column indices που μπορούν να μεταφερθούν ταυτόχρονα(συμπεριλαμβανομένου του rowbit).

Στην αρχική υλοποίηση τα values και column indices του A ενώνονται σε μία κοινή δομή, submatrix για την αποστολή στο FPGA και τα x values στέλνονται χωριστά. Στο πίνακα submatrix για κάθε ένα κελί που περιέχει values τοποθετούνται μετά RATIO_v/RATIO_-col_ind κελιά από column indices . Αυτό συμβαίνει επειδή τα column indices απαιτούν λιγότερο χώρο σε bits από τα values και άρα απαιτούνται περισσότερα κελιά για να τοποθετηθούν τα column indices που αντιστοιχούν στα values.

Όλα τα παραπάνω καθιστούν αναγκαίο το padding, δηλαδή την προσθήκη επιπλέον μηδενικών στοιχείων στον πίνακα A και στο διάνυσμα x. Πιο αναλυτικά μέσω του padding εξασφαλίζονται οι εξής προϋποθέσεις:

- Ο συνολικός αριθμός των μη-μηδενικών στοιχείων κάθε block του πίνακα A πρέπει να είναι πολλαπλάσιος του Ratio_v για να είναι εφικτή η αποστολή τους σε πακετά των 128 bits

- Όπως είδαμε για να λειτουργήσει ο αλγόριθμος πολλαπλασιασμού πρέπει κάθε γραμμή του πίνακα να περιέχει μηδενικά πολλαπλάσια του Vectorization Factor για να είναι εφικτό το loop unrolling και του Ratio_v γιατί κάθε γραμμή του πίνακα A αντιστοιχεί σε ένα στοιχείο του y.

- Τέλος το κάθε block των διανυσμάτων x και y πρέπει να έχει και αυτό στοιχεία πολλαπλάσια του Ratio_v για την μεταφορά του στο FPGA. Αυτό συνεπάγεται και την ανάγκη ύπαρξης πολλαπλάσιων του Ratio_v γραμμών και στηλών ανά block.

Δομή του κώδικα

Παρουσιάζουμε τα αρχεία από τα οποία αποτελείται η αχρική υλοποίηση μάζι με μία συνοπτική περιγραφή της λειτουργίας τους. Στο csr.cpp αρχείο βρίσκονται οι συναρτήσεις που διαβάζουν αρχικά τον πίνακα σε CSR μορφή καθώς και η εκτέλεση του SpMV σε CPU για το verification των αποτελεσμάτων και την σύγκριση των χρόνων. Στο csr_hw.cpp γίνεται η προετοιμασία των δεδομένων, δηλαδή το πακετάρισμά τους για αποστολή στο FPGA όπως ακριβώς περιγράψαμε παραπάνω. Στο csr_hw_wrapper.cpp γίνεται η κλήση των συναρτήσεων που βρίσκονται στο csr_hw.cpp για όλα τα block του πίνακα. Στο spmv.cpp βρίσκεται ο κώδικας που τρέχει στο FPGA. Αποτελείται ουσιαστικά από τον υπολογιστικό πυρήνα του πολλαπλασιασμού και τις συναρτήσεις που μεταφέρουν τα δεδομένα από και προς το FPGA. Περίεχει την entry function (spmv), που δηλώνεται στο .h αρχείο και καλείται ως hw function στο csr_hw_wrapper.cpp) Αυτή καλεί τις συναρτήσεις διαβάσματος, υπολογισμών και αποθήκευσης των αποτελεσμάτων δημιουργώντας δημιουργώντας μια pipelined ροή δεδομένων και παραλληλοποιώντας τους υπολογισμούς με την χρήση των HLS directives που παρουσιάσαμε στην εισαγωγή. Στο util.h ορίζονται οι δομές που χρησιμοποιούνται σε όλο τον κώδικα. Με χρήση της βιβλιοθήκης ap_int.h ορίζονται αριθμητικές δομές συγκεκριμένου πλήθους απο bits ώστε να δημιουργηθούν οι wide δομές που περγράψαμε. Επίσης εδώ

Matrix	Rows	Columns	Non-zeros	Application Domain
scircuit	170998	170998	958936	Circuit Simulation Problem
mac_econ_fwd500	206500	206500	1273389	Economic Problem
raefsky3	21200	21200	1488768	Computational Fluid Dynamics Problem
bbmat	38744	38744	1771722	Computational Fluid Dynamics Problem
conf5_4-8x8-15	49152	49152	1916928	Theoretical/Quantum Chemistry Problem
mc2depi	525825	525825	2100225	2D/3D Problem
rma10	46835	46835	2374001	Computational Fluid Dynamics Problem
cop20k_A	121192	121192	2624331	2D/3D Problem
webbase-1M	1000005	1000005	3105536	Directed Weighted Graph
cant	62451	62451	4007383	2D/3D Problem
pdb1HYS	36417	36417	4344765	Weighted Undirected Graph
TSOPF_RS_b300_c3	42138	42138	4413449	Power Network Problem
consph	83334	83334	6010480	2D-3D Problem
shipsec1	140874	140874	7813404	Structural Problem
PR02R	161070	161070	8185136	Computational Fluid Dynamics Problem
mip1	66463	66463	10352819	Optimization Problem
pwtk	217918	217918	11634424	Structural Problem
crankseg_2	63838	63838	14148858	Structural Problem
Si41Ge41H72	185639	185639	15011265	Theoretical/Quantum Chemistry Problem
TSOPF_RS_b2383	38120	38120	16171169	Power Network Problem
Ga41As41H72	268096	268096	18488476	Quantum Chemistry
eu-2005	862664	862664	19235140	Directed Graph
wikipedia-20051105	1634989	1634989	19753078	Directed Graph
ldoor	952203	952203	46522475	Structural Problem
bone010	986703	986703	71666325	Model Reduction Problem
cage15	5154859	5154859	99199551	Directed Weighted Graph

Πίνακας 3.1: *Συλλογή Πινάκων*

ρυθμίζεται το μέγεθος των blocks στα οποία χωρίζεται ο πίνακας ανάλογα με τον αριθμό των Compute Units.

3.2 Συλλογή Πινάκων για Αξιολόγηση

Για την αξιολόγηση των αποτελεσμάτων μας χρησιμοποιήσαμε ένα σύνολο πίνάκων από την συλλογή Αραιών πινάκων SuiteSparse[9]. Επιλέξαμε ένα ευρύ φάσμα πινάκων με διαφορετικές διαστάσεις, πλήθος και κατανομή μη-μηδενικών στοιχείων και εφαρμογή σε πληθώρα επιστημονικών πεδίων και εφαρμογών. Εξασφαλίζουμε έτσι την μελέτη της απόδοσης της υλοποίησης μας σε αρκετούς και διαφορετικούς πίνακες πραγματικών εφαρμογών. Στον Πίνακα 3.1 παρουσιάζουμε τα χαρακτηριστικά κάθε πίνακα.

3.3 Μεταφορά από το ZCU102 στο Alveo U280

3.3.1 Τροποποιήσεις στον κώδικα

Οι αλλαγές που έγιναν στον κώδικα για την μετάβαση από το ZCU102 στο Alveo U280 αφορούν κυρίως το κομμάτι του host κώδικα. Όσον αφορά την main (host.cpp) όλες οι κλήσεις των συναρτήσεων που αφορούν την εκτέλεση του spmv έχουν παραμείνει ίδιες και η μόνη αλλαγή έχει γίνει στο διάβασμα του αρχείου που τώρα γίνεται αρχικά σε COO format και στην συνέχεια μετατρέπεται σε CSR για την εκτέλεση του υπόλοιπου κώδικα. Οι συναρτήσεις που το εκτελούν αυτό (read_coo_matrix, coo_to_csr) προστέθηκαν στο αρχείο csr.cpp.

Το "πακετάρισμα" των δεδομένων συμπεριλαμβανομένου του σκαναρίσματος, του χωρισμού σε blocks και compute units έχει παραμείνει ακριβώς το ίδιο.

Οι κυρίως αλλαγές έχουν γίνει στον τρόπο που καλείται ο κώδικας που εκτελείται στο FPGA και στο πως στέλνονται τα δεδομένα σε αυτόν. Πιο συγκεκριμένα οι αλλαγές έχουν γίνει στο csr_hw_wrapper.cpp αρχείο στην συνάρτηση spmv_hw. Αρχικά στο ZCU102 ο κώδικας που εκτελείται στο FPGA καλείται ως απλή συνάρτηση ενώ στο Alveo U280 ορίζεται κατα το compile το αρχείο στο οποίο περιέχεται ο κώδικας που εκτελείται στο FPGA (spmv.cpp) από τον οποίο παράγεται το xclbin binary αρχείο με βάση το οποίο προγραμματίζεται το FPGA. Οπότε στην spmv_hw συνάρτηση στο Alveo_u280 αρχικά εντοπίζεται το device στο οποίο θα τρέξει το hw κομμάτι του κώδικα και μετα με βάση αυτό ορίζονται το context, το queue και το program, η συσκευή προγραμματίζεται με βάση το xclbin file που έχει παραχθεί και ορίζονται ίσα σε αριθμό με τα Compute Units kernels της spmv kernel συνάρτησης που έχει οριστει στο αρχείο spmv.cpp.

Στην συνέχεια γίνεται η μεταφορά των δεδομένων και η εκκίνηση των πυρήνων. Στο ZCU102 ορίζονταν διαφορετική kernel συνάρτηση για τους διαφορετικούς αριθμούς compute units μέσα στην οποία γινόταν ο διαχωρισμός των εργασιών στα compute units και είχε διαφορετικό αριθμό ορισμάτων ανάλογο των compute units. Αντίθετα στο Alveo U280 στον kernel κώδικα ορίζεται πλέον μόνο μία συνάρτηση η οποία αφορά τον κώδικα που εκτελείται σε ένα compute unit και είναι ίδια με την kernel συνάρτηση η οποία καλούνταν στον κώδικα του ZCU 102 όταν εκτελούνταν για ένα compute unit και ορίζεται ένας πυρήνας για κάθε compute unit ξεχωριστά. Κατά συνέπεια στο Alveo U280 οι συναρτήσεις spmv_hw (csr_hw_wrapper.cpp) και spmv (spmv.cpp) είναι ίδιες για τους διαφορετικό αριθμό από Compute units. Η κλήση των πολλαπλών CUs στο Alveo U280 γίνεται στο host κώδικα όπου ορίζονται πολλαπλοί πυρήνες του ίδιου kernel κώδικα και εκτελούνται ταυτόχρονα στο FPGA. Η μεταφορά των δεδομένων και η εκκίνηση των πυρήνων γίνεται στην συνάρτηση run_krnl του csr_hw_wrapper αρχείου η οποία καλείται από την spmv_hw για κάθε block ξεχωριστά.

Ακολουθεί ο κώδικας της run_krnl:

```
double run_krnl(cl::Context& context,
                cl::CommandQueue& q,
                std::vector<cl::Kernel> krnls,
                csr_hw_matrix **hw_matrix,
                csr_hw_x_vector *hw_x,
                csr_hw_vector **hw_y,
                                int block) {
    cl_int err;
    std::vector<cl::Buffer> buffer_in1(ComputeUnits);
    std::vector<cl::Buffer> buffer_in2(ComputeUnits);
    std::vector<cl::Buffer> buffer_output(ComputeUnits);

    int empty = 0;
    std::cout << "Buffers\n";
    for (int i = 0; i < ComputeUnits; i++) {
        if ((hw_matrix[i]->nr_ci[block] + hw_matrix[i]->nr_val[block]) != 0)
        {
```

```
            OCL_CHECK(err, buffer_in1[i] = cl::Buffer(context,
            CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                sizeof(BusDataType) * (hw_matrix[i]->nr_ci[block] +
                hw_matrix[i]->nr_val[block]),
                hw_matrix[i]->submatrix[block], &err));
            OCL_CHECK(err, buffer_in2[i] = cl::Buffer(context,
            CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                sizeof(ValueType) * hw_x->nr_values[block] ,
                hw_x->values[block], &err));
            OCL_CHECK(err, buffer_output[i] = cl::Buffer(context,
            CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                sizeof(BusDataType) * hw_matrix[i]->nr_rows[block] / RATIO_v,
                hw_y[i]->values[block], &err));
            empty += 1;
        }
    }
    if (empty == 0) {
        return 0;
    }

    for (int i = 0; i < ComputeUnits; i++) {
        if ((hw_matrix[i]->nr_ci[block] + hw_matrix[i]->nr_val[block]) != 0)
        {
            OCL_CHECK(err, err = (krnls[i]).setArg(0, hw_matrix[i]
            ->nr_rows[block]));
            OCL_CHECK(err, err = (krnls[i]).setArg(1, hw_matrix[i]
            ->nr_nzeros[block]));
            OCL_CHECK(err, err = (krnls[i]).setArg(2, buffer_in1[i]));
            OCL_CHECK(err, err = (krnls[i]).setArg(3, hw_matrix[i]
            ->nr_ci[block]));
            OCL_CHECK(err, err = (krnls[i]).setArg(4, hw_matrix[i]
            ->nr_val[block]));
            OCL_CHECK(err, err = (krnls[i]).setArg(5, buffer_output[i]));
            OCL_CHECK(err, err = (krnls[i]).setArg(6, buffer_in2[i]));
            OCL_CHECK(err, err = (krnls[i]).setArg(7, hw_x->nr_values[block]));

            std::cout << "Copy input data to Device Global Memory\n";
            // Copy input data to Device Global Memory
            OCL_CHECK(err, err = q.enqueueMigrateMemObjects({ buffer_in1[i],
            buffer_in2[i] }, 0 /* 0 means from host*/));
        }
    }

        OCL_CHECK(err, err = q.finish());

    std::chrono::duration<double> kernel_time(0);
    std::cout << "kernel execution\n";
    auto kernel_start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < ComputeUnits; i++) {
        if ((hw_matrix[i]->nr_ci[block] + hw_matrix[i]->nr_val[block]) != 0)
        {
            OCL_CHECK(err, err = q.enqueueTask(krnls[i]));
```

```
        }
    }
    OCL_CHECK(err, err = q.finish());
    auto kernel_end = std::chrono::high_resolution_clock::now();

    kernel_time = std::chrono::duration<double>(kernel_end - kernel_start);

    // Copy Result from Device Global Memory to Host Local Memory
    std::cout << "Data back to host\n";
    for (int i = 0; i < ComputeUnits; i++) {
        if ((hw_matrix[i]->nr_ci[block] + hw_matrix[i]->nr_val[block]) != 0)
        {
            OCL_CHECK(err, err = q.enqueueMigrateMemObjects(
            { buffer_output[i] },
            CL_MIGRATE_MEM_OBJECT_HOST));
        }
    }
    OCL_CHECK(err, err = q.finish());

    return kernel_time.count();
}
```

Η run_krnl παίρνει ως ορίσματα τα context, queue και kernels που ορίστηκαν προηγουμένως καθώς και τους πίνακες που περιέχουν τα δεδομένα πακεταρισμένα στην μορφή που θα σταλούν στο FPGA και τον αριθμό του block για το οποίο καλείται. Ορίζει αρχικά 3 buffers για κάθε compute unit, οι οποίοι δεσμεύουν την μνήμη που απαιτείται για την εκτέλεση του κώδικα στο FPGA και μέσω αυτών θα αναφερόμαστε στα δεδομένα που μεταφέρονται στο FPGA. Για κάθε compute unit ορίζεται ένας buffer για τον υποπίνακα A, ένας για τα δεδομένα του x διανύσματος και ένας για τα αποτελέσματα του SpMV που αποθηκεύονται στο διάνυσμα y. Η εντολή που χρησιμοποιήθηκε για την δημιουργία των buffers είναι της μορφής:

```
OCL_CHECK(err, buffer_in1 = cl::Buffer(context, CL_MEM_READ_ONLY |
CL_MEM_USE_HOST_PTR, sizeof(BusDataType) * matrix_size, matrix_pointer,
&err));
```

```
OCL_CHECK(err, buffer_out1 = cl::Buffer(context, CL_MEM_WRITE_ONLY |
CL_MEM_USE_HOST_PTR, sizeof(BusDataType) * matrix_size, matrix_pointer,
&err));
```

για τα δεδομένα εισόδου (A,x) και εξόδου (y) αντίστοιχα. Στην συνέχεια περνάμε τα ορίσματα του κάθε πυρήνα με την σειρά που έχουν στην δήλωση της κερνελ συνάρτησης (spmv) στο αρχείο του kernel κώδικα(spmv.cpp) με την εξης εντολή:

```
OCL_CHECK(err, err = (krnl).setArg(nr_arg, argument));
```

Ακολουθεί η αντιγραφή των δεδομένων εισόδου για κάθε compute unit στην global μνήμη του FPGA με την εντολή:

```
OCL_CHECK(err, err = q.enqueueMigrateMemObjects ({ buffer_in1[i],
buffer_in2[i] }, 0));
```

όπου το 0 σημαίνει αντιγραφή από τον host στην συσκευή.

Τέλος, γίνεται η εκκίνηση όλων των πυρήνων και μετά από αυτό η μεταφορά όλων των αποτελεσμάτων πίσω στον host με τις εντολές:

```
OCL_CHECK(err, err = q.enqueueTask(krnls[i]));
```

```
OCL_CHECK(err, err = q.enqueueMigrateMemObjects({ buffer_output[i] },
CL_MIGRATE_MEM_OBJECT_HOST))
```

Όλες οι προηγούμενες εντολές εκτελούνται μέσα σε ξεχωριστά for loops για όλα τα compute unit του κάθε block αφού ελεγχθεί ότι το εκάστοτε compute unit δεν περιέχει μηδενικό αριθμό στοιχείων. Η συνάρτηση run_krnl επιστρέφει τον συνολικό χρόνο που διήρκησε μόνο η εκτέλεση των πυρήνων και στην περίπτωση κενού block επιστρέφει απευθείας μηδέν.

Σε επίπεδο διαχείρισης της μνήμης στην υλοποίηση του ZCU 102 λόγω της χρήσης των SDSoC εργαλείων οι πίνακες που στέλνονται στο hardware του FPGA δημιουργούνται με την χρήση της εντολής sds_alloc_non_cacheable και διαγράφονται με την χρήση της εντολής sds_free οι οποίες τώρα έχουν αντικατασταθεί με τα απλά malloc και free της C++. Επίσης έχουν αφαιρεθεί πλήρως όλες οι SDS pragma εντολές οι οποίες υπήρχαν στον ορισμό της SpMV συνάρτησης που εκτελούνταν στον πυρήνα και όριζαν τον τρόπο μεταφοράς των δεδομένων στο FPGA(port, access pattern, μορφή μνήμης) όπως βλέπουμε παρακάτω.

```
#pragma SDS data sys_port(submatrix1:AFI)
```

```
#pragma SDS data access_pattern(submatrix1:SEQUENTIAL)
```

```
#pragma SDS data zero_copy(submatrix1[0:(nr_ci1+nr_val1)])
```

```
#pragma SDS data mem_attribute(x:PHYSICAL_CONTIGUOUS|NON_CACHEABLE)
```

Στην υλοποίηση που αφορά το Alveo U280 η χρήση τέτοιων εντολών δεν απαιτείται και όλα όσα χρειάζονται για τον καθορισμό των δεδομένων που θα σταλούν στο FPGA ορίζονται κατά την δημιουργία των buffers όπως ακριβώς έχει περιγραφεί παραπάνω. Προσφέρεται βέβαια η δυνατότητα για έξτρα καθορισμό του τρόπου μεταφοράς των δεδομένων και της μνήμης που αυτά θα κατανεμηθούν, αλλά αυτό αφορά την βελτίωση της απόδοσης του πυρήνα και γίνεται μέσω της δήλωσης αυτών των πραγμάτων σε άλλα αρχεία που αφορούν το configuration του κώδικα αλλά και με την χρήση #pragma HLS των Vivado εργαλείων εντός του πυρήνα.

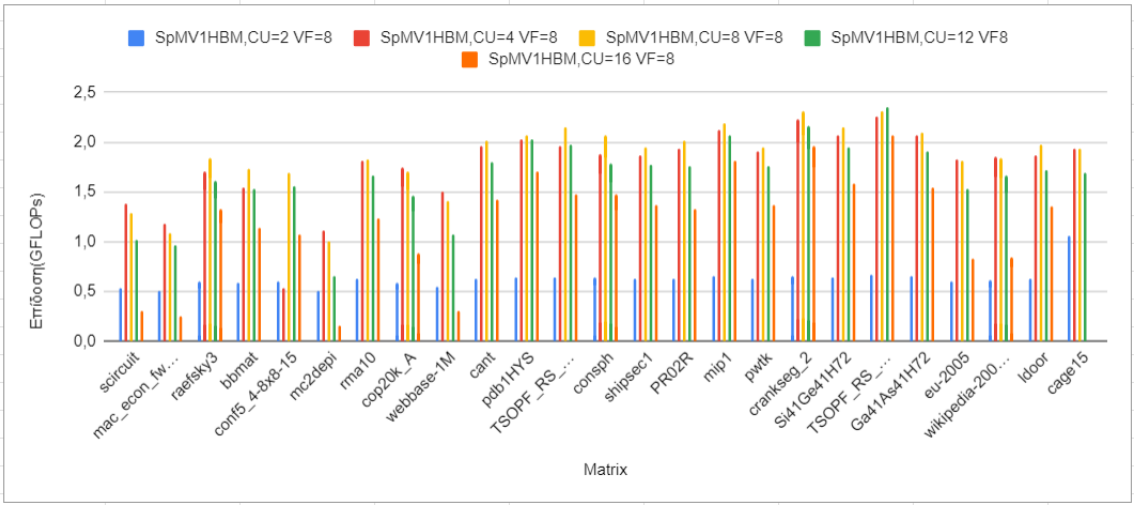
Τέλος έχει δημιουργηθεί το Makefile ακριβώς όπως στα παραδείγματα που δοθηκαν, ο αριθμός των Compute Units ορίζεται στο αρχείο spmv.cfg και μέσα στο util.h όπου ορίζεται και το Vectorization Factor. Επιπλέον, επειδή πλέον το εκτελέσιμο έχει δύο εισόδους, το xclbin file και το αρχείο με τον αραιό πίνακα, έχει χρησιμοποιηθεί parser και στην παράμετρο -x του εκτελέσιμου δίνεται το xclbin αρχείο και στην παράμετρο -i το αρχείο του αραιού πίνακα.

3.3.2 Αξιολόγηση πρώτης υλοποίησης

Σε όλες τις μετρήσεις που παρουσιάζονται παρακάτω έχουμε χρησιμοποιήσει σαν μετρική τα GFLOPs. Αποτελούν μετρική των πράξεων που γίνονται ανά μονάδα χρόνου. Για κάθε μη-μηδενικό στοιχείο του πίνακα εκτελούνται 2 πράξεις, μία του πολλαπλασιασμού και μία του

αθροίσματος, άρα για τον υπολογισμό της μετρικής πολλαπλασιάζουμε επί 2 τον αριθμό των μη-μηδενικών στοιχείων και διαιρούμε με τον χρόνο εκτέλεσης του πυρήνα.

Αρχικά στο διάγραμμα 3.2' παρατηρούμε ότι όσο αυξάνει ο αριθμός των compute units αυξάνεται και η απόδοση του FPGA μέχρι τα 8 compute units όπου παίρνει την μέγιστη τιμή της και στην συνέχεια αρχίζει να μειώνεται. Ο χρόνος εκτέλεσης μειώνεται όσο αυξάνονται τα compute units οδηγώντας σε αύξηση της απόδοσης σε GFLOPs, όπως βλέπουμε στο διάγραμμα 3.2. Παρατηρούμε ότι μετά τα 8 CUs αρχίζει να υπάρχει μείωση της απόδοσης η οποία ίσως οφείλεται στην αύξηση των padded μηδενικών με την αύξηση των CUs λόγω του μεγαλύτερου χωρισμού των στοιχείων του πίνακα.



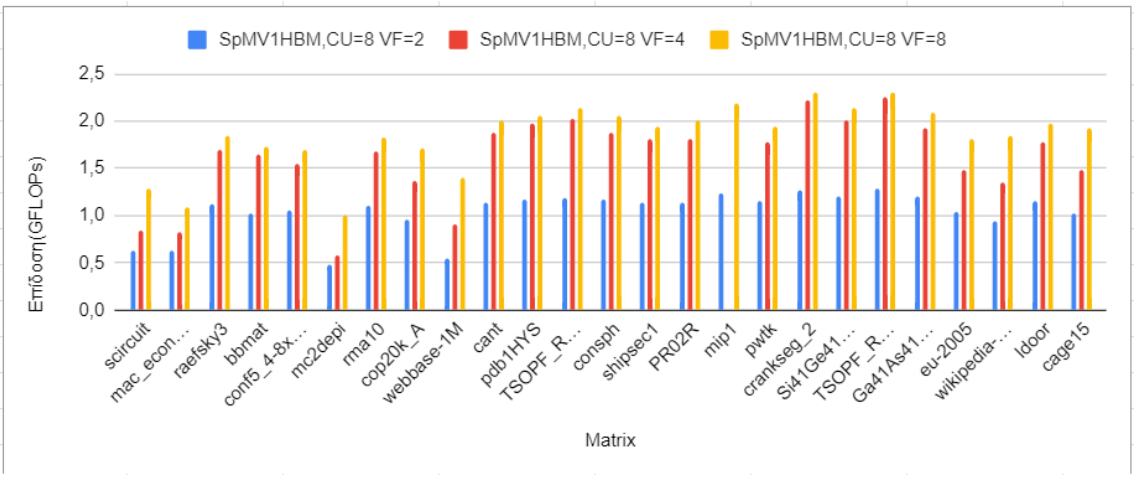
Εικόνα 3.2: *Επίδοση σε διαφορετικά πλήθη Compute Units*

Η σύγκριση της απόδοσης σε σχέση με την αλλαγή του Vectorization Factor έχει γίνει στα 8 CUs και έχουν δοκιμαστεί οι τιμές 2, 4, 8. Καθώς αυξάνουμε τον VF παρατηρούμε στην Εικόνα 3.3 αύξηση της επίδοσης του κώδικα σε GFLOPs. Με την αύξηση όμως του Vectorization Factor αυξάνεται αρκετά και ο αριθμός των μη μηδενικών στοιχείων που στέλνονται στο FPGA εξαιτιάς του padding το οποίο εξαρτάται άμεσα από το Vectorization Factor. Άρα υπάρχει αύξηση της επίδοσης σε πράξεις που γίνονται ανά χρόνο λόγω της αύξησης του αριθμού των πράξεων που γίνονται παράλληλα στα Compute Units αλλά ταυτόχρονα αυξάνεται και ο αριθμός των μη μηδενικών στοιχείων και κατ' επέκταση ο αριθμός των πράξεων που απαιτείται να γίνονται.

3.4 Πειραματισμοί για βελτίωση της επίδοσης

Στην παραπάνω υλοποίηση η διάθεση της μνήμης και των πόρων του FPGA καθορίζεται αυτόματα από τα Vitis Tools κατά το compile. Παρατηρήθηκε λοιπόν ότι ο compiler αναθέτει όλους τους buffer με τα δεδομένα που θα σταλούν στο FPGA σε ενα μόνο HBM Channel ενώ το FPGA διαθέτει 32 και εκτός αυτών υπάρχει και η PLRAM του FPGA. Για να αυξήσουμε λοιπόν την αποδοτικότητα της παραπάνω υλοποίησης αποφασίσαμε να ορίσουμε ακριβώς σε ποια μνήμη αντιστοιχείται κάθε buffer.

Αρχικά επειδή το x διανυσμα χρησιμοποιείται στις πράξεις με όλα τα στοιχεία του υπο-



Εικόνα 3.3: *Σύγκριση Vectorization Factor*

πίνακα A κάθε Compute Unit είναι αυτό το οποίο απαιτεί τις περισσότερες προσβάσεις οπότε τοποθετήθηκε στην PLRAM μνήμη για πιο ταχεία πρόσβαση, ομοίως με την ZCU υλοποίηση. Κάθε στοιχείο του υποπίνακα A χρησιμοποιείται μία μόνο φορά το οποίο σημαίνει ότι στον υποπίνακα A γίνονται πολλές σειριακές προσβάσεις. Οπότε οι buffers των A και y διανυσμάτων τοποθετούνται στα HBM κανάλια.

PLRAM Configuration

Για να είναι δυνατή η χρήση της PLRAM μνήμης απαιτείται πρώτα να γίνει κατα το compile:

- καθορισμός του μεγέθους της, ώστε να είναι επαρκής για τους buffer που θα σταλούν
- καθορισμός του είδους της PLRAM σε BlockRam ή UltraRam
- καθορισμός του data width και του read latency αναλογικά

Για την επίτευξη των παραπάνω έχουν προστεθεί στην υλοποίηση τα αρχεία plram_conf.tcl και advanced.cfg αρχεία. Το plram_conf.tcl περιλαμβάνει όλα τα παραπάνω χαρακτηριστικά για κάθε PLRAM από 0 έως 5. Αρχικά δοκιμάστηκε το μέγιστο μέγεθος για κάθε PLRAM που είναι τα 4MB και data width 512 bits και με βάση αυτά το read latency 10 όπως προτείνεται στο documentation της Xilinx χρησιμοποιώντας την παρακάτω λογική:

- 4 MB μνήμης συνολικά
- Κάθε UltraRAM είναι 32 KB (64 bits wide), άρα 4 MB επί 32 KB → 128 UltraRAMs σύνολο
- Κάθε PLRAM γραμμή έχει μήκος 512 bits άρα απαιτούνται 8 UltraRAMs για να καλύψουν το συνολικό μήκος κάθε γραμμής(512/64=8).
- Με 128 UltraRAMs συνολικά και 8 σε μήκος προκύπτει ότι θα έχουμε 16 UltraRAMs σε βάθος.
- Τέλος προτείνεται να διαλέγουμε το read latency ως εξής: read latency = depth/2 + 2 άρα τώρα διαλέγουμε read latency 10

Το αρχείο advanced.cfg χρησιμοποιείται για να περαστεί σαν παράμετρος στο compile το plram_conf.tcl αρχείο με την εντολή:

```
param=compiler.userPreSysLinkOverlayTcl=plram_conf.tcl
```

High Bandwidth Memory

Αρχικά δοκιμάσαμε να αναθέσουμε ένα HBM Channel ανά buffer για τους πίνακες A και y. Η ανάθεση των buffers στην μνήμη γίνεται στο spmv.cfg αρχείο. Στις εικόνες που ακολουθούν βλέπουμε μερικά πραδείγματα των κατανομών μνήμης που δοκιμάσαμε.

```
[connectivity]
nk=spmv:4:spmv_1.spmv_2.spmv_3.spmv_4
slr=spmv_1:SLR0
slr=spmv_2:SLR0
slr=spmv_3:SLR0
slr=spmv_4:SLR0
sp=spmv_1.submatrix1:HBM[0]
sp=spmv_1.y1:HBM[1]
sp=spmv_1.x:PLRAM[0]
sp=spmv_2.submatrix1:HBM[3]
sp=spmv_2.y1:HBM[4]
sp=spmv_2.x:PLRAM[1]
sp=spmv_3.submatrix1:HBM[6]
sp=spmv_3.y1:HBM[7]
sp=spmv_3.x:PLRAM[0]
sp=spmv_4.submatrix1:HBM[9]
sp=spmv_4.y1:HBM[10]
sp=spmv_4.x:PLRAM[1]
```

Εικόνα 3.4: *Κατανομή των HBM Channels και της PLRAM για 4 Compute Units*

Όπως είδαμε στην εισαγωγή το Alveo U280 χωρίζεται σε δυναμικές περιοχές στις οποίες μπορούν να τοποθετηθούν οι υπολογιστικοί πυρήνες. Σε αυτές τις περιοχές ανατίθενται οι υπολογιστικοί πόροι του FPGA. Το μεγαλύτερο πλεονέκτημα του Alveo U280 είναι η High Bandwidth Memory η οποία επικοινωνεί μόνο με την SLR 0. Έτσι κατά την σχεδίαση του κυκλώματος τοποθετούμε, όπως βλέπουμε στα παραδείγματα(3.4-3.6), όλους τους υπολογιστικούς μας πυρήνες στην SLR 0 με την slr compile option. Αρχικά δοκιμάσαμε απλή ανάθεση των HBM Channels, ένα ανά buffer στους πίνακες A και y και το x διάνυσμα το αναθέτουμε στις PLRAM 0 και 1, οι οποίες βρίσκονται στην SLR 0 για να αποφύγουμε την χρονοβόρα επικοινωνία μεταξύ διαφορετικών SLR.

Στην συνέχεια δοκιμάσαμε όλους τους δυνατούς συνδυασμούς. Τοποθετήσαμε και τους buffers του x διανύσματος σε HBM Channel για να εκμεταλλευτούμε το μεγαλυτερο εύρος ζώνης. Επίσης δοκιμάσαμε και την προσθήκη των PLRAM 2,3,4,5 σε περίπτωση που η επιπλέον μνήμη υπερνικούσε την καθυστέρηση του SLR Crossing. Τέλος, τα Vitis Tools δίνουν την δυνατότητα της ανάθεσης περισσότερων από ένα HBM Channels ανά buffer. Οπότε δοκιμάστηκε και αυτό στους μικρότερους αρθμούς Compute Units σε συνδυασμό και με

```
[connectivity]
nk=spmv:4:spmv_1.spmv_2.spmv_3.spmv_4
slr=spmv_1:SLR0
slr=spmv_2:SLR0
slr=spmv_3:SLR0
slr=spmv_4:SLR0
sp=spmv_1.submatrix1:HBM[0]
sp=spmv_1.y1:HBM[1]
sp=spmv_1.x:HBM[2]
sp=spmv_2.submatrix1:HBM[3]
sp=spmv_2.y1:HBM[4]
sp=spmv_2.x:HBM[5]
sp=spmv_3.submatrix1:HBM[6]
sp=spmv_3.y1:HBM[7]
sp=spmv_3.x:HBM[8]
sp=spmv_4.submatrix1:HBM[9]
sp=spmv_4.y1:HBM[10]
sp=spmv_4.x:HBM[11]
```

Εικόνα 3.5: *Κατανομή των buffers αποκλειστικά σε HBM Channels*

```
[connectivity]
nk=spmv:4:spmv_1.spmv_2.spmv_3.spmv_4
slr=spmv_1:SLR0
slr=spmv_2:SLR0
slr=spmv_3:SLR0
slr=spmv_4:SLR0
sp=spmv_1.submatrix1:HBM[0:3].1.RAMA
sp=spmv_1.y1:HBM[4:7].5.RAMA
sp=spmv_1.x:PLRAM[0:1]
sp=spmv_2.submatrix1:HBM[8:11].9.RAMA
sp=spmv_2.y1:HBM[12:15].13.RAMA
sp=spmv_2.x:PLRAM[2:3]
sp=spmv_3.submatrix1:HBM[16:19].17.RAMA
sp=spmv_3.y1:HBM[20:23].21.RAMA
sp=spmv_3.x:PLRAM[4]
sp=spmv_4.submatrix1:HBM[24:27].25.RAMA
sp=spmv_4.y1:HBM[28:31].29.RAMA
sp=spmv_4.x:PLRAM[5]
```

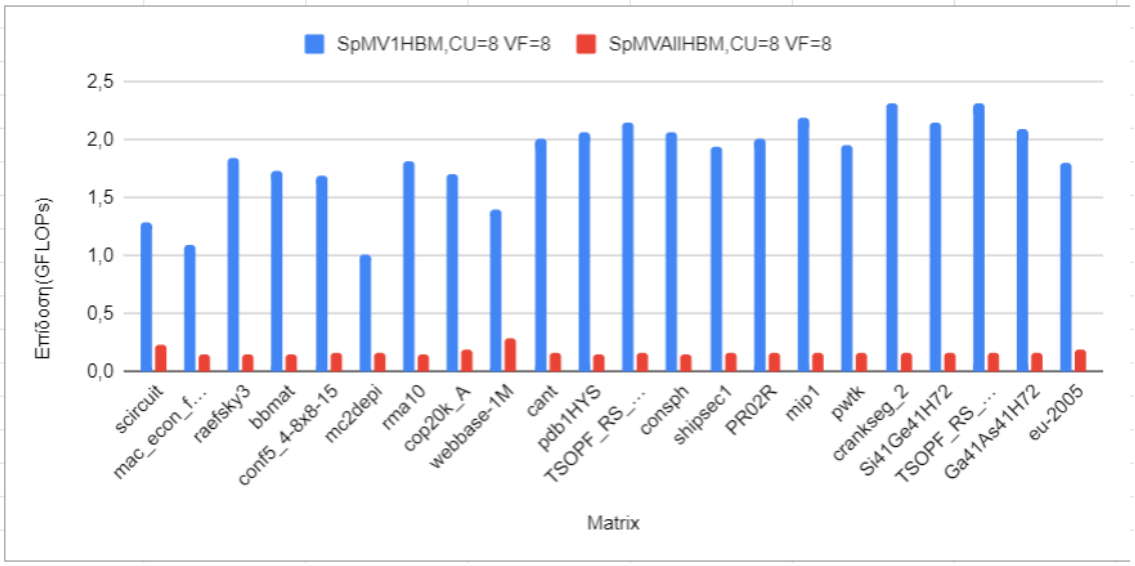
Εικόνα 3.6: *Κατανομή των buffers αποκλειστικά σε HBM Channels*

την RAMA IP τεχνική. Πιο αναλυτικά, το High Bandwidth Memory (HBM) υποσύστημα των Virtex UltraScale+ συσκευών έχει καλή απόδοση όταν πρέπει να διαχειριστεί σειριακή πρόσβαση δεδομένων στην μνήμη αλλά η απόδοση του για τυχαία πρόσβαση δεδομένων ποικίλλει ανάλογα με το μέγεθος της μνήμης, τον ρυθμό ανάγνωσης και εγγραφής, το ελάχιστο μέγεθος λέξης κλπ. Η RAMA IP τεχνική διαχειρίζεται τα παραπάνω προβλήματα με την χρήση AXI ID substitution and response reordering και ταυτόχρονα επιτρέπει και την χρήση

μεγαλύτερων από 256MB (ένα HBM Channel) buffers. Η RAMA IP επιτυγχάνεται με την προσθήκη του .<s_axilite index>.RAMA στο τέλος της δήλωσης κάθε buffer που ανατίθεται στα HBM Channels.

3.4.1 Αξιολόγηση

Παρόλες τις παραπάνω προσπάθειες η προσθήκη περισσότερων HBM καναλιών δεν μείωσε σημαντικά την επίδοση της υλοποίησης μας αντί να την αυξήσει όπως θα ήταν αναμενόμενο ανεξαρτήτως της κατανομής μνήμης που εφαρμόζαμε. Όλοι οι συνδυασμοί κατανομής μνήμης οδηγούσαν σε μετρήσεις παραπλήσιες αυτών που παρουσιάζονται στην Εικόνα 3.7. Παρατηρούμε ότι οι πίνακες που χωρίζονται σε μικρότερο αριθμό blocks, δηλαδή οι πίνακες με τον μικρότερο αριθμό στηλών, εμφανίζουν σημαντικά καλύτερες επιδόσεις σε σύγκριση με τους υπόλοιπους. Μία πιθανή εξήγηση είναι ότι η διάσπαση σε blocks, η οποία είναι απαραίτητη για την χρήση της BRAM, προσθέτει πολύ χρόνο στους υπολογισμούς.



Εικόνα 3.7: *Απόδοση υλοποίησης με όλα τα HBM Κανάλια*

Σε αυτό το σημείο θεωρήσαμε ότι η δημιουργία stream και για τις τιμές του x διανύσματος και συνεπώς η κατάργηση του χωρισμού των πινάκων σε blocks θα βοηθούσε στην καλύτερη αξιοποίηση όλης της διαθέσιμης μνήμης και στην βελτίωση της απόδοσης. Άλλωστε ένα βασικό πλεονέκτημα του Alveo U280 είναι η High Bandwidth Memory η οποία προσφέρει μεγάλο bandwidth και συμβάλλει στην ταχεία μεταφορά των δεδομένων. Είναι συνεπώς πολύ λογικό να συμφέρει το αποκλειστικό streaming όλων των δεδομένων. Αυτό μας οδήγησε στην εισαγωγή του Optima SpMV Project όπως θα αναλύσουμε στα επόμενα κεφάλαια.

3.5 Εισαγωγή Optima SpMV Project

Σε προσπάθεια βελτίωσης της επίδοσης εξετάσαμε το Optima SpMV Project και προσπαθήσαμε αρχικά να μεταφέρουμε κάποια στοιχεία του στην δική μας υλοποίηση και στην συνέχεια το αντίστροφο.

Variable	HBM channel	m_axi port	Bit-width
iat = row_ptr	HBM[0]	gmem0	64 bits
ja = col_ind	HBM[1]	gmem1	256 bits (8 integers)
coef = values	HBM[1]	gmem1	512 bits (8 doubles)
x	HBM[2]	gmem2	64 bits
b = y	HBM[0]	gmem0	64 bits

Πίνακας 3.2: *Κατανομή των HBM Channels για 1 Compute Unit στο Optima Design*

3.5.1 Περιγραφή Optima SpMV design

Το Optima SpMV Design αποτελεί μία απλούστερη υλοποίηση του SpMV. Χρησιμοποιεί την απλή CSR αναπαράσταση και τον αντίστοιχο υπολογιστικό αλγόριθμο. Ταυτόχρονα, για την υλοποίηση χρησιμοποιούνται πιο απλές δομές συγκριτικά με την δική μας υλοποίηση. Αυτό είναι εφικτό καθώς με τα Vitis Tools είναι εφικτή η μεταφορά σε wide δομές στον υπολογιστικό πυρήνα χωρίς να είναι απαραίτητη η δημιουργία wide δομών στον host κώδικα.

Βασικό πλεονέκτημα αυτού του design είναι η κατανομή των HBM Channels και των axi ports. Όπως βλέπουμε στον Πίνακα 3.2 τοποθετεί στο ίδιο channel την έξοδο y(b) με το διάνυσμα των row indices (iat) και ομοίως τα column indices(ja) με τα values(coef). Τοποθετούμε τα iat και b στο ίδιο Channel επειδή οι προσβάσεις σε αυτά είναι πιο σπάνιες από τα ja και coef. Επίσης, είναι απαραίτητο η πρόσβαση στα ja και coef να γίνεται ¨ταυτόχρονα¨. Αυτό θα μπορούσε να επιτευχθεί και με την ανάθεση τους σε διαφορετικά HBM Channels αλλά τότε θα μειώνόταν ο μέγιστος αριθμός Compute Units στις οποίες θα μπορούσαμε να χωρίσουμε τους υπολογισμούς και η υλοποίηση μας θα υστερούσε σε scalabity και επίδοση. Η βελτίωση που επιφέρει στην απόδοση η παραπάνω λογική κατανομής της HBM επιβεβαιώνεται και από μετρήσεις που θα παρουσιάσουμε στην συνέχεια.

Ο υπολογιστικός πυρήνας του Optima Design είναι βασισμένος σε μία απλή CSR αναπαράσταση και άρα ακολουθεί τον αλγόριθμο που περιγράψαμε στην εισαγωγή. Δηλαδή όλοι οι υπολογισμοί γίνονται ανά γραμμές με βάση τον row_ptr πίνακα. Τέλος, ο χωρισμός σε Compute Units γίνεται με balancing των γραμμών και όχι των στοιχείων και δεν υπάρχει χωρισμός σε blocks.

3.5.2 Διαφορές μεταξύ των δύο design

- Στο Optima-SpMV design γίνεται ξεχωριστά το streaming των column indices και των values(μη μηδενικών στοιχείων του A πίνακα) ενώ στο δικό μας αρχικό design δημιουργείται μια ενιαία δομή η οποία μεταφέρεται μέσω stream και μετά διαχωρίζεται σε column indices και values stream. Αυτό οδηγεί και στην χρήση διαφορετικού buswidth στα δύο streams, ούτως ώστε να μεταφέρεται ο ίδιος αριθμός column indices και values σε περίπτωση που ανήκουν σε διαφορετικά datatype και άρα παρουσιάζουν διαφορετικό αριθμό bit.

- Το Ratio_v ταυτίζεται με τον Vectorization Factor στο Optima-SpMV design. Ως Ratio_v επιλέγεται ο αριθμός 8, ο οποίος είναι το latency της πρόσθεσης double αριθμων στο Alveo U280, ώστε τα στοιχεία να δημιουργούν streams από wide δομές των 8 και να γίνεται vectorization (loop unrolling) ανάλογου βαθμού.

- Τέλος, η βασική διαφορά μεταξύ των δύο design βρίσκεται στην αποθήκευση του x vector στο FPGA. Στο δικό μας αρχικό design ο x vector αποθηκεύεται στην BRAM του FPGA για να επιταχύνονται οι προσβάσεις στην μνήμη για το x, το οποίο χρησιμοποιείται παραπάνω από μία φορές. Αυτό βέβαια ήταν χρήσιμο στο ZCU 102 αλλά το Alveo U280, με την High Bandwidth Memory(HBM) είναι ίσως πιο αποδοτικό να μεταφέρονται όλα τα δεδομένα μέσω streams στο FPGA αντί να γίνονται συνεχείς προσβάσεις στην PLRAM του Alveo U280. Επίσης όταν το x αποθηκεύεται στην BRAM είναι απαραίτητος ο διαχωρισμός του αρχικού πίνακα σε blocks, έτσι ώστε να χωράνε αρκετά αντίγραφα του x στην BRAM για όλα τα Compute Units, το οποίο προσθέτει επιπλέον καθυστέρηση. Στο Optima-SpMV design αποφεύγονται όλα αυτά χρησιμοποιώντας την ουρά των column indices για την εύρεση του x value που αντιστοιχεί στο κάθε μη μηδενικό στοιχείο του A και στην συνέχεια εισάγοντας αυτά τα values σε streams για να γίνει η διαδικασία του πολλαπλασιασμού.

- Η ουσιαστική διαφορά μεταξύ των δύο σχεδίων αφορά την πολυπλοκότητα του κώδικα και κυρίως των δομών που απαιτούνται για την αναπαράσταση των δεδομένων. Αυτό συμβαίνει γιατί τα Vitis Tools δίνουν την δυνατότητα δήλωσης απλών unpacked δομών στον host κώδικα και στην συνέχεια την απορρόφηση των ίδιων δεδομένων σε wide δομές περισσότερων στοιχείων στον kernel κώδικα απλά με δήλωση διαφορετικού τύπου στην είσοδο της συνάρτησης. Δεν υπάρχει, δηλαδή, η ανάγκη επεξεργασίας των δεδομένων και δημιουργίας wide δομών από τον προγραμματιστή στον host κώδικα, όπως στο SDSoC. Αντίθετα, η δήλωση wide δομών στον kernel κώδικα αρκεί για την αποστολή των αρχικών δεδομένων από τον host μέσω wide interfaces, οδηγώντας έτσι στην δημιουργία απλούστερου κώδικα με ακριβώς ίδιες λειτουργικότητες και στην διευκόλυνση του προγραμματισμού των FPGA.

3.5.3 Εισαγωγή Στοιχείων από το Optima SpMV design στην αρχική υλοποίηση

Σε μία πρώτη προσπάθεια για την μεταφορά των χρήσιμων χαρακτηριστικών του Optima-SpMV στο δικό μας design, αφαιρούμε την αποθήκευση του x και δημιουργούμε ένα stream για το x vector. Οι αλλαγές που απαιτούνται γι' αυτό έγιναν κυρίως στον κώδικα του πυρήνα και στον host κώδικα στο πακετάρισμα του x διανύσματος.

Kernel Code

Στο αρχικό design υπήρχε η read_data_submatrix συνάρτηση η οποία χωρίζει τον πίνακα submatrix που περιέχει τα column indices και τα values του A σε δύο wide streams, col_fifo_wide και values_fifo_wide. Στην συνέχεια αυτά τα streams περνάνε αντίστοιχα στις stream_data_col_ind και stream_data_values όπου τα στοιχεία γίνονται unpack στα col_fifo και values_fifo. Τέλος τα streams αυτά καταλήγουν στην compute_results η οποία με βάση τα στοιχεία του col_fifo φέρνει από την BRAM το ανάλογο στοιχείο του x και εκτελεί τον πολλαπλασιασμό. Το x έχει αποθηκευτεί στην BRAM στην συνάρτηση SpMV όπου περνιέται σε wide δομή και αποθηκεύεται αφού γίνεται unpack.

Για να γίνει το streaming του x αρχικά αλλάξαμε την μορφή που δίνεται σαν όρισμα και από wide δομή (BusDataType) την αλλάξαμε σε unpacked δομή (ValueType) για να είναι πιο εύκολο να βρίσκουμε το σωστό στοιχείο του x με βάση τα column indices. Επίσης αφαιρέσαμε από την συνάρτηση SpMV όλο το κομμάτι κώδικα το οποίο έκανε την αντιγραφή του x στην local μνήμη του FPGA. Στην συνέχεια αυτό που ουσιαστικά άλλαξε είναι η συνάρτηση stream_data_col_ind. Κάνουμε κανονικά το unpacking της col_fifo_wide αλλά αντί να περνάμε τα column indices σε ένα col_fifo βρίσκουμε με βάση αυτά τα αντίστοιχα x και περνάμε αυτά σε ένα x_values_fifo. Ταυτόχρονα επειδή σε κάθε column index το τελευταίο bit σηματοδοτεί το αν υπάρχει ή όχι αλλαγή γραμμής εκεί δημιουργούμε μία row_ptr_fifo stream στην οποία περνάμε τα τελευταία bit των column indices. Έτσι πλέον δεν χρειαζόμαστε το col_fifo για την compute_results συνάρτηση αλλά μόνο τα values_fifo, row_ptr_fifo, x_values_fifo.

Host Code

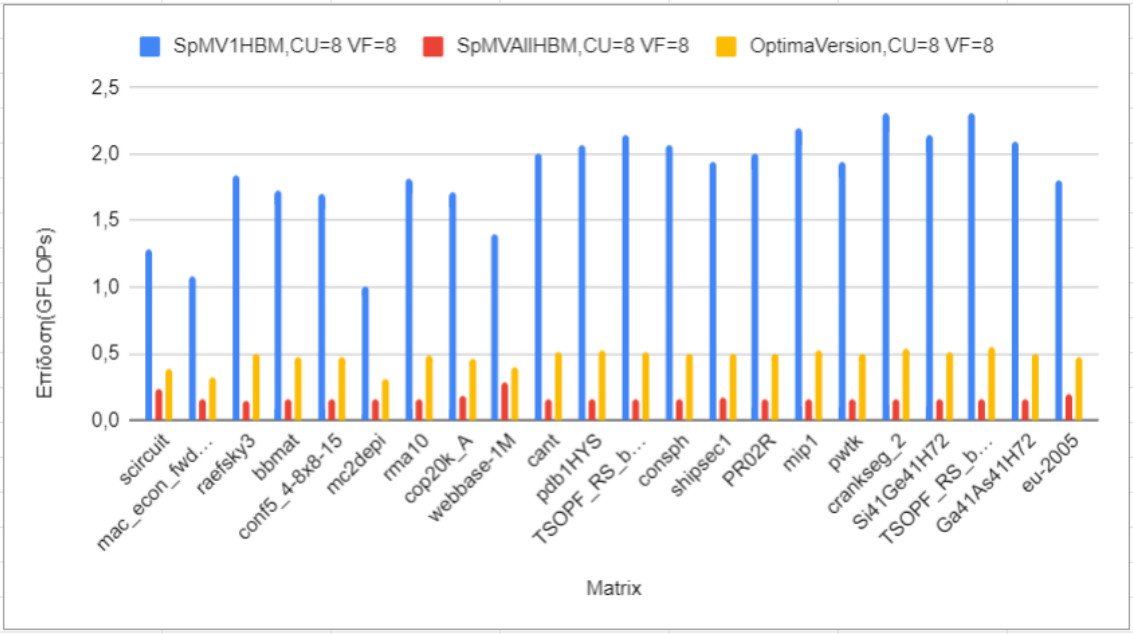
Στο αρχικό design πακετάραμε το x σε wide δομές για να το στείλουμε στο FPGA. Τώρα δεν χρειάζεται αυτό το πακετάρισμα για να διατηρήσουμε όμως τον διαχωρισμό σε blocks ακολουθούμε μία ανάλογη διαδικασία. Δημιουργούμε μία καινούρια δομή csr_hw_x_vector η οποία είναι ίδια με την csr_hw_vector αλλά αποθηκεύει τα αλυες σε ValueType αντί για BusDateType. Δημιουργούμε επίσης δύο καινούριες συναρτήσεις στο csr_hw.cpp αρχείο τις create_csr_hw_x_vector και write_csr_hw_x_vector οι οποίες καλούνται από την create_csr_hw_x_vector του csr_hw_wrapper.cpp αρχείου στην θέση των create_csr_hw_vector και write_csr_hw_x_vector. Η διαφορά τους από τις αρχικές είναι η αλλαγή των τύπων από csr_hw_vector σε csr_hw_x_vector και από BusDataType σε ValueType και η αφαίρεση της διαίρεσης με Ratio_v. Επίσης στην write_csr_hw_x_vector αφαιρέθηκε ένα εμφωλευμένο for αφού πλέον δεν χρειάζεται να περνάμε τα στοιχεία ανά ομάδες των Ratio_v στοιχείων. Τέλος δημιουργήθηκαν και οι αντίστοιχες delete συναρτήσεις.

Αλλάξαμε επίσης το BUS_BIT_WIDTH από 128 σε 512 bits για να επιτευχθεί η αποστολή 8 στοιχείων. Επειδή όμως στην περίπτωση των doubles το INDEX_TYPE_BIT_WIDTH είναι το μισό από το VALUE_TYPE_BIT_WIDTH, αυτό σημαίνει ότι τα column indices στέλνονται ανά 16 και όχι ανά 8. Αυτό θα μπορούσε να αλλάξει αν στέλνουμε χωριστά τα column indices και values.

3.5.4 Αξιολόγηση

Στο διάγραμμα 3.8 παρατηρούμε βελτίωση των αποτελεσμάτων συγκριτικά με την απλή προσθήκη των HBM Channels αλλά δυστυχώς τα αποτελέσματα συνεχίζουν να είναι χειρότερα από την χρήση ενός HBM Channel.

Υποθέτουμε ότι ίσως ο κώδικας να παρουσιάζει αρκετά περίπλοκες δομές δεδομένων και ότι το πακεταρισμα-ξεπακεταρισμα των στοιχείων μέσω της κοινής submatrix δομής προσθέτει καθυστερήσεις οι οποίες εντείνονται με την προσθήκη περισσότερων HBM Channels. Έτσι αποφασίσαμε να ακολουθήσουμε αντίστροφη πορεία και να εισάγουμε τα βέλτιστα στοιχεία της αρχικής μας υλοποίησης στο απλούστερο Optima SpMV Project.



Εικόνα 3.8: *Πρώτη Εισαγωγή του Optima και Σύγκριση με τις υπόλοιπες Υλοποιήσεις*

3.6 Εισαγωγή Στοιχείων από την αρχική υλοποίηση στο Optima SpMV design

Η μεταφορά των βασικών στοιχείων του Optima project στην δική μας υλοποίηση βελτίωσε τους χρόνους αλλά όχι σε ικανοποιητικό επίπεδο. Δεδομένου ότι ο κώδικας της αρχικής υλοποίησης προοριζόταν αρχικά για ένα FPGA παλαιότερης τεχνολογίας, όλη η διαδικασία πακεταρίσματος των δεδομένων σε wide δομές, ο χωρισμός σε block και Compute Units, το padding απαιτούσαν την δημιουργία πολύπλοκων δομών οι οποίες ίσως καθιστούσαν πιο χρονοβόρα την εκτέλεση του υπολογιστικού πυρήνα και την πρόσβαση στα δεδομένα.

Το Optima SpMV design είναι πολύ απλούστερο στις δομές που πακετάρει τα δεδομένα. Αυτό συμβαίνει γιατι όπως προαναφέραμε με τα Vitis Tools είναι εφικτή η μεταφορά σε wide δομές στον υπολογιστικό πυρήνα χωρίς να είναι απαραίτητη η δημιουργία wide δομών στον host κώδικα. Επίσης λόγω του streaming του διανύσματος x που εφαρμόζεται στο Optima SpMV project δεν είναι απαραίτητος ο χωρισμός των πινάκων σε blocks. Έτσι δοκιμάσαμε να μεταφέρουμε στο Optima Project χαρακτηριστικά από τη αρχική μας υλοποίηση που θα βελτιώσουν την επίδοση του.

Στο Optima SpMV Project χρησιμοποιούνται διαφορετικές ονομασίες για τους πίνακες τις οποίες υιοθετούμε στις περιγραφές μας απο δω και πέρα. Έτσι, ο πίνακας που αφορά τις γραμμές του πίνακα A θα αναφέρεται και ως iat, τα values ως coef, τα column indices θα ονομάζονται ja και η έξοδος y θα αναφέρεται ως b.

3.6.1 Αλλαγή του τρόπου αναπαράστασης του Αραιού Πίνακα

Το βασικό πλεονέκτημα της αρχικής μας υλοποίησης είναι η αντικατάσταση του row_ptr πίνακα της CSR αναπαράστασης με την χρήση ενός πίνακα από bits, ενδεικτικών για την αλλαγή γραμμής, για κάθε στοιχείο του πίνακα. Οπότε σαν αρχικό βήμα αλλάξαμε τον

τρόπο αναπαράστασης των row και column indices του πίνακα A. Με βάση τον πίνακα irow_-A που περιέχει την γραμμή κάθε στοιχείου δημιουργούμε το bit του κάθε στοιχείου που δείχνει αν υπάρχει αλλαγή γραμμής μετά από αυτό. Με την βοήθεια της ap_int.h βιβλιοθήκης ορίζουμε τους τύπους `ap_uint<1> RowType` και `ap_uint<32> ColType` και με την εντολή .range ενώνουμε το κάθε row bit με το αντίστοιχο column index όπως βλέπουμε στην Εικόνα 3.18.

```
{

   int j = 0;
   for ( int i = 1; i < nterm_A; i++) {
    if( irow_A[i] > j ) {
         ja_A[i-1].range(31,31) = 1;
         j++;
      }
    else{
         ja_A[i-1].range(31,31) = 0;
    }

   }
}
ja_A[nterm_A-1].range(31,31) = 1;
```

Εικόνα 3.9: *Δημιουργία του row bit και ένωση του με τα column indices*

Στην συνέχεια προσαρμόσαμε τον πυρήνα στην καινούρια αναπαράσταση. Αρχικά πλέον δεν χρειάζεται η ξεχωριστή μεταφορά του iat οπότε αφαιρούμε την δημιουργία του αντίστοιχου buffer στο host τμήμα του κώδικα και την κατανομή του σε HBM Channel καθώς και την συνάρτηση που είναι υπεύθυνη για την δημιουργία του iat stream στον πυρήνα. Αντ΄ αυτού δημιουργείται πλέον ένα καινούριο iatstream μέσα στην read_x συνάρτηση κατά την διάσπαση του column index και του row bit για την δημιουργία του x_stream όπως παρατηρούμε στην Εικόνα 3.10.

```
void read_x(hls::stream<v_dti> &ja, double *x, hls::stream<v_dt> &xstream, hls::stream<v_dtr> &iatStream, int nterm) {
    rd_x:
    ColType col;
    v_dti coltmp;
    v_dt xtmp;
    v_dtr rowtmp;

    LOOP_rd_x:
    for(int i = 0; i < nterm; i+=FADD_LAT) {
        #pragma HLS PIPELINE
        coltmp = ja.read();
        LOOP_INNER_rd_x:
        for(int k = 0; k < FADD_LAT; k++) {
            #pragma HLS UNROLL
            col = coltmp.data[k].range(0,30);
            rowtmp.data[k] = coltmp.data[k].range(31,31);
            xtmp.data[k] = x[col];
        }
        xstream << xtmp;
        iatStream << rowtmp;
    }
}
```

Εικόνα 3.10: *Δημιουργία x και iat streams*

Τέλος, η εκτέλεση του πολλαπλασιασμού παρουσιάζει την μεγαλύτερη αλλαγή. Ο καινο-

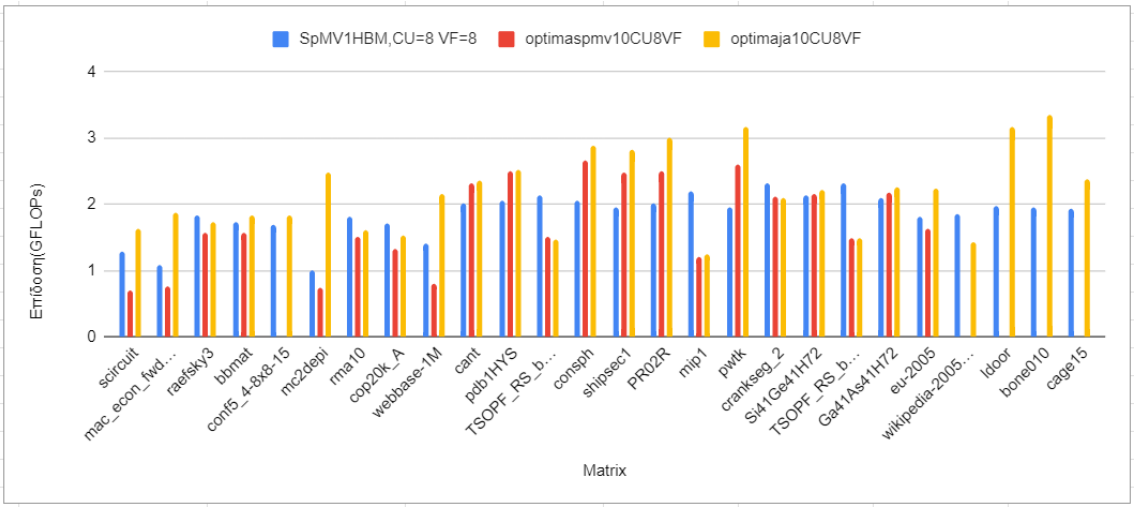
ύριος αλγόριθμος πολλαπλασιασμού είναι στην ουσία ίδιος με τον αλγόριθμο πολλαπλασιασμού της αρχικής μας υλοποίησης. Πλέον, η εκτέλεση των πράξεων γίνεται ανά οκτάδες στοιχείων και όχι ανά γραμμές και για κάθε υπολογισμό διαβάζεται από ένα στοιχείο από τις ουρές iat, x και coef. Η μόνη διαφορά είναι η αντικατάσταση του loop unroll της πρόσθεσης των επιμέρους αποτελεσμάτων των οκτάδων με την σταδιακή ανά δύο πρόσθεση τους (tree-reduction), καθώς αυτό βελτίωσε τους χρόνους εκτέλεσης.

```
// }
tmp_red1 = sum_p[0]+sum_p[4];
tmp_red2 = sum_p[1]+sum_p[5];
tmp_red3 = sum_p[2]+sum_p[6];
tmp_red4 = sum_p[3]+sum_p[7];

tmp_red5 = tmp_red1+tmp_red3;
tmp_red6 = tmp_red2+tmp_red4;

sum = tmp_red5 + tmp_red6;
```

Εικόνα 3.11: *Tree-reduction των προσθέσεων του τελικού αποτελέσματος*

3.6.2 Αξιολόγηση και Πειραματισμός με διαφορετική κατανομή των HBM Channels

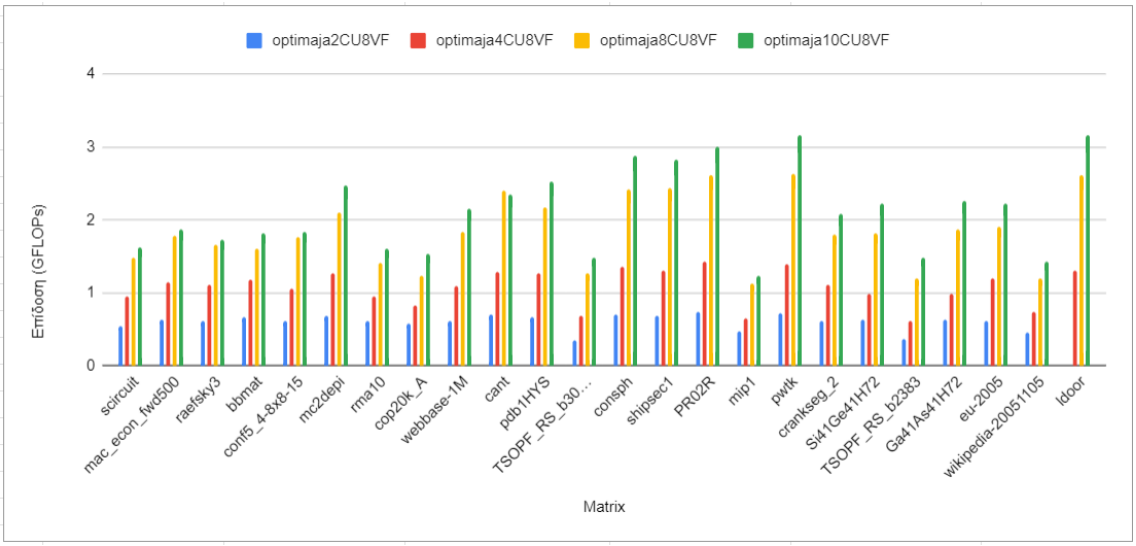
Όπως βλέπουμε στο διάγραμμα 3.12, η αλλαγή της αναπαράστασης του πίνακα A και κατ΄ επέκταση του υπολογιστικού πυρήνα βελτίωσε αισθητά την απόδοση του κώδικα σε σύγκριση με το αρχικό Optima SpMV Project αλλά και με την δική μας αρχική υλοποίηση. Παρουσιάζουμε τις βέλτιστες μετρήσεις από την μέχρι τώρα υλοποίηση (SpMV1HBM) για 8 CUs και 8 VF, το αρχικό Optima SpMV (optimaspmv) για 10 CUs και 8 VF και την τελευταία υλοποίηση που αναλύσαμε στο 3.6.1(optimaja) για 10 CUs και 8 VF.



Εικόνα 3.12: *Επίδοση Πρώτης Υλοποίησης βασισμένης στο Optima Design*

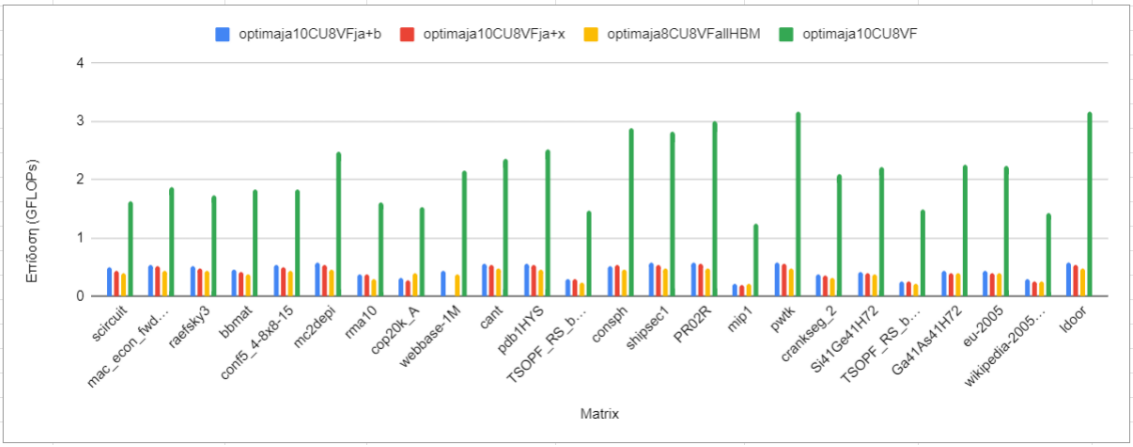
Παρατηρούμε στο διάγραμμα 3.12 ότι υπάρχει μια γενική αύξηση της επίδοσης με μερικούς πίνακες να παρουσιάζουν έως και διπλάσια απόδοση τόσο συγκριτικά με την αρχική μας υλοποίηση αλλά και σε σχέση με το Optima SpMV Design.

Σε αυτό το πλαίσιο εξετάσαμε την συμπεριφορά και σε διαφορετικά πλήθη Compute Units. Δεδομένου του ότι το FPGA παρέχει 32 HBM Channels, το μέγιστο εφικτό πλήθος Compute Units , που μπορεί το Optima SpMV να πετύχει με την κατανομή μνήμης που περιγράφηκε είναι 10. Δοκιμάζουμε και μικρότερα πλήθη Compute Units και βλέπουμε στο διάγραμμα 3.13 ότι η μέγιστη επίδοση επιτυγχάνεται για το μέγιστο αριθμό Compute Units. Αυτό πλέον είναι λογικό καθώς το padding των στοιχείων επηρεάζεται μόνο απο το Vectorization Factor, γίνεται ανά γραμμή και άρα δεν επηρεάζεται από την διάσπαση σε Compute Units.



Εικόνα 3.13: *Επίδοση σε διαφορετικά πλήθη Compute Units*

Τέλος, δοκιμάζουμε την επίδοση του κώδικα με διαφορετικές κατανομές μνήμης. Αντικαθιστούμε την τοποθέτηση των ja και coef στο ίδιο channel με την τοποθέτηση του ja και x ή του ja και b στο ίδιο channel. Επίσης, δοκιμάζουμε να τοποθετήσουμε όλους τους buffers σε ξεχωριστά HBM Channels, οπότε το μέγιστο πλήθος Compute Units γίνεται 8.



Εικόνα 3.14: *Πειραματισμός με διαφορετική κατανομή των HBM Channels*

Με βάση τις μετρήσεις στο διάγραμμα 3.14 επιβέβαιώνουμε ότι η βέλτιστη κατανομή των HBM Channels είναι η αρχική του Optima SpMV η οποία τοποθετεί μαζί τα ja και coef. Η τοποθέτηση τους στην ίδια μνήμη διευκολύνει την ταυτόχρονη προσπέλαση τους κατά το διάβασμα των streams και ευνοεί το scalability.

3.6.3 Δημιουργία του x-stream στον host κώδικα

Παρατηρήσαμε ότι για την δημιουργία του x stream στον πυρήνα γίνονται πολλές τυχαίες προσβάσεις στο διάνυσμα x οι οποίες κοστίζουν αρκετά σε χρόνο. Επίσης έχουμε, πλέον, αναθέσει το x σε HBM Channel το οποίο ευνοεί τις sequential προσβάσεις στα δεδομένα που μεταφέρει, οπότε οι random προσβάσεις που εφαρμόζονταν μέχρι τώρα δεν ευνοούν αυτήν τη αρχιτεκτονική. Σε προσπάθεια, λοιπόν, περαιτέρω βελτίωσης των αποτελεσμάτων δοκιμάσαμε να δημιουργήσουμε έναν πίνακα με τα δεδομένα που είναι απαραίτητα για το x stream πριν την αποστολή των δεδομένων στο FPGA, για να αποφύγουμε τις τυχαίες προσβάσεις στην μνήμη κατα την εκτέλεση του πυρήνα.

Για να το επιτύχουμε αυτό δημιουργήσαμε στον host κώδικα έναν πίνακα x_stream, ο οποίος περιλαμβάνει για κάθε value του πίνακα A το x value που του αντιστοιχεί στον πολλαπλασιασμό. Όπως βλέπουμε και στον κώδικα της Εικόνας 3.15, ουσιαστικά χρησιμοποιήσαμε τον πίνακα ja_A και τον διασπάσαμε δημιουργώντας έναν πίνακα με row_bits και χρησιμοποιώντας τα column indices έναν πίνακα με το x που αντιστοιχεί σε κάθε value του A. Ότι ακριβώς γινόταν μέχρι τώρα στον kernel code.

```
double* x_stream;
RowType* iatbit;
iatbit = (RowType*) OOPS_malloc((size_t)(nterm*sizeof(RowType)));
x_stream = (double*) OOPS_malloc((size_t)(nterm*sizeof(double)));
ColType col;
for(int i = 0; i < nterm; i++) {
    col = ja[i].range(0,30);
    iatbit[i] = ja[i].range(31,31);
    x_stream[i] = x[col];
}
```

Εικόνα 3.15: *Δημιουργία x stream στον host code*

Πλέον, λοιπόν, δεν υπάρχει ανάγκη μεταφοράς των column indices (ja_A) στον πυρήνα καθώς χρησιμοποιούνταν μόνο για την ευρέση του x. Έτσι, μεταφέρουμε τα values του A (coef), το x και τον πίνακα με τα row bits, ο οποίος μεταφέρεται μέσω του ίδιου Channel με τα values του A.

Τέλος το διάβασμα του x στον πυρήνα δεν περιλαμβάνει τυχαίες προσβάσεις αλλά αποτελεί ουσιαστικά μια απλή δημιουργία stream wide δομών όπως ακριβώς και το διάβασμα των coef και iat που τοποθετήσαμε σε μία συνάρτηση καθώς μεταφέρονται μέσω του ίδιου HBM Channel και του ίδιου axi port.

Η δημιουργία του x-stream στον host κώδικα έγινε με στόχο την μείωση του χρόνου εκτέλεσης του kernel code αλλά ταυτόχρονα οδήγησε σε αύξηση των δεδομένων που απο-

```
[connectivity]
sp=krnl_spmv_csr_1.iat:HBM[1]
sp=krnl_spmv_csr_1.coef:HBM[1]
sp=krnl_spmv_csr_1.x:HBM[2]
sp=krnl_spmv_csr_1.b:HBM[0]

sp=krnl_spmv_csr_2.iat:HBM[4]
sp=krnl_spmv_csr_2.coef:HBM[4]
sp=krnl_spmv_csr_2.x:HBM[5]
sp=krnl_spmv_csr_2.b:HBM[3]
```

Εικόνα 3.16: *Κατανομή Μνήμης (ενδεικτικά για 2 Compute Units)*

στέλουμε στο FPGA και αύξηση στον χρόνο προετοιμασίας των δεδομένων. Πιο συγκεκριμένα, μέχρι τώρα μεταφέραμε στο FPGA nnz column indices(32 bits) και CUs x nrows x-values(64 bits) ενώ με την κανούρια υλοποίηση μεταφέρουμε nnz x-values(64 bits). Δεδομένου ότι ο αριθμός των μη-μηδενικών στοιχείων είναι συνήθως πολύ μεγαλύτερος από τον αριθμό των γραμμών του πίνακα A η δημιουργία του x-stream οδηγεί σε αύξηση του memory traffic. Για 10 Compute Units η προσθήκη του x-stream οδηγεί σε αύξηση 40% του memory footprint κατά μέσο όρο στους πίνακες που χρησιμοποιήσαμε στις μετρήσεις μας. Επίσης, η δημιουργία του x-stream στον host κώδικα οδήγησε σε αύξηση του χρόνου προετοιμασίας των δεδομένων κατά 2,26s (περίπου 7%) κατά μέσο όρο.

Η αύξηση του χρόνου προετοιμασίας θεωρείται πολύ μικρή. Επιπλέον, λόγω της HBM μνήμης το Alveo U280 παρέχει δυνατότητα υψηλών ταχυτήτων μεταφοράς των δεδομένων. Συνεπώς, η αύξηση των δεδομένων που αποστέλλονται στο FPGA κατά 40% δεν θεωρείται σημαντική ιδιαιτέρως αν σκεφτούμε ότι η προσθήκη της δημιουργίας του x-stream στον host κώδικα οδήγησε σε αύξηση της επίδοσης έως και 50% όπως θα παρουσιάσουμε στην συνέχεια.

```
void read_input_streams_iat_coef(v_dtr *iat, hls::stream<v_dtr> &iatStream, v_dt *coef, hls::stream<v_dt> &coefStream, int nterm) {
    unsigned int vSize = ((nterm - 1) / VDATA_SIZE) + 1;
    mem_rd_ja_coef:
    for (int i = 0; i < vSize; i++) {
        #pragma HLS pipeline II=1
        iatStream << iat[i];
        coefStream << coef[i];
    }
}

void read_x(v_dt *x, hls::stream<v_dt> &xstream, int nterm) {
    unsigned int vSize = ((nterm - 1) / VDATA_SIZE) + 1;
    mem_rd_x:
    for (int i = 0; i < vSize; i++) {
        #pragma HLS pipeline II=1
        xStream << x[i];
    }
}
```

Εικόνα 3.17: *Διάβασμα Δεδομένων στον πυρήνα*

3.6.4 Μείωση των Row Bits

Δεδομένου ότι το padding που εφαρμόζουμε εξασφαλίζει ουσιαστικά ότι σε κάθε γραμμή θα υπάρχει αριθμός στοιχείων πολλαπλάσιος του 8 γνωρίζουμε ότι δεν είναι δυνατόν να υπάρξει αλλαγή γραμμής σε στοιχείο που δεν βρίσκεται σε θέση πολλαπλάσια του 8. Δοκιμάσαμε, λοιπόν να μεταφέρουμε μόνο αυτά τα row bits και έτσι να μειώσουμε το μέγεθος των δεδομένων που μεταφέρουμε καθώς και τις προσβάσεις που γίνονται στον πυρήνα για το διάβασμα άχρηστων δεδομένων.

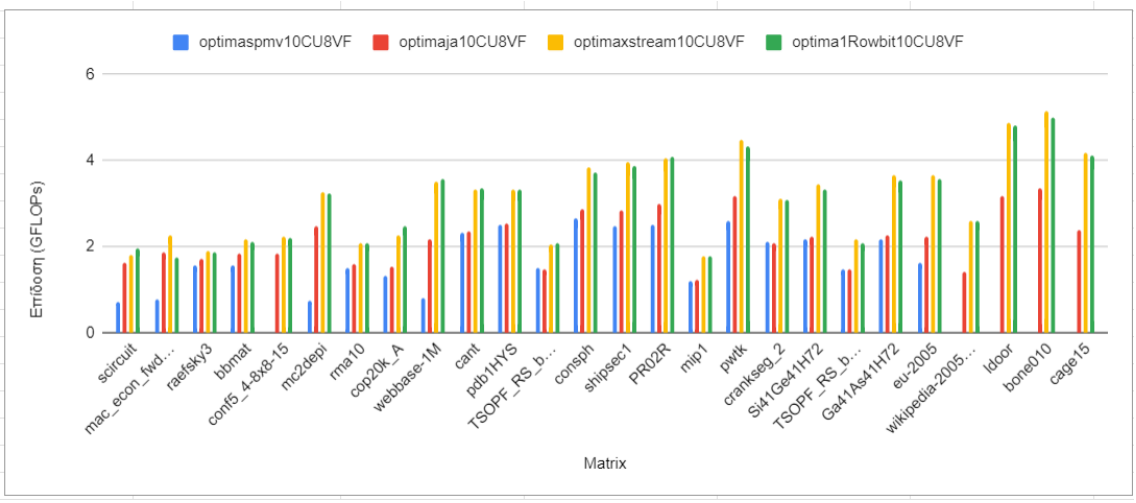
Οι αλλαγές που απαιτούνταν για αυτή την υλοποίηση ήταν μικρές. Διατηρήσαμε, όπως βλέπουμε στην Εικόνα 3.18, μόνο το τελευταίο rowbit ανά οχτάδα και στον πυρήνα στην συνάρτηση διαβάσματος τοποθετούμε στην ουρά ένα row bit σε κάθε διάβασμα αντί για την wide δομή που είχαμε προηγουμένως (`typedef struct v_datatyperow RowType data [VDATA_SIZE]; v_dtr;`). Τέλος, στον πυρήνα πάλι διαβάζουμε από την ουρά το rowbit μόνο μία φορα για κάθε οχτάδα.

```
int count=0;
iatbit = (RowType*) OOPS_malloc((size_t)((nterm/VDATA_SIZE)*sizeof(RowType)));
for(int i = 7; i < nterm; i+=VDATA_SIZE) {
    iatbit[count] = ja[i].range(31,31);
    count++;
}
```

Εικόνα 3.18: *Διατήρηση ένος Row Bit ανά οχτάδα στοιχείων*

3.6.5 Αξιολόγηση των x-stream και RowBit υλοποιήσεων

Η δημιουργία του x-stream στον host κώδικα μείωσε πάρα πολύ τις τυχαίες προσβάσεις την μνήμη κατά την εκτέλεση του πυρήνα. Αυτό είναι εμφανές και στην αύξηση της επίδοσης της υλοποιήσης η οποία φτάνει και το 50% σε κάποιους πίνακες. Επίσης, πλέον δεν υπάρχει ανάγκη αποστολής των column indices(ja). Έτσι, τα values(coef), που μεταδίδονταν μέσω του ίδιο channel με τα column indices, μπορούν να μεταδοθούν με υψηλότερη ταχύτητα τώρα που βρίσκονται στο ίδιο channel με τα row bits(iat) που απαιτούν πολύ λιγότερο εύρος ζώνης.



Εικόνα 3.19: *Επίδοση των x-stream και RowBit υλοποιήσεων*

Όσον αφορά την περαιτέρω μείωση των Row Bits παρατηρούμε στο διάγραμμα 4.3 μία μικρή βελτίωση της απόδοσης σε μερικούς από τους πίνακες αλλά όχι κάτι ιδιαίτερα αξιοσημείωτο. Αυτό είναι απολύτως λογικό καθώς ο όγκος των υπόλοιπων δεδομένων είναι πολύ μεγαλύτερος συγκριτικά με το μέγεθος των bits τα οποία αφαιρέσαμε.

Κεφάλαιο 4

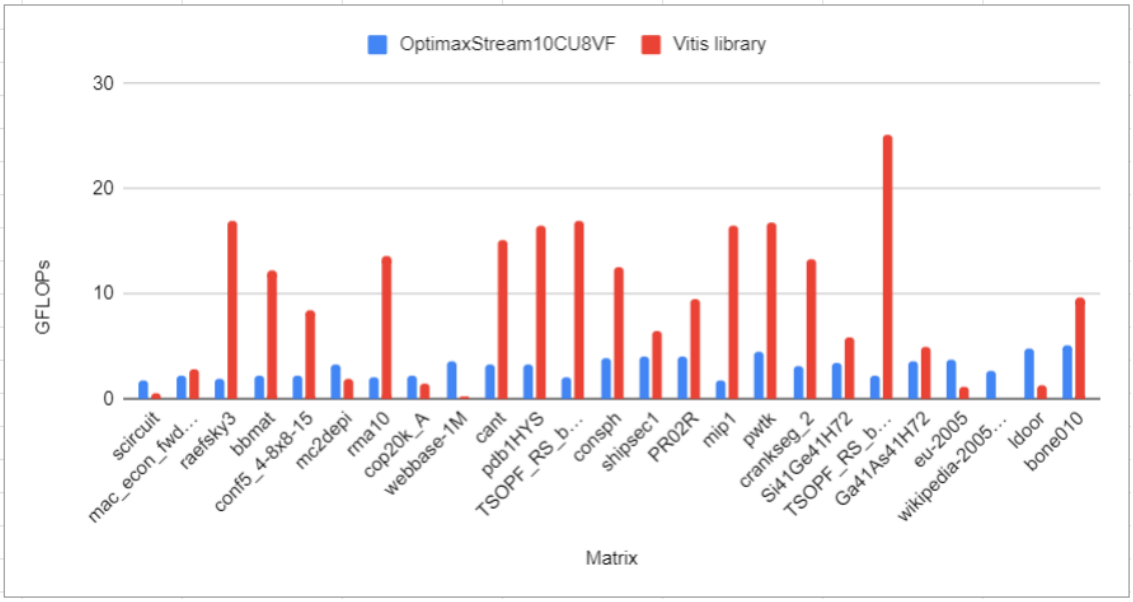
Αξιολόγηση

Για την τελική αξιολόγηση των υλοποιήσεων χρησιμοποιήσαμε την Optima x Stream υλοποίηση η οποία αποτελεί την καλύτερη υλοποίηση μας. Εξετάζουμε την χρονική επίδοση της εφαρμογής αλλά ταυτόχρονα και την ενεργειακή κατανάλωση. Παρουσιάζουμε την σύγκριση με μία CPU, μία GPU και την SpMV Vitis Library.

Για την σύγκριση της υλοποίησης μας με μία CPU χρησιμοποιήσαμε μια CSR-based υλοποίηση στον επεξεργαστή Intel Xeon Gold 5120 CPU Processor με 14 πυρήνες, συχνότητα 2.20GHz και 19.25MB L3 Cache μνήμης. Για την σύγκριση με GPU χρησιμοποιήσαμε μια CSR-based υλοποίηση στην NVIDIA Tesla V100 GPU η οποία διαθέτει 16GB RAM μνήμης.

4.1 Αξιολόγηση Επίδοσης Συγκριτικά με την Vitis SpMV Library

Αρχικά συγκρίνουμε την απόδοση της υλοποιήσης μας με την Vitis SpMV Library που παρουσιάσαμε στην εισαγωγή. Παρατηρούμε στο διάγραμμα 4.1 ότι στους περισσότερους πίνακες η βιβλιοθήκη της Vitis παρουσιάζει από 1 έως 8 φορές καλύτερη απόδοση. Αυτό είναι απολύτως λογικό καθώς η Vitis Library χρησιμοποιεί όλες τις SLR του FPGA ενώ η δική μας υλοποίηση περιορίζεται μόνο στην SLR 0 λόγω της HBM μνήμης. Επίσης, εξαιτίας της χρήσης μόνο HBM Channels η δική μας υλοποίηση μπορεί να μεταφερθεί και σε άλλα FPGAs, που είναι εξοπλισμένα με HBM Channels εν αντιθέση με την Vitis Library που έχει αναπτυχθεί αποκλειστικά για την Alveo U280. Τέλος, η Vitis Library εφαρμόζει πολύ μεγαλύτερο preprocessing στα δεδομένα το οποίο οδηγεί σε χρονοβόρα τρεξίματα όσο μεγαλώνουν οι πίνακες και καθιστά δυσκολη την συνδυαστική χρήση της βιβλιοθήκης σε άλλες εφαρμογες. Πιο αναλυτικά, όπως βλέπουνε στον Πίνακα 4.1 η δική μας υλοποίηση εφαρμόζει zero-padding της τάξης του 30% κατά μέσο όρο, ενώ της vitis library 410% κατά μέσο όρο οδηγώντας σε τρομακτική αύξηση των πράξεων που εκτελούνται. Επίσης, ο χρόνος που απαιτεί η δική μας υλοποίηση σε padding ισούται με 3.45s κατά μέσο όρο ενώ της vitis library με 468s.



Εικόνα 4.1: *Σύγκριση με Vitis SpMV*

Εμείς παρουσιάζουμε μία απλούστερη και πιο ευέλικτη υλοποίηση η οποία απαιτεί πολύ λιγότερο pre-processing και μπορεί να συνδυαστεί ευκολότερα και με υπολογιστικούς πυρήνες άλλων εφαρμογών. Επίσης, όπως βλέπουμε στις παραπάνω μετρήσεις υπάρχουν μερικοί πίνακες στους οποίους η υλοποίηση μας ξεπερνά ή αγγίζει την επίδοση της Vitis βιβλιοθήκης. Συμπεραίνουμε, λοιπόν, ότι μία τροποποίηση της υλοποίησης μας, που φροντίζει για την αξιοποίηση και των υπόλοιπών πόρων του FPGA μπορεί να παρουσίαζει καλύτερα αποτελέσματα και ταυτόχρονα να διατηρεί τα πλεονεκτήματα που περιγράψαμε.

Matrix	Stream-X		Vitis SPARSE Library	
	Time (s)	Padding Overhead	Time (s)	Padding Overhead
scircuit	0.383	60.6	45.876	1071.97
mac_econ_fwd500	0.495	56.42	47.738	538.34
raefsky3	0.288	0	15.865	33.32
bbmat	0.441	6.56	27.987	108.81
conf5_4-8x8-15	0.437	2.56	37.226	204.87
mc2depi	1.081	100.29	95.516	838.46
rma10	0.573	5.78	28.973	87.43
cop20k_A	0.448	14.14	90.275	689.01
webbase-1M	2.259	183.47	268.356	2083.39
cant	0.492	7.63	60.444	79.4
pdb1HYS	0.542	2.96	55.772	43.92
TSOPF_RS_b300_c3	1.092	5.18	47.334	63.55
consph	0.750	7.83	91.916	97.66
shipsec1	1.034	3.85	87.577	278.72
PR02R	2.129	10.70	133.94	136.44
mip1	1.538	2.35	117.624	50.37
pwtk	1.556	7.20	149.475	56.51
crankseg_2	1.757	23.52	194.319	79.34
Si41Ge41H72	2.100	10.58	366.871	338.73
TSOPF_RS_b2383	4.655	8.38	176.246	14.92
Ga41As41H72	2.639	19.92	532.046	441.29
eu-2005	5.406	23.45	494.924	384.61
wikipedia-20051105	7.646	47.35	1781.194	2029.09
ldoor	6.088	152.66	769.701	171.23
bone010	8.945	7.54	957.705	194.04
cage15	28.826	18.18	6282.698	1326.44
Average	**3.45**	**30.35 %**	**467.97**	**410.29 %**

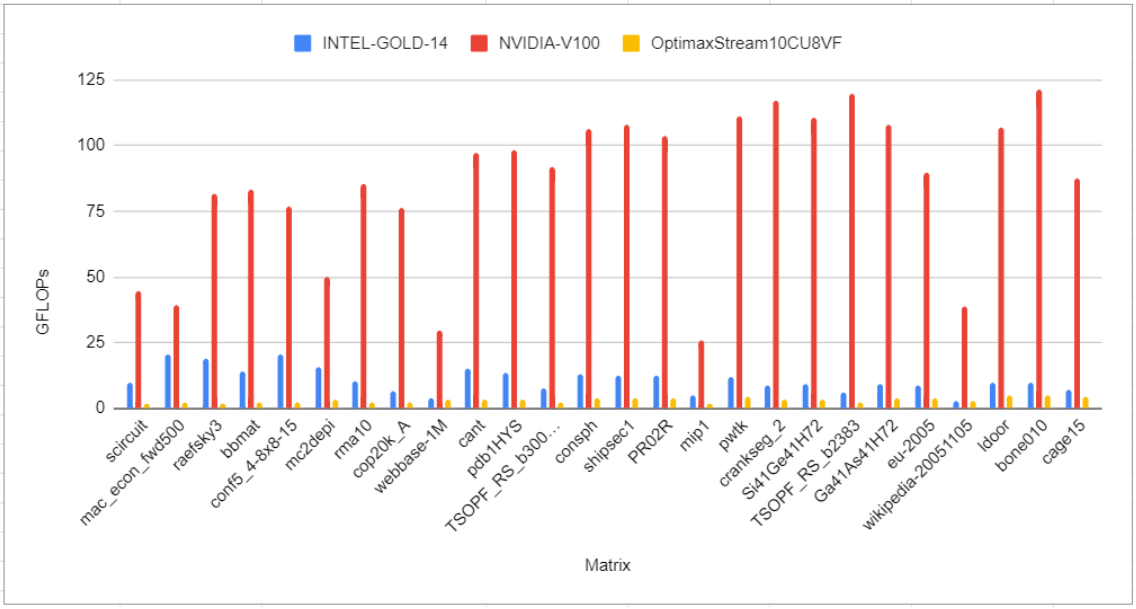
Πίνακας 4.1: *Σύγκριση preprocessing της υλοποίησής μας με την Vitis Sparse Library*

4.2 Αξιολόγηση Επίδοσης Συγκριτικά με CPU και GPU

4.2.1 Αξιολόγηση Υλοποίησης

Η υλοποίηση μας παρουσιάζει σημαντικά χειρότερη απόδοση με τις CPU και GPU που επιλέξαμε. Παρτηρούμε ότι η CPU υλοποίηση παρουσιάζει έως και 9 φορές καλύτερη απόδοση. Υπάρχουν βέβαια κάποιοι πίνακες,αυξημένου μεγέθους, που δεν χωράνε στην cache της CPU, στους οποίους η υλοποίηση μας παρουσιάζει περίπου την ίδια απόδοση. Όσον αφορά την GPU, παρουσιάζεται έως και 55 φορές καλύτερη σε σχέση με την υλοποίησή μας.

Αποδίδουμε την παραπάνω συμπερίφορα στην μείωμενη συχνότητα λειτουργίας του FPGA, στην μερική αξιοποίηση των πόρων του και ίσως στην ανάγκη για επιπλέον μηδενικά στοιχεία λόγω της αποστολής των δεδομένων σε πακέτα.

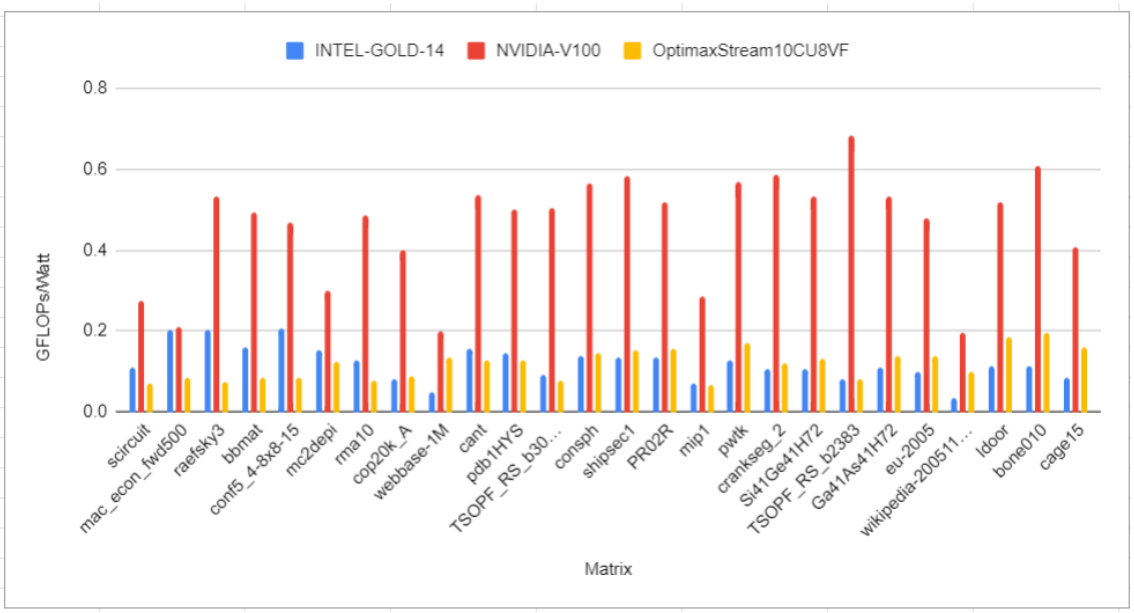


Εικόνα 4.2: *Τελική Αξιολόγηση*

4.2.2 Ενεργειακή Αξιολόγηση Υλοποίησης

Δεδομένης της χαμηλής ενεργειακής κατανάλωσης των FPGA λόγω της λειτουργίας τους σε πολύ χαμηλότερες συχνότητες η υλοποίηση μας παρουσίαζει σχετικά καλή ενεργειακή απόδοση. Η μέση κατανάλωση της GPU κυμαίνεται στα 179.57Watt, της CPU στα 86.36Watt ενώ η δική μας υλοποίηση καταναλώνει κατά μέσο όρο 26Watt. Βλέπουμε, λοιπόν ότι η υλοποίηση μας παρουσιάζει έως και 9 φορές καλύτερη κατανάλωση από την GPU και 4 από την CPU. Λαμβάνοντας, όμως, υπόψιν μας την σημαντικά καλύτερη απόδοση σε GFLOPs που παρουσιάζουν οι άλλες δύο υλοποιήσεις εισάγουμε μία ακόμη μετρική, τα GFLOPs/Watt. Τα GFLOPs/Watt αποτελούν μια μετρική ενδεικτική της ενέργειας που καταναλώνεται ανά πράξη και άρα επιτρέπει την πιο ουσιαστική σύγκριση των υλοποιήσεων.

Παρατηρούμε, λοιπόν, ότι η επίδοση της GPU συνεχίζει να ξεπερνά αρκετά την δική μας υλοποίηση. Αυτό, όμως, δεν ισχύει στην περίπτωση της CPU η οποία πλέον είναι αρκετά κοντά με την δική μας υλοποίηση, λίγο καλύτερη στους μικρότερους πίνακες και λίγο χειρότερη στους μεγαλύτερους. Συμπεραίνουμε, λοιπόν, ότι η υλοποίηση μας επιτυγχάνει αντίστοιχα αποτελέσματα με την CPU υλοποίηση αν λάβουμε υπόψιν μας την συσχέτιση πράξεων και απαιτούμενης ενέργειας.



Εικόνα 4.3: *Σύγκριση Ενεργειακής Επίδοσης σε FPGA.CPU,GPU*

Κεφάλαιο 5

Μελλοντικές Επεκτάσεις

5.1 Επέκταση του Υπολογιστικού Πυρήνα και στις υπόλοιπες SLR

Ένας βασικός λόγος για την καλύτερη επίδοση της Xilinx SpMV βιβλιοθήκης είναι η χρήση όλων των SLR και κατέπέκταση η αξιοποίηση αρκετά περισσότερων πόρων του FPGA. Συνεπώς, θα ήταν χρήσιμη η δημιουργία μίας εναλλακτικής αρχιτεκτονικής η οποία θα αξιοποιεί περισσότερους πόρους του FPGA, προσέχοντας όμως ταυτόχρονα να μην αυξήσουμε το SLR Crossing.

5.2 Μεταφορά του διανύσματος x μία φορά στο FPGA

Σε όλες τις υλοποιήσεις μας το διάνυσμα x μεταφέρεται στην μνήμη του FPGA ξεχωριστά για όλα τα Compute Units. Αυτό αυξάνει το χρόνο μεταφοράς των δεδομένων αλλά και την μνήμη που απατείται για την αποθήκευση. Σε συνδυασμό, λοιπόν, και με την επέκταση των υπολογισμών και στις υπόλοιπες SLRs θα μπορούσε η μεταφορά του x να γίνεται μόνο μία φορά σε ένα Compute Unit, από το οποίο θα μεταφέρεται στους υπόλοιπους υπολογιστικούς πυρήνες. Όλα τα παραπάνω βέβαια πρέπει να γίνουν με αρκετή προσοχή έτσι ώστε να αποφεύγονται οι ταυτόχρονες προσβάσεις στο x, και να επιτυγχάνεται σωστή διαχείριση των επικοινωνιών μεταξύ των Compute Units.

5.3 Reordering των στοιχείων του πίνακα A

Στην περίπτωση εφαρμογής των παραπάνω βελτιώσεων, επανερχόμστε στις υλοποιήσεις με τις τυχαίες προσβάσεις στο διάνυσμα x, προσβάσεις τις οποίες είχαμε αποφύγει δημιουργώντας το x stream στον host κώδικα. Η αναδιάταξη των στοιχείων του πίνακα A (reordering) και η διαφορετική κατανομή τους στα Compute Units είναι δυνατόν να μειώσει τις καθυστερήσεις των τυχαίων προσβάσεων και να βελτιώσει την απόδοση.

5.4 Πειραματισμός με τον τρόπο αποθήκευσης και τον υπολογιστικό πυρήνα

Όπως είδαμε η αλλαγή του τρόπου αποθήκευσης των στοιχείων του πίνακα A επηρέασε αρκετά την επίδοση της εφαρμογής. Χρήσιμη θα ήταν η μελέτη εναλλακτικών σχημάτων αποθήκευσης, όπως blocks of nonzeros, diagonals, banded structures, και η δημιουργία custom υπολογιστικών πυρήνων που υποστηρίζουν αυτές τις διαφορετικές δομές. Ο εντοπισμός τέτοιων δομών μπορεί να οδηγήσει σε μείωση του padding που απαιτείται και άρα σε εκτέλεση λιγότερων περιττών πράξεων και αύξηση της επίδοσης της εφαρμογής.

Μέρος III

Επίλογος

Κεφάλαιο 6

Επίλογος

Αντικείμενο αυτής της διπλωματικής αποτέλεσε η δημιουργία μιας υλοποίησης του SpMV σε ένα accelerator FPGA (Alveo U280) χρησιμοποιώντας ως βάση μία υλοποίηση του ίδιου προβλήματος σε ένα embedded FPGA (ZCU 102). Πρόκειται για δύο FPGA αρκετά διαφορετικής τεχνολογίας που απαιτούν διαφορετική διαχείριση της μνήμης, των δεδομένων και των πόρων τους. Η υλοποίηση βασίστηκε σε High Level Synthesis Tools που αυτοματοποιούν την διαδικασία προγραμματισμού του FPGA και διευκολύνουν την διαδικασία διαχείρισης των δεδομένων του SpMV.

Αρχικά, εφαρμόστηκαν τροποποιήσεις στον αρχικό κώδικα για την μεταφορά στο Alveo U280. Αυτές αφορούσαν τα εργαλεία για την μετάφραση του κώδικα και τον προγραμματισμό του FPGA καθώς και την μεταφορά των δεδομένων. Στην συνέχεια, έγινε προσαρμογή του κώδικα στους πόρους του καινούριου FPGA και αξιοποιήθηκε η High Bandwidth Memory που προσφέρει και είναι ιδανική για ένα memory bound πρόβλημα, όπως ο SpMV. Οι πολύπλοκες, όμως, δομές δεδομένων που ήταν απαραίτητες στην αρχική υλοποίηση λόγω της τεχνολογίας του ZCU102 καθιστούσαν αδύνατη την βελτίωση της επίδοσης της εφαρμογής. Τελικά, έγινε εισαγωγή του απλούστερου Optima SpMV Project και ο συνδυασμός του με την Rowbit CSR αναπαράσταση και την δημιουργία stream του x διανύσματος οδήγησαν στα βέλτιστα αποτελέσματα.

Όσον αφορά την σύγκριση της απόδοσης με CPU, GPU, αλλά και την Vitis SpMV Library που έχει σχεδιαστεί για ένα Xilinx FPGA, διαπιστώσαμε ότι η υλοποίηση μας παρουσιάζει γενικά χαμηλότερη επίδοση σε σύγκριση με τις υπόλοιπες υλοποιήσεις. Πρέπει, όμως, να λάβουμε υπόψιν ότι η υλοποιήση μας παρέχει ένα απλούστερο δεσιγν με σημαντικά λιγότερο pre-processing και παρ΄ όλα αυτά σε κάποιους πίνακες συναγωνίζεται και ξεπερνά την επίδοση της Vitis Library. Όσον αφορά την επίδοση της υλοποίησης μας συγκριτικά με τις CPU και GPU όντως υστερεί αρκετά σε επίδοση αλλά σε όρους ενεργειακής απόδοσης καταφέρνει να συναγωνιστεί την CPU. Με εφαρμογή των προτάσεων του Κεφαλαίου 5, καλύτερη αξιοποίηση των πόρων του FPGA και μείωση του zero-padding θα μπορούσαμε να παρατηρήσουμε περαιτέρω βελτίωση στην επίδοση της υλοποίησης μας.

Τέλος, συμπεραίνουμε ότι αν και τα FPGA συνεχίζουν να υστερούν σε επίδοση συγκριτικά με κλασσικά επεξεργαστικά συστήματα θα μπορούσαν με την εφαρμογή περαιτέρω βελτιώσεων και την προσαρμογή του κώδικα να αποτελέσουν εναλλακτική επιλογή ίδιως για εφαρμογές οι οποίες στοχεύουν σε χαμηλότερη ενεργειακή κατανάλωση.

Βιβλιογραφία

[1] *Introduction to FPGA Design with Vivado High Level Synthesis*. `https://nickbrown.online/?page_id=25`. 2023.

[2] *All About FPGAs*. `https://www.eetimes.com/all-about-fpgas/`. 2023.

[3] *Alveo U280 Data Center Accelerator Card, User Guide*. `https://www.mouser.com/pdfDocs/u280userguide.pdf`. 2023.

[4] *Introduction to FPGA Design with Vivado High Level Synthesis*. `https://docs.xilinx.com/v/u/en-US/ug998-vivado-intro-fpga-design-hls`. 2023.

[5] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas και Nectarios Koziris. *CSX: An Extended Compression Format for Spmv on Shared Memory Systems. SIGPLAN Not.*, 46(8):247–256, 2011.

[6] Mehmood R. Katib I. et al. ZAKI Usman, S. *A Smart Method and Tool for Automatic Performance Optimization of Parallel SpMV Computations on Distributed Memory Machine*. 2019.

[7] *CSCMV Overview*. `https://xilinx.github.io/Vitis_Libraries/sparse/2020.2/user_guide/L2_intro.html`. 2023.

[8] Π. Μπάκος. *Υλοποίηση Υπολογιστικού Πυρήνα Πολλαπλασιασμού Αραιού Πίνακα με Διάνυσμα σε FPGA*. Διπλωματική εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, 2019.

[9] Timothy A. Davis και Yifan Hu. *The University of Florida Sparse Matrix Collection*. 38(1), 2011.

[10] Mohammad Hosseinabady και Jose Luis Nunez-Yanez. *A Streaming Dataflow Engine for Sparse Matrix-Vector Multiplication Using High-Level Synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(6):1272–1285, 2020.

[11] Atefeh Sohrabizadeh. *Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication*. 2021.

[12] Zhongchun Zhou Zhiru Zhang Yixiao Du, Yuwei Hu. *High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '22)*, Virtual Event, CA, USA. ACM, New York, NY, USA, 2022.

[13] *Xilinx/Vitis_Accel_Examples*. https://github.com/Xilinx/Vitis_Accel_Examples. 2023.

[14] *Overview of Arbitrary Precision Integer Data Types*. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Integer-Data-Types. 2023.