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

Υλοποίηση Υπολογιστικού Πυρήνα Πολλαπλασιασμού Αραιού Πίνακα με Διάνυσμα σε FPGA

Διπλωματικη Εργασια

ΜΠΑΚΟΣ ΠΑΝΑΓΙΩΤΗΣ

Επιβλέπων: Γκούμας Γεώργιος
Επ. Καθηγητής Ε.Μ.Π.

Εργαστηριο Υπολογιστικων Συστηματων
Αθήνα, Σεπτέμβριος 2019

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

Υλοποίηση Υπολογιστικού Πυρήνα Πολλαπλασιασμού Αραιού Πίνακα με Διάνυσμα σε FPGA

Διπλωματικη Εργασια

ΜΠΑΚΟΣ ΠΑΝΑΓΙΩΤΗΣ

Επιβλέπων: Γκούμας Γεώργιος
Επ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25/9/2019.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Γκούμας Γεώργιος	Κοζύρης Νεκτάριος	Πνευματικάτος Διονύσιος
Επ. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Πολ. Κρήτης

Αθήνα, Σεπτέμβριος 2019

(Υπογραφή)

.....

Μπακος Παναγιωτης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
© 2019 – All rights reserved

Copyright © Μπάκος Παναγιώτης, 2019.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα (SpMV) αποτελεί βασικό κομμάτι πολλών εφαρμογών του High Performance Computing. Η αλγοριθμική του φύση όμως περιορίζει τη μέγιστη δυνατή επίδοση που μπορεί να επιτευχθεί από επεξεργαστές γενικού σκοπού (CPU).

Τα τελευταία χρόνια έχει αναπτυχθεί η τάση του συνδυασμού CPUs με προγραμματιζόμενους επιταχυντές ειδικού σκοπού, για υπολογιστικές εφαρμογές υψηλής έντασης. Πλέον, στόχος δεν είναι μόνο η βελτίωση της επίδοσης, αλλά και η ενεργειακή αποδοτικότητα των συστημάτων αυτών. Οι επαναδιαμορφούμενες αρχιτεκτονικές (FPGA) προβάλλουν ως μια ισχυρή εναλλακτική επιλογή για συστήματα υψηλής απόδοσης και χαμηλής κατανάλωσης. Το κύριο μειονέκτημα των FPGA είναι ότι απαιτείται περισσότερη προσπάθεια για τη σχεδίαση μιας εφαρμογής που προορίζεται για αυτά, συγκρινόμενο με CPU ή GPU. Παρ΄ όλα αυτά, με την ανάπτυξη εργαλείων σύνθεσης υψηλού επιπέδου (High Level Synthesis - HLS) και τον προγραμματισμό τους σε γλώσσες υψηλού επιπέδου (C++), τα FPGA έχουν γίνει πιο προσιτά προς την HPC κοινότητα.

Αντικείμενο της διπλωματικής εργασίας είναι η υλοποίηση και βελτιστοποίηση του αλγορίθμου SpMV σε FPGA, κάνοντας χρήση εργαλείων HLS. Επιπλέον, μελετάται η απόδοση της υλοποίησης μας, με όρους επίδοσης και ενεργειακής αποδοτικότητας, συγκρινόμενη με σύγχρονες CPU και GPU.

Λέξεις Κλειδιά

Πολλαπλασιασμός Αραιού Πίνακα με Διάνυσμα, SpMV, Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών, FPGA, Υπολογιστική Υψηλών Επιδόσεων, HPC

Abstract

Sparse Matrix-Vector Multiplication (SpMV) is a key component of many High Performance Computing applications. Its algorithmic nature, though, limits the maximum performance that can be achieved by general purpose processors (CPUs).

In recent years, a trend towards the combination of CPUs with programmable hardware accelerators has developed for high performance computing tasks. The goal of these systems is not only performance, but also energy efficiency. Reconfigurable architectures (FPGAs) have emerged as a powerful alternative for high performance-low power systems. The main disadvantage of FPGAs is that they require more programming and design effort, compared to a CPU or a GPU. However, with High Level Synthesis (HLS) tools and programming with higher level languages (C++), FPGAs have become friendlier to the HPC community.

This thesis focuses on implementing the SpMV algorithm for FPGAs, using HLS tools. We progressively evaluate and optimize our design, and compare its performance and energy-efficiency against modern CPUs and GPUs.

Keywords

Sparse Matrix-Vector Multiplication, SpMV, Field Programmable Gate Array, FPGA, High Performance Computing, HPC.

Ευχαριστίες

Με την ευκαιρία της ολοκλήρωσης της διπλωματικής μου εργασίας, θα ήθελα να ευχαριστήσω τον κ. Γεώργιο Γκούμα, για την ευκαιρία που μου έδωσε να εκπονήσω τη διπλωματική μου εργασία στο Εργαστήριο Υπολογιστικών Συστημάτων.

Ένα μεγάλο ευχαριστώ στις Νικέλα Παπαδοπούλου και Χλόη Αλβέρτη, για την καθοδήγηση και υποστήριξη που προσέφεραν σε όλη τη διάρκεια της διπλωματικής εργασίας.

Ευχαριστώ ακόμη όλους εκείνους τους φίλους, παλιούς και νέους, που έκαναν τα φοιτητικά χρόνια μια όμορφη και ξεχωριστή περίοδο.

Τέλος, θέλω να ευχαριστήσω την οικογένεια μου, που είναι πάντα δίπλα μου, δείχνοντας αγάπη, ενδιαφέρον και εμπιστοσύνη. Ό,τι καταφέρνω, το οφείλω σε αυτούς τους ανθρώπους.

Παναγιώτης Μπάκος

Περιεχόμενα

Κατάλογος Σχημάτων

Κεφάλαιο 1

Εισαγωγή

Την τελευταία δεκαετία η αύξηση της πολυπλοκότητας μιας ευρείας κατηγορίας υπολογιστικών εφαρμογών και η κατάρρευση της κλιμάκωσης της ενεργειακής επίδοσης των επεξεργαστών γενικού σκοπού προκάλεσε μια στροφή προς τα ετερογενή υπολογιστικά συστήματα.

Η ενορχηστρωμένη χρήση επεξεργαστών γενικού σκοπού (CPU) και προγραμματιζόμενων επιταχυντών ειδικού σκοπού (HW Accelerators) για υπολογιστικές εφαρμογές υψηλής έντασης εφαρμόζεται ήδη σε υπολογιστικά συστήματα κέντρων δεδομένων (Data centers), αλλά και σε συστήματα υψηλών επιδόσεων (High Performance Computing). Στόχος είναι η αύξηση της απόδοσης με όρους επίδοσης άλλά και κατανάλωσης ισχύος.

Οι επαναδιαμορφούμενες αρχιτεκτονικές, γνωστές και ως FPGA (Field Programmable Gate Arrays), είναι ένα ισχυρό υπολογιστικό υλικό, το οποίο επιτρέπει στους σχεδιαστές τη δημιουργία συστημάτων εξειδικευμένων εφαρμογών. Παρά τη μεγάλη ευελιξία και τη συχνή υπεροχή τους σε ενεργειακή κατανάλωση/επίδοση, συγκριτικά με τους επεξεργαστές γενικού σκοπού και άλλους επιταχυντές (GPU), ο προγραμματισμός τους με μοντέλα επιπέδου μεταφοράς καταχωρητή (RTL) δρούσε ανασταλτικά στην ευρεία χρήση τους.

Τα τελευταία χρόνια όμως αναπτύχθηκαν προηγμένα εργαλεία σύνθεσης (High Level Synthesis - HLS), με στόχο την επιτάχυνση και αυτοματοποίηση της διαδικασίας σχεδιασμού και προγραμματισμού υπολογιστικών εφαρμογών. Η εξέλιξη αυτή καθιστά τα FPGA μια ελκυστική τεχνολογία επιταχυντών ευρείας χρήσης, λαμβάνοντας επιπλέον υπόψιν την ενεργειακή αποδοτικότητά τους, συγκρινόμενα με άλλες αρχιτεκτονικές.

1.1 Αντικείμενο της διπλωματικής

Στα πλαίσια της διπλωματικής αυτής, θα υλοποιήσουμε τον αλγόριθμο πολλαπλασιασμού αραιού πίνακα με διάνυσμα (Sparse Matrix-Vector Multiplication - SpMV) σε FPGA, κάνοντας χρήση των εργαλείων HLS. Ο αλγόριθμος SpMV αποτελεί το βασικό υπολογιστικό πυρήνα σε πληθώρα εφαρμογών του HPC. Η αλγοριθμική φύση του όμως είναι τέτοια, που περιορίζει την απόδοση των ισχυρών επεξεργαστικών συστημάτων, μιας και για κάθε πράξη κινητής υποδιαστολής απαιτούνται πολλές προσβάσεις στην ιεραρχία μνήμης (χαμηλή αναλογία flop : byte).

Στην υλοποίησή μας προσπαθούμε να επιτύχουμε την κατά το δυνατόν αποδοτική μεταφορά του αραιού πίνακα στο FPGA, καθώς το διαθέσιμο εύρος ζώνης μνήμης είναι περιορισμένο. Στη συνέχεια, εφαρμόζουμε τεχνικές παραλληλοποίησης των υπολογισμών, με σκοπό τη βελτίωση της επίδοσης. Τέλος, συγκρίνουμε την υλοποίησή μας με μία CPU και μία GPU, και δείχνουμε ότι το FPGA μπορεί να αποτελέσει μια ενεργειακά αποδοτική επιλογή για το SpMV.

1.2 Οργάνωση του τόμου

Το υπόλοιπο της διπλωματικής οργανώνεται ως εξής :

Το **Κεφάλαιο 2** ξεκινά με την παρουσίαση της αρχιτεκτονικής και του τρόπου λειτουργίας ενός FPGA. Στη συνέχεια γίνεται η εισαγωγή στην έννοια του αλγορίθμου SpMV και των διαφόρων σχημάτων αποθήκευσης αραιών πινάκων. Τέλος, παρουσιάζεται η μέχρι στιγμής έρευνα πάνω στην υλοποίηση του αλγορίθμου SpMV σε FPGA.

Στο **Κεφάλαιο 3** παρουσιάζεται η πορεία της υλοποίησης που αναπτύχθηκε στα πλαίσια της διπλωματικής αυτής. Σε κάθε στάδιο αυτής παρουσιάζονται επιμέρους αποτελέσματα. Το κεφάλαιο ολοκληρώνεται με τη σύγκριση της υλοποίησης με μία σύγχρονη CPU και GPU, με όρους επίδοσης και ενεργειακής αποδοτικότητας.

Στο **Κεφάλαιο 4** προτείνονται μελλοντικές επεκτάσεις και πιθανές βελτιώσεις της παρούσας υλοποίησης, ενω στο **Κεφάλαιο 5**, συγκεντρώνονται τα συμπεράσματα της εργασίας.

Κεφάλαιο 2

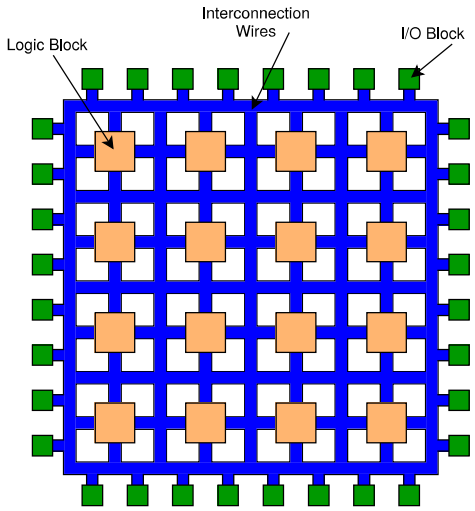
Θεωρητικό υπόβαθρο

2.1 Field-Programmable Gate Arrays (FPGAs)

2.1.1 Εισαγωγή στα FPGA

Το FPGA ή Field-Programmable Gate Array ή Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών είναι ένα ολοκληρωμένο κύκλωμα, το οποίο ο σχεδιαστής μπορεί να επαναπρογραμματίσει, ανάλογα με την εφαρμογή για την οποία προορίζεται. Η λειτουργία ενός τέτοιου κυκλώματος βασίζεται σε μπλοκ προγραμματιζόμενης λογικής CLBs (Configurable Logic Blocks), τα οποία συνδέονται μεταξύ τους μέσω προγραμματιζόμενων διασυνδέσεων (Programmable Interconnects). Η επικοινωνία με τον έξω κόσμο γίνεται μέσω ρυθμιζόμενων μπλοκ εισόδου/εξόδου (I/O blocks).

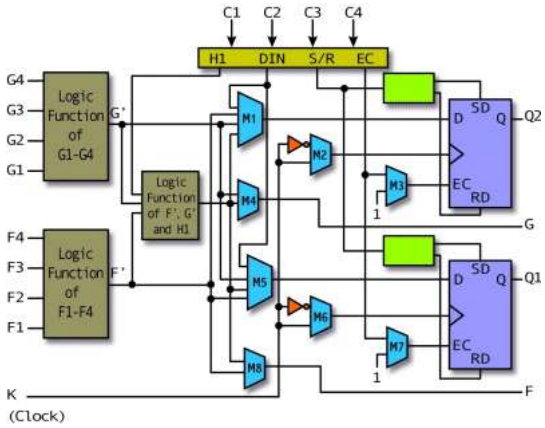
Σε ένα FPGA δίνεται η δυνατότητα στον προγραμματιστή-σχεδιαστή να κάνει αλλαγές στην εφαρμογή που καλείται να εκτελέσει και να βελτιώσει την απόδοσή της, σε αντίθεση με ένα κύκλωμα ASIC (Application Specific Integrated Circuit), που κατασκευάζεται για συγκεκριμένη εφαρμογή και δεν επιδέχεται αλλαγής.



Σχήμα 2.1: Βασική δομή FPGA [1]

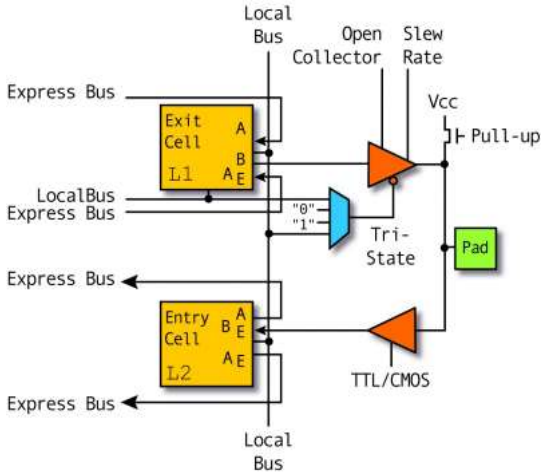
Αναλυτικά, ένα FPGA αποτελείται από τα εξής στοιχεία :

- **Configurable Logic Block** : Αναλαμβάνει την εκτέλεση βασικών συναρτήσεων της εκάστοτε εφαρμογής. Ένα CLB αποτελείται από Look-Up Tables, Flip-Flops, και πολυπλέκτες, όπως φαίνεται στο Σχήμα 2.2. Το Look-Up Table είναι ο πίνακας αληθείας της εκάστοτε συνάρτησης που υλοποιείται στο συγκεκριμένο μπλοκ, τα Flip-Flops αναλαμβάνουν την αποθήκευση συγκεκριμένων μεταβλητών, και οι πολυπλέκτες αναλαμβάνουν τη δρομολόγηση της λογικής εντός του μπλοκ και προς τους εξωτερικούς πόρους.



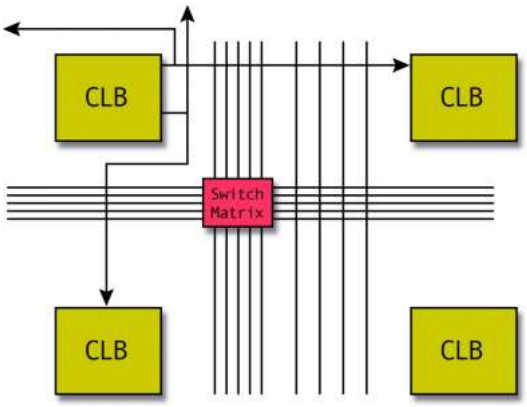
Σχήμα 2.2: Δομή CLB

- **I/O Block** : Μέσω αυτού εκτελείται η επικοινωνία του chip με εξωτερικές συσκευές. Εντοπίζεται στην περιφέρεια του chip.



Σχήμα 2.3: Δομή I/O Block

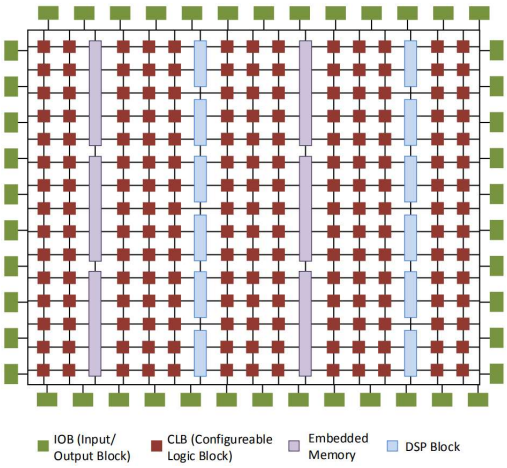
- **Interconnection Network** : Μέσω αυτού γίνεται η σύνδεση των CLBs μεταξύ τους, ώστε να εκτελεστούν πιο πολύπλοκες λογικές συναρτήσεις, μέσω προγραμματιζόμενων καλωδιώσεων. Επιπλέον, αναλαμβάνει τη σύνδεση των CLBs με τα μπλοκ I/O.



Σχήμα 2.4: Δίκτυο διασύνδεσης

Οι σύγχρονες αρχιτεκτονικές FPGA συνδυάζουν τα ανωτέρω δομικά στοιχεία με επιπλέον μπλοκ εξειδικευμένα για αποθήκευση, μεταφορά και υπολογισμούς δεδομένων, που βελτιώνουν την υπολογιστική πυκνότητα και απόδοση της συσκευής. Τα συγκεκριμένα μπλοκ δεν είναι επαναπρογραμματιζόμενα από το σχεδιαστή, οδηγώντας σε μείωση της απαιτούμενης περιοχής του chip για αυτά και σε επιτάχυνση της λειτουργίας τους. Είναι τα εξής :

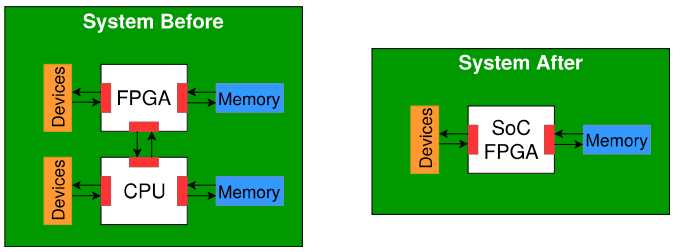
- **BlockRAM** : Πρόκειται για μια ενσωματωμένη μνήμη, προσβάσιμη σε κάθε κύκλο λειτουργίας του chip. Η διασύνδεσή της μέσω δύο θυρών dual-port, επιτρέπει την παράλληλη πρόσβαση σε δύο διαφορετικές περιοχές της μνήμης στον ίδιο κύκλο ρολογιού.
- **Θύρες I/O υψηλής ταχύτητας** : Πρόκειται για εξειδικευμένες διεπαφές, με τις οποίες επιτυγχάνεται ταχύτερη ροή δεδομένων από τον εξωτερικό κόσμο στο chip.
- **Digital Signal Processor (DSP)** : Είναι το μπλοκ με την υψηλότερη πολυπλοκότητα, το οποίο συνθέτει μια Αριθμητική και Λογική Μονάδα (ALU), συνδυάζοντας αθροιστές, αφαιρέτες και πολλαπλασιαστές. Μπορεί να αναλάβει υψηλό υπολογιστικό φόρτο.



Σχήμα 2.5: Δομή σύγχρονου FPGA [2]

2.1.2 Ετερογενείς αρχιτεκτονικές System-on-Chip

Στο παρελθόν, η ύπαρξη του FPGA και του επεξεργαστή που το τροφοδοτεί με δεδομένα σε διαφορετικά chip καθιστούσε κρίσιμο σημείο της εφαρμογής την επικοινωνία μεταξύ των δύο. Για το λόγο αυτό έχει αναπτυχθεί η τάση η επαναδιαμορφούμενη λογική ενός FPGA να συνδυάζεται με ενσωματωμένους επεξεργαστές και περιφερειακές συσκευές, χτισμένα σε μία κοινή πλατφόρμα, που ονομάζεται System-on-Chip (SoC). Με αυτόν τον τρόπο επιτυγχάνεται σημαντικά ταχύτερη επικοινωνία μεταξύ τους, χαμηλότερη κατανάλωση ενέργειας, μικρότερες απαιτήσεις σε μέγεθος πλατφόρμας (μικρότερο κόστος κατασκευής). Επίσης, δίνεται η δυνατότητα για παράλληλη λειτουργία των δύο υποσυστημάτων που το απαρτίζουν, επιτυγχάνοντας υψηλότερη απόδοση.



Σχήμα 2.6: Σύγκριση μεταξύ ανεξάρτητων CPU/FPGA και SoC FPGA [3]

Ένα SoC FPGA χωρίζεται σε δύο διακριτά υποσυστήματα, το επεξεργαστικό σύστημα (Processing System - PS) και την προγραμματίσιμη λογική (Programmable Logic - PL).

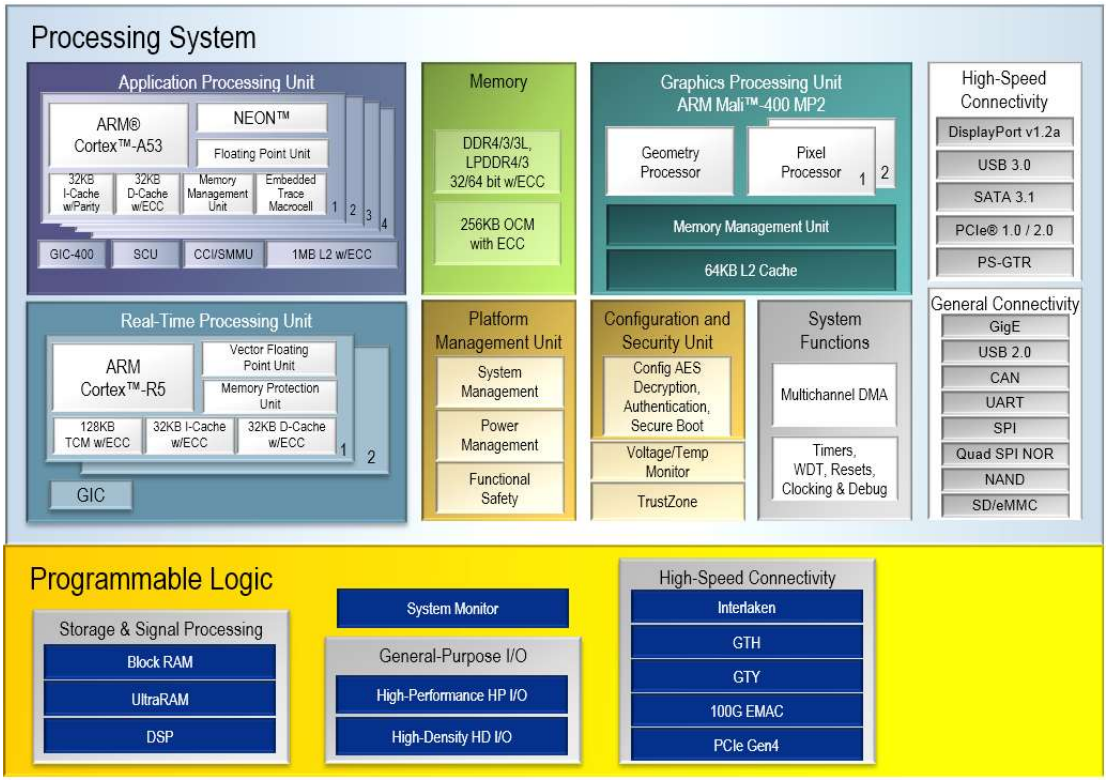
2.1.3 Zynq UltraScale+ MPSoC

Στα πλαίσια της συγκεκριμένης διπλωματικής το SoC FPGA που χρησιμοποιήθηκε είναι το Zynq UltraScale+ MPSoC (Multi-Processor System-on-Chip) ZCU102 της Xilinx.

Στο PS βρίσκουμε έναν τετραπύρηνο ARM επεξεργαστή Cortex-A53, συχνότητας λειτουργίας 1.5 GHz, ο οποίος υποστηρίζει υψηλού επιπέδου λειτουργικά συστήματα (Linux) διευκολύνοντας την ανάπτυξη εφαρμογών που θα αναθέτουν συγκεκριμένο κομμάτι της εκτέλεσης τους στο υποσύστημα PL. Διαθέτει ακόμη 2 επεξεργαστικές μονάδες πραγματικού χρόνου Cortex-R5 και επεξεργαστή γραφικών Mali-400. Η διαθέσιμη μνήμη του PS είναι 4 GB DDR4 64-bit SODIMM. [4]

Στο PL το FPGA που χρησιμοποιείται είναι το ZU9EG της Xilinx και οι διαθέσιμοι πόροι είναι οι εξής :

- Look-Up Tables : 274080
- Flip-Flops : 548160
- BlockRAM (# of 18Kb blocks) : 32.1 Mb (1824)
- DSPs : 2520



Σχήμα 2.7: Zynq UltraScale+ MPSoC ZCU102 Block Diagram

2.1.4 Πρωτόκολλο επικοινωνίας AXI

Η επικοινωνία μεταξύ των δύο υποσυστημάτων εκτελείται μέσω AXI (Advanced eXtensible Interface) διεπαφών, οι οποίες εξασφαλίζουν υψηλό εύρος και χαμηλή καθυστέρηση στις μεταφορές δεδομένων. Το πρωτόκολλο AXI είναι μέρος της ARM AMBA (Advanced Microcontrollers Bus Architecture), που είναι μια οικογένεια μικροελεγκτών διαύλων σε System-on-Chip αρχιτεκτονικές.

Στο συγκεκριμένο πρωτόκολλο η μεταφορά δεδομένων γίνεται με «χειραψία» μεταξύ των δύο πλευρών (master-slave), ενώ ακόμη δίνεται η δυνατότητα για μεταφορά πολλών δεδομένων σε μορφή «ριπής» (burst-type), χρησιμοποιώντας μόνο μία διεύθυνση μνήμης. Τα τελευταία χρόνια χρησιμοποιείται η έκδοση του πρωτοκόλλου που ονομάζεται AXI4, η οποία περιλαμβάνει 3 τύπους διεπαφών :

- AXI4 : Χρησιμοποιείται για memory-mapped συνδέσεις και επιτρέπει αποστολή έως και 256 λέξεων δεδομένων με μία διεύθυνση μνήμης (burst)
- AXI4-Lite : Ομοίως, χρησιμοποιείται για memory-mapped συνδέσεις, αλλά για μεταφορά μόνο ενός στοιχείου
- AXI4-Stream : Χρησιμοποιείται για ροή δεδομένων (streaming) σε υψηλή ταχύτητα μεταξύ των δύο υποσυστημάτων. Ενδείκνυται για εφαρμογές που υπάρχει συνεχής ροή δεδομένων (όπως και η δική μας)

2.1.5 Προγραμματισμός σε FPGA

Μία από τις κύριες διαφορές ανάμεσα σε έναν επεξεργαστή και σε ένα FPGA είναι ότι η αρχιτεκτονική του επεξεργαστή είναι σταθερή. Σε έναν επεξεργαστή, ο μεταγλωττιστής αναλαμβάνει να προσαρμόσει βέλτιστα την εφαρμογή λογισμικού στις διαθέσιμες υπολογιστικές δομές. Αντιθέτως, σε ένα FPGA, ο μεταγλωττιστής προσαρμόζει την υπολογιστική αρχιτεκτονική στις ανάγκες της εφαρμογής λογισμικού, χρησιμοποιώντας τους διαθέσιμους πόρους του FPGA, όπως αυτοί αναλύθηκαν προηγουμένως. Σε αυτό το κεφάλαιο θα αναλυθούν έννοιες σχεδίασης υλικού.

Συχνότητα ρολογιού

Μία από τις πρώτες έννοιες που απασχολεί έναν προγραμματιστή κατά την υλοποίηση μιας εφαρμογής σε μία πλατφόρμα εκτέλεσης είναι η συχνότητα ρολογιού. Υψηλότερη συχνότητα ρολογιού μεταφράζεται σε ταχύτερη εκτέλεση της εφαρμογής. Εδώ, το πλεονέκτημα φαίνεται αρχικά να το έχουν οι κλασικοί επεξεργαστές, στους οποίους η συχνότητα του ρολογιού συνήθως κυμαίνεται στα 2 GHz, ενώ σε FPGA δεν ξεπερνά τα 500 MHz. Όμως, ο τρόπος εκτέλεσης μίας εφαρμογής διαφέρει σημαντικά ανάμεσα σε αυτές τις δύο πλατφόρμες.

Σε έναν κλασικό επεξεργαστή, ο μεταγλωττιστής, γνωρίζοντας την αρχιτεκτονική του, μετατρέπει την εφαρμογή λογισμικού σε ένα σύνολο εντολών. Το σύνολο εντολών εκτελείται πάντα με συγκεκριμένη σειρά, όπως φαίνεται παρακάτω.

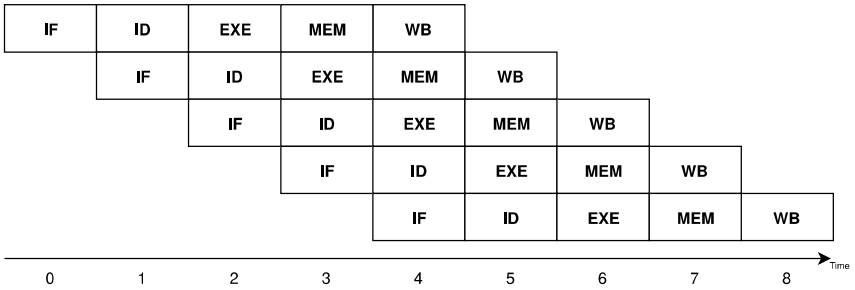
Σχήμα 2.8: Στάδια εκτέλεσης εντολής σε επεξεργαστή [5]

Ανεξαρτήτως του τύπου του επεξεργαστή, η εκτέλεση κάθε εντολής μιας εφαρμογής λογισμικού περνά από τα εξής στάδια :

1. Instruction Fetch (IF) : Φόρτωση εντολής από τη μνήμη του προγράμματος
2. Instruction Decode (ID) : Αποκωδικοποίηση της εντολής, καθορισμός της διεργασίας και των τελεστών της
3. Execution (EXE) : Εκτέλεση της εντολής στο διαθέσιμο υλικό
4. Memory Operations (MEM) : Φόρτωση της επόμενης εντολής με διεργασίες μνήμης
5. Write Back (WB) : Αποθήκευση των αποτελεσμάτων της εντολής σε τοπικούς καταχωρητές ή στη μνήμη

Σε σύγχρονους επεξεργαστές, χρησιμοποιούνται πολλαπλές μονάδες εκτέλεσης των παραπάνω σταδίων, εκτελώντας τα με κάποιου βαθμού επικάλυψη. Στις περισσότερες περιπτώσεις,

διαδοχικές εντολές της εφαρμογής εξαρτώνται μεταξύ τους, και η επικάλυψη περιορίζεται. Επιπλέον, οι διαθέσιμες υπολογιστικές μονάδες του υλικού, που χρησιμοποιούνται στο στάδιο EXE, είναι περιορισμένες, επιτρέποντας μόνο μία εντολή να εκτελείται κάθε στιγμή στο στάδιο αυτό. Αυτό έχει σαν αποτέλεσμα μόνο μία εντολή της εφαρμογής να ολοκληρώνεται σε κάθε κύκλο ρολογιού, όπως φαίνεται και από το παρακάτω σχήμα.

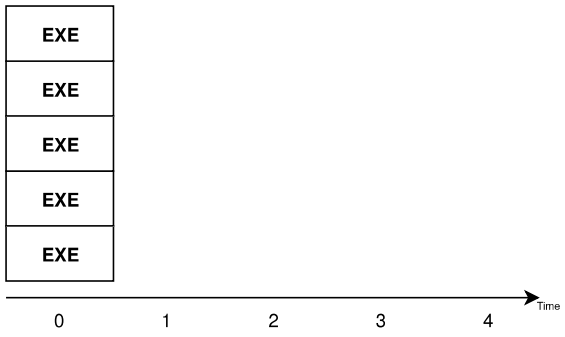


Σχήμα 2.9: Πολλαπλές μονάδες εκτέλεσης σταδίων σε επεξεργαστή

Σε ένα FPGA η εκτέλεση μιας εφαρμογής γίνεται σε ένα κύκλωμα, διαμορφωμένο για αυτήν την εφαρμογή. Έπομένως, αλλαγή στην εφαρμογή αλλάζει και το κύκλωμα που υλοποιείται στο FPGA.

Σχήμα 2.10: Στάδια εκτέλεσης εντολής σε FPGA

Ο μεταγλωττιστής σε ένα FPGA δεν χρειάζεται να λαμβάνει υπόψιν του τα στάδια όπως αναλύθηκαν προηγουμένως, αλλά προσπαθεί να επιτύχει τη μέγιστη παραλληλία εντολών. Για παράδειγμα, διαδοχικές εντολές της εφαρμογής, εκτελούνται σε διαφορετικό κύκλωμα πάνω στο FPGA, και επιτυγχάνεται η μέγιστη δυνατή παραλληλία, αρκεί φυσικά να μην είναι μεταξύ τους αλληλοεξαρτώμενες.



Σχήμα 2.11: Πολλαπλές μονάδες εκτέλεσης σταδίων σε FPGA

Καταλαβαίνουμε από τα παραπάνω ότι η χαμηλότερη συχνότητα ρολογιού σε ένα FPGA δεν συνεπάγεται άμεσα χαμηλότερη επίδοση συγκρινόμενη με κλασικό επεξεργαστή, καθώς ο

τρόπος που εκτελούνται οι εντολές διαφέρει αρκετά. Επίσης, ένα άλλο πλεονέκτημα αυτής της χαμηλότερης συχνότητας ρολογιού, είναι ότι επιτυγχάνεται αρκετά χαμηλότερη κατανάλωση ενέργειας, μιας και η σχέση μεταξύ κατανάλωσης και συχνότητας είναι γραμμική.

$$P = \frac{1}{2}cFV^2 \tag{2.1}$$

Χρονοδρομολόγηση

Η χρονοδρομολόγηση είναι η διαδικασία προσδιορισμού των εξαρτήσεων δεδομένων και ελέγχου μεταξύ διαφορετικών εντολών, προκειμένου να καθοριστεί πότε θα εκτελεστεί καθεμία. Ο μεταγλωττιστής για το FPGA αναλύει τις εξαρτήσεις μεταξύ γειτονικών εντολών, καθώς και μεταξύ διαφορετικών χρονικών στιγμών. Αυτό επιτρέπει στο μεταγλωττιστή να ομαδοποιήσει εντολές που θα εκτελούνται στον ίδιο κύκλο ρολογιού και να ρυθμίζει το υλικό ώστε να επιτρέπεται η επικάλυψη τους.

Πλέον, κατά την εκτέλεση ενός σετ εντολών (για παράδειγμα εντολές εντός ενός επαναληπτικού βρόχου), δεν απαιτείται η ολοκλήρωση του σετ αυτού, προκειμένου να ξεκινήσει η εκτέλεση του επόμενου. Η διαδικασία αυτή ονομάζεται pipelining και θα αναλυθεί στη συνέχεια.

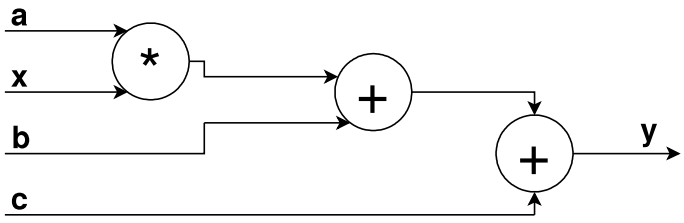
Καθυστέρηση και Pipelining

Ως καθυστέρηση ορίζουμε τον αριθμό των κύκλων που απαιτούνται για την ολοκλήρωση μιας εντολής ή ενός σετ εντολών, προκειμένου να παραχθεί κάποιο αποτέλεσμα. Για παράδειγμα, στο σχήμα 2.8, η καθυστέρηση της εντολής είναι 5 κύκλοι ρολογιού.

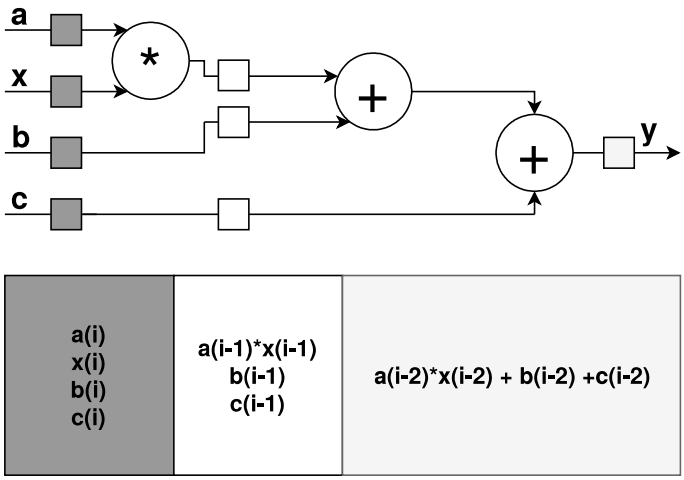
Η καθυστέρηση μιας εφαρμογής είναι από τις κύριες μετρικές απόδοσης, τόσο σε κλασικούς επεξεργαστές, όσο και σε FPGA. Ο περιορισμός της καθυστέρησης γίνεται με εφαρμογή του pipelining (σωλήνωσης). Σε έναν επεξεργαστή, αυτό σημαίνει ότι μπορεί να αρχίσει η εκτέλεση της επόμενης εντολής προτού τελειώσει η τρέχουσα, όπως βλέπουμε στο σχήμα 2.9, σε βέλτιστη μορφή. Η εξοικονόμηση σε κύκλους ρολογιού είναι εμφανής, καθώς απο 25 κύκλους, που θα είχαμε εάν εκτελούνταν οι 5 εντολές διαδοχικά, πλέον έχουμε 9.

Σε ένα FPGA το pipelining εφαρμόζεται με λίγο διαφορετικό τρόπο. Για κάθε εντολή ή σετ εντολών δημιουργείται ένα κύκλωμα, που αναλαμβάνει την εκτέλεση των υπολογισμών. Για παράδειγμα για την εκτέλεση της εντολής $\mathbf{y = a*x + b + c}$ προκύπτει το κυκλωμα υπολογισμού που βλέπουμε στην παρακάτω εικόνα. Σε έναν κλασσικό επεξεργαστή, όλα τα δεδομένα (a, x, b, c) πρέπει να είναι γνωστά πριν την έναρξη των υπολογισμών, και μόνο ένα αποτέλεσμα (y) μπορεί να υπολογιστεί κάθε φορά.

Αντιθέτως, στην FPGA-Pipelined έκδοση, τα κουτιά αναπαριστούν ενδιάμεσους καταχωρητές που αποθηκεύουν δεδομένα διαδοχικών χρονικών στιγμών. Αυτό σημαίνει ότι η περιοχή με τον πολλαπλασιαστή και η περιοχή με τους δύο αθροιστές μπορούν να λειτουργούν παράλληλα, υπολογίζοντας παράλληλα τα στοιχεία y(i-2) και y(i-1). Στο κάτω μέρος της εικόνας διακρίνουμε την κατάσταση των καταχωρητών μία δεδομένη χρονική στιγμή.



Σχήμα 2.12: Κύκλωμα υπολογισμού της εντολής $y = a*x+b+c$ [5]



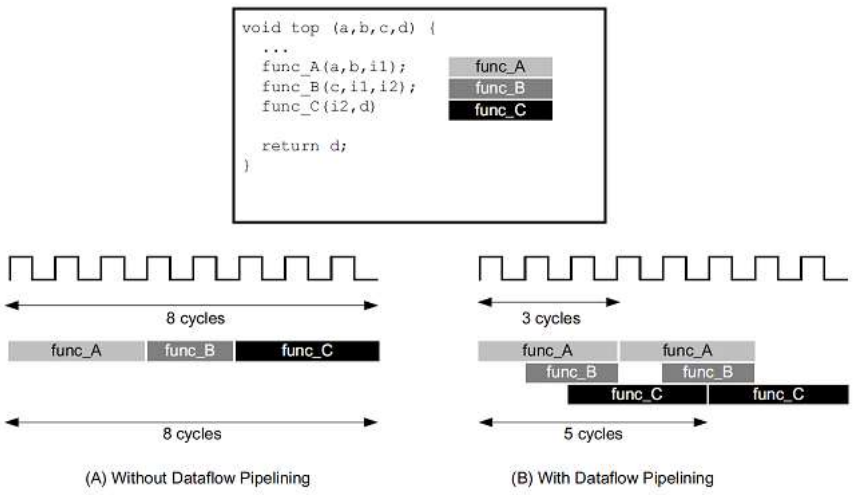
Σχήμα 2.13: Εφαρμογή pipelining σε εντολή υπολογισμού [5]

Έτσι ορίζεται το Pipelining σε επίπεδο εντολών. Σε πιο coarse-grain επίπεδο, ορίζουμε την έννοια του **Dataflow**, σπάζοντας το πρόγραμμα σε επιμέρους υπολογιστικά τμήματα (συναρτήσεις ή βρόχους επαναλήψεων), και εκτελώντας τα κατά το δυνατόν παράλληλα. Ο μεταγλωττιστής αναλαμβάνει να εξάγει τις εξαρτήσεις μεταξύ των υπολογιστικών τμημάτων και να δρομολογήσει την εκτέλεσή τους το συντομότερο δυνατόν. Έπιτυγχάνεται επομένως περαιτέρω βελτίωση της απόδοσης.

Αρχιτεκτονική και Διάταξη Μνήμης

Η αρχιτεκτονική της μνήμης και οι προσβάσεις σε αυτήν αποτελούν από τους κρίσιμους παράγοντες απόδοσης, σε οποιαδήποτε υπολογιστική πλατφόρμα. Σε έναν κλασικό επεξεργαστή, ο προγραμματιστής αναλαμβάνει να προσαρμόσει την εφαρμογή του στην αρχιτεκτονική μνήμης, Αυτή συνήθως διακρίνεται σε 3 κατηγορίες, τη γρήγορη, που είναι η μνήμη cache, την αργή, που είναι οι συσκευές μαζικής αποθήκευσης, και την ενδιάμεση, που είναι οι DDR μνήμες, ανάλογα με τον αριθμό κύκλων ρολογιού που χρειάζονται για να μεταφέρουν δεδομένα στον επεξεργαστή. Για τη βελτίωση της απόδοσης μιας εφαρμογής στη συγκεκριμένη πλατφόρμα, ο προγραμματιστής πρέπει να αναμορφώσει τον αλγόριθμο της εφαρμογής, ώστε να επαναχρησιμοποιεί όσο πιο αποδοτικά δεδομένα που έχουν αποθηκευτεί στη γρήγορη μνήμη cache.

Σε ένα FPGA, η πρόσβαση στην αργή ή ενδιάμεση μνήμη δεν άλλάζει. Όσον αφορά τη



Σχήμα 2.14: Εφαρμογή dataflow σε συνάρτηση [5]

γρήγορη μνήμη, δεν υπάρχει σταθερή αρχιτεκτονική, αλλά για κάθε εφαρμογή δημιουργείται διαφορετική αρχιτεκτονική μνήμης, που θα ταιριάζει καλύτερα στον τρόπο πρόσβασης στα δεδομένα της εφαρμογής. Δημιουργούνται επομένως εσωτερικά τμήματα μνήμης, διαφορετικού μεγέθους κάθε φορά, επιτρέποντας την παράλληλη πρόσβαση σε αυτά από επιμέρους υπολογιστικά τμήματα της εφαρμογής, όπως αυτά ορίστηκαν προηγουμένως για το dataflow. Ένας περιορισμός αυτής της γρήγορης μνήμης είναι ότι δεν επιτρέπει δυναμική κατανομή μνήμης, όπως συνηθίζεται σε κλασικούς επεξεργαστές, ενώ ακόμη ειναι περιορισμένη σε μέγεθος, και χρειάζεται να γίνει μελέτη για την κατά το δυνατόν αποτελεσματικότερη αξιοποίηση της.

2.1.6 High-Level Synthesis Tools

Η συνηθισμένη μέθοδος ανάπτυξης εφαρμογών σε FPGA είναι μέσω γλωσσών περιγραφής υλικού, όπως η Verilog ή η VHDL. Σε αυτές, ο προγραμματιστής-σχεδιαστής περιγράφει το κύκλωμα της εφαρμογής σε επίπεδο μεταφοράς (δεδομένων) καταχωρητών (Register Transfer Level - RTL), και απαιτεί προσεκτικό προγραμματισμό σε επίπεδο bit. Συγκεκριμένα, χρησιμοποιούνται πολλές παράλληλες διεργασίες, οι οποίες περιγράφουν συνδυαστική λογική, βασικές αριθμητικές πράξεις και δρομολογούνται από το ρολόι του FPGA. Με τον τρόπο αυτό, ο προγραμματιστής-σχεδιαστής μπορεί πράγματι να ελέγξει τη διάταξη των λογικών πυλών πάνω στο διαθέσιμο υλικό.[6]

Ο συγκεκριμένος τρόπος προγραμματισμού είναι όμως αρκετά απαιτητικός και χρονοβόρος, σε σύγκριση με τη διαδικασία ανάπτυξης μιας εφαρμογής σε μια κλασική υπολογιστική πλατφόρμα. Επίσης, είναι δύσκολο να αλλάξει ο κώδικας, ώστε να βελτιωθεί μεταγενέστερα. Για το λόγο αυτό, τα τελευταία χρόνια έχουν αναπτυχθεί εργαλεία High-Level Synthesis (HLS), τα οποία αναλαμβάνουν τη μετάφραση ενός αλγορίθμου γραμμένου σε γλώσσα υψηλού επιπέδου (C/C++) σε γλώσσα περιγραφής υλικού RTL. Φυσικά, ο προγραμματιστής πρέπει κατά το σχεδιασμό της εφαρμογής να λαμβάνει υπόψιν τους διαθέσιμους πόρους του FPGA, ώστε να επιτύχει την επιθυμητή επιτάχυνση και παραλληλοποίηση, αλλά πλέον μπορεί να το κάνει σε

ένα πιο εύχρηστο για αυτόν περιβάλλον ανάπτυξης. Στα πλαίσια αυτής της διπλωματικής, θα χρησιμοποιηθεί ο **Vivado HLS Compiler** της **Xilinx**.

Μέσω του περιβάλλοντος SDSoC της Xilinx, δίνεται η δυνατότητα στον προγραμματιστή να αναπτύξει την εφαρμογή προς εκτέλεση στο PS, σε πυρήνα ARM που διαθέτει το χρησιμοποιούμενο SoC, και στη συνέχεια να επιταχύνει ένα υπολογιστικά έντονο κομμάτι, ορίζοντας την εκτέλεση του στο PL (FPGA). Επίσης δίνεται η δυνατότητα επαλήθευσης ορθής λειτουργίας της εφαρμογής, κάτι που ήταν αρκετά δύσκολο να υλοποιηθεί σε γλώσσα περιγραφής υλικού.

Παρακάτω θα παρουσιαστούν ορισμένες οδηγίες (directives) για τον μεταγλωττιστή, μέσω των οποίων καθορίζεται ο τρόπος λειτουργίας των συναρτήσεων που θα εκτελούνται στο PS, όπως και ο τρόπος μεταφοράς δεδομένων από το PS στο PL, ώστε να επιτευχθεί η μέγιστη δυνατή απόδοση. Επίσης μέσω του περιβάλλοντος SDSoC έχουν προταθεί κάποια επιπλέον directives, τα οποία ακολουθούν.[7]

- **Array Partition** : Κατακερματίζει έναν πίνακα σε μικρότερους υποπίνακες, αυξάνοντας τον αριθμό θυρών I/O που είναι διαθέσιμες για τον πίνακα. Με τον τρόπο αυτό ο πίνακας είναι προσβάσιμος από παραπάνω του ενός σημεία του κώδικα.

- **Stream** : Τα streams χρησιμοποιούνται όταν η πρόσβαση στα δεδομένα της εφαρμογής είναι ακολουθιακή και τα δεδομένα χρησιμοποιούνται μία φορά το καθένα. Αποτελεί στην ουσία μία ουρά FIFO.

- **Pipeline** : Αυξάνει την απόδοση της εφαρμογής, ενεργοποιώντας το Pipelining, όπως αυτό αναπτύχθηκε προηγουμένως. Στόχος η μείωση του διαστήματος έναρξης (Initation Interval - II), έτσι ώστε διαδοχικές επαναλήψεις να ξεκινούν το συντομότερο δυνατόν, και η εφαρμογή να μπορεί να δέχεται εισόδους δεδομένων με ίδιο ρυθμό.

- **Dataflow** : Αυξάνει την απόδοση της εφαρμογής, ενεργοποιώντας το Dataflow, όπως αυτό αναπτύχθηκε προηγουμένως.

- **Unroll** : Το «ξετύλιγμα» (Unrolling) ενός βρόχου γίνεται δημιουργώντας πολλαπλά αντίγραφα των εντολών του βρόχου, μειώνοντας το πλήθος των επαναλήψεων κατά τον ίδιο παράγοντα (unroll factor). Με αυτόν τον τρόπο, αξιοποιούνται καλύτερα οι δυνατότητες του FPGA, αφού κάθε νέα εντολή ανατίθεται σε ξεχωριστό μέρος από τους διαθέσιμους πόρους, και εκτελείται παράλληλα με τις υπόλοιπες. Επίσης, μειώνεται το πλήθος των φορών που γίνεται έλεγχος ολοκλήρωσης του βρόχου

- **Inline** : Χρησιμοποιώντας αυτό, ανεβάζει ένα επίπεδο στην ιεραρχία του RTL σχεδίου τα επιμέρους υπολογιστικά τμήματα της εφαρμογής.

- **Data Access Pattern** : Καθορίζεται το μοτίβο πρόσβασης στα δεδομένα (ακολουθιακό ή τυχαίο)

- **Zero Copy** : Η μεταφορά δεδομένων μεταξύ PS και PL γίνεται μέσω της μοιραζόμενης μνήμης, διαμέσου μιας AXI master bus διεπαφής. Πρέπει να ορίζεται το μέγεθος

πινάκων που μεταφέρονται στην hardware function, είτε με σταθερό αριθμό, είτε μέσω μεταβλητής της hardware function

- **Mem Attribute** : Ο μεταγλωττιστής ενημερώνεται εάν η μνήμη που έχει δεσμευθεί για έναν πίνακα είναι φυσικά (physically) συνεχόμενη, ώστε να επιλέξει το αποδοτικότερο μέσο μεταφοράς δεδομένων.

2.2 Πολλαπλασιασμός Αραιού Πίνακα με Διάνυσμα (SpMV)

Η αραιότητα ενός πίνακα μπορεί να συνδεθεί εννοιολογικά με συστήματα χαλαρής ζεύξης (loosely-coupled). Τέτοια συστήματα εμφανίζονται σε επιστημονικές περιοχές όπως Θεωρία Δικτύων, Ρευστοδυναμική, Όραση Υπολογιστών και σε εφαρμογές που έχουν σχέση με την επίλυση Μερικών Διαφορικών Εξισώσεων. Αραιοί πίνακες μπορούν ακόμη να χρησιμοποιηθούν για την αναπαράσταση μεγάλων γράφων, χρησιμοποιώντας λίστες γειτνίασης. Ένα παράδειγμα τέτοιου γράφου είναι ο παγκόσμιος ιστός (World Wide Web), όπου ακμές μεταξύ κόμβων δηλώνουν τη σύνδεση σελίδων μέσω hyperlink.

2.2.1 Αραιοί πίνακες και Αναπαράστασή τους

Αραιός χαρακτηρίζεται ένας πίνακας που αποτελείται κυρίως απο μηδενικά. Η αραιότητα (sparsity) ενός πίνακα προκύπτει ως ο λόγος του πλήθους μηδενικών στοιχείων προς το συνολικό πλήθος στοιχείων του πίνακα, και με ανάλογο τρόπο ορίζεται και η πυκνότητα (density) του πίνακα. Για παράδειγμα ο ακόλουθος πίνακας αποτελείται από 9 μη-μηδενικά στοιχεία σε σύνολο 25 στοιχείων, και έχει 64% αραιότητα και 36% πυκνότητα.

$$A = \begin{bmatrix} 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 3 & 0 & 2 \\ 7 & 0 & 0 & 5 & 0 \\ 0 & 9 & 0 & 8 & 0 \\ 0 & 0 & 6 & 0 & 0 \end{bmatrix}$$

Σχήμα 2.15: Αραιός πίνακας 5x5

Δεν υπάρχει κάποιο αυστηρό κριτήριο, σύμφωνα με το οποίο κάτω από ορισμένο ποσοστό πυκνότητας χαρακτηρίζεται ένας πίνακας ως αραιός. Ένας πίνακας μπορεί να θεωρηθεί αραιός όταν προκύπτουν πλεονεκτήματα από τη διαφορετική διαχείρισή του (π.χ. αποθήκευση σε διαφορετική μορφή για εξοικονόμηση κατανάλωσης μνήμης). Ειδικές δομές δεδομένων έχουν δημιουργηθεί, για την αναπαράσταση αυτού του είδους πινάκων, οι οποίες συγκρατούν μόνο τη χρήσιμη πληροφορία που προκύπτει από αυτούς.

Coordinate (COO)

Ο πιο απλός τρόπος αναπαράστασης ενός αραιού πίνακα. Αποτελείται από τρία διανύσματα, μεγέθους όσο το πλήθος των μη-μηδενικών στοιχείων. Τα διανύσματα ***row_ind*** και ***col_ind***

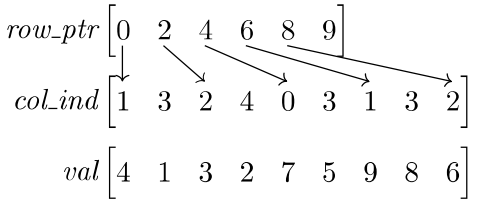
περιέχουν τη γραμμή και τη στήλη κάθε μη-μηδενικού στοιχείου αντίστοιχα, και το διάνυσμα ***val*** περιέχει τις τιμές των μη-μηδενικών στοιχείων. Τα δύο πρώτα διανύσματα έχουν ακέραιο τύπο δεδομένων, ενώ το τελευταίο έχει τύπο δεδομένων κινητής υποδιαστολής μονής (32-bits single) ή διπλής (64-bits double) ακρίβειας.

$$
\begin{aligned}
row\_ind &\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 \end{bmatrix} \\
col\_ind &\begin{bmatrix} 1 & 3 & 2 & 4 & 0 & 3 & 1 & 3 & 2 \end{bmatrix} \\
val &\begin{bmatrix} 4 & 1 & 3 & 2 & 7 & 5 & 9 & 8 & 6 \end{bmatrix}
\end{aligned}
$$

Σχήμα 2.16: Αναπαράσταση του πίνακα A (σχήμα 2.15) σε COO

Compressed Sparse Row (CSR)

Είναι το πιο ευρέως χρησιμοποιούμενο σχήμα αποθήκευσης για αραιούς πίνακες. Πλέον δεν υπάρχει το διάνυσμα *row_ind* του COO, αλλά ένα διάνυσμα δεικτών στην αρχή κάθε γραμμής του πίνακα, το ***row_ptr***. Το διάνυσμα αυτό έχει μέγεθος ίσο με το πλήθος γραμμών του πίνακα, αυξημένο κατά 1 (το τελευταίο στοιχείο είναι το πλήθος μη-μηδενικών στοιχείων). Τα διανύσματα ***col_ind*** και ***val*** παραμένουν ως έχουν.

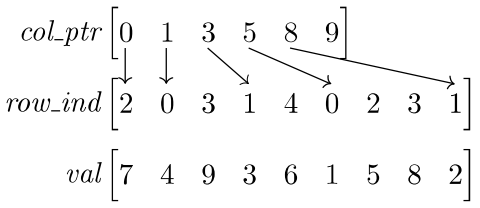


Σχήμα 2.17: Αναπαράσταση του πίνακα A (σχήμα 2.15) σε CSR

Μιας και ο αριθμός των γραμμών ενός πίνακα είναι συνήθως αρκετά μικρότερος από τα μη-μηδενικά στοιχεία του, παρατηρούμε την εξοικονόμηση στην κατανάλωση μνήμης σε σχέση με το COO. Επιπλέον, μέσω του διανύσματος row_ptr διευκολύνεται η τυχαία προσπέλαση των μη-μηδενικών στοιχείων μίας συγκεκριμένης γραμμής του πίνακα. Από την άλλη, απαιτεί τα δεδομένα να εισαχθούν ταξινομημένα, βάσει της γραμμής στην οποία ανήκουν, καθιστώντας δύσκολη την εισαγωγή νέων μη-μηδενικών στοιχείων.

Η δομή **CSC** παρουσιάζει ομοιότητες με τη CSR. Πλέον, γίνεται χρήση δεικτών στο διάνυσμα col_ind, αντί για το row_ind. Δεν θα αναλυθεί περαιτέρω γιατί δεν προσφέρει κάποια επιπλέον χρησιμότητα.

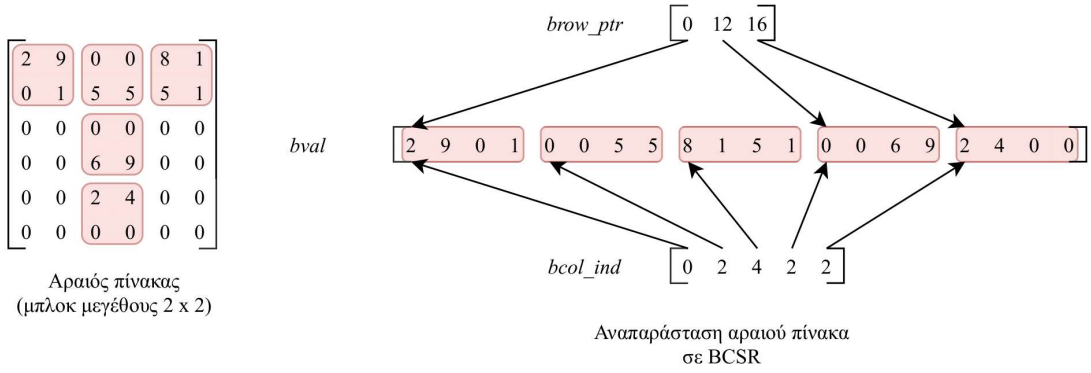
.



Σχήμα 2.18: Αναπαράσταση του πίνακα A (σχήμα 2.15) σε CSC

Εναλλακτικά σχήματα αποθήκευσης αραιών πινάκων

Παρά τη συμπαγή αναπαράσταση του CSR, υπάρχει συχνά πλεονάζουσα πληροφορία στη δομή *col_ind*. Τα μη-μηδενικά στοιχεία αραιών πινάκων που προκύπτουν από πραγματικές εφαρμογές εμφανίζονται σε μικρές πυκνές ομάδες (οριζόντιες, κάθετες ή διαγώνιες ακολουθίες, ή και δισδιάστατα μπλοκ). Επομένως έχει νόημα να μειωθεί περαιτέρω ο όγκος της πληροφορίας που αποθηκεύεται σχετικά με τη θέση των μη-μηδενικών στοιχείων στον πίνακα. Έτσι προέκυψε το σχήμα αποθήκευσης **Blocked Sparse Compressed Row (BCSR)**[8]. Σε αυτό, αντί να αποθηκεύονται δείκτες σε μη-μηδενικά στοιχεία, αποθηκεύονται δείκτες σε μπλοκ r × c), που περιέχουν τουλάχιστον ένα μη-μηδενικό στοιχείο. Μιας και το μέγεθος του μπλοκ είναι προκαθορισμένο, γίνεται, όπου είναι απαραίτητο, χρήση μηδενικών στοιχείων, για να σχηματιστούν πλήρη μπλοκ.

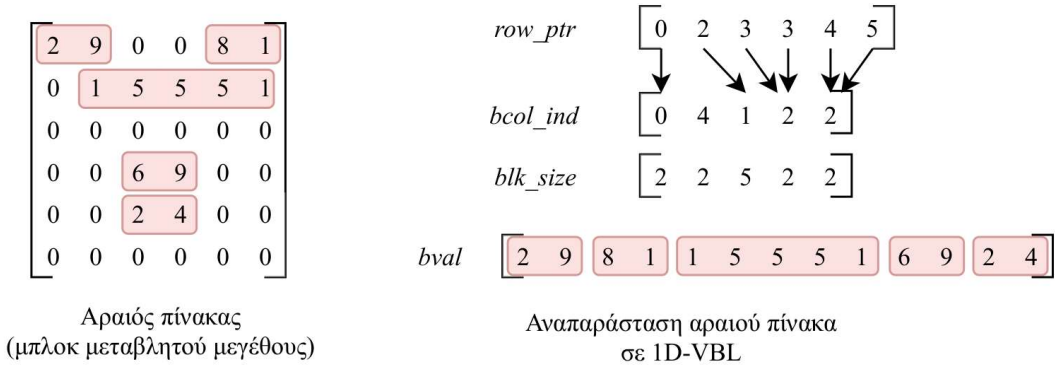


Σχήμα 2.19: Σχήμα αποθήκευσης BCSR

Ομοίως έχουν προταθεί σχήματα αποθήκευσης που εκμεταλλεύονται διαγώνιες ακολουθίες μη-μηδενικών στοιχείων **(RSDIAG/BCSD)**[9]. Μπορούμε εύκολα να καταλάβουμε ότι η επιλογή του ιδανικού μεγέθους μπλοκ απαιτεί προ-επεξεργασία του πίνακα, ώστε να περιοριστεί η εισαγωγή των μηδενικών κατά το δυνατό.

Περαιτέρω βελτιώσεις του σχήματος αυτού προσπαθούν να ομαδοποιούν μη-μηδενικά στοιχεία, αποφεύγοντας τη χρήση μεγάλου αριθμού επιπρόσθετων μηδενικών. Αυτό το επιτυγχάνουν χρησιμοποιώντας μπλοκ όχι σταθερού, αλλά μεταβλητού μεγέθους, διατηρώντας

σε μία νέα δομή το μέγεθος κάθε μπλοκ. Στο σχήμα **1-Dimensional Variable Block Length(1D-VBL)**[10] εντοπίζονται διαδοχικά μη-μηδενικά στοιχεία του πίνακα και αποθηκεύονται ως πλήρη μπλοκ.



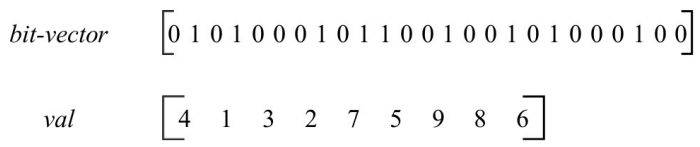
Σχήμα 2.20: Σχήμα αποθήκευσης 1D-VBL

Τέλος, έχει προταθεί το **Compress Sparse eXtended (CSX)**[11], στο οποίο γίνεται εκμετάλλευση των πυκνών υποπεριοχών ενός πίνακα, οποιουδήποτε είδους, μειώνοντας περαιτέρω την κατανάλωση μνήμης.

Σχήματα αποθήκευσης αραιών πινάκων βελτιστοποιημένα για FPGA

Μέχρι στιγμής, τα σχήματα αποθήκευσης που έχουν παρουσιαστεί είναι σχεδιασμένα για συστήματα κλασικών επεξεργαστών, χρησιμοποιώντας κωδικοποίηση επιπέδου λέξης (word-level encoding). Τα FPGA χειρίζονται καλύτερα δεδομένα κωδικοποιημένα σε επίπεδο bit.

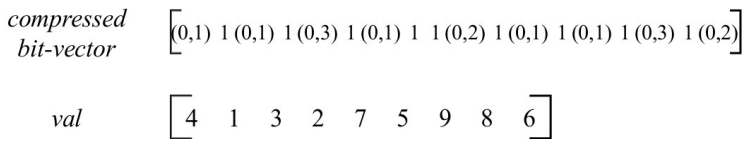
Το πιο απλό σχήμα αποθήκευσης χρησιμοποιεί ένα διάνυσμα bit, οι τιμές του οποίου υποδηλώνουν εάν το στοιχείο στη θέση αυτή του πίνακα είναι μηδενικό ή όχι. Συγκεκριμένα, για μη-μηδενικά στοιχεία χρησιμοποιείται η τιμή 1, ενώ για μηδενικά η τιμή 0, όπως φαίνεται και παρακάτω.



Σχήμα 2.21: Αναπαράσταση του πίνακα A (σχήμα 2.15) με Bit-Vector

Στο σχήμα αποθήκευσης **Compressed Bit-Vector** εφαρμόζεται συμπίεση Run-Length Encoding στο διάνυσμα bit, όπου εμφανίζονται ακολουθίες απο μηδενικά στοιχεία. Βλέπουμε τώρα να υπάρχουν ζευγάρια της μορφής *(0,X)*, όπου X ο αριθμός των συνεχόμενων μηδενικών που εμφανίζονται στον αραιό πίνακα.

Τέλος στο [12] προτείνεται η βελτιωμένη εκδοχή **Compressed Variable-Length Bit-Vector (CVBV)**, όπου η συμπίεση σε run-length encoding αποθηκεύεται σε ένα μεταβλητού μεγέθους πεδίο, μειώνοντας περαιτέρω το μέγεθος των δεδομένων που πρέπει να σταλούν



Σχήμα 2.22: Αναπαράσταση του πίνακα A (σχήμα 2.15) με Compressed Bit-Vector

στο FPGA για την εκτέλεση του SpMV. Το συγκεκριμένο σχήμα αποθήκευσης επιτυγχάνει συμπίεση έως και 43% σε σχέση με το CSR.

2.2.2 Υπολογιστικός πυρήνας SpMV

Ο υπολογιστικός πυρήνας του Πολλαπλασιασμού Αραιού Πίνακα με Διάνυσμα (SpMV) βρίσκεται στην καρδιά της επίλυσης γραμμικών συστημάτων μεγάλης κλίμακας σε διάφορα επιστημονικά πεδία. Είναι επομένως αναγκαία η κατά το δυνατόν αποτελεσματικότερη εκτέλεση του και είναι κάτι που αποτελεί πεδίο έρευνας για τον τομέα του High Performance Computing. Μιας και η υλοποίηση που προέκυψε στα πλαίσια της διπλωματικής στηρίζεται στη δομή αποθήκευσης CSR, θα εστιάσουμε στον υπολογιστικό πυρήνα αυτής.

Algorithm 1 CSR-SpMV

Input: (*row_ptr*, *col_ind*, *val*), sparse array of size $N \times M$ with NNZ non-zero elements in CSR format
Input: x, vector of size M
Output: y, vector of size N
for $i \leftarrow 0$ **to** N **do**
  for $j \leftarrow row\_ptr[i]$ **to** $row\_ptr[i+1]$ **do**
    y[i] $\leftarrow y[i] + val[j] \times x[col\_ind[j]]$
  end
end

Πρώτον, παρατηρούμε ότι η πρόσβαση στο διάνυσμα x είναι τυχαία και εξαρτάται από τη δομή των μη-μηδενικών στοιχείων του πίνακα. Επομένως, η χρήση του x πρέπει να γίνεται όσο πιο αποδοτικά γίνεται. Επιπλέον, καθένα από τα στοιχεία των διανυσμάτων *row_ptr*, *col_ind*, *val* χρησιμοποιείται μόνο μια φορά κατά την εκτέλεση του υπολογιστικού πυρήνα. Τέλος, μπορούμε ακόμη να παρατηρήσουμε ότι, αν και ο SpMV μπορεί να παραλληλοποιηθεί κατά τις γραμμές του πίνακα, ενδεχομένως να προκύψουν φαινόμενα άνισης κατανομής του υπολογιστικού φόρτου, καθώς δεν λαμβάνεται υπόψιν το πλήθος των μη-μηδενικών στοιχείων ανά γραμμή.

Συμπερασματικά, λόγω της αλγοριθμικής του φύσης, ο υπολογιστικός πυρήνας CSR-SpMV εμφανίζει χαμηλή υπολογιστική ένταση, σε σχέση με αντίστοιχους πυρήνες που αφορούν πυκνούς πίνακες.

2.3 Σχετική Έρευνα

Η streaming φιλοσοφία του αλγορίθμου SpMV και οι ακανόνιστες προσβάσεις στο διάνυσμα πολλαπλασιασμού στην κρυφή μνήμη (cache) ενός κλασικού επεξεργαστικού συστήματος οδήγησαν τους ερευνητές στα FPGA και στα πλεονεκτήματα που η συγκεκριμένη αρχιτεκτονική προσφέρει. Συγκεκριμένα, πέρα από την καλή απόδοση των FPGA σε υπολογισμούς κινητής υποδιαστολής, η on-chip μνήμη και η πληθώρα μπλοκ I/O, που αυτά διαθέτουν, περιορίζουν τη μεγάλη καθυστέρηση που συνέβαινε λόγω των cache-misses. Οι πρώτες υλοποιήσεις έγιναν σε γλώσσα περιγραφής υλικού (HDL).

Στο [13] υλοποιείται ο αλγόριθμος CSR-SpMV, όπου χρησιμοποιούνται παράλληλα k πολλαπλασιαστές, και κύκλωμα συγκέντρωσης αποτελεσμάτων (reduction circuit), το οποίο φροντίζει να αθροίζονται τα επιμέρους αποτελέσματα με ορθό τρόπο και να αντιμετωπίζονται οι Read After Write (RAW) κίνδυνοι.

Στο [14], επιχειρείται παραλληλοποίηση του αλγορίθμου, αναθέτοντας μία γραμμή του πίνακα σε κάθε ζεύγος πολλαπλασιαστή και αθροιστή που χρειάζεται στο CSR-SpMV. Το μέγιστο πλήθος γραμμών, των οποίων το αποτέλεσμα μπορεί να υπολογιστεί παράλληλα περιορίζεται από το μέγιστο εύρος ζώνης μνήμης του εκάστοτε FPGA που χρησιμοποιείται. Η μετάδοση (streaming) του αραιού πίνακα στις υπολογιστικές μονάδες του FPGA γίνεται σε ζεύγη τιμής-δείκτη στήλης, από τα διανύσματα *val*, *col_ind* αντίστοιχα. Για την τιμή χρησιμοποιείται αριθμητική 64 bit, ενώ για το δείκτη της στήλης αριθμητική 16 bit, για εξοικονόμηση εύρους ζώνης μνήμης.

Το πρόβλημα σε ένα FPGA είναι το περιορισμένο εύρος ζώνης για επικοινωνία με την εξωτερική μνήμη, από την οποία τροφοδοτείται με τα δεδομένα του αραιού πίνακα και του διανύσματος. Στο [15] γίνεται προσπάθεια περιορισμού του φαινομένου αυτού, εφαρμόζοντας συμπίεση στα δεδομένα του αραιού πίνακα που μεταφέρονται στο FPGA. Συγκεκριμένα, δημιουργείται ένα λεξικό με τις k πιο συχνά εμφανιζόμενες τιμές του πίνακα, το οποίο αποθηκεύεται στη BRAM του FPGA, για ταχύτερη πρόσβαση σε αυτές. Συμπίεση εφαρμόζεται ακόμη στο [16], αυτή τη φορά στους δείκτες θέσης των στοιχείων του πίνακα.

Τέλος, ο αλγόριθμος SpMV υλοποιήθηκε σε περιβάλλον High Level Synthesis (HLS) στο [17], κάνοντας χρήση των οδηγιών που τα HLS εργαλεία προσφέρουν, όπως Pipelining, Loop-Unrolling, Dataflow, και σε μία γλώσσα πιο εύχρηστη και ευέλικτη σε βελτιστοποιήσεις. Η υλοποίηση, που θα αναλυθεί στο επόμενο κεφάλαιο, βασίζεται στη δημοσίευση αυτή.

Κεφάλαιο 3

Υλοποίηση και Αξιολόγηση

Στο κεφάλαιο αυτό θα παρουσιαστεί η πορεία που ακολουθήθηκε για την υλοποίηση του αλγορίθμου Πολλαπλασιασμού Αραιού Πίνακα με Διάνυσμα (SpMV) σε FPGA. Αρχικά, θα εξετάσουμε τα αποτελέσματα της απευθείας μεταφοράς κώδικα γραμμένου για κλασικούς επεξεργαστές σε FPGA. Στη συνέχεια, εξετάζουμε ένα εναλλακτικό σχήμα αποθήκευσης αραιού πίνακα, και περαιτέρω βελτιστοποιήσεις, προκειμένου να εκμεταλλευτούμε τα πλεονεκτήματα που το προγραμματιστικό μοντέλο των FPGA προσφέρει. Σε κάθε βήμα της υλοποίησης θα παρουσιαστούν ενδιάμεσα αποτελέσματα της εκτέλεσης του αλγορίθμου SpMV για διαφορετικού μεγέθους αραιούς πίνακες, και η σύγκριση τους με τους χρόνους εκτέλεσης του αλγορίθμου στο επεξεργαστικό σύστημα (PS).

3.1 Εισαγωγή

Σε όλες τις υλοποιήσεις που θα παρουσιαστούν στη συνέχεια, το διάβασμα του αραιού πίνακα και του διανύσματος γίνεται στο επεξεργαστικό σύστημα (PS), και οι υπολογισμοί στη προγραμματίσιμη λογική (PL). Η μεταφορά δεδομένων μεταξύ των δύο υποσυστημάτων γίνεται μέσω των θυρών υψηλής απόδοσης (HP Ports) που η πλατφόρμα μας διαθέτει. Τα directives που χρησιμοποιούμε για τη μεταφορά των διανυσμάτων αναπαράστασης του αραιού πίνακα *row_ptr*, *col_ind*, *val*, όπως και του διανύσματος-αποτελέσματος y είναι τα εξής :

- **Zero Copy**
- **Access Pattern** : Sequential, μιας και το μοτίβο προσβάσεων στα διανύσματα αναπαράστασης του αραιού πίνακα είναι ακολουθιακό.
- **Mem Attribute** : Ενημερώνουμε το μεταγλωττιστή ότι η μνήμη που έχει δεσμευθεί για τα διανύσματα αυτά είναι σε φυσικά συνεχόμενες θέσεις μνήμης. Στη μεριά του PS η δέσμευση μνήμης έχει γίνει με την εντολή **sds_alloc_non_cacheable**.

Με τον τρόπο αυτό επιτυγχάνουμε την κατά το δυνατόν αποδοτικότερη μεταφορά των δεδομένων μεταξύ των δύο υποσυστημάτων. Για το διάνυσμα x, στο οποίο το μοτίβο προσβάσεων είναι τυχαίο, δεν γίνεται να ενεργοποιήσουμε αυτά τα directives.

3.2 Συλλογή πινάκων

Για να αξιολογήσουμε την απόδοση των υλοποιήσεών μας, επιλέξαμε 20 πίνακες από τη συλλογή αραιών πινάκων SuiteSparse[18], που καλύπτουν ένα ευρύ φάσμα επιστομονικών πεδίων και εφαρμογών. Οι συγκεκριμένοι πίνακες παρουσιάζουν διαφορετικές ιδιότητες, που επηρεάζουν την απόδοση του αλγορίθμου SpMV, όπως διαστάσεις πίνακα, αραιότητα, αριθμός μη-μηδενικών στοιχείων ανά γραμμή.

Matrix	**Dimensions**	**Non-zeros**	**Application Domain**
large-dense	2000 × 2000	4000000	Synthetic Matrix
human_gene1	22283 × 22283	12345963	Undirected Weighted Graph
nd24k	72000 × 72000	14393817	2D-3D Problem
JP	87616 × 67320	13734559	Tomography
consph	83334 × 83334	3046907	2D-3D Problem
poisson3Db	85623 × 85623	2374949	Computational Fluid Dynamics
barrier2-12	115625 × 115625	3897557	Semiconductor Devices
FEM_3D_thermal2	147900 × 147900	3489300	Thermal Energy
SiO2	155331 × 155331	5719417	Quantum Chemistry
degme	185501 × 659415	8127528	Linear Programming
offshore	259789 × 259789	2251231	Electromagnetics
Ga41As41H72	268096 × 268096	9378286	Quantum Chemistry
parabolic_fem	525825 × 525825	2100225	Computational Fluid Dynamics
rajat30	643994 × 643994	6175377	Circuit Simulation
ASIC_680k	682862 × 682862	3871773	Circuit Simulation
Hardesty2	929901 × 303645	4020731	Computer Graphics/Vision
boneS10	914898 × 914898	28191660	Model Reduction
audikw_1	943695 × 943695	39297771	Structural Problem
webbase-1M	1000005 × 1000005	3105536	Directed Weighted Graph
thermal2	1228045 × 1228045	4904179	Thermal Energy
G3_circuit	1585478 × 1585478	4623152	Circuit Simulation

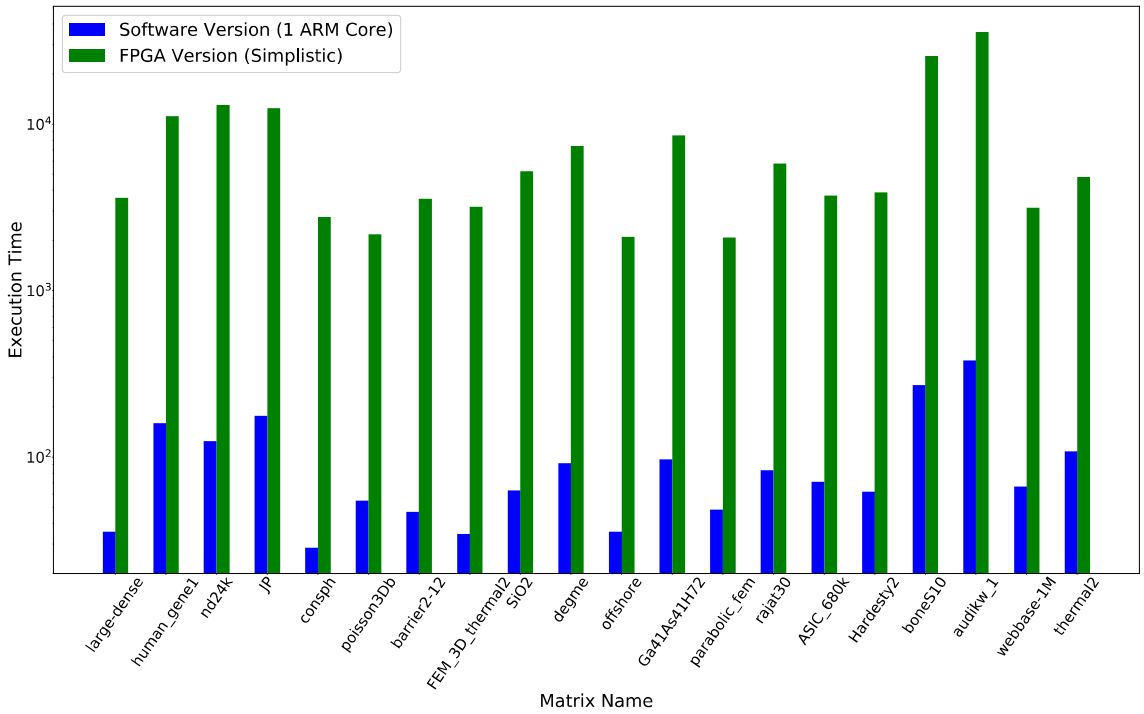
3.3 Αρχική υλοποίηση

Ως μία πρώτη προσέγγιση, αποφασίσαμε να μεταφέρουμε κώδικα γραμμένο για σύστημα κλασικού επεξεργαστή απευθείας προς εκτέλεση στο FPGA.

```
int spmv(
        float   * values,   int     * col_indices,  int     * row_ptr,
        float   * x,        float   * y,
        int     row_size,   int     col_size,       int     data_size
){
    int i=0, j=0, rowStart=0, rowEnd=row_size;
L1: for (i = rowStart; i < rowEnd; ++i) {
        float y0 = 0.0;
L2:     for (j = row_ptr[i] ; j < row_ptr[i+1]; ++j) {
            y0 += values[j] * x[col_indices[j]];
        }
        y[i] = y0;
    }
}
```

Listing 3.1: Αρχική υλοποίηση του CSR-SpMV

Μεταφέροντας αυτούσιο τον κώδικα, δεν λαμβάνουμε υπόψιν τους διαθέσιμους πόρους του FPGA. Επιπλέον, οι συνεχείς τυχαίες προσβάσεις στο διάνυσμα x υποχρεώνουν σε συνεχή επικοινωνία τα δύο υποσυστήματα PS και PL, η οποία είναι κοστοβόρα. Όπως είναι αναμενόμενο, οι χρόνοι εκτέλεσης για όλους τους πίνακες είναι αρκετές τάξεις μεγέθους μεγαλύτεροι σε σχέση με τους χρόνους εκτέλεσης σε έναν πυρήνα του PS.



Σχήμα 3.1: Σύγκριση CSR-SpMV για 1 πυρήνα ARM και για FPGA(μη βελτιστοποιημένο)

3.3.1 Βελτιστοποίηση με HLS Directives

Στην προσπάθειά μας να βελτιώσουμε την απόδοση του CSR-SpMV, εξετάσαμε τη χρήση των οδηγιών (directives) που τα εργαλεία HLS προσφέρουν. Συγκεκριμένα, δεν μπορούμε να εφαρμόσουμε Pipelining στον εξωτερικό βρόχο L1, καθώς κάτι τέτοιο θα απαιτούσε το πλήρες ξετύλιγμα(Unroll) του εσωτερικού βρόχου, για τον οποίο δεν γνωρίζουμε από πριν τον πλήρη αριθμό επαναλήψεων(εξαρτάται από τους δείκτες *row_ptr[i]* και *row_ptr[i+1]*. Για το λόγο αυτό χρησιμοποιήσαμε τις οδηγίες **Pipeline** και **Unroll** (με παράγοντα 2) στον εσωτερικό βρόχο **L2**.

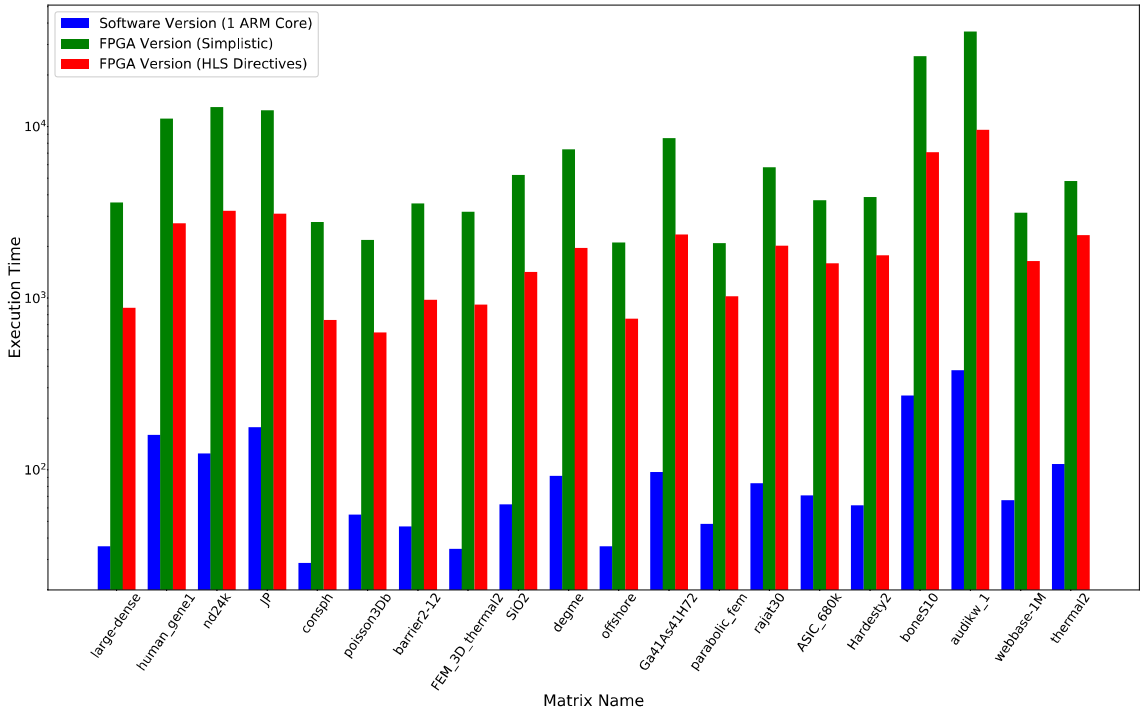
```
int spmv(
        float   * values,   int     * col_indices,  int     * row_ptr,
        float   * x,        float   * y,
        int     row_size,   int     col_size,       int     data_size
){
    int i=0, j=0, rowStart=0, rowEnd=row_size;
L1: for (i = rowStart; i < rowEnd; ++i) {
        float y0 = 0.0;
L2:     for (j = row_ptr[i] ; j < row_ptr[i+1]; ++j) {
            #pragma HLS PIPELINE II=1
            #pragma HLS unroll factor=2
            y0 += values[j] * x[col_indices[j]];
        }
        y[i] = y0;
    }
}
```

Listing 3.2: Υλοποίηση του CSR-SpMV με οδηγίες HLS

Στον παρακάτω πίνακα βλέπουμε τη σύγκριση της αξιοποίησης των διαθέσιμων πόρων του FPGA. Παρατηρούμε ότι και στις δύο υλοποιήσεις που έχουν παρουσιαστεί μέχρι στιγμής αξιοποιείται πολύ μικρό μέρος της διαθέσιμης περιοχής του.

Version	BRAM_18K	DSP48E	FF	LUT
CSR(Simplistic)	10	5	3904	4157
CSR(Optimized)	10	5	4434	4620
Available	1824	2520	548160	274080

Η βελτίωση στους χρόνους εκτέλεσης είναι σε σχέση με την αρχική FPGA υλοποίηση ελάχιστη για όλους τους πίνακες. Τα αποτελέσματα είναι και πάλι τάξεις μεγέθους χειρότερα σε σχέση με την έκδοση που εκτελείται στο υποσύστημα PS.

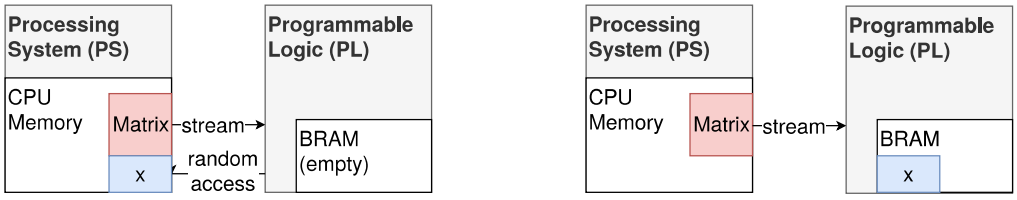


Σχήμα 3.2: Σύγκριση χρόνων εκτέλεσης CSR-SpMV για 1 πυρήνα ARM και για FPGA (βελτιστοποιημένο και μη)

Χρειάζεται να προσεγγίσουμε το πρόβλημα λαμβάνοντας υπόψιν τα πλεονεκτήματα αλλά και τους περιορισμούς της FPGA αρχιτεκτονικής.

- Σημαντικό ρόλο στην ταχύτητα εκτέλεσης του αλγορίθμου παίζουν οι συνεχείς τυχαίες προσβάσεις στο διάνυσμα x με το οποίο εκτελείται ο πολλαπλασιασμός του αραιού πίνακα. Για το λόγο αυτό, αποφασίσαμε να αποθηκεύουμε το διάνυσμα αυτό στη διαθέσιμη BRAM μνήμη του υποσυστήματος PL. Βέβαια, αυτό θα περιορίσει το μέγιστο μέγεθος προβλήματος που θα μπορεί να επιλυθεί, καθώς η διαθέσιμη BRAM είναι περιορισμένη σε μέγεθος (το SoC FPGA του εργαστηρίου διαθέτει 32 Mb μνήμης).

  Όπως βλέπουμε και στο παρακάτω σχήμα, τόσο ο πίνακας όσο και το διάνυσμα είναι αποθηκευμένα στο υποσύστημα PS. Από τις επόμενες υλοποιήσεις, πριν την εκτέλεση του πολλαπλασιασμού, θα μεταφέρουμε το διάνυσμα στη BRAM.



(α′) Μέχρι στιγμής υλοποιήσεις (β′) Υλοποιήσεις από εδώ και στο εξής

Σχήμα 3.3: Θέσεις αποθήκευσης αραιού πίνακα (*Matrix*) και διανύσματος (x) στα υποσυστήματα PS και PL

Όπως και προηγουμένως, φροντίζουμε τα στοιχεία του αραιού πίνακα να «ρέουν» (stream) από το PS στο PL κατά την εκτέλεση του αλγορίθμου.

- Πρέπει ακόμη να αξιοποιήσουμε πλήρως το διαθέσιμο εύρος ζώνης μνήμης του διαύλου επικοινωνίας του PS με το PL. Για το λόγο αυτό δημιουργήσαμε ένα νέο σχήμα αποθήκευσης, το οποίο βασίζεται στο CSR και θα αναφέρεται από εδώ και στο εξής ως **packedCSR**.
- Η παράλληλη εκτέλεση των πράξεων βελτιώνει περαιτέρω την απόδοση της υλοποίησης μας, εφαρμόζοντας **Vectorization** στους υπολογισμούς ανά γραμμή.
- Προκειμένου να αξιοποιηθούν καλύτερα οι διαθέσιμοι πόροι του FPGA, δημιουργούμε πολλαπλές υπολογιστικές μονάδες (Compute Units), μοιράζοντας τον υπολογιστικό φόρτο σε αυτές. Κάθε υπολογιστική μονάδα αναλαμβάνει συγκεκριμένο μέρος του πίνακα, με λογική **1D-Blocking**.
- Για την αντιμετώπιση του περιορισμένου μεγέθους του διανύσματος x που μπορούμε να αποθηκεύσουμε στη BRAM του FPGA, εφαρμόζουμε **2D-Blocking** στον πίνακα.

3.4 Αναπαράσταση packedCSR και packedCSR-SpMV

3.4.1 Σχήμα αποθήκευσης packedCSR

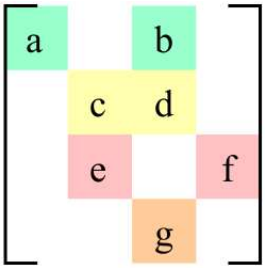
Όπως έχουμε ήδη αναφέρει, κατά την εκτέλεση του αλγορίθμου CSR-SpMV, η πρόσβαση στα δεδομένα του αραιού πίνακα είναι μοναδική και ακολουθιακή κατά τον τρόπο που αυτά έχουν αποθηκευτεί. Για το λόγο αυτό, επιλέγουμε να γίνεται μετάδοση (streaming) του πίνακα στο υποσύστημα PL μέσω των θυρών υψηλής απόδοσης (HP Ports) που διαθέτει το SoC-FPGA του εργαστηρίου.

Οι συγκεκριμένες θύρες έχουν εύρος ζώνης 128 bits, κάτι που σημαίνει ότι για να αξιοποιούνται πλήρως, θα πρέπει τα δεδομένα να μεταφέρονται από το PS στο PL μέσω μιας δομής, που κάθε στοιχείο της θα έχει εύρος 128 bits.

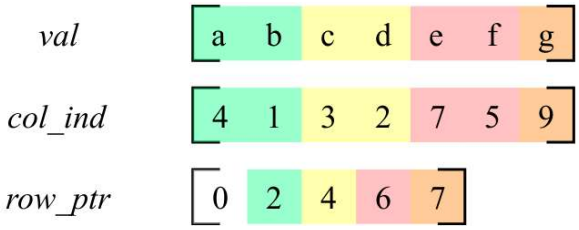
Για το λόγο αυτό δημιουργήσαμε το σχήμα αποθήκευσης **packedCSR**, το οποίο συνδυάζει τα τρια διανύσματα αναπαράστασης του σχήματος CSR, *row_ptr*, *col_ind* και *val*, σε μία δομή εύρους 128 bits. Η δομή αυτή δημιουργήθηκε με τη βοήθεια της βιβλιοθήκης της Xilinx[19] για ορισμένους από το χρήστη τύπους δεδομένων. Στον κώδικα που θα παρουσιαστεί στη συνέχεια αναφέρεται ως **uintbuswidth_t**.

Η συγκεκριμένη δομή προκύπτει ως εξής :

- Ο τύπος δεδομένων που χρησιμοποιούμε για τις τιμές (*val*) του πίνακα είναι **float** (απλής ακρίβειας), εύρους 32 bits. Το ίδιο εύρος ορίζουμε να έχουν και οι δείκτες θέσεων των τιμών του πίνακα (*col_ind*) όπως και ο αριθμός των μη-μηδενικών ανά γραμμή (*row_ptr*), χρησιμοποιώντας τη βιβλιοθήκη της Xilinx **ap_int.h**. Επομένως κάθε στοιχείο της δομής packedCSR αποτελείται από 4 αριθμούς.



(α′) Αραιός πίνακας



(β′) Αναπαράσταση CSR

0	2	a	0
b	2	0	2
c	1	d	2
0	2	e	1
f	3	0	1
g	2	0	0

(γ′) Αναπαράσταση packedCSR

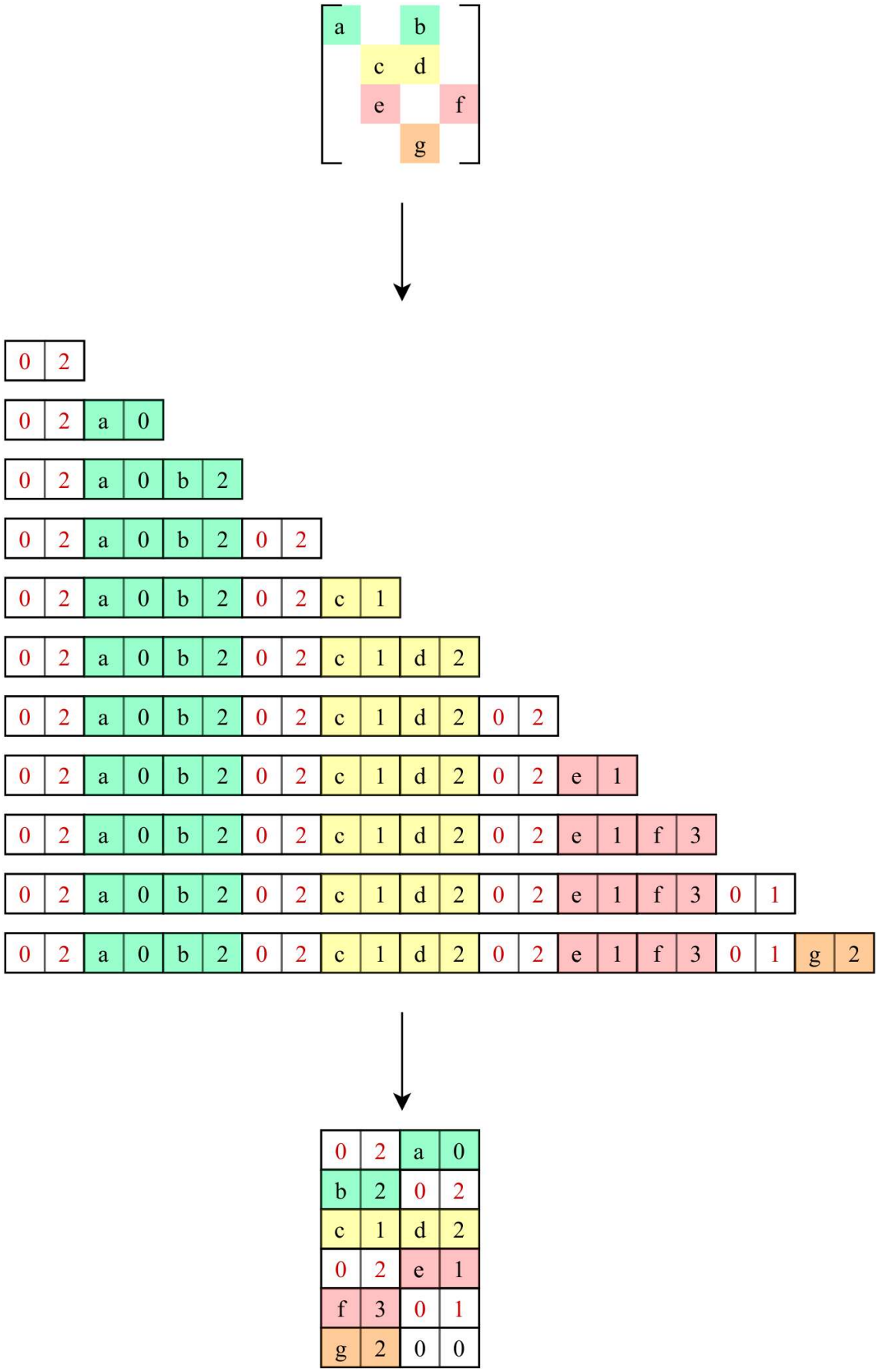
Σχήμα 3.4: Μετατροπή ενός αραιού πίνακα σε μορφή CSR και μετέπειτα σε μορφή packedCSR

- Όταν υπάρχει αλλαγή γραμμής στον πίνακα, στη δομή αποθηκεύεται ένα ζεύγος της μορφής **(0 , X)**, όπου X είναι ο αριθμός των μη-μηδενικών στοιχείων της τρέχουσας γραμμής. Ο αριθμός 0 χρησιμοποιείται ως «αναγνωριστικό» αλλαγής γραμμής, όπως θα δούμε αργότερα στον κώδικα. Ο αριθμός των μη-μηδενικών στοιχείων της τρέχουσας γραμμής προκύπτει από την αφαίρεση δύο διαδοχικών στοιχείων, στην κατάλληλη θέση, του διανύσματος *row_ptr*.

- Στη συνέχεια στη δομή αποθηκεύονται τα μη-μηδενικά στοιχεία της γραμμής, σε ζεύγη της μορφής **(τιμή , θέση-στη-γραμμή)**, δεδομένα που παίρνουμε από τα διανύσματα *val* και *col_ind* αντίστοιχα.

- Εάν είναι ανάγκη, γίνεται γέμισμα (padding) με ζεύγος της μορφής (0,0) προκειμένου κάθε στοιχείο της δομής να έχει τιμές και στα 128 bits του, όπως βλέπουμε στην τελευταία γραμμή του Σχήματος 3.4 (γ′).

Στην επόμενη σελίδα ακολουθεί σχηματική αναπαράσταση του τρόπου με τον οποίο δημιουργείται η δομή packedCSR για τον πίνακα του Σχήματος 3.4. Διαχειριζόμαστε τα δεδομένα από τα τρία διανύσματα του σχήματος CSR ως ένα stream δεδομένων.



Σχήμα 3.5: Αναλυτική δημιουργία αναπαράστασης packedCSR για αραιό πίνακα

3.4.2 Παρουσίαση αλγορίθμου SpMV με σχήμα αποθήκευσης packedCSR

Για να αξιοποιήσουμε περαιτέρω τις θύρες υψηλής απόδοσης του FPGA, αποφασίσαμε το διάνυσμα x όπως και το διάνυσμα y (το αποτέλεσμα του πολλαπλασιασμού), να αποθηκευτούν σε δομή που θα έχει εύρος 128 bits, η οποία ονομάζεται **uintbuswidth_t**. Μέσω αυτής θα μεταδίδονται από το PS στο PL και αντίστροφα με τον πιο αποδοτικό τρόπο. Μιας και κάθε τιμή των δύο διανυσμάτων έχει εύρος 32 bits, κάθε στοιχείο της δομής θα περιέχει 4 τιμές των δύο αυτών διανυσμάτων.

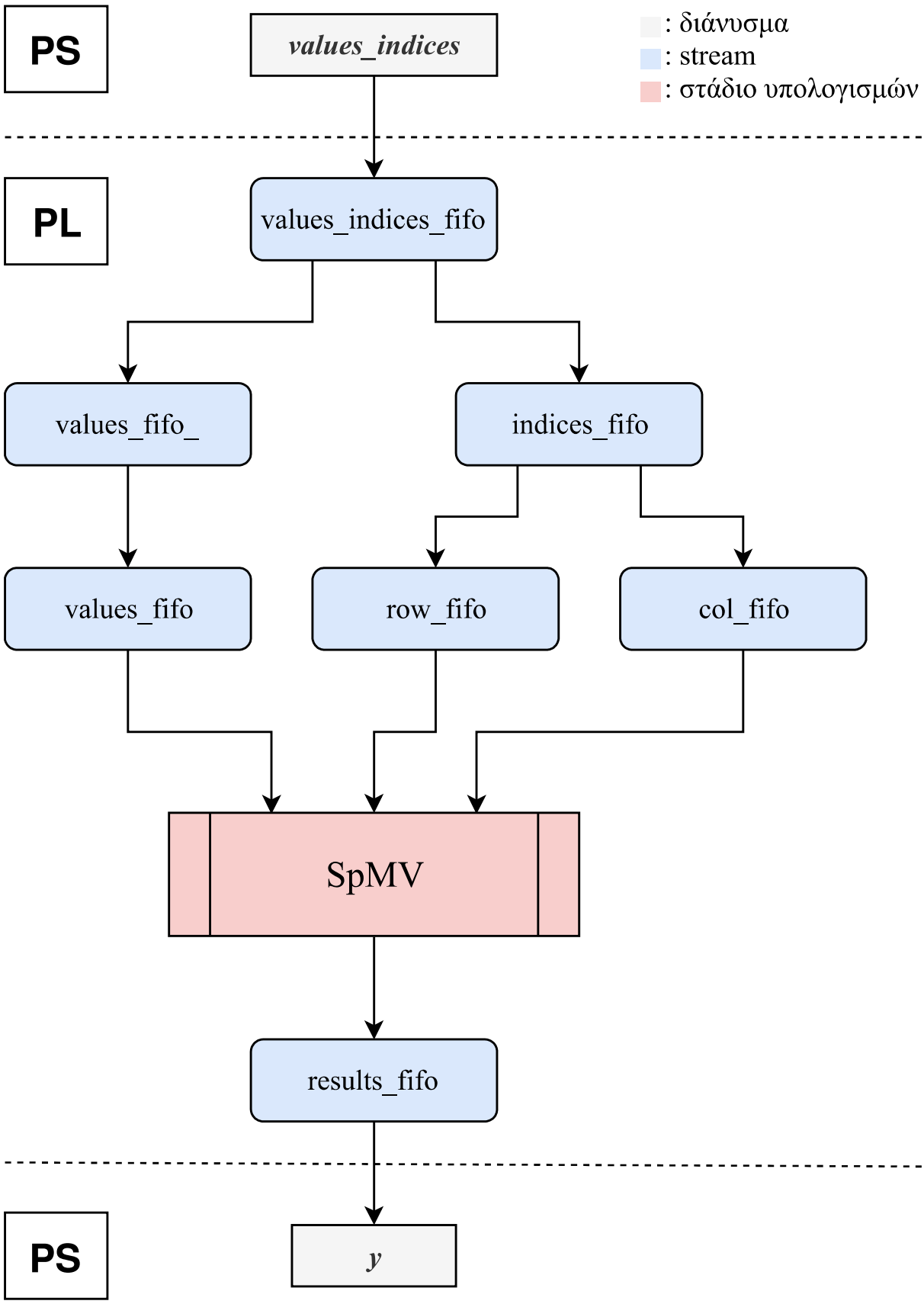
Αποθήκευση διανύσματος x στη BRAM του FPGA

Στο στάδιο αυτό αποθηκεύεται στη BRAM το διάνυσμα x που από το PS. Μιας και το διάνυσμα μεταφέρεται μέσω δομής τύπου **uintbuswidth_t**, αποθηκεύουμε σε ζευγάρια των 4 τις τιμές του διανύσματος στο τοπικό αντίγραφο ***x_local***.

```
  DATA_TYPE x_local[COLS_DIV_BLOCKS];
  #pragma HLS ARRAY_PARTITION variable=x_1_local cyclic factor=2 dim=1
L0: for (u32 i = 0; i < exp_col_size/4; i+=1) {
    #pragma HLS PIPELINE
    uintbuswidth_t x128_tmp = *(x+i);

    DATA_TYPE_ap_uint32_t f2uint_tmp_1;
    DATA_TYPE_ap_uint32_t f2uint_tmp_2;
    DATA_TYPE_ap_uint32_t f2uint_tmp_3;
    DATA_TYPE_ap_uint32_t f2uint_tmp_4;

    f2uint_tmp_1.apint = x128_tmp.range(31, 0);
    f2uint_tmp_2.apint = x128_tmp.range(63, 32);
    f2uint_tmp_3.apint = x128_tmp.range(95, 64);
    f2uint_tmp_4.apint = x128_tmp.range(127, 96);

    DATA_TYPE x_tmp_1 = f2uint_tmp_1.f;
    DATA_TYPE x_tmp_2 = f2uint_tmp_2.f;
    DATA_TYPE x_tmp_3 = f2uint_tmp_3.f;
    DATA_TYPE x_tmp_4 = f2uint_tmp_4.f;
    x_local[i*4+0] = x_tmp_1;
    x_local[i*4+1] = x_tmp_2;
    x_local[i*4+2] = x_tmp_3;
    x_local[i*4+3] = x_tmp_4;
  }
```

Listing 3.3: Αντιγραφή διανύσματος x στη BRAM

Στη συνέχεια, ξεκινάει η ροή των δεδομένων του αραιού πίνακα από το PS στο PL. Τα δεδομένα μεταφέρονται μέσω της δομής **values_indices**. Η ροή των δεδομένων γίνεται μέσω streams που προσφέρουν τα εργαλεία HLS, και ο διαμοιρασμός των δεδομένων στα κατάλληλα streams γίνεται όπως φαίνεται παρακάτω.



Σχήμα 3.6: Ροή δεδομένων κατά την εκτέλεση του αλγορίθμου packedCSR-SpMV

Πέρασμα τιμών στο stream *values_indices_fifo*

Η δομή **values_indices**, μεγέθους ind_size περιλαμβάνει ζεύγη της μορφής **(0,X)**, **(τιμή,θέση)** ή **(0,0)** όπως παρουσιάσαμε προηγουμένως. Κάθε στοιχείο της δομής περιέχει δύο τέτοια ζεύγη, και με τον τρόπο που παρουσιάζεται παρακάτω τα δεδομένα αυτά προωθούνται στο stream ***values_indices_fifo***.

```
  hls::stream<uintbushalfwidth_t> values_indices_fifo;
  #pragma HLS STREAM variable=values_indices_fifo depth=BUFFER_SIZE dim=1
L1: for (i = 0; i < ind_size; i++) {
    #pragma HLS PIPELINE
    uintbuswidth_t  tmp_value_indices = values_indices[i];
    values_indices_fifo << tmp_value_indices.range(63, 0);
    values_indices_fifo << tmp_value_indices.range(127, 64);
  }
```

Listing 3.4: Προώθηση δεδομένων αραιού πίνακα στο stream *values_indices_fifo*

Πέρασμα τιμών στα streams *values_fifo_* και *indices_fifo*

Στη συνέχεια, γίνεται διαχωρισμός των ζευγών στα streams ***indices_fifo*** και ***values_fifo_***. Τα ζεύγη (0,0) που είχαν χρησιμοποιηθεί για γέμισμα (padding) της δομής αγνοούνται και δεν προωθούνται στα νέα streams. Για τα άλλα δύο ζεύγη, τα πρώτα 32 bits προωθούνται στο stream (float) τιμών ***values_fifo_***, και τα υπόλοιπα 32 bits στο stream (ακεραίων) δεικτών ***indices_fifo***.

```
  hls::stream<DATA_TYPE> values_fifo_;
  #pragma HLS STREAM variable=values_fifo_ depth=BUFFER_SIZE dim=1
  hls::stream<u32>       indices_fifo;
  #pragma HLS STREAM variable=indices_fifo depth=BUFFER_SIZE dim=1
L2: for (i = 0; i < 2*ind_size; i++) {
    #pragma HLS PIPELINE
    uintbushalfwidth_t  tmp_value_indices = values_indices_fifo.read();

    DATA_TYPE_ap_uint32_t tmp_DATA_TYPE_uint32;
    tmp_DATA_TYPE_uint32.apint = tmp_value_indices.range(31, 0);
    u32 tmp_u32 = tmp_value_indices.range(63, 32);

    if (i < values_size+row_size) {
      values_fifo_ << tmp_DATA_TYPE_uint32.f;
      indices_fifo << tmp_u32;
    }
  }
```

Listing 3.5: Προώθηση δεδομένων στα streams *values_fifo_* και *indices_fifo*

Πέρασμα τιμών στα streams *row_fifo*, *col_fifo* και *values_fifo*

Πλέον μπορούμε να διαχωρίσουμε τα δεδομένα στα 3 streams που θα μας χρειαστούν για την εκτέλεση των υπολογισμών. Γίνεται έλεγχος εάν ο αριθμός μη-μηδενικών στοιχείων της τρέχουσας γραμμής είναι 0 (col_left = 0). Αυτό υποδηλώνει ότι υπήρξε αλλαγή γραμμής στον πίνακα, και προωθούμε στο stream ***row_fifo*** τον αριθμό των μη-μηδενικών στοιχείων της τρέχουσας γραμμής, από το stream *indices_fifo*. Φροντίζουμε παράλληλα να αφαιρέσουμε από το *values_fifo_* το 0 που ήταν το δεύτερο μέλος του συγκεκριμένου ζεύγους. Στην άλλη περίπτωση, έχουμε να κάνουμε με ζεύγος τιμής και θέσης της στον πίνακα, επομένως γεμίζουμε τα streams ***col_fifo*** και ***values_fifo*** με τα κατάλληλα δεδομένα. Πλέον είμαστε έτοιμοι να εκτελέσουμε τους υπολογισμούς ανά γραμμή.

```
  hls::stream<u32>    row_fifo;
  #pragma HLS STREAM variable=row_fifo depth=BUFFER_SIZE dim=1
  hls::stream<u32>    col_fifo;
  #pragma HLS STREAM variable=col_fifo depth=BUFFER_SIZE dim=1
  hls::stream<u32>    values_fifo;
  #pragma HLS STREAM variable=values_fifo depth=BUFFER_SIZE dim=1
  u32 col_left = 0;
L3: for(r=0; r< values_size+row_size; r++){
    #pragma HLS PIPELINE
    u32 index = indices_fifo.read();
    if(col_left == 0 ) {
      col_left = index;
      row_fifo << col_left;
      DATA_TYPE tmp = values_fifo_.read();
    } else{
      col_fifo << index;
      col_left--;
      values_fifo << values_fifo_.read();
    }
  }
```

Listing 3.6: Προώθηση δεδομένων στα streams *row_fifo*, *col_fifo* και *values_fifo*

Στάδιο υπολογισμών

Η διαδικασία του υπολογισμού κάθε αποτελέσματος ανά γραμμή έχει ως εξής :

1. Ελέγχεται εάν έχουμε αλλαγή γραμμής στον πίνακα, ομοίως με πριν, όταν *col_left = 0*. Στην περίπτωση αυτή, αντλούμε από το stream *row_fifo* τον αριθμό μη-μηδενικών στοιχείων της τρέχουσας γραμμής.

2. Στη συνέχεια, αντλούμε από τα streams *values_fifo* και *col_fifo* την τιμή και τη στήλη των μη-μηδενικών στοιχείων της τρέχουσας γραμμής. Εκτελείται ο πολλαπλασιασμός με το κατάλληλο στοιχείο του διανύσματος x, και το αποτέλεσμα αποθηκεύεται στον πίνακα *term*.

3. Όλοι οι όροι που έχουν αποθηκευτεί στον πίνακα *term* αποτελούν επιμέρους στοιχεία του αποτελέσματος που χρειάζεται να υπολογίσουμε για την τρέχουσα γραμμή. Αθροίζοντας τα, παίρνουμε βήμα-βήμα το τελικό αποτέλεσμα.

4. Όταν πλέον τελειώσουν τα στοιχεία της τρέχουσας γραμμής ($col\_left = 0$), γίνεται προώθηση του αποτελέσματος στο stream ***results_fifo*** και στην επόμενη επανάληψη του εξωτερικού βρόχου θα αρχίσει ο υπολογισμός του αποτελέσματος της επόμενης γραμμής.

Σημείωση : Ο αριθμός II, που είναι ο αριθμός των επαναλήψεων στους δύο εσωτερικούς βρόχους, είναι στην περίπτωση που παρουσιάζουμε ίσος με 1. Στη συνέχεια, στο πλαίσιο του Vectorization των υπολογισμών, θα αυξήσουμε τον αριθμό αυτόν, με σκοπό την ταυτόχρονη εκτέλεση περισσότερων του ενός πολλαπλασιασμών.

```
  hls::stream<DATA_TYPE> results_fifo;
  #pragma HLS STREAM variable=results_fifo depth=BUFFER_SIZE dim=1
  col_left = 0;
  DATA_TYPE sum;
  DATA_TYPE term[II];
  #pragma HLS ARRAY_PARTITION variable=term complete dim=1
L4: for (r = 0; r < values_size; r+=II) {
    #pragma HLS PIPELINE
    if (col_left == 0) {
      col_left = row_fifo.read();
      sum = 0;
    }
    DATA_TYPE value;
    u32 col;
    for (i = 0; i < II; i++) {
      #pragma HLS dependence variable=term inter false
      #pragma HLS unroll
      value = values_fifo.read();
      col   = col_fifo.read();
      term[i] = value * x[col];
    }

    DATA_TYPE sum_tmp = 0;
    for (i = 0; i < II; i++) {
      #pragma HLS dependence variable=term inter false
      #pragma HLS unroll
      sum_tmp  += term[i];
    }
    sum += sum_tmp;
    col_left -= II;

    if (col_left == 0)
      results_fifo << sum;
  }
```

Listing 3.7: Υπολογισμός αποτελεσμάτων ανά γραμμή

Αποθήκευση αποτελεσμάτων στο διάνυσμα y

Στο τελευταίο στάδιο γίνεται άντληση των αποτελεσμάτων από το stream results_fifo και αποθήκευση τους στον πίνακα που θα σταλεί από το PL στο PS. Όπως και στο πρώτο στάδιο, τα δεδομένα στέλνονται σε ζεύγη των 4, μέσω της δομής $\boldsymbol{y}$, τύπου *uintbuswidth_t*.

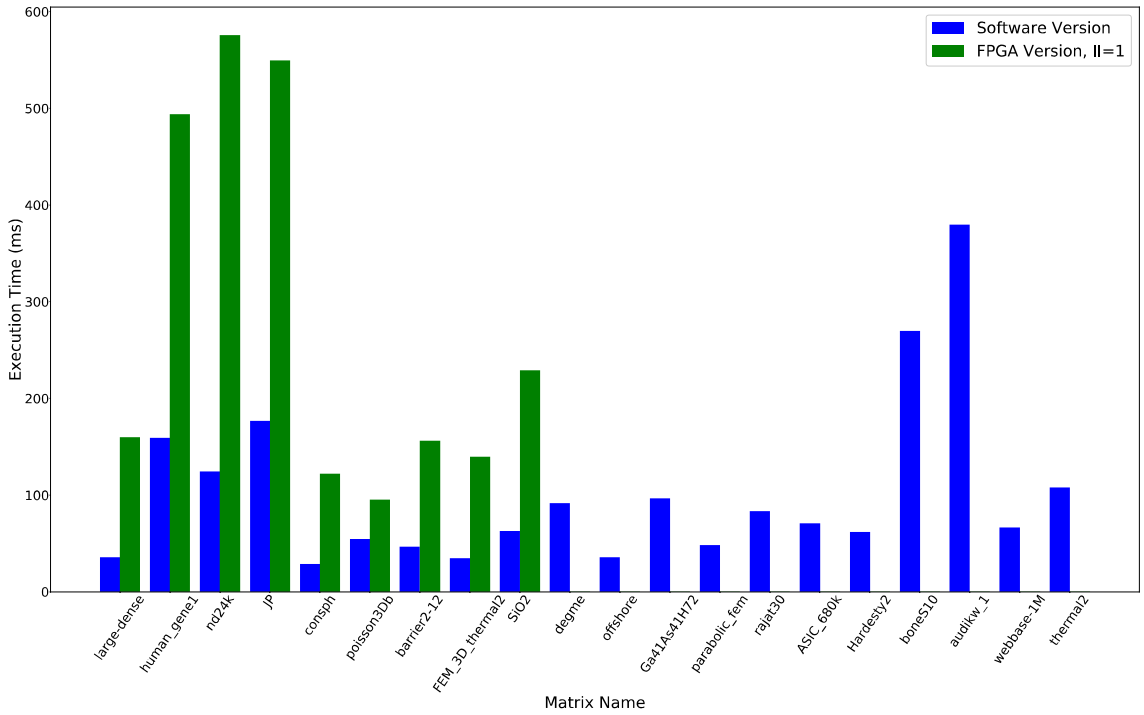
```
L5: for (i = 0; i < (row_size)/4; i++) {
    #pragma HLS PIPELINE

    DATA_TYPE_ap_uint32_t f2int;
    uintbuswidth_t  y_128;

    f2int.f = results_fifo.read();
    y_128.range(31,  0)   = f2int.apint;

    f2int.f = results_fifo.read();
    y_128.range(63,  32)  = f2int.apint;

    f2int.f = results_fifo.read();
    y_128.range(95,  64)  = f2int.apint;

    f2int.f = results_fifo.read();
    y_128.range(127,  96) = f2int.apint;

    y[i] = y_128;
  }
```

Listing 3.8: Αποθήκευση αποτελεσμάτων στο διάνυσμα αποτελέσματος y

3.4.3 Αξιολόγηση υλοποίησης

Συγκρίνοντας τους χρόνους εκτέλεσης της υλοποίησης μας με τους αντίστοιχους της εκτέλεσης του CSR-SpMV σε έναν πυρήνα ARM του PS, παρατηρούμε ότι δεν έχουμε ακόμη κάποια βελτίωση. Η βελτίωση σε σχέση με τις CSR υλοποιήσεις για FPGA που παρουσιάσαμε προηγουμένως είναι εμφανής, αλλά υπάρχει ακόμη μεγάλο περιθώριο βελτίωσης.

Παρατηρούμε ακόμη ότι δεν γίνεται να πάρουμε αποτελέσματα για όλους τους πίνακες στη συλλογή μας. Αυτό οφείλεται στο γεγονός ότι δεν μπορούμε να αποθηκεύσουμε τα αντιστοίχου μεγέθους διανύσματα x στη BRAM του FPGA, κάτι που θα επιλύσουμε στην τελευταία φάση της υλοποίησης μας.



Σχήμα 3.7: Σύγκριση χρόνων εκτέλεσης CSR-SpMV για 1 πυρήνα ARM και packedCSR-SpMV για FPGA

3.5 Vectorization

Σε μία πρώτη προσπάθεια βελτίωσης της απόδοσης, αποφασίσαμε να εκμεταλλευτούμε τους υπολογιστικούς πόρους του FPGA. Θέλουμε δηλαδή να εκτελούμε ταυτόχρονα περισσότερους του ενός υπολογισμούς.

Για να το επιτύχουμε αυτό, θέτουμε στο κομμάτι που πριν ορίσαμε ως **Στάδιο Υπολογισμών** τον αριθμό II ίσο με 2, 4 ή 8, επιδιώκοντας την ταυτόχρονη εκτέλεση αντίστοιχου πλήθους πολλαπλασιασμών μη-μηδενικών στοιχείων με στοιχεία του διανύσματος. Η χρήση του pragma UNROLL στους δύο εσωτερικούς βρόχους του κώδικα 3.7 «ξετυλίγει» πλήρως τους βρόχους, δεσμεύοντας αντίστοιχους πόρους που εκτελούν πράξεις.

a	b	0	0	c
0	d	e	f	g
0	h	0	0	0
0	0	i	j	0
0	k	0	l	m

(α′) Αραιός πίνακας

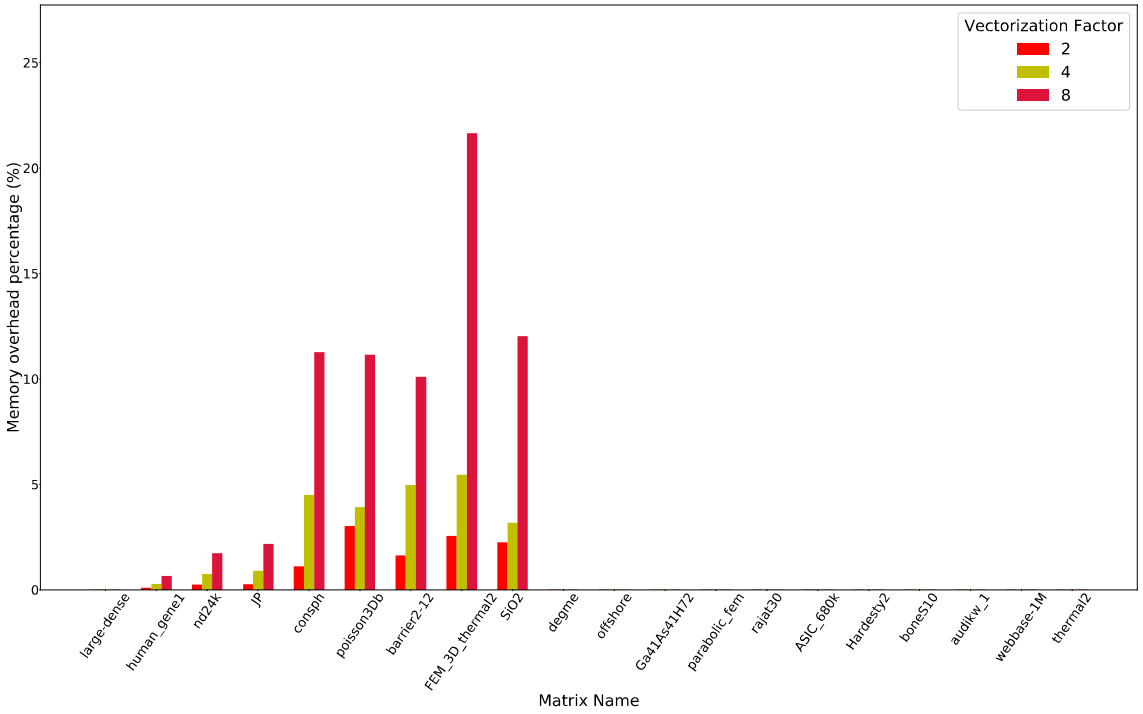
0	4	a	0
b	1	c	4
0	0	0	4
d	1	e	2
f	3	g	4
...			

...			
0	2	h	1
0	0	0	2
i	2	j	3
0	4	k	1
l	3	m	4
0	0	0	0

(β′) Αναπαράσταση packedCSR (II=2)

Σχήμα 3.8: Μετατροπή ενός αραιού πίνακα σε μορφή packedCSR για II = 2

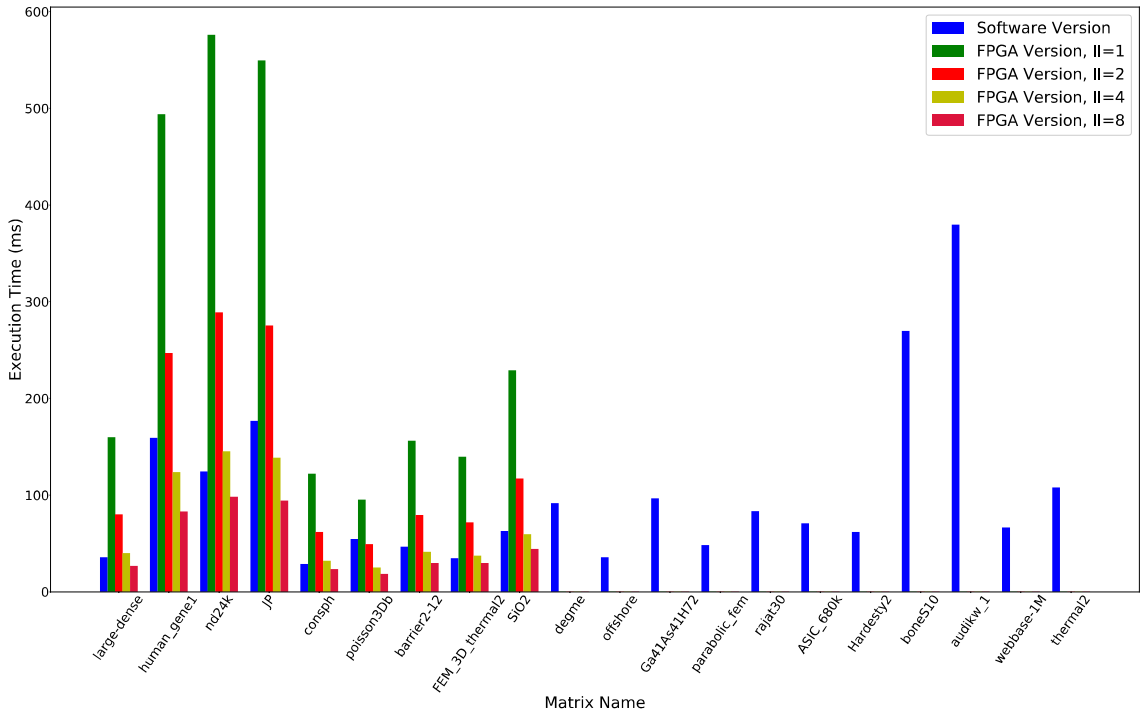
Για να συμβεί αυτό, απαιτείται κάθε γραμμή του πίνακα να διαθέτει πλήθος μη-μηδενικών στοιχείων που θα είναι ακέραιο πολλαπλάσιο του παράγοντα Vectorization (από εδώ και στο εξής θα αναφέρεται ως II). Αυτό το επιτυγχάνουμε, εφαρμόζοντας «γέμισμα» (padding) με μηδενικά στοιχεία ανά γραμμή, όπου αυτό είναι απαραίτητο. Αναμενόμενα, κάτι τέτοιο αυξάνει τη μνήμη που απαιτείται για τη δημιουργία της δομής, επομένως και του πλήθους των δεδομένων που θα χρειαστεί να μεταφερθούν από το PS στο PL.



Σχήμα 3.9: Επιρροή αύξησης II στη δεσμευμένη μνήμη για τον αραιό πίνακα (αναπαράσταση packedCSR)

Παρατηρούμε ότι όταν το II είναι ίσο με 8, χρειάζεται να δεσμεύσουμε αρκετή παραπάνω μνήμη για την αναπαράσταση του αραιού πίνακα, αύξηση που φτάνει μέχρι και 22% σε σχέση με το αρχικό μέγεθος του πίνακα. Για τα υπόλοιπα II, η αύξηση στην κατανάλωση μνήμης είναι μικρότερη και αποδεκτή, εφόσον υπάρχει βελτίωση στην απόδοση.

Στο Σχήμα 3.10, παρατηρούμε γραμμική μείωση στους χρόνους εκτέλεσης, καθώς αυξάνουμε το II από 1 εώς και 4. Η μείωση παύει να είναι γραμμική κατά τη μετάβαση από II=4 στο II=8, καθώς παρά το γεγονός ότι έχουμε ταυτόχρονη εκτέλεση διπλάσιων υπολογισμών, έχουμε αυξήσει κατά πολύ το πλήθος των δεδομένων που χρειάζεται να μεταφερθούν από το PS στο PL. Τα μηδενικά αυτά καταλαμβάνουν ακόμη χρόνο κατά την εκτέλεση των υπολογισμών. Βλέπουμε όμως ότι πλέον επιτυγχάνουμε μικρότερους χρόνους εκτέλεσης από τους αντίστοιχους του CSR-SpMV στο PS.



Σχήμα 3.10: Σύγκριση χρόνων εκτέλεσης για διαφορετικό παράγοντα Vectorization (II)

3.5.1 Αξιοποίηση πόρων

Στον παρακάτω πίνακα βλέπουμε τη σύγκριση της αξιοποίησης των διαθέσιμων πόρων του FPGA. Χρησιμοποιούμε το μέγιστο δυνατό τμήμα της BRAM που επιτρέπεται από το εργαλείο σύνθεσης του bitstream που θα εκτελεστεί στο FPGA. Επιπλέον αυξάνοντας το II, παρατηρούμε ότι μόνο όταν αυτό γίνει 8, αξιοποιούμε περισσότερες DSPs. Παρατηρούμε ακόμη αύξηση στην περιοχή που καταλαμβάνεται όσον αφορά τα Flip-Flops και τις μονάδες LUT. Παρόλα αυτά, μικρό μέρος των δυνατοτήτων του FPGA έχει αξιοποιηθεί μέχρι στιγμής.

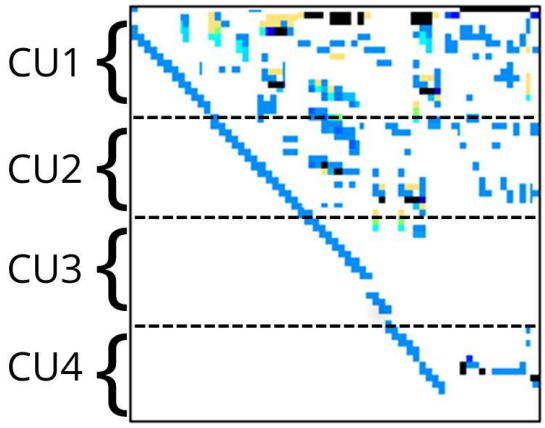
Version (II)	BRAM_18K	DSP48E	FF	LUT
1	910	5	4400	5729
2	910	5	4626	5835
4	910	5	5174	6040
8	910	7	6186	6600
Available	1824	2520	548160	274080

Συμπερασματικά, επιλέγουμε ως την καλύτερη υλοποίηση αυτήν που επιτυγχάνει τους μικρότερους χρόνους εκτέλεσης για όσους πίνακες μπορούν να ελεγχθούν μέχρι στιγμής. Στις επόμενες υλοποιήσεις θα έχουμε σαν βάση ότι ο παράγοντας Vectorization θα είναι **II = 8**.

3.6 1D-Blocking

Μέχρι στιγμής, ο υπολογισμός του διανύσματος αποτελέσματος y γινόταν σειριακά, ανά γραμμή του πίνακα. Μπορούμε όμως να μοιράσουμε τον υπολογιστικό φόρτο σε 2, 4 ή 8 Υπολογιστικές Μονάδες (Compute Units). Κάθε υπολογιστική μονάδα αναλαμβάνει ξεχωριστή περιοχή του πίνακα, και εκτελεί υπολογισμούς ανεξάρτητα από (και ταυτόχρονα με) τις υπόλοιπες.

Ο διαχωρισμός γίνεται έτσι ώστε, εάν ο πίνακας αποτελείται από n γραμμές και διαθέτουμε p υπολογιστικές μονάδες, καθεμία θα αναλάβει τον υπολογισμό αποτελεσμάτων για n / p γραμμές.

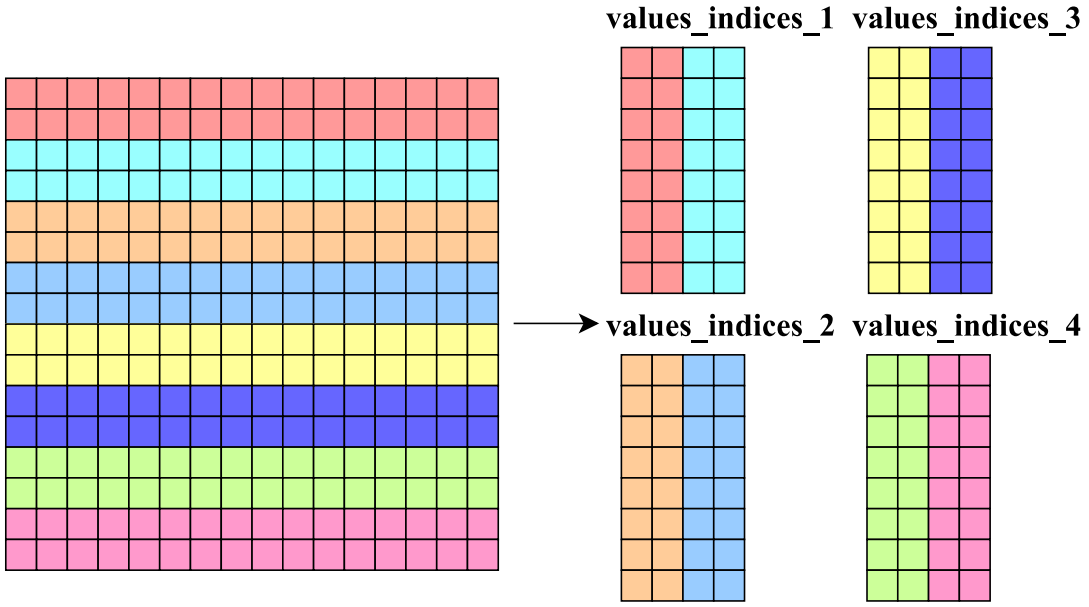


Σχήμα 3.11: Διαμοιρασμός φόρτου εργασίας σε 4 υπολογιστικές μονάδες

Για να συμβεί αυτό, δημιουργούμε πολλαπλά αντίγραφα του υπολογιστικού πυρήνα, όπως αυτός παρουσιάστηκε στην Ενότητα 3.4.2. Δημιουργούμε ξεχωριστή δομή ***values_indices*** για κάθε υπολογιστική μονάδα που θα χρησιμοποιηθεί. Καθεμία διαθέτει μόνο τα μη-μηδενικά στοιχεία των γραμμών που της ανατίθενται.

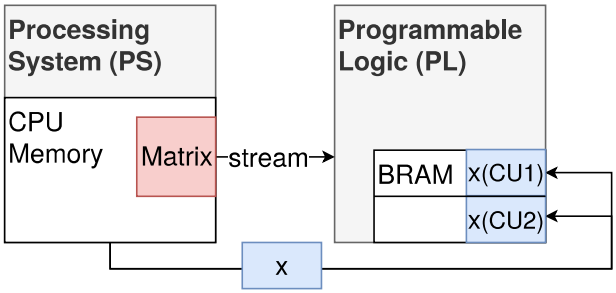
Μιας και το FPGA του εργαστηρίου διαθέτει 4 θύρες υψηλής απόδοσης, μπορούμε να δημιουργούμε μόνο μέχρι 4 τέτοιες δομές *values_indices*, που θα μεταφέρουν δεδομένα με τον αποδοτικότερο δυνατό τρόπο. Αυτό δεν μας δημιουργεί πρόβλημα όταν χρησιμοποιούμε 1, 2 ή 4 υπολογιστικές μονάδες. Όταν όμως θέλουμε να χρησιμοποιήσουμε 8 υπολογιστικές μονάδες, χρειάζεται να τροποποιήσουμε την αναπαράσταση του αραιού πίνακα.

Χρησιμοποιούμε 4 δομές *values_indices*, και καθεμία αναλαμβάνει τα δεδομένα δύο περιοχών του πίνακα. Για παράδειγμα, για τη δομή *values_indices_1* του Σχήματος 3.12, τα πρώτα 64 bits κάθε στοιχείου της περιλαμβάνουν δεδομένα που αφορούν την 1η υποπεριοχή του πίνακα, και τα υπόλοιπα 64 για τη 2η υποπεριοχή του πίνακα. Εάν είναι απαραίτητο, και η περιοχή των πρώτων 64 bits διαθέτει περισσότερη πληροφορία από την άλλη, γίνεται κατάλληλο γέμισμα με μηδενικά, έτσι ώστε κάθε στοιχείο της δομής να έχει πληροφορία. Κατάλληλη μετατροπή γίνεται και στον κώδικα που παρουσιάστηκε στο Listing 3.4.



Σχήμα 3.12: Αναπαράσταση αραιού πίνακα για 8 υπολογιστικές μονάδες

Προκειμένου να εξασφαλιστεί η ανεξάρτητη λειτουργία των Υπολογιστικών Μονάδων, απαιτείται καθεμία να διαθέτει ξεχωριστό αντίγραφο του διανύσματος x, καθώς οι Υπολογιστικές Μονάδες δεν έχουν τη δυνατότητα να μοιράζονται δεδομένα μεταξύ τους.

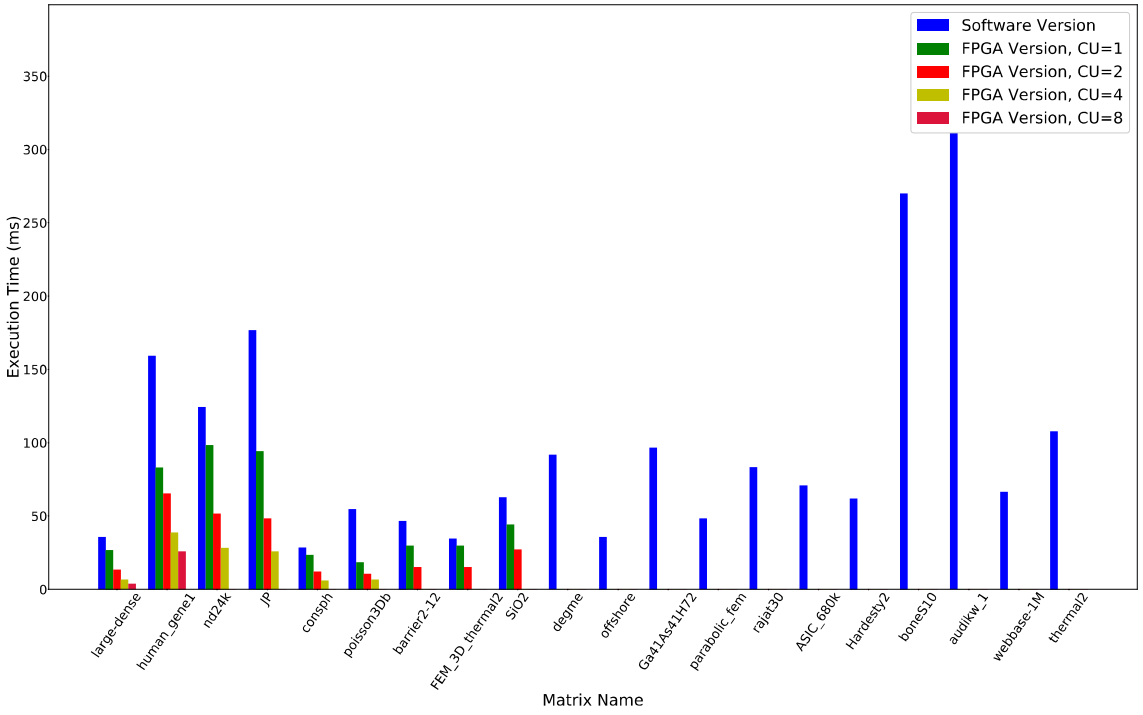


Σχήμα 3.13: Αποθήκευση πολλαπλών αντιγράφων του διανύσματος x στη BRAM (CU = 2)

Η ύπαρξη περισσότερων του ενός αντιγράφων του διανύσματος x στην BRAM του FPGA περιορίζει περαιτέρω το μέγιστο χώρο που μπορούμε να διαθέσουμε για καθένα από αυτά. Πιο συγκεκριμένα, το μέγιστο μέγεθος διανύσματος που μπορεί να αποθηκευτεί στη BRAM διαμορφώνεται, καθώς αυξάνουμε τις Υπολογιστικές Μονάδες, ως εξής :

Υπολογιστικές Μονάδες (CU)	Μέγιστο μέγεθος διανύσματος
1	491520
2	225280
4	112640
8	49152

Στο παρακάτω διάγραμμα βλέπουμε τη σύγκριση των χρόνων εκτέλεσης καθώς αυξάνουμε τον αριθμό των Υπολογιστικών Μονάδων, όταν ο παράγοντας Vectorization είναι ίσος με 8. Παρατηρούμε ότι όντως η χρήση πολλαπλών Υπολογιστικών Μονάδων πετυχαίνει περαιτέρω μείωση του χρόνου εκτέλεσης. Από την άλλη, περιορίζεται αρκετά το μέγιστο μέγεθος προβλήματος που μπορεί να διαχειριστεί η υλοποίηση μας. Συγκεκριμένα, όταν οι Υπολογιστικές Μονάδες είναι 8, μπορούσαμε να πάρουμε αποτελέσματα για μόνο 2 από τους πίνακες της συλλογής μας.



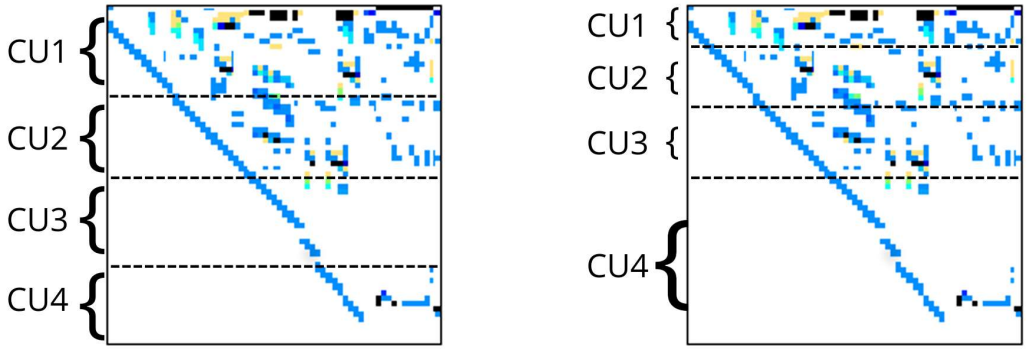
Σχήμα 3.14: Σύγκριση χρόνων εκτέλεσης για διαφορετικό πλήθος Υπολογιστικών Μονάδων (CU)

3.6.1 Δίκαιος διαμοιρασμός φόρτου εργασίας

Συνεχίζοντας την προσπάθεια βελτίωσης της απόδοσης, παρατηρούμε ότι ο τρόπος με τον οποίο μοιράζεται η εργασία ανάμεσα στις υπολογιστικές μονάδες δεν είναι ο βέλτιστος δυνατός. Αυτό συμβαίνει καθώς μοιράζουμε την εργασία με βάση το πλήθος των γραμμών, και χωρίς να λαμβάνουμε υπόψιν το πώς τα μη-μηδενικά στοιχεία είναι εμφανίζονται μέσα στον πίνακα.

Για παράδειγμα, μπορεί να υπάρχει μεγάλη συγκέντρωση μη-μηδενικών στοιχείων στην άνω υποπεριοχή του πίνακα, όπως φαίνεται στο Σχήμα 3.15. Εάν κάνουμε το διαμοιρασμό όπως φαίνεται στην εικόνα (α'), τότε οι πρώτες δύο Υπολογιστικές Μονάδες θα έχουν να φέρουν εις πέρας μεγαλύτερο όγκο υπολογισμών.

Μιας και ο συνολικός χρόνος εκτέλεσης εξαρτάται από την πιο αργή Υπολογιστική Μονάδα, χρειάζεται να βρούμε έναν άλλο τρόπο με τον οποίο θα αναθέτουμε τους υπολογισμούς ανά υπολογιστική μονάδα.



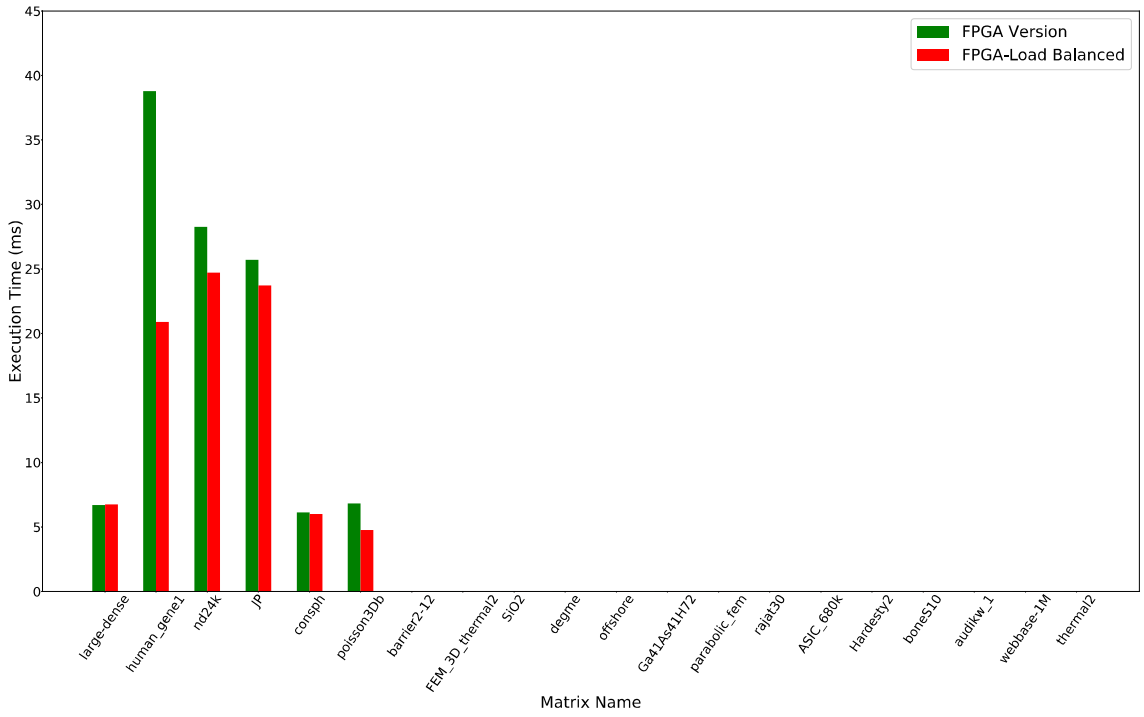
(α′) Διαμοιρασμός φόρτου εργασίας βάσει πλήθους γραμμών

(β′) Διαμοιρασμός φόρτου εργασίας βάσει μη-μηδενικών στοιχείων

Σχήμα 3.15: Στρατηγικές διαμοιρασμού φόρτου εργασίας (CU=4)

Αποφασίσαμε επομένως, κάθε Υπολογιστική Μονάδα να μην αναλαμβάνει ίσο πλήθος γραμμών, αλλά ίσο (προσεγγιστικά) πλήθος μη-μηδενικών στοιχείων. Προσεγγιστικά, γιατί φροντίζουμε ο διαμοιρασμός να γίνεται έτσι ώστε να μην υπάρχουν μη-μηδενικά στοιχεία μοιρασμένα μεταξύ δύο διαδοχικών Υπολογιστικών Μονάδων, καθώς κάτι τέτοιο θα απαιτούσε περισσότερους ελέγχους και επιπλέον καθυστέρηση στο χρόνο εκτέλεσης του πολλαπλασιασμού.

Με τη νέα στρατηγική διαμοιρασμού, όπως βλέπουμε στο παρακάτω διάγραμμα, πετυχαίνουμε περαιτέρω μείωση στους χρόνους εκτέλεσης.



Σχήμα 3.16: Σύγκριση στρατηγικών διαμοιρασμού εργασίας (II = 8, CU = 4)

3.6.2 Αξιοποιήση πόρων

Στον παρακάτω πίνακα βλέπουμε τη σύγκριση της αξιοποίησης των διαθέσιμων πόρων του FPGA, όταν ο παράγοντας Vectorization είναι ίσος με 8. Καθώς διπλασιάζουμε τον αριθμό των Υπολογιστικών Μονάδων, αντίστοιχα δεσμεύουμε περισσότερους διαθέσιμους πόρους σε DSPs, Flip-Flops και LUTs. Τελικά, το μέγιστο των πόρων που δεσμεύουμε για την υλοποίησή μας είναι 2% DSPs, 7% Flip-Flops και 14% LUTs.

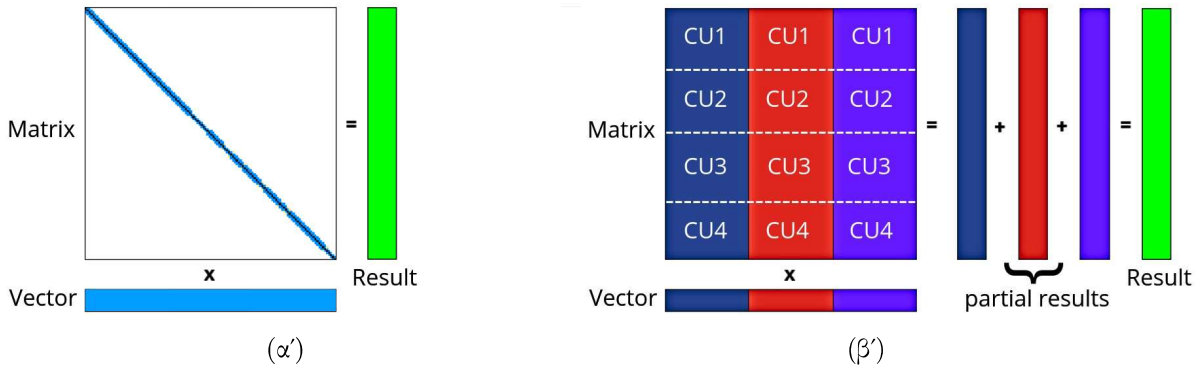
Version (CU)	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
1	910	7	6186	6600
2	872	14	11437	11969
4	944	28	21941	22704
8	936	56	39119	39251
Available	1824	2520	548160	274080

3.7 2D-Blocking

Μετά τις τροποποιήσεις στον κώδικα του υπολογιστικού πυρήνα, χρειάζεται πλέον να βρούμε έναν τρόπο η υλοποίησή μας να μπορεί να δέχεται σαν είσοδο πίνακες οποιουδήποτε μεγέθους. Μέχρι στιγμής, το μέγιστο μέγεθος πίνακα που μπορούσε να δεχτεί η υλοποίηση με τις 8 Υπολογιστικές Μονάδες είναι 49152, ενώ σε πραγματικές συνθήκες οι πίνακες που προκύπτουν έχουν πολύ μεγαλύτερο μέγεθος. Για το λόγο αυτό εισάγουμε την έννοια του **2D-Blocking**.

Ο πίνακας σπάει σε μπλοκ, ανάλογα με τον αριθμό στηλών που διαθέτει. Δημιουργούνται ξεχωριστές δομές *values_indices* για κάθε μπλοκ, και οι υπολογισμοί των επιμέρους αποτελεσμάτων εκτελούνται σειριακά για κάθε μπλοκ. Η διάσπαση σε μπλοκ γίνεται προκειμένου να μην απαιτείται ολόκληρο το διάνυσμα x να είναι αποθηκευμένο στη BRAM του FPGA, αλλά μόνο το μέρος αυτού που αντιστοιχεί στο μπλοκ για το οποίο εκτελούμε υπολογισμούς κάθε στιγμή.

Ο πίνακας της παρακάτω εικόνας θα διασπαστεί σε 3 μπλοκ, καθένα εκ των οποίων το διαχειριζόμαστε σαν ανεξάρτητο αραιό πίνακα, ο οποίος θα πολλαπλασιαστεί με το αντίστοιχο κομμάτι του διανύσματος x. Θα υπολογιστούν κατά σειρά τα αποτελέσματα της μπλε περιοχής, κατόπιν της κόκκινης περιοχής και τελικά της μωβ περιοχής. Στη συνέχεια, αυτά τα επιμέρους αποτελέσματα θα αθροιστούν στο PS και θα μας δώσουν το τελικό αποτέλεσμα.



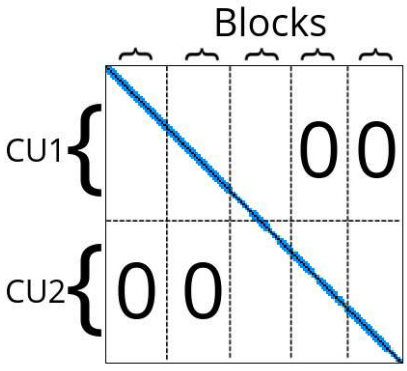
Σχήμα 3.17: Σχηματική αναπαράσταση διάσπασης σε μπλοκ αραιού πίνακα μεγάλου μεγέθους

3.7.1 Μειονεκτήματα 2D-Blocking

Η διάσπαση σε μπλοκ, αν και μας επιτρέπει να δοκιμάζουμε την υλοποίηση μας σε μεγάλους πίνακες, φέρνει αρκετά μειονεκτήματα στην υλοποίηση μας.

Πρώτον, όπως παρουσιάσαμε προηγουμένως, κάθε γραμμή του πίνακα πρέπει να διαθέτει πλήθος μη-μηδενικών στοιχείων που θα είναι πολλαπλάσιο του παράγοντα Vectorization(II). Ακόμη και αν κάποια γραμμή δεν διαθέτει κάποιο μη-μηδενικό στοιχείο, θα χρειαστεί να αποκτήσει II στοιχεία, ώστε οι υπολογισμοί να εκτελεστούν σωστά. Αυτό οδηγεί, σε πίνακες μεγάλου μεγέθους που θα σπάσουν σε πολλά μπλοκ, στη δημιουργία πολλών περιοχών που, προκειμένου να αποκτήσουν μη-μηδενικά στοιχεία, θα χρειαστεί να τις γεμίσουμε με μηδενικά.

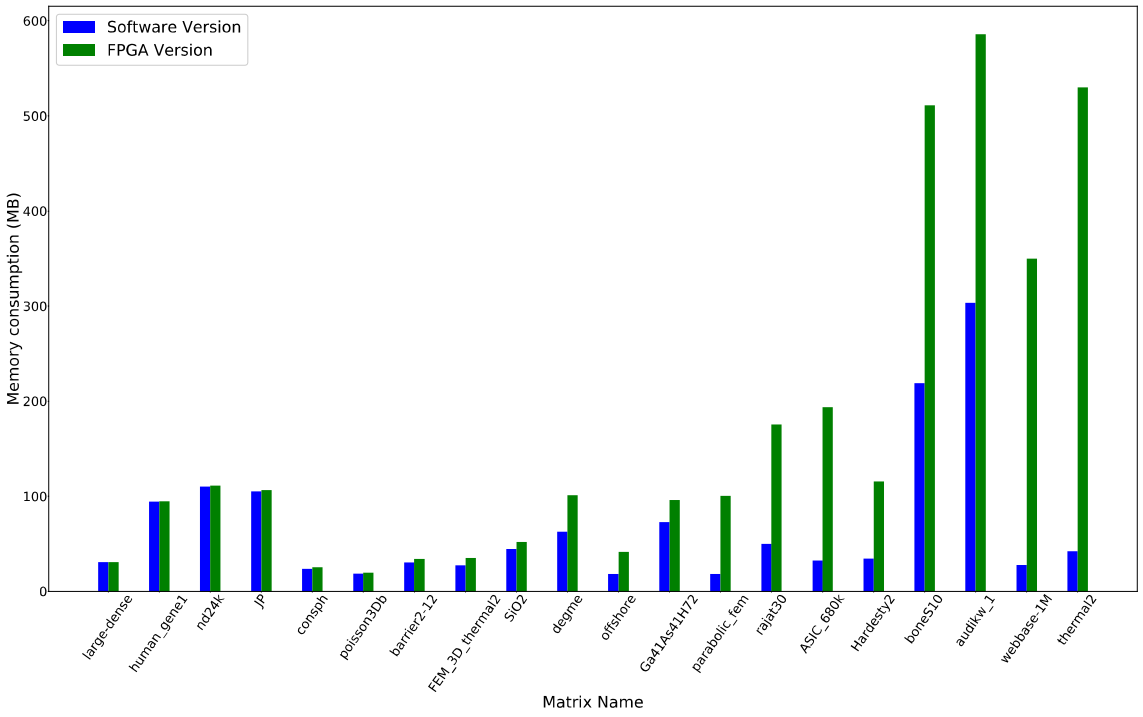
Στον παρακάτω διαγώνιο πίνακα, η διάσπαση σε πολλα μπλοκ δημιούργησε πολλές περιοχές που έχουν μόνο μηδενικά στοιχεία, σημειωμένες με 0 στην εικόνα. Οι συγκεκριμένες περιοχές, αν και θα μπορούσε να αποφευχθεί, καταλαμβάνουν κανονικά χρόνο από τους υπολογισμούς, παρά το γεγονός ότι το αποτέλεσμα του πολλαπλασιασμού τους με το διάνυσμα θα είναι μηδενικό.



Σχήμα 3.18: Μηδενικές περιοχές λόγω διάσπασης σε πολλά μπλοκ

Επιπλέον, η αύξηση των Υπολογιστικών Μονάδων, όπως είδαμε προηγουμένως, απαιτεί όλο και μικρότερο μέγεθος διανύσματος που θα αποθηκεύεται κάθε φορά στη BRAM. Αυτό από την πλευρά του σημαίνει ότι για πίνακες μεγάλου μεγέθους, η αύξηση των Υπολογιστικών

Μονάδων, οδηγεί στη διάσπασή τους σε όλο και περισσότερα μπλοκ. Μάλιστα, στο παρακάτω διάγραμμα βλέπουμε ότι η κατανάλωση μνήμης για την αναπαράσταση του αραιού πίνακα είναι ιδιαίτερα υψηλή, όσο μεγαλύτερος γίνεται ο πίνακας, λόγω της ύπαρξης πολλών αποκλειστικά-μηδενικών περιοχών.

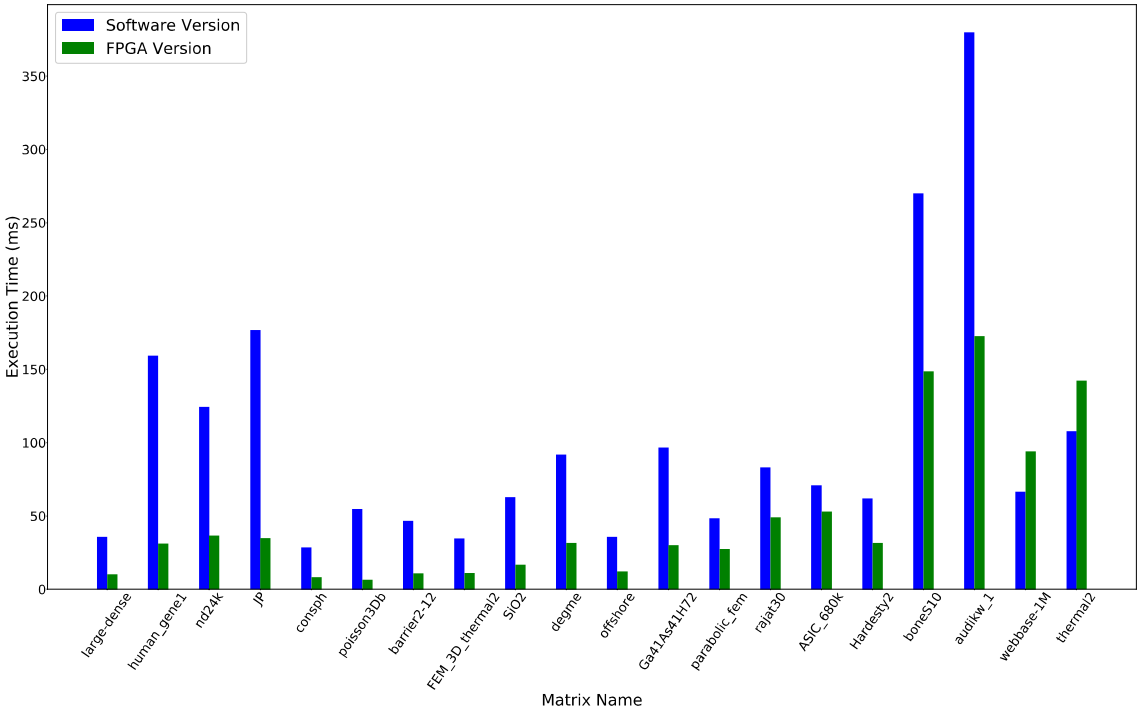


Σχήμα 3.19: Σύγκριση κατανάλωσης μνήμης αναπαραστάσεων CSR και packedCSR με 2D-Blocking (II = 4 , CU = 4)

Ειδικά για τους δύο μεγαλύτερους πίνακες στη συλλογή μας, που είναι διαγώνιοι πίνακες, επομένως και με πολλές μηδενικές περιοχές, παρατηρούμε ότι η κατανάλωση μνήμης είναι τάξεις μεγέθους μεγαλύτερη. Το 2D Blocking είναι όμως απαραίτητο, αφού πλέον μπορούμε να δοκιμάσουμε την υλοποίηση σε πίνακες οποιουδήποτε μεγέθους

3.8 Τελική αξιολόγηση

Παρακάτω παρουσιάζουμε την τελική σύγκριση της υλοποίησης μας με την υλοποίηση CSR για 1 πυρήνα ARM του PS. Ο λόγος που επιλέξαμε τη συγκεκριμένη υλοποίηση να παρουσιάσουμε είναι επειδή για αυτήν καταφέραμε να πάρουμε αποτελέσματα για όλους τους πίνακες της συλλογής. Για πιο «βαριές» υλοποιήσεις, όπως τη (II=4,CU=8), (II=8,CU=4) ή (II=8,CU=8), οι οποίες έχουν μεγαλύτερες απαιτήσεις σε κατανάλωση μνήμης, το FPGA του εργαστηρίου δεν είχε την απαιτούμενη μνήμη διαθέσιμη. Η μέγιστη μνήμη, που δεσμεύουμε στο PS για την αναπαράσταση του αραιού πίνακα με τη βοήθεια της εντολής ***sds_alloc_non_cacheable***, στο συγκεκριμένο FPGA είναι 1 Gigabyte, το οποίο για ορισμένες περιπτώσεις πινάκων δεν επαρκούσε.



Σχήμα 3.20: Σύγκριση χρόνων εκτέλεσης CSR-SpMV για 1 πυρήνα ARM και packedCSR-SpMV για FPGA (II = 4, CU=4)

Στο διάγραμμα αυτό βλέπουμε ότι για τις περισσότερες περιπτώσεις πινάκων επιτυγχάνουμε όντως επιτάχυνση του πολλαπλασιασμού. Για τους δύο μεγαλύτερους πίνακες ο χρόνος εκτέλεσης ξεπερνά τον αντίστοιχο του PS, κάτι που οφείλεται στη διάσπαση του πίνακα σε πολλά μπλοκ και την ύπαρξη πολλών μηδενικών περιοχών μέσα σε αυτά, όπως αναλύθηκε προηγουμένως.

3.9 Ενεργειακή αξιολόγηση

Στόχος της παρούσας διπλωματικής δεν ήταν μόνο η επιτάχυνση του αλγορίθμου SpMV σε FPGA, αλλά και η αξιοποίηση των ενεργειακών οφελών της συγκεκριμένης πλατφόρμας. Για το σκοπό αυτό, θα συγκρίνουμε την ενεργειακή αποδοτικότητα της υλοποίησης μας με υλοποιήσεις CSR-SpMV για συστήματα με CPU και GPU.

Για τη σύγκριση με CPU, χρησιμοποιήθηκε η υλοποίηση CSR-SpMV της βιβλιοθήκης **Intel MKL** [20] στον επεξεργαστή **Intel Xeon E5-2630V4** (Broadwell), ο οποίος λειτουργεί σε συχνότητα 2.20 GHz και διαθέτει 10 πυρήνες, 25 MB κρυφής μνήμης και 256 GB μνήμης.

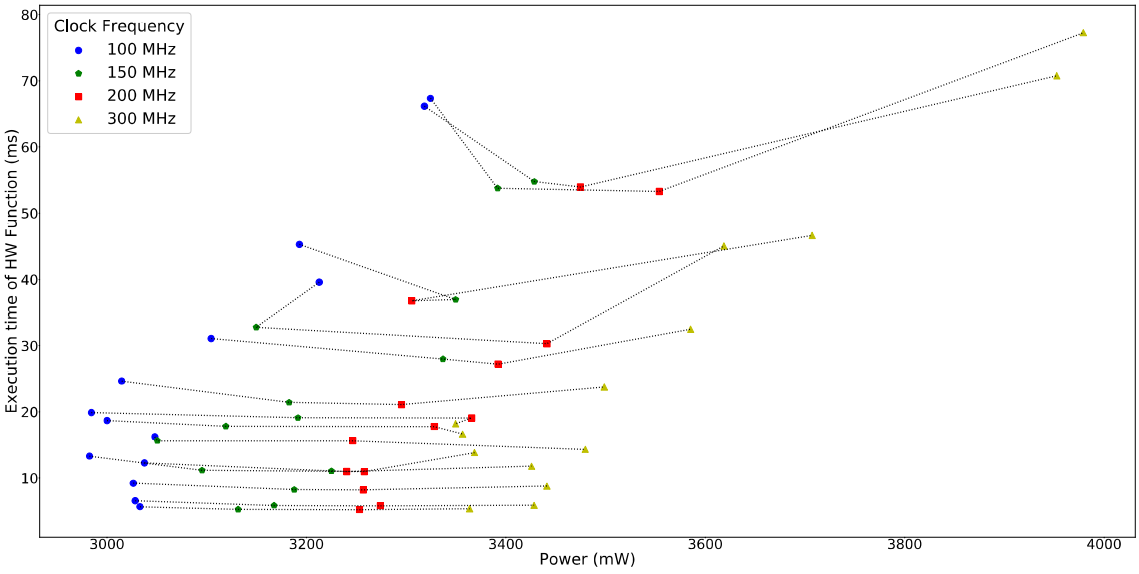
Για τη σύγκριση με GPU, χρησιμοποιήθηκε η υλοποίηση CSR-SpMV της βιβλιοθήκης **cuSPARSE** [21] στη GPU **NVIDIA Tesla K40**, η οποία διαθέτει 12 GB DDR5 μνήμης.

Χρησιμοποιούμε μετρητές ισχύος που η Intel και η Xilinx προσφέρουν για τη CPU και το FPGA αντίστοιχα. Για τη GPU, γνωρίζουμε την ισχύ από τα τεχνικά χαρακτηριστικά της, η οποία συνδυασμένη με τους χρόνους εκτέλεσης του CSR-SpMV, θα μας δώσει τις ζητούμενες μετρήσεις ενέργειας.

Η σύγκριση με τη CPU και τη GPU θα γίνει τόσο σε επίπεδο χρόνων εκτέλεσης όσο και σε επίπεδο κατανάλωσης ενέργειας.

3.9.1 Συχνότητα λειτουργίας FPGA

Το FPGA του εργαστηρίου δίνει τη δυνατότητα να θέτουμε τη συχνότητα λειτουργίας του, σε συχνότητες που κυμαίνονται από 75 έως 600 MHz. Μετά από δοκιμές, βρήκαμε ότι οι συχνότητες που ικανοποιούν τις απαιτήσεις χρονισμού της υλοποίησης μας, για τις διάφορες παραμέτρους που θέσαμε κατά τη διάρκειά της (II και CU), είναι στο εύρος συχνοτήτων 100-300 MHz.



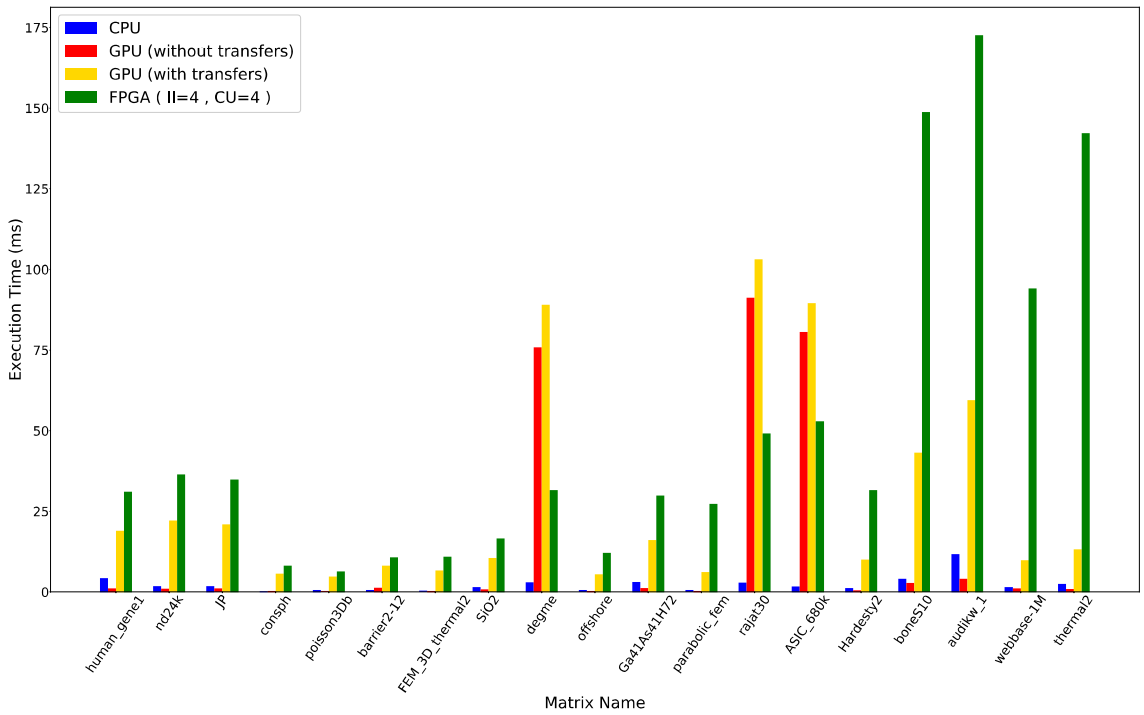
Σχήμα 3.21: Σχέση χρόνου-ισχύος για διάφορες συχνότητες λειτουργίας του FPGA

Στο παραπάνω διάγραμμα χρόνου-ισχύος, με διακεκομμένες γραμμές συνδέονται μετρήσεις που πήραμε για κάθε αραιό πίνακα σε διάφορες συχνότητες. Παρατηρούμε ότι καθώς αυξάνουμε τη συχνότητα λειτουργίας, μόνο στη μετάβαση από τα 100 στα 150 MHz επιτυγχάνεται μείωση των χρόνων εκτέλεσης, ιδίως για τους μεγαλύτερους πίνακες (άνω δεξιά περιοχή του διαγράμματος). Η αύξηση περαιτέρω της συχνότητας λειτουργίας θα λέγαμε ότι περισσότερο αυξάνει την κατανάλωση ισχύος παρά μειώνει αισθητά τους χρόνους εκτέλεσης. Μάλιστα η συχνότητα των 300 MHz απαιτούσε πόρους από τη BRAM, μειώνοντας το διαθέσιμο χώρο για αποθήκευση του διανύσματος x. Αυτό με τη σειρά του απαιτούσε τη διάσπαση σε περισσότερα μπλοκ ενός πίνακα μεγάλου μεγέθους. Για το λόγο αυτό εμφανίζεται το φαινόμενο της αύξησης του χρόνου εκτέλεσης για τους μεγάλους πίνακες (άνω δεξιά περιοχή).

Καταλήξαμε επομένως στο συμπέρασμα ότι η βέλτιστη συχνότητα λειτουργίας, λαμβάνοντας ταυτόχρονα υπόψιν την απόδοση όσο και την κατανάλωση ισχύος είναι τα **150 MHz**.

3.9.2 Σύγκριση με CPU και GPU

Στο παρακάτω διάγραμμα βλέπουμε τη σύγκριση στους χρόνους εκτέλεσης με τη CPU και τη GPU. Η υλοποίηση που χρησιμοποιήσαμε για το FPGA έχει παράγοντα Vectorization 4 και 4 Υπολογιστικές Μονάδες. Στο χρόνο εκτέλεσης του FPGA συμπεριλαμβάνουμε τις μεταφορές δεδομένων από το PS στο PL και αντίστροφα. Για το λόγο αυτό, στις μετρήσεις της GPU παραθέτουμε χρόνους εκτέλεσης σε αυτήν τόσο χωρίς (καθαρός χρόνος υπολογισμών) όσο και με τις μεταφορές δεδομένων μέσω του διαύλου PCIe.



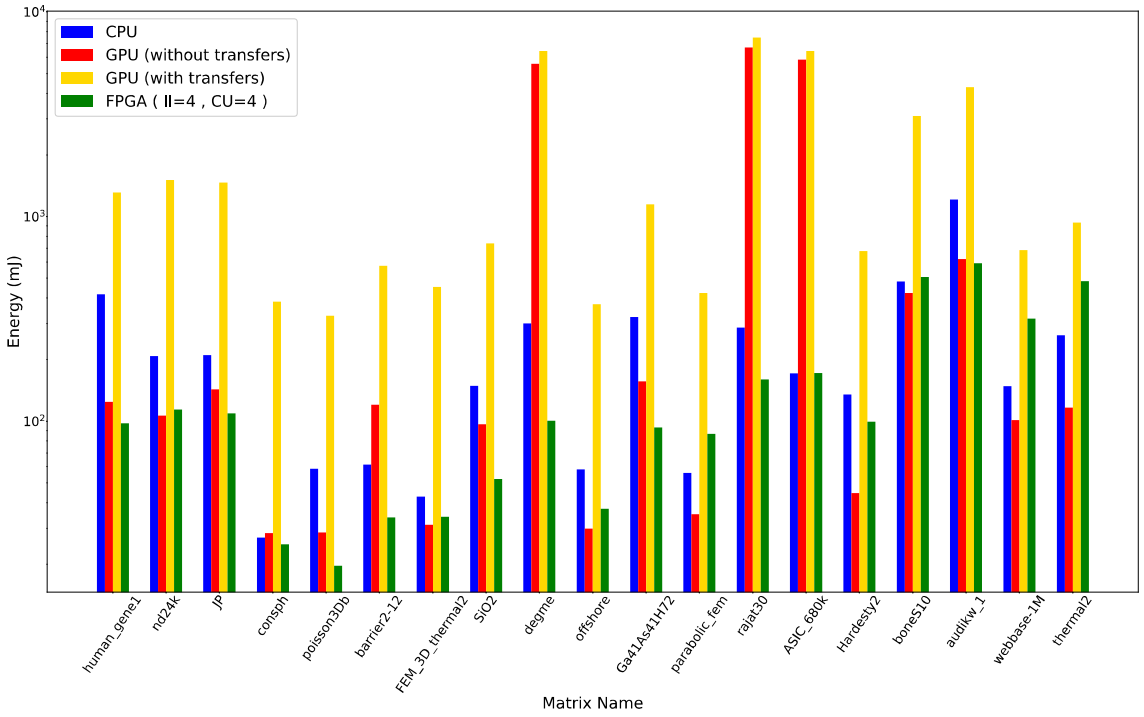
Σχήμα 3.22: Σύγκριση χρόνων εκτέλεσης SpMV για διαφορετικές αρχιτεκτονικές

Παρατηρούμε ότι η υλοποίησή μας είναι από 7 έως και 62 φορές πιο αργή σε σχέση με τη CPU, κατά μέσο όρο 25 φορές. Το FPGA έχει 2.5 φορές λιγότερες Υπολογιστικές Μονάδες (4 έναντι 10 πυρήνων της CPU) και λειτουργεί σε συχνότητα 15 φορές μικρότερη. Επομένως, η διαφορά στην απόδοση μπορεί να δικαιολογηθεί.

Η GPU εμφανίζει παρόμοια συμπεριφορά με τη CPU εκτός από τους πίνακες degme, rajat30 και ASIC_680K, οι οποίοι λόγω της δομής τους, προκαλούν αστάθεια στην απόδοση της GPU. Για αυτούς τους τρεις πίνακες, το FPGA πέτυχε σημαντικά καλύτερους χρόνους. Αν συμπεριλάβουμε και τους χρόνους μεταφοράς δεδομένων για τη GPU, το FPGA είναι ταχύτερο από τη GPU για τους πίνακες μικρού μεγέθους. Για τους 5 μεγαλύτερους πίνακες της συλλογής μας, το FPGA εμφανίζει χειρότερη συμπεριφορά.

Από τη σκοπιά κατανάλωσης ενέργειας τώρα, η CPU, παρά την εμφανή υπεροχή της στο χρόνο εκτέλεσης, καταναλώνει περισσότερη ενέργεια, μέχρι και 5 φορές παραπάνω σε σχέση με το FPGA. Το FPGA, λόγω των υψηλών χρόνων εκτέλεσης για τους δύο μεγαλύτερους πίνακες της συλλογής, καταναλώνει περισσότερη ενέργεια σε σχέση με τη CPU σε αυτές τις περιπτώσεις.

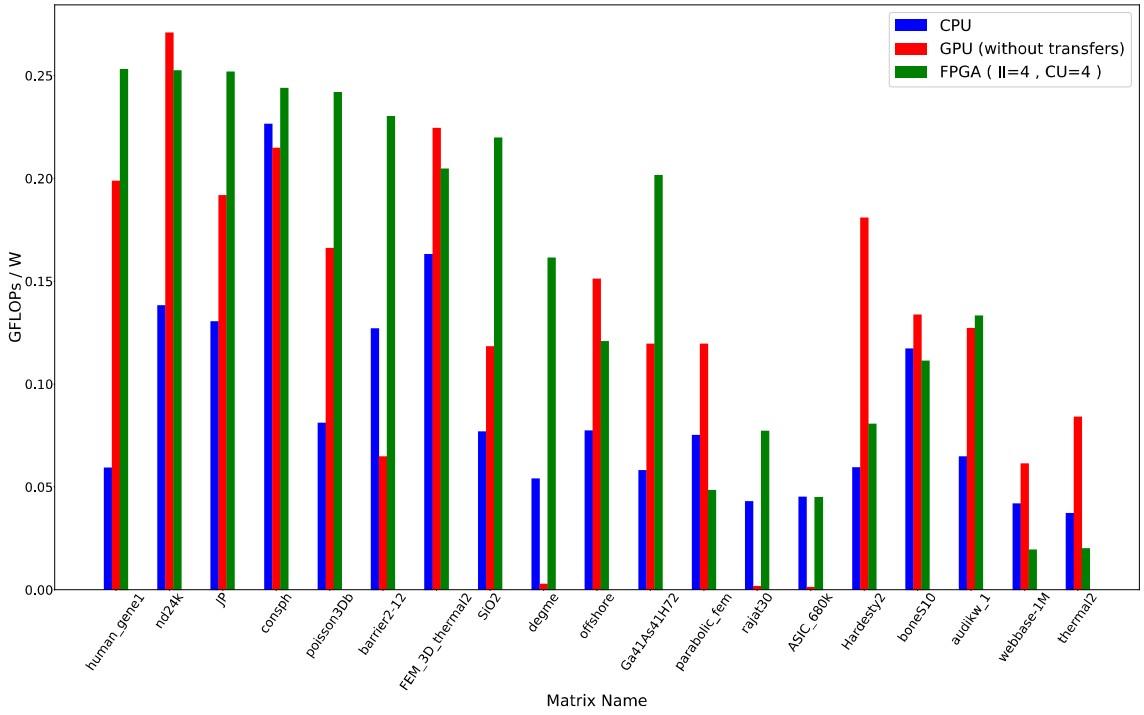
Η GPU, λαμβάνοντας υπόψιν μόνο το υπολογιστικό κομμάτι, εμφανίζει κατανάλωση περίπου παρόμοια με το FPGA, με την εξαίρεση των 3 πινάκων όπου εμφανίζει ακανόνιστη συμπεριφορά. Εάν όμως συμπεριλάβουμε τη φάση μεταφοράς δεδομένων, τότε η GPU γίνεται η λιγότερο ενεργειακά αποδοτική επιλογή ανάμεσα στις εξεταζόμενες αρχιτεκτονικές.



Σχήμα 3.23: Σύγκριση κατανάλωσης ενέργειας SpMV για διαφορετικές αρχιτεκτονικές

Μία ακόμη μετρική που μπορούμε να χρησιμοποιήσουμε για να ποσοτικοποιήσουμε την ενεργειακή απόδοση κάθε αρχιτεκτονικής είναι τα FLOPs per Watt [22]. Στο SpMV, για κάθε

μη-μηδενικό στοιχείο του πίνακα, εκτελούνται δύο πράξεις κινητής υποδιαστολής· πολλαπλασιασμός με το αντίστοιχο στοιχείο του διανύσματος x, και ενημέρωση του αποτελέσματος στην αντίστοιχη θέση του διανύσματος αποτελέσματος y. Επομένως, ο αριθμός των FLOPSs υπολογίζεται ως το πηλίκο του αριθμού των μη-μηδενικών στοιχείων του πίνακα, πολλαπλασιασμένο επί δύο, προς το χρόνο εκτέλεσης του SpMV για τον πίνακα αυτόν.



Σχήμα 3.24: Σύγκριση ενεργειακής απόδοσης SpMV για διαφορετικές αρχιτεκτονικές

Στο παραπάνω διάγραμμα παρατηρούμε ότι το FPGA, παρά τους χειρότερους χρόνους εκτέλεσης σε σχέση με CPU και GPU, επιτυγχάνει ικανοποιητικά αποτελέσματα, λαμβάνοντας υπόψιν την επίδοση και την κατανάλωση ενέργειας που απαιτείται για την επίτευξή της, ταυτόχρονα.

Κεφάλαιο 4

Μελλοντικές επεκτάσεις

4.1 Εισαγωγή

Ένα σημαντικό μειονέκτημα της υλοποίησης μας, όπως παρουσιάστηκε προηγουμένως, είναι το εκτεταμένο γέμισμα με μηδενικά, προκειμένου κάθε γραμμή του πίνακα να διαθέτει πλήθος μη-μηδενικών στοιχείων που θα είναι πολλαπλάσιο του παράγοντα Vectorization. Ο συγκεκριμένος περιορισμός όμως αυξάνει κατά πολύ τη μνήμη που απαιτείται για την αναπαράσταση του αραιού πίνακα σε μορφή packedCSR, αυξάνοντας το χρόνο επικοινωνίας μεταξύ PS και PL, και επιβαρύνοντας το PL με υπολογισμούς που έχουν μηδενικό αποτέλεσμα.

4.2 Συμπίεση μηδενικών στοιχείων

Σε μια πρώτη προσπάθεια μείωσης του όγκου των μηδενικών στοιχείων, αποφασίσαμε να εφαρμόσουμε κωδικοποίηση μήκους σειράς (run-length encoding) στα μηδενικά στοιχεία που χρησιμοποιούμε για το «γέμισμα» (padding) των γραμμών. Με τον τρόπο αυτό εξοικονομούμε μνήμη για την αναπαράσταση του αραιού πίνακα, ιδιαίτερα όταν ο παράγοντας Vectorization είναι ίσος με 4 ή 8, όπως φαίνεται στο Σχήμα 4.1 (γ′) (τα ζεύγη τιμών που είναι σε τονισμένη και πλάγια γραφή).

a	b	0	0	c
0	d	e	f	g
0	h	0	0	0
0	0	i	j	0
0	k	0	0	0

(α′) Αραιός πίνακας

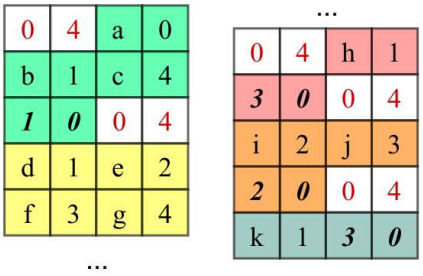
0	4	a	0
b	1	c	4
0	0	0	4
d	1	e	2
f	3	g	4
0	4	h	1
0	0	0	0

...

...

0	0	0	4
i	2	j	3
0	0	0	0
0	4	k	1
0	0	0	0
0	0	0	0

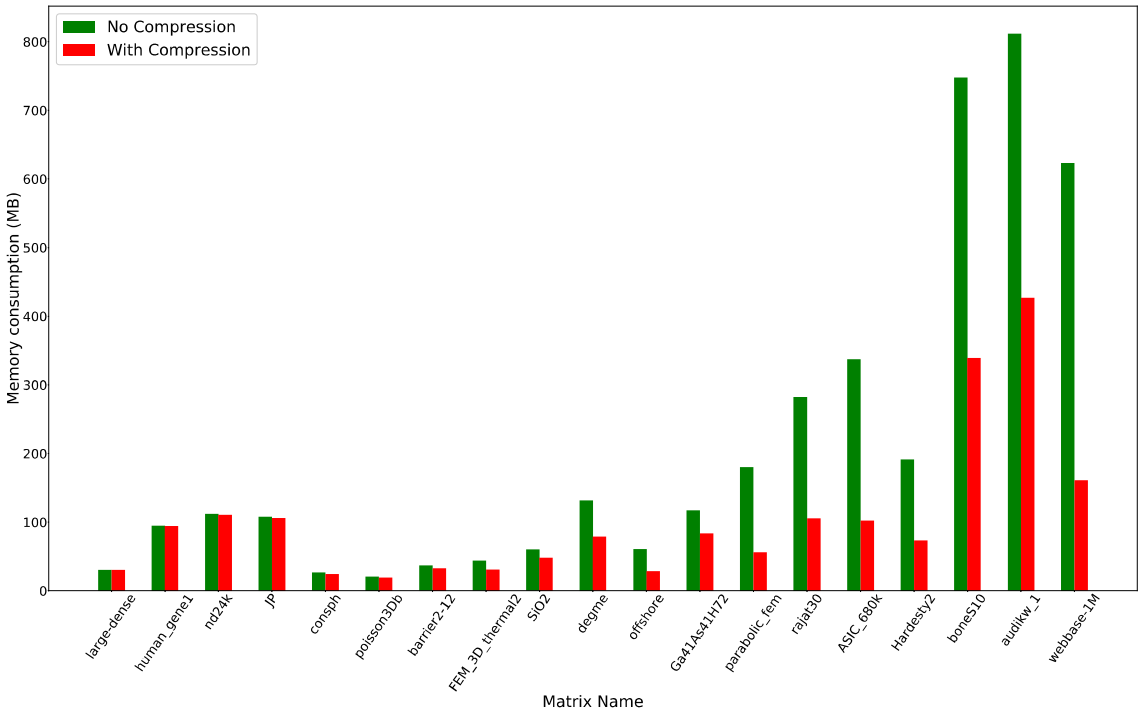
(β′) Αναπαράσταση packedCSR



(γ′) Αναπαράσταση packedCSR με συμπίεση στα μηδενικά στοιχεία

Σχήμα 4.1: Σύγκριση αναπαραστάσεων χωρίς και με συμπίεση μηδενικών στοιχείων (II=4)

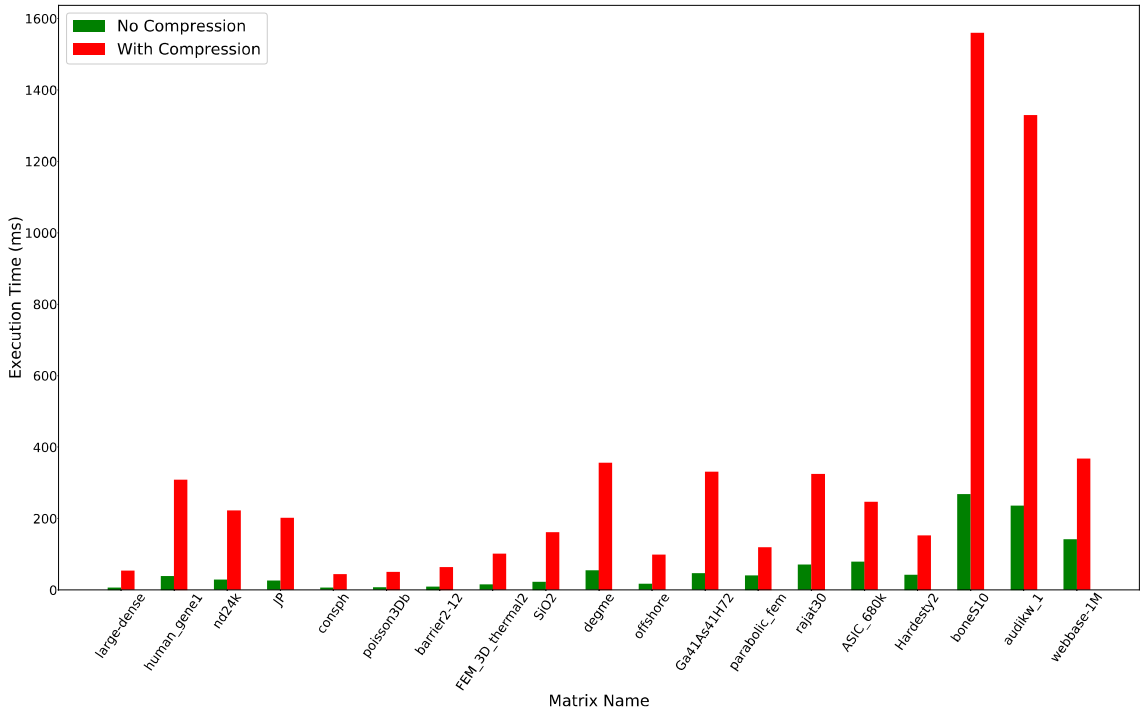
Ο συγκεκριμένος τρόπος αναπαράστασης, αν και όντως επιτυγχάνει μείωση στην κατανάλωση μνήμης, όπως φαίνεται και παρακάτω, χρησιμοποιεί έναν μηχανισμό αποκωδικοποίησης στη μεριά του PL, ο οποίος δεν είναι καθόλου φιλικός προς την FPGA αρχιτεκτονική.



Σχήμα 4.2: Σύγκριση κατανάλωσης μνήμης για αναπαράσταση χωρίς και με συμπίεση μηδενικών στοιχείων (II=8, CU=4)

Συγκεκριμένα, γίνεται χρήση πολλών if-conditionals, στο στάδιο περάσματος τιμών στα streams ***col_fifo*** και ***values_fifo***. Ανάλογα την τιμή του ζεύγους ***(X,0)***, όπου ***X*** το πλήθος των μηδενικών στοιχείων που χρειάζεται για να συμπληρωθεί η συγκεκριμένη γραμμή του

πίνακα, γίνεται χρήση του αντίστοιχου «μονοπατιού» στον κώδικα. Η συγκεκριμένη τεχνική δεν ενδείκνυται για προγραμματισμό σε FPGA, και τα αποτελέσματα στους χρόνους εκτέλεσης μας δείχνουν ότι μελλοντικά θα χρειαστεί να ακολουθήσουμε διαφορετική μέθοδο συμπίεσης.



Σχήμα 4.3: Σύγκριση χρόνων εκτέλεσης για αναπαράσταση χωρίς και με συμπίεση μηδενικών στοιχείων (II=8, CU=4)

Ιδανικά, θα θέλαμε η υλοποίησή μας να μην εξαρτάται από μηδενικά στοιχεία που χρησιμοποιούνται για γέμισμα των γραμμών του αραιού πίνακα. Στόχος μας είναι η μείωση κατά το δυνατόν της επικοινωνίας μεταξύ PS και PL και των δεδομένων που χρειάζεται να μεταφερθούν από το πρώτο στο δεύτερο υποσύστημα, μιας και ο αλγόριθμος SpMV είναι ένα αρκετά εξαρτημένο από τη μνήμη (memory-bound) πρόβλημα.

4.3 Αριθμητική κινητής υποδιαστολής διπλής ακρίβειας

Επόμενο βήμα για τη βελτίωση της υλοποίησής μας είναι η χρήση αριθμητικής κινητής υποδιαστολής διπλής ακρίβειας (double precision). Με τον τρόπο αυτό η ακρίβεια των εκτελούμενων πράξεων θα αυξηθεί.

4.4 Έλεγχος και αξιολόγηση άλλων σχημάτων αποθήκευσης

Μελλοντικά, θα θέλαμε να εξετάσουμε την απόδοση εναλλακτικών σχημάτων αποθήκευσης. Ένα σχήμα αποθήκευσης που αναπτύχθηκε και βελτιστοποιήθηκε για χρήση σε FPGA είναι το **Compressed Variable-Length Bit-Vector** (CVBV) που παρουσιάστηκε στην Ενότητα 2.2.1. Με το συγκεκριμένο επιτεύχθηκε συμπίεση έως και 43% σε σχέση με το σχήμα αποθήκευσης CSR, βήμα στην κατεύθυνση που θα θέλαμε και εμείς να ακολουθήσουμε.

Πέραν αυτού, θα θέλαμε να εξετάσουμε τη συμπεριφορά του σχήματος αποθήκευσης **CSX** [11], το οποίο αξιοποιεί πυκνές υποπεριοχές οποιουδήποτε είδους του πίνακα.

4.5 Ανάπτυξη βιβλιοθήκης FPGA-SpMV

Στόχος είναι ακόμη η ανάπτυξη μιας βιβλιοθήκης SpMV για FPGA ή η ένταξη των διαφόρων υλοποιήσεων σε ήδη υπάρχουσες βιβλιοθήκες γραμμικής άλγεβρας. Οι πιθανές παράμετροι των υλοποιήσεων είναι ο παράγοντας Vectorization και το πλήθος των Υπολογιστικών Μονάδων που χρησιμοποιούνται. Με κατάλληλη προεπεξεργασία του πίνακα, θα γίνεται επιλογή της βέλτιστης υλοποίησης, όσον αφορά την κατανάλωση μνήμης (λόγω του γεμίσματος με μηδενικά) ή την απόδοση.

4.6 Έλεγχος απόδοσης σε διαφορετικό FPGA

Η υλοποίηση, όσον αφορά το κομμάτι του 2D-Blocking, εξαρτάται άμεσα από τη διαθέσιμη BRAM που διαθέτει το FPGA του εργαστηρίου. Η αύξηση της διαθέσιμης BRAM σε ένα βελτιωμένο FPGA, θα μπορούσε να οδηγήσει στη βελτίωση της απόδοσης, καθώς ο αριθμός των μπλοκ, στα οποία θα πρέπει να διασπαστεί ένας πίνακας, θα μειωθεί.

Κεφάλαιο 5

Επίλογος

Στην παρούσα διπλωματική έγινε μια πρώτη προσπάθεια υλοποίησης του πολλαπλασιασμού αραιού πίνακα με διάνυσμα (SpMV) σε FPGA, με χρήση εργαλείων HLS που επιταχύνουν τη διαδικασία σχεδιασμού και προγραμματισμού στο FPGA. Ο συγκεκριμένος αλγόριθμος είναι ένα αρκετά εξαρτημένο από τη μνήμη (memory-bound) πρόβλημα, και τα FPGA διαθέτουν περιορισμένο εύρος ζώνης μεταφοράς δεδομένων. Για το λόγο αυτό, χρειάζεται προσεκτική μελέτη των τεχνικών που θα εφαρμοστούν, προκειμένου η εκτέλεσή του στο FPGA να δώσει ενθαρρυντικά αποτελέσματα.

Αρχικά, με το σχήμα αποθήκευσης packedCSR γίνεται πλήρης αξιοποίηση του εύρους ζώνης μνήμης, επιτυγχάνοντας την ταχύτερη δυνατή μεταφορά των δεδομένων από το υποσύστημα PS στις Υπολογιστικές Μονάδες του FPGA. Στη συνέχεια, με την παράλληλη εκτέλεση περισσότερων της μίας πράξεων ανά γραμμή του πίνακα, γίνεται η πρώτη προσπάθεια στην κατεύθυνση της βελτίωσης της απόδοσης του υπολογιστικού πυρήνα. Περαιτέρω βελτίωση επιτεύχθηκε με τη δημιουργία πολλαπλών Υπολογιστικών Μονάδων και το διαμοιρασμό του φόρτου εργασίας σε αυτές, που λειτουργούν παράλληλα και ανεξάρτητα η μία από την άλλη. Τέλος, με τη διάσπαση του πίνακα σε μπλοκ 2 διαστάσεων (2D-Blocking), έγινε εφικτή η εκτέλεση του SpMV για οποιοδήποτε μέγεθος πίνακα, χωρίς περιορισμό από τη διαθέσιμη μνήμη BRAM του FPGA.

Κατά τη σύγκριση της υλοποίησής μας με μία σύγχρονη CPU και GPU, διαπιστώσαμε ότι από άποψη επίδοσης το FPGA επιτυγχάνει χρόνους εκτέλεσης που είναι αρκετά χειρότεροι από τις άλλες δύο αρχιτεκτονικές. Το συγκεκριμένο γεγονός μπορεί να εξηγηθεί από την εκτεταμένη εφαρμογή του γεμίσματος με μηδενικά που εφαρμόζεται, και καθιστά ακόμη πιο memory-bound το SpMV για το FPGA. Η απαλλαγή της υλοποίησης από τα μηδενικά ενδεχομένως να βελτιώσει αρκετά την επίδοσή της. Από τη σκοπιά της κατανάλωσης ενέργειας όμως, το FPGA, αν και με αρκετά χειρότερους χρόνους εκτέλεσης, αποδείχθηκε η πιο ενεργειακά αποδοτική επιλογή ανάμεσα στις τρεις αρχιτεκτονικές.

Συμπερασματικά, το FPGA δείχνει ότι μπορεί να αποτελέσει μια εναλλακτική επιλογή για το SpMV, εάν στόχος είναι ο συνδυασμός επίδοσης και ενεργειακής αποδοτικότητας. Με την εφαρμογή βελτιώσεων, όπως αυτές που προτάθηκαν στο Κεφάλαιο 4, τα FPGA μπορούν να αποδειχθούν μία ακόμα πιο συμφέρουσα λύση για το SpMV.

Βιβλιογραφία

[1] EE Times. All about FPGAs https://www.eetimes.com/document.asp?doc_id=1274496, 2006.

[2] R. Bajaj and S. Fahmy, Mapping for maximum performance on FPGA DSP blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35. 1-1. 10.1109/TCAD.2015.2474363, 2015.

[3] Intel, Altera. Architecture Brief. What is an SoC FPGA? https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ab/ab1_soc_fpga.pdf, 2014.

[4] Xilinx. Zynq UltraScale+ MPSoC Data Sheet : Overview https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf, 2018.

[5] Xilinx. Introduction to FPGA Design with Vivado High-Level Synthesis https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf, 2019.

[6] David Gschwend. ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network https://github.com/dgschwend/zynqnet, 2016.

[7] Xilinx. SDx Pragma Reference Guide https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1253-sdx-pragma-reference.pdf, 2019.

[8] E. J. Im, K. Yelick. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. *Proceedings of the International Conference on Computational Sciences – Part I*, pages 127–136. Springer-Verlag, 2001.

[9] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*, PhD thesis, University of California, Berkeley, 2003.

[10] A. Pinar, M. T. Heath Improving Performance of Sparse Matrix-Vector Multiplication. *SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, Portland, OR, USA, 1999, pp. 30-30.

[11] K. Kourtis, V. Karakasis, G. Goumas, N. Koziris. CSX: An Extended Compression Format for SpMV on Shared Memory Systems. *16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*, San Antonio, TX, USA, February 12–16, 2011.

[12] S. Kestur, J. D. Davis and E. S. Chung. Towards a universal FPGA matrix-vector multiplication architecture. *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, Toronto, ON, 2012, pp. 9-16.

[13] Z. Ling, V. K. Prasanna. Sparse Matrix-Vector multiplication on FPGAs. *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays (FPGA '05)*, ACM, New York, NY, USA, 63-74.

[14] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar and J. D. Bakos. FPGA vs. GPU for sparse matrix vector multiply. *2009 International Conference on Field-Programmable Technology*, Sydney, NSW, 2009, pp. 255-262.

[15] P. Grigoras, P. Burovskiy, E. Hung and W. Luk. Accelerating SpMV on FPGAs by Compressing Nonzero Values. *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, Vancouver, BC, 2015, pp. 64-67.

[16] J. Willcock, A. Lumsdaine. Accelerating sparse matrix computations via data compression. *Proceedings of the 20th annual international conference on Supercomputing (ICS '06)*, ACM, New York, NY, USA.

[17] M. Hosseinabady, J. L. 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*, .

[18] SuiteSparse Matrix Collection `https://sparse.tamu.edu`.

[19] HLS Arbitrary Precision Types Library `https://github.com/Xilinx/HLS_arbitrary_Precision_Types`.

[20] Intel® Math Kernel Library `https://software.intel.com/en-us/mkl`.

[21] NVIDIA CUDA Sparse Matrix library `https://docs.nvidia.com/cuda/cusparse/index.html`.

[22] H. Giefers, P. Staar, C. Bekas and C. Hagleitner. Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA. *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Uppsala, 2016, pp. 46-56.