Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Accelerating Connected Components Graph Algorithms in Reconfigurable Logic

Διπλωματική Εργασία

ΑΛΕΞΑΝΔΡΟΣ Κ. ΚΟΥΡΗΣ

Επιβλέπων: Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων
Αθήνα, Απρίλιος 2019

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

Accelerating Connected Components Graph Algorithms in Reconfigurable Logic

Διπλωματική Εργασία

ΑΛΕΞΑΝΔΡΟΣ Κ. ΚΟΥΡΗΣ

Επιβλέπων: Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 4 Απριλίου 2019.

.....	.....	.....
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Παναγιώτης Τσανάκας
Αναπληρωτής Καθηγητής	Καθηγητής	Καθηγητής

Αθήνα, Απρίλιος 2019

(Υπογραφή)

……
Αλέξανδρος Κ. Κουρής
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

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

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή κ. Δημήτριο Σούντρη, που μου έδωσε την ευκαιρία να συνεργαστώ μαζί του σε ένα πολύ ενδιαφέρον αντικείμενο και εργαστήριο.

Επίσης ευχαριστώ ιδιαίτερα τον μεταδιδακτορικό ερευνητή κ. Χριστόφορο Κάχρη για την καθοδήγηση και βοήθεια που μου προσέφερε κατά την διάρκεια εκπόνησης αυτής της διατριβής.

Τέλος θα ήθελα να ευχαριστήσω την οικογένειά μου και τους φίλους μου, για τη στήριξή τους και όλα όσα μου προσέφεραν κατά την διάρκεια φοίτησης μου στην σχολή.

Περίληψη

Ένα από τα πιο χρήσιμα αντικείμενα στη σύγχρονη επιστήμη των υπολογιστών και το οποίο έχει επανέλθει στην επικαιρότητα λόγω της διάδοσης και ανάγκης ανάλυσης μεγάλου όγκου δεδομένων (Big Data) είναι η θεωρία των γράφων. Η έρευνα έχει πλέον στραφεί στην εύρεση νέων αποδοτικότερων αλγορίθμων και στη βελτιστοποίηση των υπαρχόντων, ειδικότερα αυτών που επικεντρώνονται στην επίλυση προβλημάτων και ανάλυση γράφων.

Οι γράφοι αναλαμβάνουν να αναπαραστήσουν τις σχέσεις μεταξύ των στοιχείων ενός συνόλου. Γύρω μας οι γράφοι διέπουν μεγάλο μέρος της καθημερινότητας μας χωρίς να το αντιλαμβανόμαστε, καθώς γράφος μπορεί να είναι ένα δίκτυο υπολογιστών, μία αλυσίδα καταστημάτων, ένα κοινωνικό δίκτυο ή όποιας μορφής δίκτυο δημιουργείται μέσω μιας σχέσης σε ένα σύνολο. Χαρακτηριστικά των γράφων είναι κυρίως το μεγάλο τους μέγεθος και η τυχαιότητα που διέπει της διασυνδέσεις τους. Λόγω των παραπάνω, ο χρόνος επίλυσης και ο υπολογιστικός φόρτος που απαιτείται για την επίλυση αλγορίθμων αυξάνεται εκθετικά κάθε χρόνο. Πλέον, για να επεξεργαστούμε τα δεδομένα και για να εξάγουμε πληροφορίες από γράφους που αναπαριστούν παραδείγματος χάρη, μέσα κοινωνικής δικτύωσης, απαιτείται τεράστια υπολογιστική ισχύ, που πριν από μερικά χρόνια δεν θα μπορούσαμε να το φανταστούμε. Αυτό το φαινόμενο και η συνεχής ροή και παραγωγή δεδομένων δεν πρόκειται να σταματήσουν, οπότε χρειαζόμαστε νέες μεθόδους και τρόπους ανάλυσης των γράφων. Η πολυπλοκότητα πλέον των αλγορίθμων γράφων καθιστούν μη αποδοτική την εκτέλεση τους από επεξεργαστές γενικού σκοπού.

Σε αυτή τη διπλωματική διερευνούμε τη δυνατότητα και την αποδοτικότητα υλοποίησης τους από υλικό ειδικού σκοπού, FPGAs. Πιο συγκεκριμένα, θα ασχοληθούμε με τον αλγόριθμο των Shiloach-Vishkin για την εύρεση του αριθμού των διασυνδεμένων στοιχείων ενός γράφου. Για την υλοποίηση του αλγορίθμου θα χρησιμοποιήσουμε FPGA της εταιρίας Xilinx και συγκεκριμένα το ZYNQ ZC702. Θα διερευνήσουμε διάφορες μορφές του αλγορίθμου αλλά και τη χρήση του FPGA. Στόχος μας είναι η επιτάχυνσή του με όσο το δυνατό μικρότερη χρήση πόρων του συστήματος.

Λέξεις Κλειδιά

Θεωρία γράφων, Shiloach-Vishkin, Διασυνδεδεμένα Στοιχεία, FPGA, επεξεργαστής υλικού, επιτάχυνση αλγορίθμου, ZYNQ ZC702

Abstract

One of the most useful subjects in modern computer science and which has come back to the news due to the dissemination and need of large data analysis (Big Data) is graph theory. Research has now turned to finding new, more efficient algorithms and optimizing existing ones, especially those related to problem solving and graph analysis.

The graphs undertake to represent the relationships between the elements of a set. Around us, the graphs govern much of our everyday life without us perceiving it, as a graph can be a computer network, chain stores, a social network or any form of network that is created through a relationship in a whole. Characteristics of the graphs are mainly their large size and the randomness that governs their interconnections. Due to the above, the solving time and computational load required to solve algorithms grows exponentially every year. Now, in order to process data and extract information from graphs depicting, for example, social media, a huge amount of computational power is required, which a few years ago we could not imagine. This phenomenon and continuous flow and production of data are not going to stop, so we need new methods and ways of analyzing graphs. The complexity of graph algorithms renders their performance unprofitable by general purpose processors.

In this diploma, we investigate the potential and efficiency of their implementation on special purpose hardware, called FPGAs. More specifically, we will deal with the Shiloach-Vishkin algorithm to find the number of connected components of a graph. To implement the algorithm, we will use Xilinx's FPGA, namely the ZYNQ ZC702. We will explore various forms of the algorithm but also the use of the FPGA. Our goal is to accelerate it with the least possible use of system resources.

Keywords

Graph theory, Shiloach-Vishkin, Connected Components, FPGA, hardware processing, acceleration, ZYNQ ZC702

Περιεχόμενα

Κατάλογος Σχημάτων

Κεφάλαιο 1

Εισαγωγή

1.1 Αντικείμενο της διπλωματικής

Τα τελευταία χρόνια υπάρχει μεγάλη ζήτηση και έντονη έρευνα γύρω από τα κατανεμημένα συστήματα και την αποτελεσματική επεξεργασία γράφων μεγάλης κλίμακας (Big Data), καθώς όλο και περισσότερες εταιρίες και εφαρμογές στρέφονται προς το cloud και τα data centers για την αποθήκευση και την επεξεργασία των δεδομένων τους. Εφαρμογές όπως τα κοινωνικά δίκτυα, η αναζήτηση στον ιστό, η μηχανική μάθηση κ.ά., διαθέτουν και παράγουν έναν τεράστιο όγκο δεδομένων καθημερινά, ο οποίος μπορεί να διαμορφωθεί σε γράφους. Η γραφή κατανεμημενών εφαρμογών γράφων είναι εγγενώς δύσκολη και απαιτεί μοντέλα προγραμματισμού που μπορούν να καλύψουν ένα διαφορετικό σύνολο τομέων προβλημάτων, συμπεριλαμβανομένων επαναληπτικών αλγορίθμων βελτίωσης, μετασχηματισμών γράφων, συνόλων γράφων, αντιστοίχησης προτύπων, ανάλυσης δικτύων γράφων και διαδρομών γράφων. Σε μεγάλες πλατφόρμες δεδομένων έχουν υλοποιηθεί πολλές εφαρμογές για την επεξεργασία γράφων σε κατανεμημένα συστήματα. Παρόλα που έχει γίνει σημαντική πρόοδος σε πειραματικό επίπεδο στην επεξεργασία γράφων δεν έχει βρεθεί κάποια βέλτιστη και μόνιμη λύση. Στην παρούσα διπλωματική θα ασχοληθούμε με τη θεωρία των γράφων και πιο συγκεκριμένα θα μελετήσουμε τον αλγόριθμο των Shiloach-Vishkin για την εύρεση του πλήθους των διασυνδεδεμένων στοιχείων (Connected Components) ενός γράφου. Παράλληλα, θα αναλύσουμε τη χρήση των συστημάτων FPGA για την υλοποίηση αλγορίθμων, πλεονεκτήματα από τη χρήση αυτών και τέλος θα υλοποιήσουμε έναν επιταχυντή υλικού του παραπάνω αλγορίθμου.

1.2 Οργάνωση του τόμου

Η διατριβή αυτή οργανώνεται ως εξής:

1. Στο Κεφάλαιο 2 αναλύουμε το θεωρητικό υπόβαθρο που χρειαζόμαστε σχετικά με τα FPGAs και ευνοϊκές τεχνικές για την χρήση τους.
2. Στο Κεφάλαιο 3 αναφερόμαστε στον αλγόριθμο των Shiloach-Vishkin για την εύρεση του αριθμού των Connected Components. Αρχίζουμε από την θεωρία των γράφων, αναπαράσταση γράφων και παρουσίαση της αρχικής

δημοσίευσης των Shiloach-Vishkin. Τέλος, παραθέτουμε τη βελτιωμένη μορφή του αλγορίθμου που πρότειναν οι Bader et al.

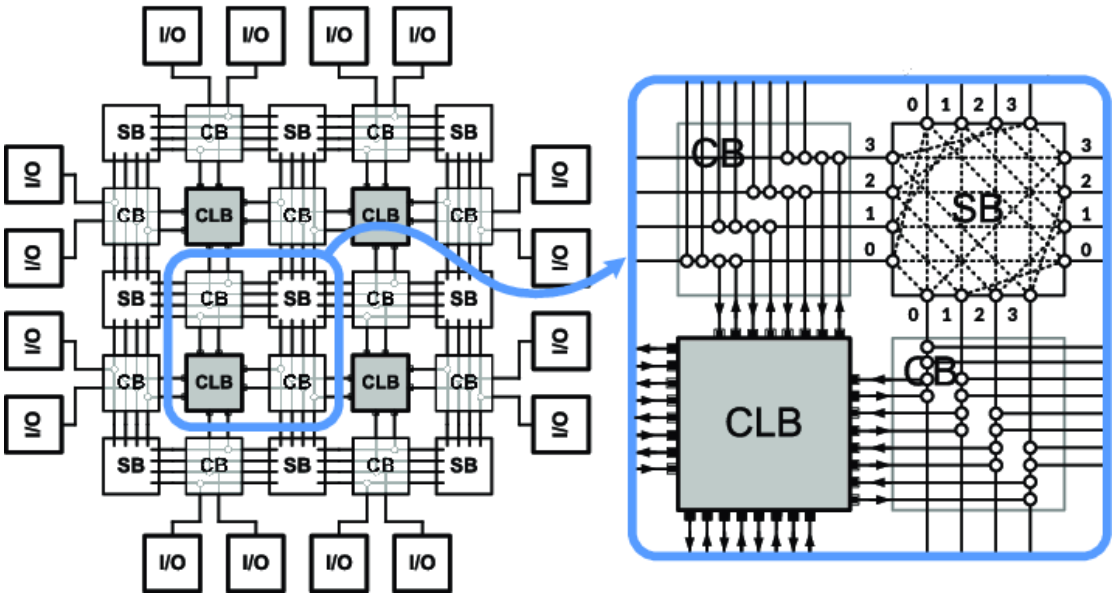
3. Στο Κεφάλαιο 4 αναλύουμε την σχεδίαση της αρχιτεκτονικής του αλγορίθμου στο υλικό.
4. Στο Κεφάλαιο 5 παραθέτουμε τα αποτελέσματα και αξιολογούμε τους επιταχυντές υλικού που υλοποιήσαμε.
5. Στο Κεφάλαιο 6 ερευνούμε δημοσιεύσεις σχετικές με την υλοποίηση αλγορίθμων γράφων σε FPGAs και άλλες τεχνολογίες.
6. Στο Κεφάλαιο 7 συνοψίζουμε τα συμπεράσματα αυτής της διατριβής.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

2.1 Εισαγωγή στα FPGA

Τα FPGAs (Field Programmable Gate Arrays) είναι ένας τύπος προγραμματιζόμενων ολοκληρωμένων κυκλωμάτων που επιτρέπουν στον χρήστη, την υλοποίηση διάφορων εφαρμογών. Είναι ημιαγώγιμες συσκευές που περιλαμβάνουν στοιχεία προγραμματιζόμενης λογικής CLBs (Configurable Logic Blocks),προγραμματιζόμενες διασυνδέσεις και μπλοκ εισόδου/εξόδου. Κύριο χαρακτηριστικό τους είναι ότι μπορούν να επαναπρογραμματίζονται κατά το δοκούν για αναβάθμιση ήδη υλοποιημένων εφαρμογών ή ακόμα και για υλοποίηση διαφορετικών εφαρμογών, σε αντίθεση με τα ολοκληρωμένα κυκλώματα εφαρμογών ASIC (Application-Specific Integrated Circuit), που κατασκευάζονται για συγκεκριμένη εφαρμογή και δεν επιδέχονται αλλαγής. Τα FPGAs περιέχουν μεγάλο αριθμό στοιχείων προγραμματιζόμενης λογικής (CLBs). Τα CLBs οργανώνονται σε έναν δύο διαστάσεων πίνακα. Κατά αυτόν τον τρόπο διαμορφώνεται η βασική δομή των FPGA όπως φαίνεται στο παρακάτω σχήμα:



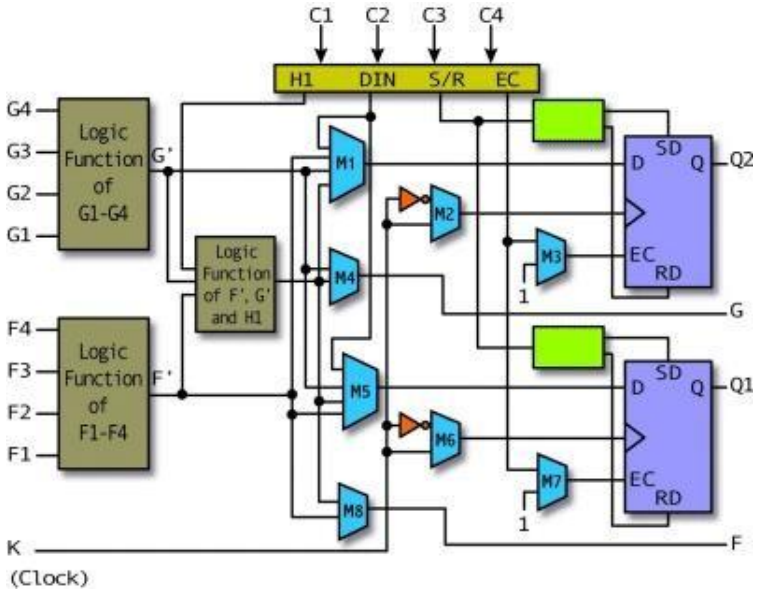
Σχήμα 2.1: Βασική δομή FPGA

2.2 Δομικά στοιχεία των FPGAs

Η βασική δομή ενός FPGA [1] περιλαμβάνει τα εξής στοιχεία:

- **Configurable Logic Blocks (CLB)**

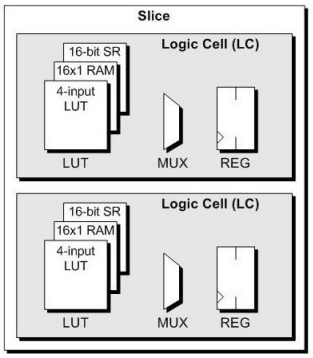
Τα μπλοκ αυτά περιέχουν την λογική για τα FPGA και δημιουργούν μία μικρή μηχανή καταστάσεων, όπως φαίνεται στο παρακάτω σχήμα. Περιέχουν πίνακες αναζήτησης (LUTs), Flip-Flops, πολυπλέκτες για την δρομολόγηση της λογικής εντός του μπλοκ και προς εξωτερικούς πόρους. Οι πολυπλέκτες επιτρέπουν επίσης την επιλογή πολικότητας, την επαναφορά και την εκκαθάριση της επιλογής εισόδου.



Σχήμα 2.2: Δομή CLB

- **Slices**

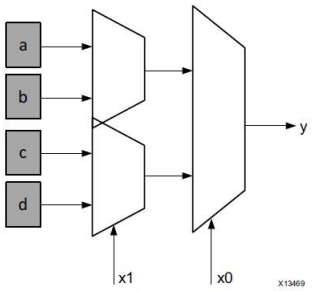
Υπομονάδα των CLBs. Στα FPGA της Xilinx δύο ή τέσσερα slices συνθέτουν ένα CLB.



Σχήμα 2.3: Δομή slice

- **Look-up tables (LUTs)**

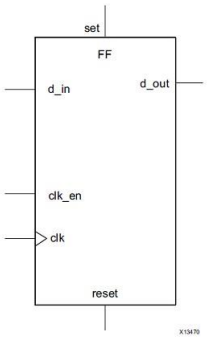
Στοιχεία που είναι ικανά να υλοποιήσουν οποιαδήποτε λογική λειτουργία. Ουσιαστικά, αυτό το στοιχείο είναι ένας πίνακας αληθείας στον οποίο οι διάφοροι συνδυασμοί των εισόδων που εφαρμόζονται, υλοποιούν διαφορετικές λειτουργίες για να δώσουν τιμές εξόδου. Λόγω της μεγάλης ευελιξίας τους, δύναται να χρησιμοποιούνται ως 64-bit μνήμες και αναφέρονται συχνά ως διαμοιρασμένες μνήμες. Αποτελούν την ταχύτερη μνήμη που συναντάται στα FPGAs.



Σχήμα 2.4: Δομή LUT

- **Flip-Flops (FFs)**

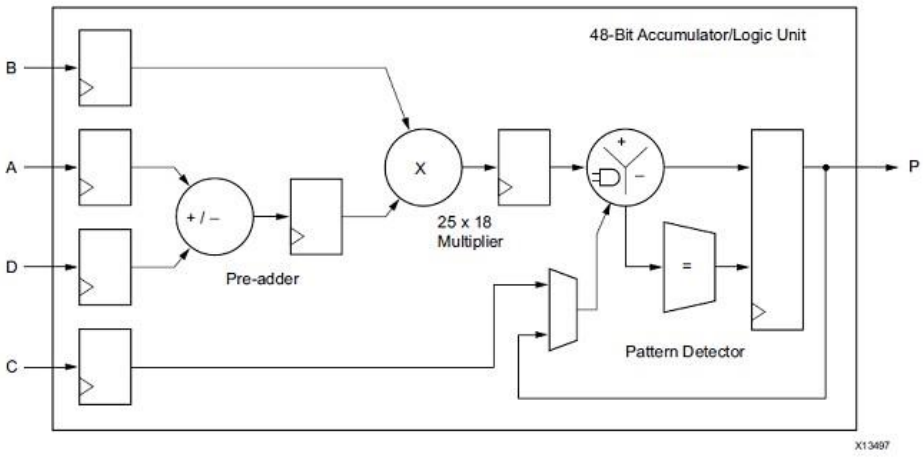
Η βασική δομή τους περιλαμβάνει είσοδο δεδομένων, είσοδο ρολογιού, ενεργοποιητή ρολογιού, επαναφορά και έξοδο δεδομένων. Αποτελούν καταχωρητή 1 bit. Έχουν την ιδιότητα να μεταδίδουν την είσοδο όταν επιτρέπει το ρολόι του συστήματος καθώς και να αποθηκεύουν το 1bit για παραπάνω από έναν παλμό ρολογιού. Συστάδα αυτών υλοποιούν τους καταχωρητές.



Σχήμα 2.5: Δομή Flip-Flop

- **DSP48 Block**

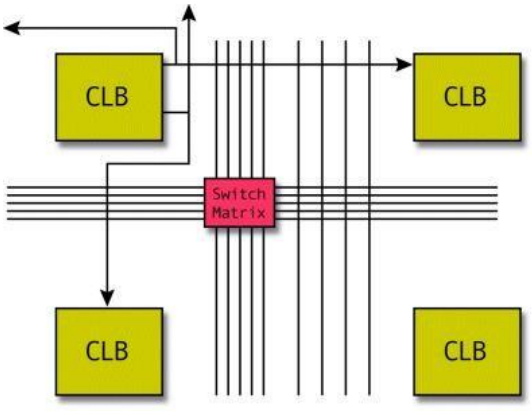
Το πιο περίπλοκο, υπολογιστικό μπλοκ σε ένα FPGA της Xilinx. Είναι μία αριθμητική λογική μονάδα (ALU) που αποτελείται από μία αλυσίδα τριών διαφορετικών μπλοκ. Αυτή η υπολογιστική αλυσίδα περιλαμβάνει μια μονάδα πρόσθεσης/αφαίρεσης συνδεδεμένη με έναν πολλαπλασιαστή που συνδέεται στην τελική μονάδα πρόσθεσης / αφαίρεσης / συσσώρευσης. Αυτή η αλυσίδα επιτρέπει να υλοποιούνται σε ένα μόνο DSP48, συναρτήσεις της μορφής: $P = Bx(A + D) + C$ και να επωμιστεί μεγάλο υπολογιστικό φόρτο.



Σχήμα 2.6: Δομή DSP48

- **Προγραμματιζόμενες Διασυνδέσεις**

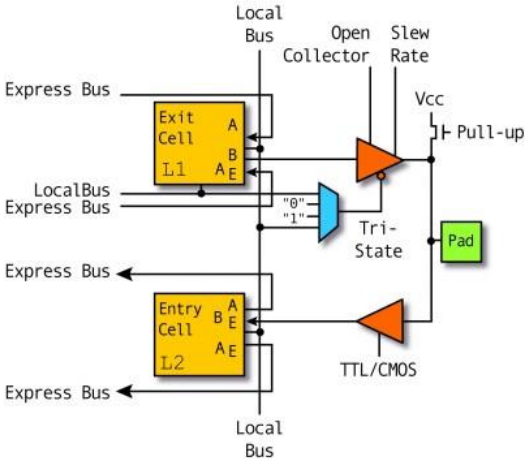
Είναι γραμμές (καλώδια) που διασυνδέουν τα στοιχεία μεταξύ τους χωρίς καθυστέρηση. Χρησιμοποιούνται και ως δίαυλοι μέσα στο chip. Τρανζίστορ χρησιμοποιούνται για να συνδεθούν ή να αποσυνδεθούν διαφορετικές γραμμές. Επίσης υπάρχουν προγραμματιζόμενοι πίνακες αλλαγής (Switch Matrices - SM), όπως φαίνεται στο παρακάτω σχήμα, για να συνδέουν τις γραμμές σε συγκεκριμένο, ευέλικτο τρόπο.



Σχήμα 2.7: Προγραμματιζόμενη διασύνδεση

- **Μπλοκ Εισόδου/Εξόδου (IOBs)**

Τα μπλοκ αυτά μεταφέρουν τα δεδομένα εντός και εκτός του FPGA. Περιέχουν ένα buffer εισόδου, ένα buffer εξόδου τριών καταστάσεων και έξοδο ανοικτού συλλέκτη. Τα μπλοκ αυτά βρίσκονται περιφερειακά στο FPGA.



Σχήμα 2.8: Μπλοκ Εισόδου/Εξόδου (IOB)

- **Registers**

Συνθέτονται από μία συστάδα Flip-Flops. Αποτελούν την ταχύτερη δομή μνήμης. Ενσωματώνονται στον υπολογισμό όπου χρησιμοποιούνται χωρίς την ανάγκη διευθυνσιοδότησης ή πρόσθετων καθυστερήσεων.

- **Shift Registers**

Είναι μία αλυσίδα καταχωρητών συνδεδεμένων μεταξύ τους. Σκοπός αυτών είναι η επαναχρησιμοποίηση δεδομένων κατά μήκος ενός υπολογιστικού μονοπατιού.

- **Block RAM (BRAM)**

Είναι μία δύο-θυρών μνήμη τυχαίας προσπέλασης (RAM), ενσωματωμένη στο FPGA, που προσφέρει την δυνατότητα αποθήκευσης σχετικά μεγάλου όγκου δεδομένων. Η διασύνδεση της μέσω δύο θυρών επιτρέπει την παράλληλη, σε ίδιο κύκλο ρολογιού πρόσβαση σε διαφορετικές τοποθεσίες μνήμης.

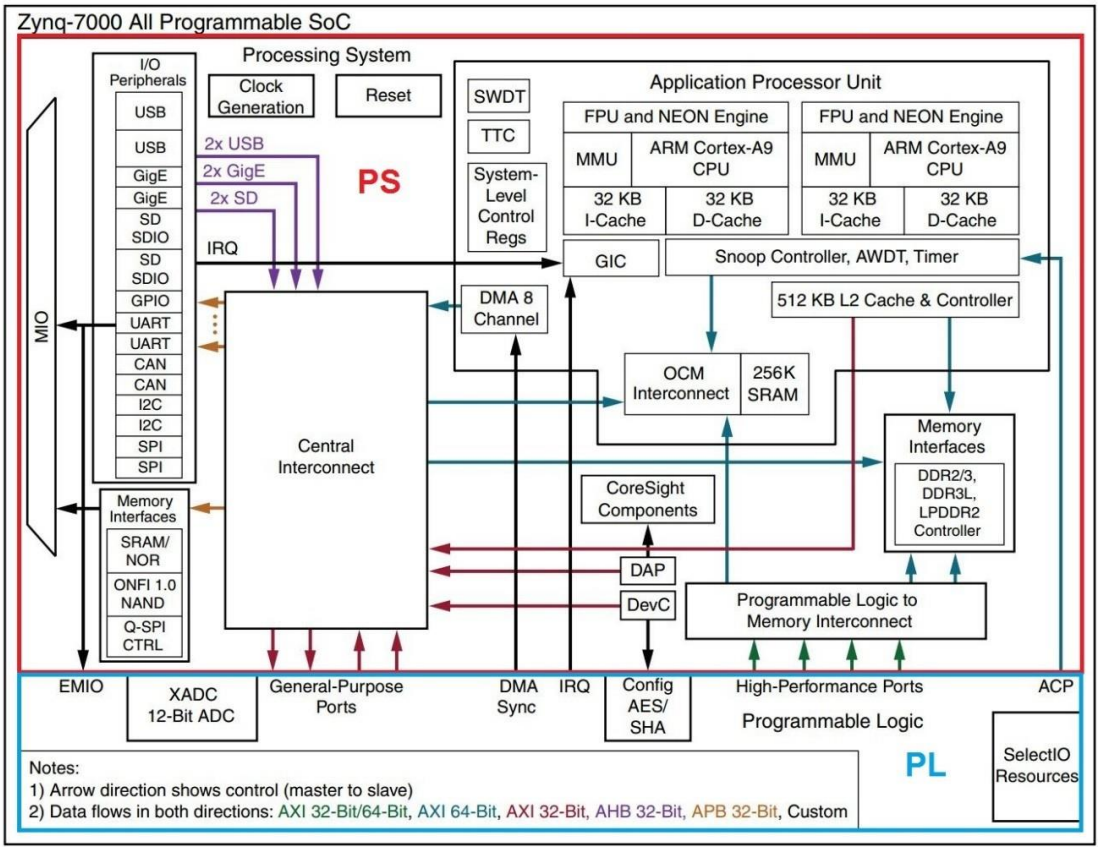
- **Fist In First Out (FIFO)**

Μπορεί να θεωρηθεί ως ουρά με ένα μόνο σημείο εισόδου και ένα μόνο σημείο εξόδου. Αυτό το είδος δομής χρησιμοποιείται συνήθως για τη μετάδοση δεδομένων μεταξύ βρόχων προγράμματος ή λειτουργιών. Δεν υπάρχει σχετική

λογική διευθυνσιοδότησης και οι λεπτομέρειες υλοποίησης διεκπεραιώνονται πλήρως από τον μεταγλωττιστή HLS.

2.3 Zynq-7000 All Programmable SoC

Η γενιά των Zynq-7000 All Programmable SoC περιλαμβάνει ετερογενείς υπολογιστικές πλατφόρμες που συνδυάζουν τα χαρακτηριστικά ενός επεξεργαστή ARM με ένα FPGA, επιτρέποντας την επιτάχυνση υλικού. Ο ARM επεξεργαστής αποτελεί την καρδιά του επεξεργαστικού συστήματος (Processing System - PS) και το FPGA την βάση της προγραμματιζόμενης λογικής (Programmable Logic - PL), έχοντας όλα τα χαρακτηριστικά που αναπτύξαμε παραπάνω. Δίνουν την δυνατότητα παραλληλίας και διοχέτευσης (pipeline) στις εφαρμογές κατά μήκος ολόκληρης της πλατφόρμας, αφού οι πόροι της μπορούν να χρησιμοποιούνται ταυτόχρονα. Παράλληλα, ενσωματώνουν κεντρική μονάδα επεξεργασίας CPU, ψηφιακή επεξεργασία σήματος DSP, κυκλώματα-προϊόντα συγκεκριμένης εφαρμογής ASSP και λειτουργία μικτού σήματος σε μία συσκευή. Σαν αποτέλεσμα προσφέρουν την ευελιξία και την επεκτασιμότητα ενός FPGA, ενώ παράλληλα παρέχουν χαμηλή κατανάλωση και υψηλές επιδόσεις.



Σχήμα 2.9: Αρχιτεκτονική ZYNQ-7000 [2]

2.3.1 Σύστημα Επεξεργασίας PS

Το τμήμα του Συστήματος Επεξεργασίας (PS) [2] είναι ένα πλήρες σύστημα βασισμένο σε ένα επεξεργαστή διπλού πυρήνα ARM Cortex-A9 που υλοποιεί αρχιτεκτονική ARMv7-A. Τους πυρήνες συνοδεύουν και άλλες επεξεργαστικές μονάδες, που μαζί συντελούν την μονάδα επεξεργασίας εφαρμογών (APU) όπως φαίνεται και στην εικόνα 2.9. Πιο συγκεκριμένα κάθε πυρήνας συνοδεύεται από μια μονάδα επεξεργασίας μέσων NEON και κινητής υποδιαστολής (Floating Point Unit - FPU), μια μονάδα διαχείρισης μνήμης (Memory Management - MMU) και μία 32kB μνήμη (cache) πρώτου επιπέδου και τεσσάρων δρόμων συσχέτισης. Την μονάδα επεξεργασίας εφαρμογών απαρτίζουν ακόμα ο Snoop Controller, που αναλαμβάνει την επικοινωνία μεταξύ των δύο πυρήνων, μία διαμοιραζόμενη μνήμη 512kB δεύτερου επιπέδου, οκτώ δρόμων συσχέτισης και μία μνήμη OCM (On-Chip Memory) 256kB SRAM που είναι προσβάσιμη και από την προγραμματιζόμενη λογική. Εκτός της μονάδας εφαρμογών συναντάμε και μνήμη τύπου DDR2/DDR3 με μέγεθος που διαφέρει ανάλογα με την συσκευή της γενιάς Zynq-7000.

Στο Σύστημα Επεξεργασίας συναντάμε επίσης τις μονάδες εισόδου/εξόδου περιφερειακών συσκευών (I/O Peripherals) που αναλαμβάνουν την διεπαφή με τον εξωτερικό κόσμο, για λήψη και αποστολή δεδομένων, προς και από περιφερειακές συσκευές. Αυτές οι διεπαφές περιγράφονται παρακάτω:

- **USB (Universal Serial Bus)**

  Πρωτόκολλο επικοινωνίας σε δίαυλο με περιφερειακά συστήματα.

- **GigE (Gigabit Ethernet)**

  Τεχνολογία μετάδοσης βασισμένη στην Ethernet frame μορφή και πρωτόκολλο που χρησιμοποιείται σε τοπικά δίκτυα υπολογιστών.

- **SD/SDIO (Secure Digital Input Output)**

  Πρωτόκολλο επικοινωνίας για την επικοινωνία με κάρτα μνήμης SD

- **GPIO (General-Purpose Input/Output)**

  Είναι μια ευέλικτη θύρα που η συμπεριφορά της ελέγχεται από τον χρήστη κατά τον χρόνο εκτέλεσης.

- **UART (Universal Asynchronous Receiver-Transmitter)**

  Διεπαφή που επιτρέπει την ασύγχρονη σειριακή επικοινωνία.

- **CAN (Controller Area Network)**

  Αποτελεί ένα πρωτόκολλο επικοινωνίας που επιτρέπει στους μικροεπεξεργαστές να επικοινωνούν μεταξύ τους, χωρίς να επεμβαίνει ο κεντρικός επεξεργαστής.

- **I2C**

Ένας αμφίδρομος σειριακός δίαυλος δύο καναλιών, που παρέχει απλή και αποδοτική μεταφορά δεδομένων σε μικρές αποστάσεις, ανάμεσα σε διάφορες συσκευές.

- **SPI (Serial Peripheral Interface)**

Επιτρέπει την σύγχρονη σειριακή αμφίδρομη επικοινωνία σε έναν δίαυλο.

2.3.2 Πρωτόκολλο επικοινωνίας AXI

Η Xilinx για την μεταφορά δεδομένων μεταξύ των PS-PL καθώς και μεταξύ IP cores (προσυσκευασμένα τμήματα κώδικα VHDL), χρησιμοποιεί πρωτόκολλο επικοινωνίας AXI (Advanced eXtensible Interface) [3]. Το πρωτόκολλο AXI είναι μέρος της ARM AMBA (Advanced Microcontroller Bus Architecture) που είναι μια οικογένεια μικροελεγκτών διαύλων. Η πρώτη έκδοση AXI συμπεριλήφθηκε στο πρότυπο AMBA 3.0 που κυκλοφόρησε το 2003. Το πρότυπο AMBA 4.0 κυκλοφόρησε το 2010 και περιέχει την δεύτερη έκδοση AXI, την AXI4. Υπάρχουν τρεις τύποι διεπαφής AXI4.

- **AXI4**

Χρησιμοποιείται για memory mapped συνδέσεις και επιτρέπει burst (μαζική αποστολή) έως και 256 λέξεων δεδομένων, δίνοντας μόνο μία διεύθυνση μνήμης. Προσφέρει υψηλή απόδοση στις memory mapped συνδέσεις.

- **AXI4-Lite**

Απλή μεταφορά μίας λέξης δεδομένων σε μία διεύθυνση δεδομένων, σε memory mapped συνδέσεις. Προσφέρει χαμηλή απόδοση στις memory mapped συνδέσεις.

- **AXI4-Stream**

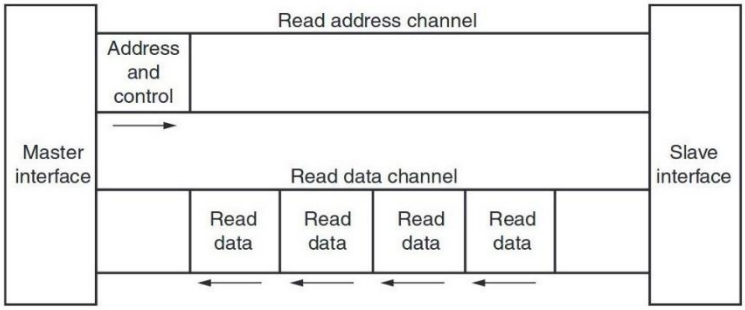
Αφαιρεί την υποχρέωση για διευθυνσιοδότηση μνήμης και επιτρέπει απεριόριστου μεγέθους burst. Προσφέρει υψηλές ταχύτητες σε streaming δεδομένα.

Οι προδιαγραφές AXI περιγράφουν την διεπαφή μίας μονάδας AXI master και μίας AXI slave, που αντιπροσωπεύουν IP cores και ανταλλάσσουν πληροφορίες μεταξύ τους. Αμφότερες οι διεπαφές τύπου AXI4 και AXI4-Lite περιέχουν πέντε κανάλια.

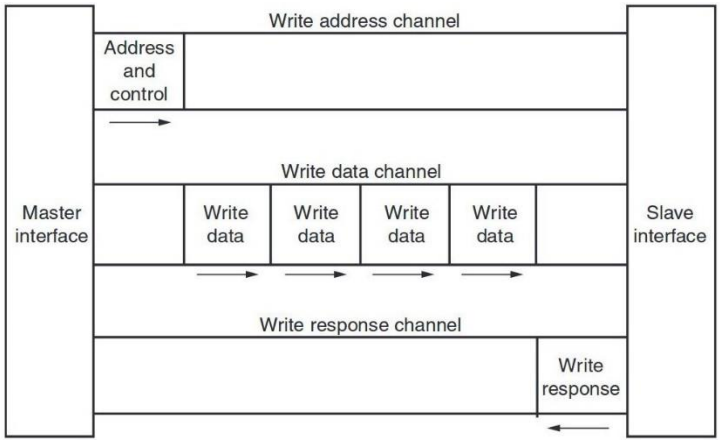
- Κανάλι Ανάγνωσης Διεύθυνσης

- Κανάλι Εγγραφής Διεύθυνσης
- Κανάλι Ανάγνωσης Δεδομένων
- Κανάλι Εγγραφής Δεδομένων
- Κανάλι Εγγραφής Απάντησης

Τα δεδομένα μπορούν να μεταφέρονται προς και τις δύο κατευθύνσεις ανάμεσα σε master και slave, καθώς και το μέγεθος των μεταφορών μπορεί να διαφέρει. Ο περιορισμός είναι ότι το AXI4 επιτρέπει burst ως 256 λέξεις δεδομένων, ενώ το AXI4-Lite επιτρέπει μεταφορά μόνο μίας λέξης δεδομένων ανά συναλλαγή.



Σχήμα 2.10: Αρχιτεκτονική Καναλιού Ανάγνωσης



Σχήμα 2.11: Αρχιτεκτονική Καναλιού Εγγραφής

Όπως φαίνεται και στα παραπάνω σχήματα, το AXI4 παρέχει ξεχωριστές συνδέσεις δεδομένων και διευθύνσεων και για εγγραφή και ανάγνωση, προσφέροντας έτσι την δυνατότητα ταυτόχρονης, αμφίδρομης μεταφοράς δεδομένων. Σε επίπεδο Hardware, το AXI4 επιτρέπει την χρήση διαφορετικών ρολογιών για κάθε ζευγάρι AXI master-slave. Επιπρόσθετα, το πρωτόκολλο AXI

επιτρέπει την χρήση slices, συχνά αποκαλούμενα στάδια pipeline, για να επιτυγχάνονται οι χρονικοί περιορισμοί.

2.3.3 Διεπαφές Συστήματος Επεξεργασίας και Προγραμματιζόμενης Λογικής

Στην εικόνα του σχήματος 2.9 φαίνονται οι τρεις τύπου διεπαφών με τις οποίες επικοινωνούν η Προγραμματιζόμενη Λογική με το Σύστημα Επεξεργασίας.

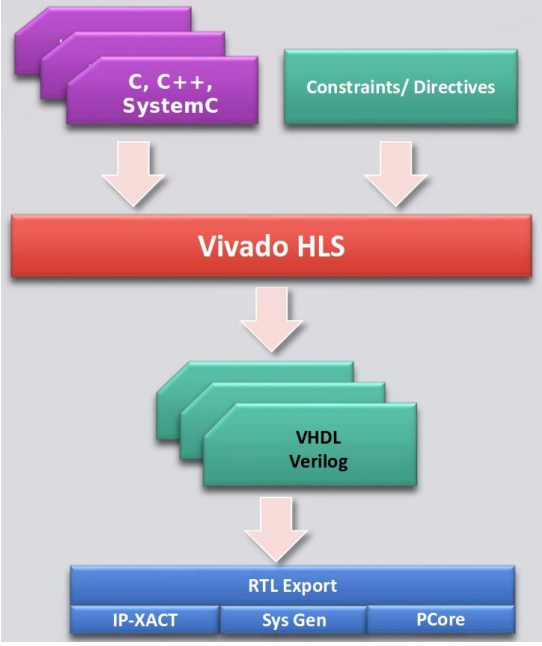
- **General Purpose Ports:** Δίαυλοι δεδομένων 32-bit επικοινωνίας. Η οικογένεια Zynq-7000 περιέχει τέσσερις τέτοιους διαύλους. Οι δύο από αυτούς έχουν ως master το PS και ως slave το PL και οι άλλοι δύο αντίστροφα. Επιτρέποντας έτσι και στα δύο μέρη να διευθετήσουν ή να ξεκινήσουν μια μεταφορά δεδομένων ως masters.

- **High Performance Ports:** Υψηλής απόδοσης δίαυλοι δεδομένων 32/64bit. Παρέχουν πρόσβαση στο PL στις DDR και OCM μνήμες του PS. Περιέχουν FIFO buffer που υποστηρίζει έως και 32 λέξεις δεδομένων για την ανάγνωση. Η οικογένεια Zynq-7000 περιέχει τέσσερις τέτοιους διαύλους, στους οποίους master είναι πάντα το PL.

- **Acceleration Coherency Ports:** Δίαυλος δεδομένων 64bit επικοινωνίας του PL με τον Snoop Controller του επεξεργαστή ARM. Αυτή η σύνδεση επιτυγχάνει συνάφεια στις μνήμες cache L1, L2. Κατά αυτόν τον τρόπο ο επεξεργαστής ενημερώνεται άμεσα για αλλαγές που πραγματοποιεί το PL στις τιμές των δεδομένων. Η οικογένεια Zynq-7000 περιέχει έναν τέτοιο δίαυλο, όπου master είναι το PL.

2.4 High Level Synthesis (HLS)

Η σύνθεση υψηλού επιπέδου (HLS) είναι μία αυτοματοποιημένη διαδικασία σχεδιασμού που μεταφράζει έναν αλγόριθμο γραμμένο σε υψηλού επιπέδου γλώσσα, όπως C/C++, σε γλώσσα περιγραφής υλικού RTL (Register Transfer Level) και συντίθεται στο FPGA. Η διαδικασία αυτή πραγματοποιείται με το εργαλείο σχεδιασμού της Xilinx, Vivado HLS. Ο προγραμματιστής θα πρέπει να δομήσει τον αλγόριθμό του με τρόπο που οδηγεί σε αποδοτική παραλληλοποίηση και εκτέλεση, αναλογιζόμενος την δομή του FPGA.



Σχήμα 2.12: Vivado HLS

2.4.1 Βασικές Έννοιες Σχεδίασης Υλικού

Μία από τις σημαντικότερες διαφορές ενός επεξεργαστή με ένα FPGA είναι ότι η αρχιτεκτονική του επεξεργαστή είναι συγκεκριμένη. Η διαφορά αυτή επηρεάζει άμεσα το τρόπο που λειτουργεί ο μεταγλωττιστής στις δύο περιπτώσεις. Στην περίπτωση του επεξεργαστή, η υπολογιστική αρχιτεκτονική είναι συγκεκριμένη και σκοπός του μεταγλωττιστή είναι να ταιριάξει καλύτερα την εφαρμογή λογισμικού στις υπάρχουσες επεξεργαστικές δομές. Σε αντίθεση, στην περίπτωση του FPGA, ο HLS μεταγλωττιστής δομεί την επεξεργαστική αρχιτεκτονική ώστε να ταιριάζει στην εφαρμογή λογισμικού. Η διαδικασία κατεύθυνσης του HLS μεταγλωττιστή στην δημιουργία της επεξεργαστικής δομής, απαιτεί εις βάθος γνώση στις έννοιες της σχεδίασης υλικού [4][5].

Συχνότητα Ρολογιού

Η συχνότητα ρολογιού είναι από τα πρώτα πράγματα που σκεφτόμαστε όταν δημιουργούμε την πλατφόρμα εκτέλεσης ενός συγκεκριμένου αλγορίθμου. Στοχεύουμε συνήθως σε υψηλές συχνότητες ρολογιού που μεταφράζονται σε ταχύτερη εκτέλεση του αλγορίθμου. Οι επεξεργαστές κυμαίνονται σε συχνότητες ρολογιού άνω των 2GHz, ενώ τα FPGA δεν ξεπερνούν τα 500MHz. Αναλύοντας περαιτέρω τις δύο πλατφόρμες, η διαφορά τους δεν έγκειται μόνο στην συχνότητα του ρολογιού. Η μεγάλη διαφορά βρίσκεται στον τρόπο που εκτελείται ένα πρόγραμμα λογισμικού στις δύο πλατφόρμες. Αναφερόμενοι στους επεξεργαστές, ο μεταγλωττιστής, γνωρίζοντας την αρχιτεκτονική του επεξεργαστή, συντάσσει το λογισμικό του χρήστη σε ένα σύνολο εντολών. Το σετ εντολών εκτελείται πάντα σε αυτήν την δομική σειρά όπως φαίνεται και στο σχήμα 2.13.

Σχήμα 2.13: Στάδια Εκτέλεσης Εντολών Επεξεργαστή

Ανεξαρτήτως του τύπου του επεξεργαστή, η εκτέλεση των εντολών είναι πάντα ίδια. Κάθε εντολή πρέπει να εκτελέσει τα στάδια του σχήματος 2.13.

- **Instruction Fetch (IF):** Φόρτωση εντολής από τη μνήμη.

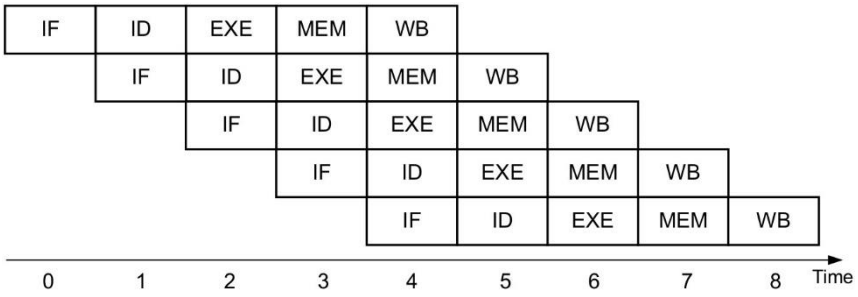
- **Instruction Decode (ID):** Αποκωδικοποίηση της εντολής, καθορισμός της διαδικασίας και των ορισμάτων της.

- **Execution (EXE):** Εκτέλεση της εντολής σε διαθέσιμο υλικό, συγκεκριμένα στην Αριθμητική Λογική Μονάδα (ALU) ή στην Μονάδα Κινητής Υποδιαστολής (FPU).

- **Memory Operations (MEM):** Λήψη δεδομένων για την επόμενη εντολή χρησιμοποιώντας χειριστές μνήμης.

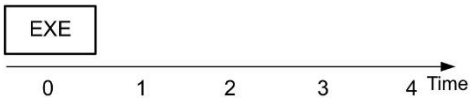
- **Write Back (WB):** Εγγραφή των αποτελεσμάτων της εντολής είτε σε τοπικούς καταχωρητές είτε στην μνήμη.

Οι περισσότεροι επεξεργαστές περιλαμβάνουν πολλαπλές μονάδες εκτέλεσης των σταδίων των εντολών και μπορούν να τις εκτελούν μέχρι ενός βαθμού επικάλυψης. Επειδή όμως οι εντολές αλληλοεξαρτώνται συνήθως μεταξύ τους, περιορίζεται αυτή η επικάλυψη. Τα στάδια EXE, που είναι υπεύθυνα για τον υπολογισμό της εφαρμογής, εκτελούνται διαδοχικά. Η διαδοχική αυτή εκτέλεση οφείλεται στους περιορισμένους πόρους στο στάδιο EXE και στην εξάρτηση μεταξύ των εντολών. Στο σχήμα 2.14 βλέπουμε την καλύτερη περίπτωση εκτέλεσης εντολών σε επεξεργαστή. Η τεχνική αυτή ονομάζεται pipelining (διοχέτευση). Ακόμα και αν ο μεταγλωττιστής αποφάσιζε ότι πολλά στάδια EXE μπορούν να εκτελεστούν ταυτόχρονα, η δομή των σετ εντολών δεν θα το επέτρεπε. Αποτέλεσμα αυτών είναι ότι στον επεξεργαστή, μέχρι μία εντολή μπορεί να ολοκληρώνεται ανά κύκλο ρολογιού.



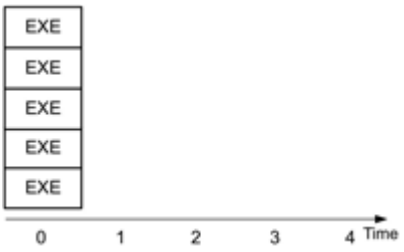
Σχήμα 2.14: Επεξεργαστής με Πολλαπλές Μονάδες Εκτέλεσης Σταδίων

Το FPGA δεν εκτελεί όλο το πρόγραμμα σε μια κοινή υπολογιστική πλατφόρμα. Εκτελεί μια εντολή σε ένα διαμορφωμένο κύκλωμα για αυτήν την εφαρμογή. Κατά αυτόν τον τρόπο αλλάζοντας το πρόγραμμα, αλλάζει και το κύκλωμα του FPGA. Η παρουσία του σταδίου MEM εξαρτάται από την εφαρμογή.



Σχήμα 2.15: Στάδια Εκτέλεσης Εντολών FPGA

Λόγω τις ευελιξίας του, ο HLS μεταγλωττιστής δεν χρειάζεται να δομήσει το υλικό με βάση συγκεκριμένα στάδια, όπως στον επεξεργαστή, αλλά το δομεί όπως αυτός κρίνει. Μπορεί λοιπόν να βρει τρόπους για μεγαλύτερη παραλληλία.



Σχήμα 2.16: FPGA με Πολλαπλές Μονάδες Εκτέλεσης

Ένα άλλο ζήτημα, που προκύπτει από την συχνότητα ρολογιού, είναι της κατανάλωσης ενέργειας. Μία προσέγγιση της κατανάλωσης ενέργειας δίνεται από την σχέση 2.1 . Όπου '*P* ' η κατανάλωση ενέργειας, όπου '*cF* ' η συχνότητα ρολογιού και όπου '*V*' η ηλεκτρική τάση.

$$P = \frac{1}{2} cFV^2 \qquad (2.1)$$

Χρονοδρομολόγηση

Η χρονοδρομολόγηση είναι η διαδικασία ταυτοποίησης των δεδομένων και εξαρτήσεων ελέγχου μεταξύ διαφορετικών λειτουργιών για να προσδιοριστεί πότε θα εκτελείται κάθε μία από αυτές. Στην παραδοσιακή σχεδίαση FPGA, αυτή είναι μια χειροκίνητη διαδικασία που επίσης αναφέρεται ως παραλληλισμός του αλγορίθμου λογισμικού για υλοποίηση υλικού.

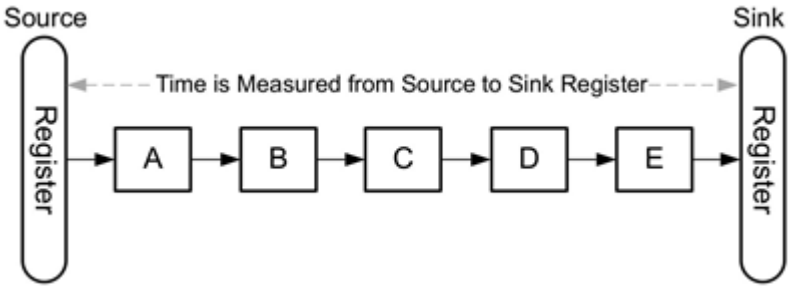
Ο μεταγλωττιστής HLS αναλύει τις εξαρτήσεις μεταξύ παρακείμενων λειτουργιών καθώς και για όλη την διάρκεια εκτέλεσης. Αυτό επιτρέπει στον μεταγλωττιστή να ομαδοποιεί λειτουργίες που εκτελούνται στον ίδιο κύκλο ρολογιού και να ρυθμίζει το υλικό ώστε να επιτρέπει την επικάλυψη κλήσεων λειτουργίας. Η επικάλυψη των εκτελέσεων κλήσεων λειτουργίας καταργεί τον περιορισμό του επεξεργαστή που απαιτεί την ολοκλήρωση της κλήσης της τρέχουσας λειτουργίας πριν να ξεκινήσει η επόμενη κλήση λειτουργίας στο ίδιο σύνολο λειτουργιών. Αυτή η διαδικασία ονομάζεται pipelining.

Latency & Pipelining

Latency είναι η καθυστέρηση δράσης, ο αριθμός των κύκλων ρολογιού που χρειάζεται για να ολοκληρωθεί ένα σετ εντολών και να παραχθεί μία τιμή αποτελέσματος της εφαρμογής. Χρησιμοποιώντας τη βασική δομή εκτέλεσης εντολών σε επεξεργαστή του σχήματος 2.13, έχουμε Latency πέντε κύκλους ρολογιού. Εάν η εφαρμογή είχε 5 εντολές, θα είχαμε συνολικό Latency 25 κύκλους σε απλό μοντέλο (χωρίς pipeline).

Το Latency των εφαρμογών αποτελεί βασική μέτρηση της απόδοσης και στους επεξεργαστές και στα FPGA. Σε αμφότερες τις περιπτώσεις το Latency βελτιώνεται με τη χρήση Pipelining. Στους επεξεργαστές το Pipelining σημαίνει ότι η επόμενη εντολή μπορεί να τεθεί σε εκτέλεση πριν τελειώσει η τρέχουσα. Αυτό επιτρέπει την επικάλυψη των σταδίων που απαιτούνται στην εκτέλεση των σετ εντολών. Το καλύτερο αποτέλεσμα χρήσης pipeline, φαίνεται στο σχήμα 2.14. Σε αυτήν την περίπτωση ο επεξεργαστής επιτυγχάνει Latency εννέα κύκλων ρολογιού για μία εφαρμογή πέντε εντολών. Σε ένα FPGA, οι κύκλοι επιβάρυνσης (Overhead Cycles) που σχετίζονται με την επεξεργασία εντολών δεν υπάρχουν. Το Latency μετράται με τον αριθμό των κύκλων ρολογιού που χρειάζεται για να εκτελεστεί το στάδιο EXE, της πρώτυπης εντολής του επεξεργαστή. Στην περίπτωση του σχήματος 2.15 έχουμε Latency έναν κύκλο. Ο παραλληλισμός παίζει επίσης σημαντικό ρόλο στην καθυστέρηση. Για την ολοκλήρωση μιας εφαρμογής πέντε εντολών, σε FPGA, έχουμε Latency επίσης ένα κύκλο ρολογιού, όπως φαίνεται και στο σχήμα 2.16. Με την καθυστέρηση ενός κύκλου ρολογιού στο FPGA, ίσως δεν είναι ξεκάθαρο γιατί το Pipeline είναι πλεονεκτικό. Ωστόσο, ο λόγος για τον οποίο χρησιμοποιείται το Pipeline σε ένα FPGA είναι ο ίδιος όπως σε έναν επεξεργαστή, να βελτιωθεί η εκτέλεση της εφαρμογής.

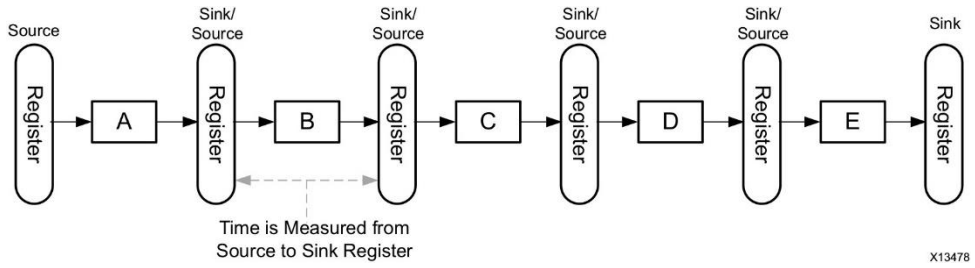
Κατά την υλοποίηση του υλικού στο FPGA, οι εντολές δομούνται σε ένα κύκλωμα. Κάθε εντολή αντιστοιχεί σε ένα μονοπάτι - υποκύκλωμα. Αυτά τα μονοπάτια ορίζονται ανάμεσα σε έναν καταχωρητή προέλευσης και σε έναν καταχωρητή προορισμού (Sink Register). Ο χρόνος μετάδοσης του σήματος σε αυτά τα μονοπάτια καθορίζει και τον κύκλο του ρολογιού. Όπως προαναφέρθηκε και παραπάνω, στα FPGA, οι εντολές εκτελούνται σε ένα κύκλο ρολογιού. Κατά αυτόν τον τρόπο, η πιο αργή εντολή καθορίζει την συχνότητα ρολογιού. Στο παράδειγμα του σχήματος 2.17, αν κάθε δομή του μονοπατιού χρειάζεται 2ns, θα χρειαστούν 10ns για την εκτέλεση του μονοπατιού. Έτσι η συχνότητα ρολογιού που θα εκτελεστεί το κύκλωμα είναι 100MHz.



Σχήμα 2.17: FPGA, Υλοποίηση Χωρίς Pipeline

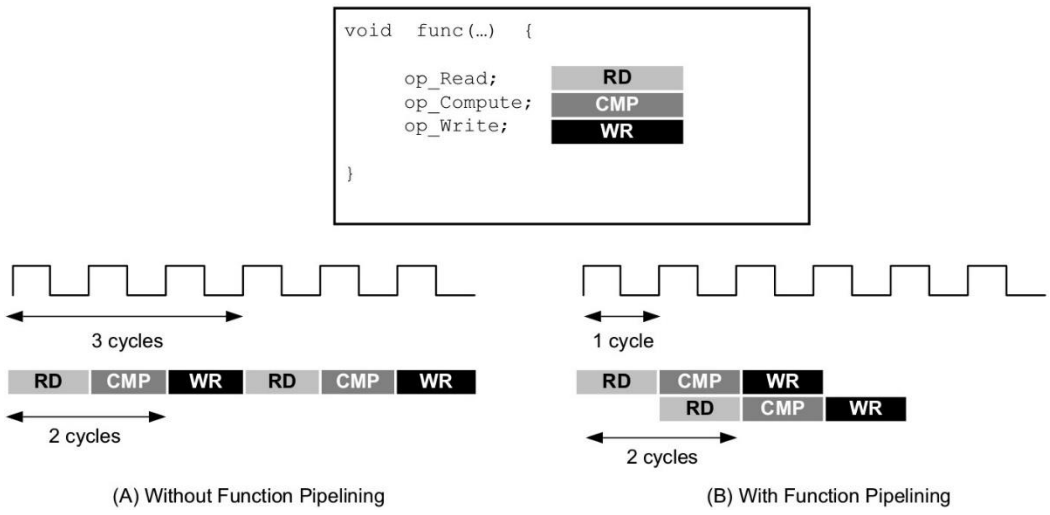
Το Pipelining είναι μια τεχνική ψηφιακού σχεδιασμού που επιτρέπει στον σχεδιαστή να αποφεύγει εξαρτήσεις δεδομένων και να αυξάνει το επίπεδο παραλληλισμού σε μια υλοποίηση υλικού του αλγορίθμου. Η εξάρτηση δεδομένων στην αρχική υλοποίηση του λογισμικού διατηρείται για λειτουργική ισοδυναμία, αλλά το απαιτούμενο κύκλωμα διαιρείται σε μια αλυσίδα ανεξάρτητων σταδίων. Όλα τα στάδια της αλυσίδας λειτουργούν παράλληλα στον ίδιο κύκλο ρολογιού. Η μόνη διαφορά είναι η πηγή δεδομένων για κάθε στάδιο. Κάθε στάδιο του υπολογισμού, λαμβάνει τις τιμές δεδομένων του από το αποτέλεσμα που υπολογίστηκε από το προηγούμενο στάδιο, κατά τον προηγούμενο κύκλο ρολογιού.

Για την υλοποίηση του pipeline στο FPGA, προστίθενται ενδιάμεσοι καταχωρητές ώστε να διαιρέσουν τα μεγάλα υπολογιστικά μονοπάτια σε μικρότερα. Αυτή η διαίρεση αυξάνει το Latency σε απόλυτο αριθμό, αλλά αυξάνει την απόδοση επιτρέποντας στο κύκλωμα να τρέξει σε μεγαλύτερη συχνότητα ρολογιού. Έτσι αν υλοποιήσουμε Pipeline στο μονοπάτι του σχήματος 2.17, παράγουμε το σχήμα 2.18. Δεχόμαστε ξανά ότι κάθε δομή χρειάζεται 2ns για να εκτελεστεί. Τώρα κάθε δομή αποτελεί ένα μονοπάτι και καθορίζει τον κύκλο του ρολογιού. Σε αυτήν την περίπτωση επιτυγχάνουμε συχνότητα ρολογιού 500MHz.



Σχήμα 2.18: FPGA, Υλοποίηση με Pipeline

Το Pipeline χρησιμοποιείται μέσα σε συναρτήσεις ή βρόχους για να βελτιώσει τον αριθμό throughput.



Σχήμα 2.19: Pipeline

Εδώ αξίζει να αναφερθούμε σε έναν σημαντικό όρο, ίσως το σημαντικότερο, σχετικά με την υλοποίηση pipelining σε επαναληπτικούς βρόγχους, το Initiation Interval (II) ή Διάστημα Εκκίνησης. Το (II), όπως θα αναφέρεται από δω και στο εξής, είναι ο αριθμός των κύκλων ρολογιού μεταξύ των αρχικών εντολών συνεχόμενων επαναλήψεων του βρόγχου. Δεδομένου αυτού μπορούμε να συμπεράνουμε ότι, χωρίς να υπάρχουν παράλληλα pipelines, πως το βέλτιστο (II) ισούται με 1, όπως φαίνεται παραπάνω στο σχήμα 19.

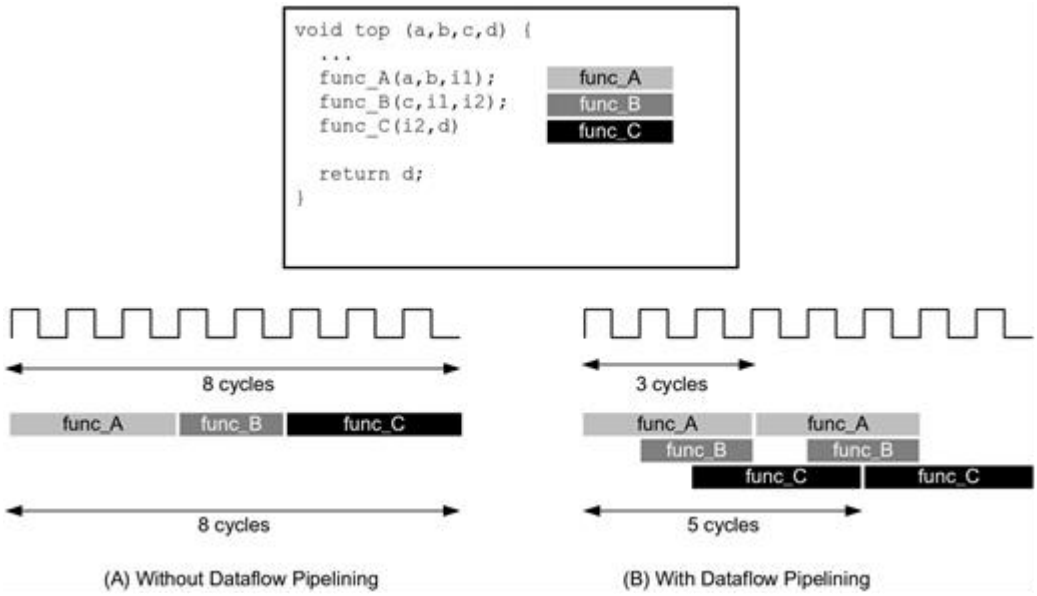
Dataflow

Το Dataflow είναι μια άλλη τεχνική ψηφιακού σχεδιασμού, η οποία είναι παρόμοια με την έννοια της αγωγιμότητας. Ο στόχος της ροής δεδομένων είναι να εκφράσει τον παραλληλισμό σε ένα coarse-grain επίπεδο. Στον coarse-grained παραλληλισμό το πρόγραμμα χωρίζεται σε επιμέρους μεγάλα υπολογιστικά τμήματα. Όσον αφορά την εκτέλεση του λογισμικού, αυτή η μετατροπή ισχύει για την παράλληλη εκτέλεση λειτουργιών μέσα σε ένα πρόγραμμα. Το HLS επιτυγχάνει αυτό το επίπεδο παραλληλισμού αξιολογώντας τις αλληλεπιδράσεις μεταξύ διαφορετικών λειτουργιών ενός προγράμματος με βάση τις εισόδους και τις

εξόδους τους. Η απλούστερη περίπτωση παραλληλισμού είναι όταν οι λειτουργίες αφορούν διαφορετικά σύνολα δεδομένων και δεν επικοινωνούν μεταξύ τους. Σε αυτή την περίπτωση, το HLS διαθέτει τους λογικούς πόρους του FPGA για κάθε λειτουργία και στη συνέχεια εκτελεί τα μπλοκ ανεξάρτητα. Η πιο περίπλοκη περίπτωση, η οποία είναι χαρακτηριστική στα προγράμματα λογισμικού, είναι όταν μία λειτουργία παρέχει αποτελέσματα για άλλη λειτουργία. Η περίπτωση αυτή αναφέρεται ως σενάριο καταναλωτή-παραγωγού.

Το HLS υποστηρίζει δύο μοντέλα χρήσης για το σενάριο καταναλωτή-παραγωγού. Στο πρώτο μοντέλο χρήσης, ο παραγωγός δημιουργεί ένα πλήρες σύνολο δεδομένων πριν αρχίσει να λειτουργεί ο καταναλωτής. Ο παραλληλισμός επιτυγχάνεται δημιουργώντας ένα ζεύγος μνημών BRAM ως τράπεζες μνήμης. Κάθε λειτουργία μπορεί να έχει πρόσβαση μόνο σε μία τράπεζα μνήμης για τη διάρκεια μιας κλήσης λειτουργίας. Όταν αρχίσει μια νέα κλήση λειτουργίας, το κύκλωμα που παράγεται από το HLS μεταβαίνει στις συνδέσεις μνήμης τόσο για τον παραγωγό όσο και για τον καταναλωτή. Αυτή η προσέγγιση εγγυάται τη λειτουργική ορθότητα αλλά περιορίζει το επίπεδο του εφικτού παραλληλισμού σε όλες τις κλήσεις λειτουργίας. Στο δεύτερο μοντέλο χρήσης, ο καταναλωτής μπορεί να αρχίσει να εργάζεται με μερικά αποτελέσματα από τον παραγωγό και το επιτεύξιμο επίπεδο παραλληλισμού επεκτείνεται ώστε να συμπεριλαμβάνει την εκτέλεση μέσα σε μια κλήση λειτουργίας. Οι μονάδες που παράγονται με HLS και για τις δύο λειτουργίες συνδέονται μέσω της χρήσης ενός κυκλώματος FIFO. Αυτό το κύκλωμα μνήμης, το οποίο λειτουργεί ως ουρά στον προγραμματισμό του λογισμικού, παρέχει συγχρονισμό σε επίπεδο δεδομένων μεταξύ των μονάδων. Σε οποιοδήποτε σημείο κατά τη διάρκεια μιας κλήσης λειτουργίας, και οι δύο μονάδες υλικού εκτελούν τον προγραμματισμό τους. Η μόνη εξαίρεση είναι ότι η ενότητα καταναλωτών αναμένει ορισμένα στοιχεία να είναι διαθέσιμα από τον παραγωγό πριν από τον υπολογισμό. Στην ορολογία HLS, ο χρόνος αναμονής του καταναλωτή αναφέρεται ως το διάστημα έναρξης (Initiation Interval - II).

Το Dataflow χρησιμοποιείται σαν Pipeline επικαλύπτοντας συναρτήσεις και βρόχους, αντί για εντολές όπως έχουμε στην περίπτωση του απλού Pipeline.



Σχήμα 2.20: Dataflow

Throughput

Το throughput αποτελεί μία ακόμα μονάδα μέτρησης της συνολικής απόδοσης της υλοποίησης μας. Ισούται με τον αριθμό των κύκλων ρολογιού που χρειάζεται η προγραμματιζόμενη λογική για να δεχτεί κάθε επόμενο δείγμα δεδομένων εισόδου. Με αυτήν την τιμή, είναι σημαντικό να καταλάβουμε ότι η συχνότητα ρολογιού του κυκλώματος αλλάζει την έννοια του μεγέθους του throughput.

Για παράδειγμα, τόσο το σχήμα 2.17 όσο και το σχήμα 2.18 δείχνουν εφαρμογές που απαιτούν έναν κύκλο ρολογιού μεταξύ δειγμάτων δεδομένων εισόδου. Η βασική διαφορά είναι ότι η υλοποίηση στο σχήμα 2.17 απαιτεί 10ns μεταξύ των δειγμάτων εισόδου, ενώ το κύκλωμα στο σχήμα 2.18 απαιτεί μόνο 2ns μεταξύ των δειγμάτων δεδομένων εισόδου. Αφού γίνει γνωστή η χρονική βάση, είναι σαφές ότι η δεύτερη εφαρμογή έχει υψηλότερη απόδοση, επειδή μπορεί να δεχθεί υψηλότερο ρυθμό δεδομένων εισόδου.

Αρχιτεκτονική και Διάταξη Μνήμης

Η αρχιτεκτονική της μνήμης της επιλεγμένης πλατφόρμας υλοποίησης, είναι ένα από τα φυσικά στοιχεία που μπορούν να επηρεάσουν την απόδοση μιας εφαρμογής λογισμικού. Η αρχιτεκτονική μνήμης καθορίζει το ανώτερο όριο της επιτεύξιμης απόδοσης. Σε κάποιο σημείο εκτέλεσης, όλες οι εφαρμογές είτε σε επεξεργαστή είτε σε FPGA δεσμεύονται από την μνήμη ανεξάρτητα από τον τύπο και τον αριθμό των διαθέσιμων υπολογιστικών πόρων. Μια στρατηγική στο σχεδιασμό FPGA είναι η κατανόηση του που βρίσκεται η μνήμη και του πώς μπορεί να επηρεαστεί από την διάταξη δεδομένων και την οργάνωση της μνήμης.

Σε ένα σύστημα βασισμένο σε επεξεργαστή, ο μηχανικός του λογισμικού πρέπει να εφαρμόζει την εφαρμογή σε ίδια αρχιτεκτονική μνήμης ανεξάρτητα από τον συγκεκριμένο τύπο επεξεργαστή. Αυτή η στρατηγική απλοποιεί τη διαδικασία της ενσωμάτωσης των εφαρμογών σε διαφορετικά συστήματα, σε βάρος της απόδοσης. Η κοινή αρχιτεκτονική μνήμης που είναι εξοικειωμένη με τους μηχανικούς λογισμικού αποτελείται από μνήμες που είναι αργές, μεσαίες ή γρήγορες με βάση τον αριθμό των κύκλων ρολογιού που χρειάζονται για να μεταφέρουν τα δεδομένα στον επεξεργαστή. Στις αργές μνήμες έχουμε τις συσκευές μαζική αποθήκευσης, όπως οι σκληροί δίσκοι. Στις μεσαίες έχουμε τις μνήμες DDR και στις γρήγορες, τις cache μνήμες που βρίσκονται στο chip του επεξεργαστή. Σε αυτή την αρχιτεκτονική μνήμης ο χρήστης αντιλαμβάνεται έναν ενιαίο μεγάλο χώρο μνήμης. Μέσα σε αυτό το χώρο μνήμης ο χρήστης κατανέμει και ανακατευθύνει περιοχές για την αποθήκευση δεδομένων του προγράμματος. Η φυσική τοποθεσία των δεδομένων και ο τρόπος με τον οποίο μετακινούνται μεταξύ των διαφόρων επιπέδων της ιεραρχίας μνήμης, χειρίζονται από την υπολογιστική πλατφόρμα και η διαδικασία είναι διαφανής για τον χρήστη. Σε αυτό το είδος του συστήματος, ο μόνος τρόπος για να αυξήσουμε την απόδοση είναι να επαναχρησιμοποιούμε όσο το δυνατόν περισσότερο τα δεδομένα στην κρυφή μνήμη.

Η πρώτη διαφορά που συναντά ένας μηχανικός λογισμικού όταν διευθετεί την μνήμη σε ένα FPGA, είναι η έλλειψη σταθερής αρχιτεκτονικής μνήμης on-chip. Τα συστήματα που βασίζονται σε FPGA μπορούν να συνδεθούν σε αργές και μεσαίες μνήμες, αλλά παρουσιάζουν τον μεγαλύτερο βαθμό διαφοροποίησης όσον αφορά τις διαθέσιμες γρήγορες μνήμες. Πιο συγκεκριμένα, αντί να αναδιαρθρώνει το λογισμικό για να χρησιμοποιήσει καλύτερα μια υπάρχουσα μνήμη cache, ο HLS μεταγλωττιστής δημιουργεί μια γρήγορη αρχιτεκτονική μνήμης για να ταιριάζει καλύτερα στη διάταξη δεδομένων στον αλγόριθμο. Η προκύπτουσα υλοποίηση FPGA μπορεί να έχει μία ή περισσότερες εσωτερικές τράπεζες (υλοποιημένα μικρά κομμάτια μνήμης) διαφορετικών μεγεθών που μπορούν να προσπελαστούν ανεξάρτητα η μία από την άλλη.

Ο κώδικας FPGA δεν δύναται να υλοποιήσει δυναμική κατανομή μνήμης. Η χρήση της δυναμικής κατανομής μνήμης είναι βέλτιστης πρακτικής για συστήματα με βάση τον επεξεργαστή, λόγω της καθορισμένης αρχιτεκτονικής της μνήμης. Αντίθετα στην περίπτωση των FPGA, ο μεταγλωττιστής HLS δημιουργεί μια αρχιτεκτονική μνήμης προσαρμοσμένη στην εφαρμογή. Αυτή η προσαρμοσμένη αρχιτεκτονική μνήμης διαμορφώνεται τόσο από το μέγεθος των μπλοκ μνήμης στο πρόγραμμα όσο και από τον τρόπο με τον οποίο τα δεδομένα χρησιμοποιούνται καθ' όλη τη διάρκεια εκτέλεσης του προγράμματος. Οι σύγχρονοι μεταγλωττιστές τελευταίας γενιάς για FPGA, όπως το HLS, απαιτούν οι απαιτήσεις μνήμης μιας εφαρμογής να αναλύονται πλήρως κατά τον χρόνο σύνταξης. Το πλεονέκτημα της κατανομής στατικής μνήμης είναι ότι ο μεταγλωττιστής HLS μπορεί να διαμορφώσει τη μνήμη για έναν στατικό πίνακα με διαφορετικούς τρόπους.

Ανάλογα με τους υπολογισμούς που πραγματοποιούνται κατά τη διάρκεια εκτέλεσης του αλγορίθμου, ο μεταγλωττιστής HLS μπορεί να υλοποιήσει τη μνήμη για τον πίνακα ως καταχωρητές, καταχωρητές μετατόπισης, FIFOs ή BRAM.

2.4.2 Μεθοδολογία Βελτιστοποίησης

Το Vivado HLS παρέχει μια σειρά οδηγιών βελτιστοποίησης και διαμορφώσεων που χρησιμοποιούνται για την κατεύθυνση της σύνθεσης προς το επιθυμητό αποτέλεσμα. Με αυτές τις οδηγίες μπορούμε να προκαθορίσουμε συνολικά την αρχιτεκτονική του υλικού που θέλουμε να σχεδιάσουμε. Στο Vivado HLS οι οδηγίες αυτές δίνονται με την εντολή #pragma και κατευθύνουν τον μεταγλωττιστή σχετικά με την αρχιτεκτονική του υλικού.

Στον παρακάτω πίνακα παραθέτουμε με σειρά τα βήματα της μεθοδολογίας για βελτιστοποίηση της Σχεδίασης Υψηλού Επιπέδου.

Προσομοίωση Σχεδίασης	Επαλήθευση του C κώδικα
Σύνθεση Σχεδίασης	Αρχική Σχεδίαση
Αρχικές Βελτιστοποιήσεις	Καθορισμός Διεπαφών
	Καθορισμός επαναλήψεων βρόχων
Pipeline	Pipeline & Dataflow
Βελτιστοποίηση δομών	Διαμέριση μνημών και θυρών
	Αφαίρεση ψευδών εξαρτήσεων
Μείωση Latency	Προαιρετικός καθορισμός απαιτήσεων του Latency
Βελτιστοποίηση του χώρου	Προαιρετική ανάκτηση πόρων μέσω κοινής χρήσης

Πίνακας 2.1: Μεθοδολογία Βελτιστοποίησης Σχεδίασης

Αφού πρώτα έχουμε επαληθεύσει τον αλγόριθμο μας γραμμένο σε C/C++ και έχουμε κάνει μία πρώτη σύνθεση του υλικού, αρχίζουμε να διευθετούμε την αρχιτεκτονική. Παρακάτω αναλύουμε τα directives (οδηγίες προς τον compiler).

Αρχικές Βελτιστοποιήσεις

- **INTERFACE -** Καθορίζει πώς δημιουργούνται οι θύρες RTL από την περιγραφή της συνάρτησης.
- **DATA_PACK -** Συσκευάζει τα πεδία δεδομένων μιας δομής σε μία μεταβλητή με ευρύτερο πλάτος λέξης.
- **LOOP_TRIPCOUNT -** Χρησιμοποιείται για βρόχους που έχουν μεταβλητά όρια. Παρέχει μια εκτίμηση για τον αριθμό επανάληψης του βρόχου. Αυτό δεν έχει αντίκτυπο στη σύνθεση, μόνο στην αναφορά.
- **Config Interface -** Αυτή η διαμόρφωση ρυθμίζει τις θύρες εισόδου/εξόδου που δεν σχετίζονται με τα ορίσματα της top-level συνάρτησης και επιτρέπει την εξάλειψη των αχρησιμοποίητων θυρών από το τελικό RTL.

Η διεπαφή σχεδιασμού ορίζεται συνήθως από τα άλλα μπλοκ του συστήματος. Δεδομένου ότι ο τύπος του πρωτοκόλλου εισόδου/εξόδου βοηθάει να προσδιοριστεί τι μπορεί να επιτευχθεί με σύνθεση, συνιστάται η χρήση της οδηγίας **INTERFACE** για να διευκρινιστεί αυτό πριν προχωρήσουμε στη βελτιστοποίηση του σχεδιασμού.

Pipeline

Σε αυτό το στάδιο της διαδικασίας βελτιστοποίησης θέλουμε να δημιουργήσουμε όσο το δυνατόν περισσότερη ταυτόχρονη λειτουργία. Μπορούμε να εφαρμόσουμε την οδηγία **PIPELINE** σε λειτουργίες και βρόχους. Επίσης μπορούμε να χρησιμοποιήσουμε την οδηγία **DATAFLOW** στο επίπεδο που περιέχει τις λειτουργίες και τους βρόχους ώστε να λειτουργούν παράλληλα.

- **PIPELINE -** Μειώνει το διάστημα έναρξης (Initiation Interval - II) επιτρέποντας την ταυτόχρονη εκτέλεση λειτουργιών εντός ενός βρόχου ή μιας λειτουργίας.
- **DATAFLOW -** Επιτρέπει το pipeline σε επίπεδο εργασίας, επιτρέποντας την εκτέλεση λειτουργιών και βρόχων ταυτόχρονα. Χρησιμοποιείται για την ελαχιστοποίηση του διαστήματος έναρξης (II).
- **RESOURCE -** Καθορίζει ένα πακέτο πόρων συστήματος (core) που χρησιμοποιείται για την υλοποίηση μιας μεταβλητής (πίνακα, αριθμητική λειτουργία, ή όρισμα συνάρτησης) στο RTL.
- **Config Compile -** Επιτρέπει στους βρόχους να κάνουν αυτόματα pipeline με βάση την επανεξέταση τους.

Βελτιστοποίηση Δομών

Ο κώδικας C μπορεί να περιέχει περιγραφές που εμποδίζουν την εκτέλεση μιας διαδικασίας ή βρόχου με την απαιτούμενη απόδοση. Σε ορισμένες περιπτώσεις, αυτό ενδέχεται να απαιτεί τροποποίηση κώδικα, αλλά στις περισσότερες περιπτώσεις τα ζητήματα αυτά μπορούν να αντιμετωπιστούν χρησιμοποιώντας άλλες οδηγίες βελτιστοποίησης.

- **ARRAY_PARTITION -** Διαμέριση μεγάλων πινάκων σε πολλούς μικρότερους ή σε ξεχωριστούς καταχωρητές, για τη βελτίωση της πρόσβασης στα δεδομένα και την απομάκρυνση των σημείων συμφόρησης της RAM.
- **DEPEDENCE -** Χρησιμοποιείται για την παροχή πρόσθετων πληροφοριών που μπορούν να ξεπεράσουν τις εξαρτήσεις βρόχων και να επιτρέψουν το pipeline των βρόχων, ακόμα και με μεγαλύτερο διάστημα έναρξης (II).
- **INLINE -** Χρησιμοποιείται αντί της απλής κλήσης μιας συνάρτησης. Με την οδηγία **INLINE** ο compiler ξαναδομεί την συνάρτηση στο σημείο που είναι το **INLINE**. Αυτή η τακτική χρησιμοποιείται για την ενεργοποίηση της βελτιστοποίησης της λογικής μεταξύ των συνόρων των συναρτήσεων και για τη βελτίωση των latencies και (II) μειώνοντας το κόστος των κλήσεων συναρτήσεων.
- **UNROLL -** Ξετυλίγει τους βρόχους για να δημιουργήσει πολλαπλές λειτουργίες αντί για μία μόνο συλλογή λειτουργιών.
- **Config Array Partition -** Αυτή η διαμόρφωση προσδιορίζει τον τρόπο κατανομής των πινάκων, συμπεριλαμβανομένων των Global πινάκων και εάν η διαμέριση επηρεάζει τις θύρες αυτών.
- **Config Compile -** Ελέγχει τη σύνθεση συγκεκριμένων βελτιστοποιήσεων όπως είναι οι αυτόματες βελτιστοποιήσεις pipeline σε βρόχους, όπως και σε μαθηματικές πράξεις κινούμενης υποδιαστολής.
- **Config Schedule -** Καθορίζει το επίπεδο της βελτιστοποίησης που πρέπει να χρησιμοποιηθεί κατά τη διάρκεια της φάσης χρονοδρομολόγησης της σύνθεσης. Διευθετεί την τιμή του (II) στο pipeline με σκοπό την επίτευξη του χρονοδιαγράμματος. Ιδανικά το (II) ισούται με 1. Ακόμα, καθορίζει την δομή των μηνυμάτων εξόδου.
- **CONFIG_UNROLL -** Επιτρέπει σε όλες τις επαναλήψεις βρόχων κάτω από έναν αριθμό να ξετιλυχτούν αυτόματα.

Μείωση Latency

- **LATENCY -** Επιτρέπει τον καθορισμό περιορισμού ελάχιστου και μέγιστου Latency.
- **LOOP_FLATTEN -** Επιτρέπει την μετατροπή ενσωματωμένων βρόχων σε έναν βρόχο με βελτιωμένη καθυστέρηση.
- **LOOP_MERGE -** Συγχώνευση διαδοχικών βρόχων για να μειωθεί το συνολικό Latency, να αυξηθεί η κοινή χρήση και να βελτιωθεί η λογική.

Βελτιστοποίηση Χώρου

- **ALLOCATION -** Καθορίζει ένα όριο για τον αριθμό των λειτουργιών, πυρήνων ή λειτουργιών που χρησιμοποιούνται. Αυτό μπορεί να αναγκάσει την ανταλλαγή πόρων υλικού και μπορεί να αυξήσει το Latency
- **ARRAY_MAP -** Συνδυάζει πολλαπλούς μικρότερους πίνακες σε ένα μεγάλο πίνακα για τη μείωση των πόρων μνήμης RAM.

- **ARRAY_RESHAPE -** Μετασχηματίζει έναν πίνακα από ένα με πολλά στοιχεία σε ένα με μεγαλύτερο πλάτος λέξης. Χρήσιμο για τη βελτίωση των προσπελάσεων μπλοκ RAM χωρίς τη χρήση περισσότερης μνήμης RAM.
- **OCCURRENCE -** Χρησιμοποιείται σε περιπτώσεις pipeline σε συναρτήσεις ή βρόχους, για να καθορίσουμε ότι ο κώδικας σε μια περιοχή, εκτελείται με μικρότερο ρυθμό από τον κώδικα που περικλείει την συνάρτηση ή τον βρόχο.
- **STREAM -** Καθορίζει ότι ένα συγκεκριμένο κανάλι μνήμης πρόκειται να εφαρμοστεί ως FIFO ή RAM κατά τη βελτιστοποίηση της ροής δεδομένων.
- **Config Bind -** Καθορίζει το επίπεδο βελτιστοποίησης που χρησιμοποιείται κατά τη διάρκεια της φάσης binding της σύνθεσης και μπορεί να χρησιμοποιηθεί για να ελαχιστοποιήσει τον αριθμό των χρησιμοποιούμενων λειτουργιών.
- **Config Dataflow -** Αυτή η διαμόρφωση προσδιορίζει το προεπιλεγμένο κανάλι μνήμης και το βάθος FIFO στη βελτιστοποίηση ροής δεδομένων.

Κεφάλαιο 3

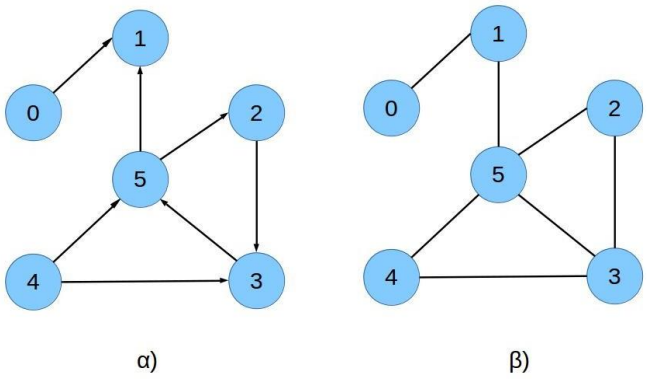
Connected Components – Shiloach-Vishkin

3.1 Γράφοι

3.1.1 Ορισμοί

Οι γράφοι είναι ένας αφηρημένος τρόπος για να αναπαραστήσουμε τις σχέσεις μεταξύ ενός συνόλου στοιχείων. Τα στοιχεία που συνθέτουν τους γράφους ονομάζονται κόμβοι ή κορυφές. Μερικά στοιχεία συνδέονται μεταξύ τους με δεσμούς που ονομάζονται ακμές. Θα αναφέρουμε ως γείτονες δύο κόμβους, όταν υπάρχει απευθείας ακμή που τους συνδέει. Δίνοντας έναν πιο αυστηρό ορισμό, ένας γράφος είναι ένα διατεταγμένο ζεύγος $G = (V,E)$, που αποτελείται από ένα σύνολο κόμβων $V = \{1,\ldots,N\}$ και ένα σύνολο ακμών $E = \{ (i,j) : i, j \in V\}$ που συνδέουν τους κόμβους μεταξύ τους. Μία ακμή από τον κόμβο *i* στον κόμβο *j* συμβολίζεται και ως *ij*.

Υπάρχουν δύο κατηγορίες γράφων, οι κατευθυνόμενοι και οι μη κατευθυνόμενοι. Αν υπάρχει συμμετρία μεταξύ των κόμβων, δηλαδή αν με την ύπαρξη ακμής από τον κόμβο *i* στον κόμβο *j*, συνεπάγεται και η ύπαρξη της ακμής *ji*, τότε ο γράφος ονομάζεται *μη κατευθυνόμενος*. Αντίθετα, όταν δεν υπάρχει αυτή η συμμετρία, ο γράφος ονομάζεται *κατευθυνόμενος*.



Σχήμα 3.1: Παράδειγμα κατευθυνόμενου και μη κατευθυνόμενου γράφου αντίστοιχα

3.1.2 Αναπαράσταση γράφου

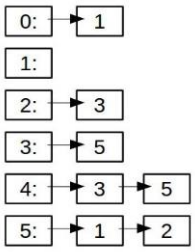
Οι μέθοδοι που μπορούν να αναπαρασταθούν οι γράφοι είναι αρκετοί. Οι πιο σύνηθεις είναι με χρήση πίνακα γειτνίασης, με λίστα γειτνίασης και με μη διατεταγμένη ακολουθία ακμών. Παρακάτω θα χρησιμοποιήσουμε το παράδειγμα του σχήματος 1 και συγκεκριμένα τον κατευθυνόμενο γράφο α) για την περιγραφή των μεθόδων.

- Για την αναπαράσταση ενός γράφου *n-κόμβων* με πίνακα γειτνίασης δημιουργούμε έναν πίνακα διαστάσεων *nxn*. Όπου κάθε στοιχείο *ij* αντιπροσωπεύει την ύπαρξη ακμής από τον κόμβο *i* στον κόμβο *j*. Συνεπώς όταν υπάρχει η ακμή θέτουμε το στοιχείο *ij* του πίνακα ίσο με 1 και όταν δεν υπάρχει ίσο με το 0. Κατά αυτόν τον τρόπο στο παράδειγμα μας διαμορφώνεται όπως φαίνεται κάτωθι:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Σχήμα 3.2: Παράδειγμα πίνακα γειτνίασης

- Με την χρήση ***λίστας γειτνίασης*** για να αναπαραστήσουμε έναν γράφο *n-κόμβων*, δημιουργούμε μία λίστα αποτελούμενη από *n* λίστες. Σε κάθε κόμβο *i* αντιστοιχεί μία λίστα με τους γειτονικούς κόμβους *j*, όπου υπάρχουν οι ακμές *ij*. Συνεπώς το παράδειγμά μας αναπαρίσταται όπως φαίνεται παρακάτω.



Σχήμα 3.3: Παράδειγμα λίστας γειτνίασης

- Στην ***μη διατεταγμένη ακολουθία ακμών*** για την αναπαράσταση ενός γράφου *m-ακμών*, δημιουργούμε μία λίστα με *m* ζευγάρια *[i, j]*. Το κάθε ζευγάρι *[i, j]* αντιπροσωπεύει τις ακμές *ij*. Επομένως, το παράδειγμά μας διαμορφώνεται ως εξής.

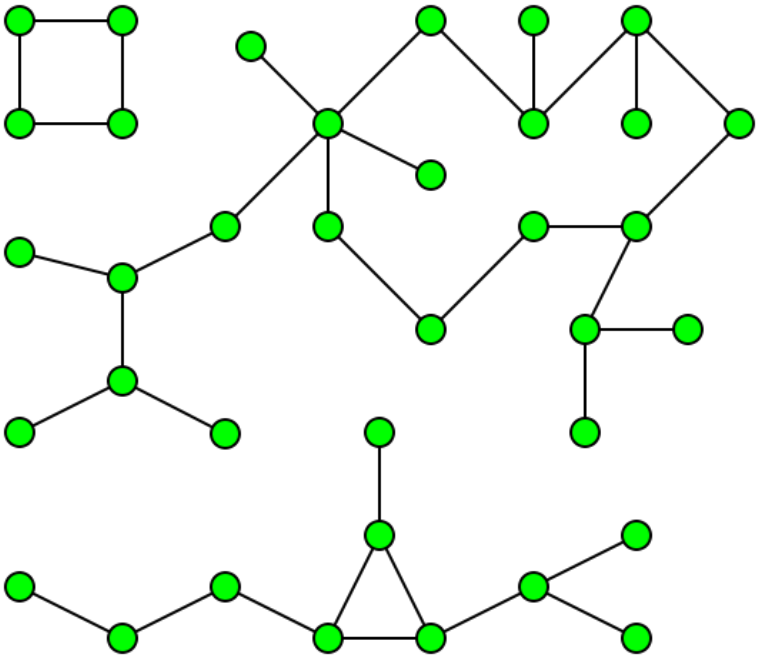
```
[[0,1],[2,3],[3,5],[4,3],[4,5],[5,1],[5,2]]
```

Σχήμα 3.4: Παράδειγμα μη διατεταγμένης λίστας ακμών

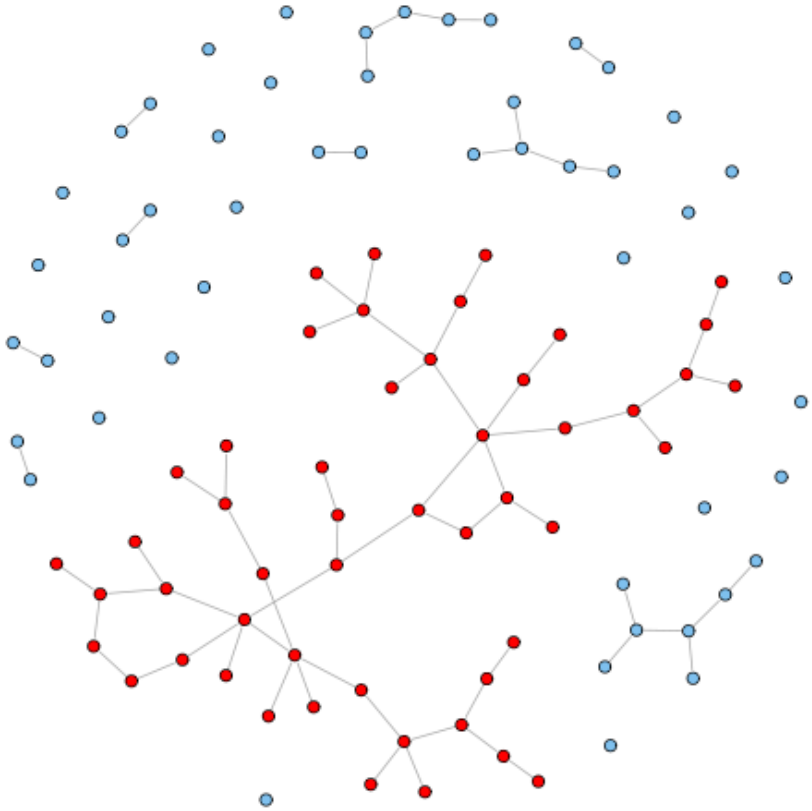
Εμείς, στην παρούσα διατριβή, θα ασχοληθούμε και θα χρησιμοποιήσουμε την αναπαράσταση γράφου ως λίστα γειτνίασης. Βάση του αλγορίθμου που θα παρουσιάσουμε παρακάτω, η χρήση της λίστας γειτνίασης αποτελεί μονόδρομος. Βέβαια, όπως αναφέρθηκε εκτενώς στο προηγούμενο κεφάλαιο, ο χειρισμός της μνήμης, του μεγέθους των δεδομένων και πως αυτά μεταφέρονται στο FPGA απαιτούν ιδιαίτερο και προσεκτικό χειρισμό. Το μέγεθος της λίστας είναι το μικρότερο δυνατό bytewise, αλλά παράλληλα απαιτείται η ολική μεταφορά του γράφου στην πλατφόρμα του FPGA, γεγονός που μας περιορίζει βάση του hardware και της μνήμης αυτού. Παρακάτω, θα εξετάσουμε την μελέτη και τις προσπάθειες που έγιναν για τη σωστή λειτουργία του αλγόριθμου όσο και της αύξησης του μεγέθους του γράφου καθώς και τα πλεονεκτήματα και τα μειονεκτήματα της κάθε προσέγγισης.

3.2 Διασυνδεδεμένα Στοιχεία – Connected Components

Στη θεωρία γράφων, ένα διασυνδεδεμένο στοιχείο (connected component), ενός μη κατευθυνόμενου γράφου είναι ένας υπογράφος του συνολικού γράφου (υπεργράφου), όπου οποιεσδήποτε δύο κορυφές (κόμβοι) είναι συνδεδεμένες μεταξύ τους με μονοπάτια, και δεν συνδέεται με επιπλέον κορυφές του υπεργράφου. Πιο απλά, είναι ένα μέγιστο σύνολο από κόμβους, όπου κάθε ζευγάρι κόμβων συνδέεται με ένα μονοπάτι. Τα διασυνδεμένα στοιχεία σχηματίζουν μία υποδιαίρεση του συνόλου των κόμβων ενός γράφου, το οποίο σημαίνει πως κάθε διασυνδεμένο στοιχείο δεν μπορεί να είναι κενό, ανά ζεύγη δεν έχουν κανένα κοινό στοιχείο και η ένωση όλων σχηματίζει το σύνολο όλων των κορυφών, δηλαδή των υπεργράφο. Βάση των παραπάνω, μπορούμε να πούμε ότι μία κορυφή που δεν έχει συνδεμένες ακμές είναι και αυτή ένα διασυνδεμένο στοιχείο καθώς και ότι αν όλες οι κορυφές ενός γράφου είναι συνδεδεμένες μεταξύ τους, τότε έχουμε μόνο ένα διασυνδεδεμένο στοιχείο το οποίο αποτελείται από ολόκληρο το γράφο. Παρακάτω βλέπουμε κάποια γραφικά παραδείγματα διασυνδεδεμένων γράφων:



Σχήμα 3.5: Γράφος με τρία διασυνδεμένα στοιχεία



Σχήμα 3.6: Γράφος με χρωματισμένο το διασυνδεμένο στοιχείο με τις περισσότερες κορυφές. Σύνολο 31 διασυνδεμένα στοιχεία.

Ένας εναλλακτικός τρόπος ορισμού των διασυνδεδεμένων στοιχείων ενός γράφου περιλαμβάνει τις ισοδύναμες κλάσεις μίας σχέσης ισοδυναμίας, η οποία ορίζεται στις κορυφές του γράφου. Σε ένα μη-κατευθυνόμενο γράφο, μία κορυφή *V* είναι *προσβάσιμη* από μία κορυφή *U* εάν υπάρχει ένα μονοπάτι από το *U* στο *V.* Σε αυτόν το ορισμό, μία μοναδική κορυφή μετράτε ως ένα μονοπάτι μηδενικού μήκους, και η ίδια κορυφή μπορεί να εμφανιστεί παραπάνω από μία φορές σε ένα μονοπάτι. Η προσβασιμότητα είναι μία σχέση ισοδυναμίας εφόσον:

- Είναι *αυτοπαθής*: Υπάρχει ένα μονοπάτι μήκους μηδέν από οποιαδήποτε κορυφή προς τον εαυτό της.
- Είναι *συμμετρική*: Εάν υπάρχει ένα μονοπάτι από το *U* το *V,* τότε οι ίδιες ακμές σχηματίζουν ένα μονοπάτι από το *V* στο *U*.
- Είναι *μεταβατική*: Εάν υπάρχει ένα μονοπάτι από το *U* στο *V* και ένα μονοπάτι από το *V* στο *W,* τότε τα δύο μονοπάτια μπορούν να ενωθούν ώστε να σχηματίσουν ένα μονοπάτι από το *U* στο *W*.

Τότε, τα διασυνδεμένα στοιχεία είναι οι επαγόμενοι υπογράφοι που σχηματίζονται από τις ισοδύναμες κλάσεις της σχέσης ισοδυναμίας.

3.3 Αλγόριθμος Shiloach-Vishkin

Ο αριθμός των διασυνδεδεμένων στοιχείων είναι ένα σημαντικό τοπολογικό χαρακτηριστικό ενός γράφου. Ο υπολογισμός τους είναι ένα σχετικά ευθύ και απλό πρόβλημα αν αναφερόμαστε σε γραμμικό χρόνο (με βάση τον αριθμό των κορυφών και των ακμών του γράφου) χρησιμοποιώντας *αναζήτηση κατά πλάτος (Breadth-First Search - BFS)* ή *αναζήτηση κατά βάθος (Depth-First Search - DFS).* Σε κάθε περίπτωση, μία αναζήτηση που ξεκινάει σε κάποιο συγκεκριμένο κόμβο *V* θα υπολογίσει ολόκληρο το διασυνδεμένο στοιχείο που περιέχει το *V* πριν επιστρέψει. Για να υπολογίσουμε όλα τα διασυνδεδεμένα στοιχεία ενός γράφου, θα πρέπει να επαναλάβουμε την παραπάνω διαδικασία για κάθε κόμβο ο οποίος δεν έχει συμπεριληφθεί σε προηγουμένως εβρισκόμενο διασυνδεδεμένο στοιχείο. Οι *Hopcroft & Tarjan (1973)* περιγράφουν ενδελεχώς την παραπάνω διαδικασία, αναφέροντας πως είναι πλέον σωστά και πλήρως καταγεγραμμένη.

Υπάρχουν, επίσης, αποτελεσματικοί αλγόριθμοι για τον δυναμικό υπολογισμό των διασυνδεμένων στοιχείων ενός γράφου καθώς προσθέτουμε ή αφαιρούμε κορυφές και ακμές από αυτόν.

3.3.1 Shiloach-Viskin

Στην παρούσα πτυχιακή θα ασχοληθούμε με τον αλγόριθμο των Yossi Shiloach & Uzi Vishkin [6] για τον παράλληλο υπολογισμό των διασυνδεμένων στοιχείων

ενός γράφου, ο οποίος έχει βελτιστοποιηθεί βασιζόμενος στη δημοσίευση των D.Bader, G. Cong και J. Feo [7].

Ο αρχικός αλγόριθμος των Shiloach-Vishkin είναι ένας παράλληλος αλγόριθμος ο οποίος χρησιμοποιεί *O(n+m)* επεξεργαστές, συγκεκριμένα *n+2m*, για να βρει τα διασυνδεδεμένα στοιχεία ενός μη κατευθυνόμενου γράφου με *n* κόμβους και *m* ακμές σε χρόνο *O(log n)*, υποθέτοντας πως οι επεξεργαστές έχουν πρόσβαση σε κοινή μνήμη και πως η ταυτόχρονη πρόσβαση σε αυτή για λειτουργίες εγγραφής και διαβάσματος επιτρέπεται.

3.3.2 Βασικά του αλγορίθμου

Κατά τη διάρκεια όλου του αλγορίθμου κάθε κορυφή V έχει ένα πεδίο δεικτών D(V) μέσω του οποίου "δείχνει" σε κάποια άλλη κορυφή ή τον εαυτό της. Θα μπορούσαμε να πούμε πως *V->D(V)* είναι σαν μία κατευθυνόμενη ακμή σε ένα βοηθητικό γράφο, το λεγόμενο "γράφο δεικτών". Ο γράφος δεικτών αλλάζει από τη μία φάση του αλγόριθμου στην άλλη. Παρόλα αυτά, είναι ένα δάσος από ριζωμένα δέντρα (δέντρο είναι ένας μη κατευθυνόμενος γράφος, στον οποίο οποιεσδήποτε δύο κορυφές συνδέονται με ένα και μόνο απλό μονοπάτι), μαζί με βρόγχους που συνδέονται με τον εαυτό τους, πράγμα το οποίο συμβαίνει μόνο στις ρίζες. Καθώς ο αλγόριθμος προχωράει, ο αριθμός των δέντρων μειώνεται ενώ κάθε δέντρο αυξάνει (ή εξαφανίζεται). Αυτό προκαλείται από μία διεργασία προσάρτησης κατά την οποία κάθε δέντρο προσαρτάται σε κάποιο άλλο. Επιπλέον, τα δέντρα υπόκεινται και σε άλλη μία διεργασία κατά την οποία καταρρέουν προς τη ρίζα τους. Όλος ο αλγόριθμος αποτελείται από ενδιάμεσες εφαρμογές των παραπάνω βασικών διεργασιών.

Στο τέλος του αλγορίθμου, οι κορυφές κάθε διασυνδεδεμένου στοιχείου σχηματίζουν ένα ριζωμένο αστέρι στο γράφο των δεικτών. Αποτέλεσμα αυτού, είναι κάθε ερώτηση τη μορφής "Η κορυφή V_i και η κορυφή V_j ανήκουν στο ίδιο διασυνδεδεμένο στοιχείο;" να μπορεί να απαντηθεί σε σταθερό χρόνο.

3.3.3 Απλή Περιγραφή του Αλγορίθμου

Ας υποθέσουμε ότι *V={1,2,...,n}*. Ο αλγόριθμος πραγματοποιεί το πολύ *log3/2n + 2* επαναλήψεις. Η γραφή *Ds(i)=j* σημαίνει πως η κορυφή *i* δείχνει στην κορυφή *j* μετά την *s-στη* επανάληψη. Αρχικά $D_0(i) = i, i = 1,2,...,n$.

Βήμα 1: Κατάρρευση προς τη ρίζα

$D_s(i) <- D_{s-1}(D_{s-1}(i))$

Βήμα 2: Προσάρτηση δέντρων σε χαμηλότερες κορυφές άλλων δέντρων

Όλες οι κορυφές που δείχνουν σε μία ρίζα στο τέλος μίας προηγούμενης επανάληψης ελέγχουν αν οι γείτονές τους δείχνουν σε μικρότερες κορυφές. Αν

κάποιος βρει έναν τέτοιο γείτονα *j*, προσπαθεί να προσαρτήσει το δέντρο του στο $D_s(j)$.

Εδώ θα πρέπει να αναφέρουμε πως ένα δέντρο ονομάζεται στάσιμο στην *s-στη* επανάληψη εάν δεν έχει υποστεί κάποια αλλαγή στα δύο πρώτα βήματα αυτής της επανάληψης *(s)*, δηλαδή δεν έχει καταρρεύσει στη ρίζα του, δεν έχει προσαρτηθεί σε κάποιο άλλο δέντρο, ούτε κάποιο άλλο δέντρο έχει προσαρτηθεί σε αυτό. Η ρίζα ενός στάσιμου δέντρου ονομάζεται στάσιμη ρίζα.

Βήμα 3: Προσάρτηση στάσιμων δέντρων

Όλες οι κορυφές που δείχνουν σε μία στάσιμη ρίζα, ελέγχουμε αν οι γειτονικές κορυφές τους δείχνουν σε μία κορυφή ενός άλλου δέντρου. Εάν βρεθεί τέτοια κορυφή , προσπαθούμε να προσαρτήσουμε το δέντρο της στο $D_s(j)$.

Βήμα 4: Δεύτερη κατάρρευση προς τη ρίζα

$D_s(i) \leftarrow D_s(D_s(i))$

3.3.4 Αναλυτική Περιγραφή του Αλγορίθμου

Εισαγωγή δεδομένων: Υποθέτοντας ότι οι κορυφές του γράφου αντιπροσωπεύονται από τους αριθμούς *1,...,n*, κρατάμε μόνο τον αριθμό *n* και το διάνυσμα *E* μήκους *2m* στο οποίο κάθε ακμή *(i,j)* εμφανίζεται δύο φορές, μία σαν το κατευθυνόμενο ζευγάρι *<i,j>* και μία σαν το κατευθυνόμενο ζευγάρι *<j,i>*. Η σειρά εμφάνισης των κατευθυνόμενων ζευγών στο διάνυσμα *E* δεν έχει σημασία. Οι επεξεργαστές αντιπροσωπεύονται ως $P_1,...,P_{n+2m}$.

Θα χρησιμοποιήσουμε επίσης ένα βοηθητικό διάνυσμα *Q* μήκους *n*. Κατά τη διάρκεια εκτέλεσης του αλγορίθμου το *Q* ικανοποιεί τα εξής:

Q(i) = s, εάν μετά το δεύτερο βήμα της s-στης επανάληψης, υπάρχει τουλάχιστον μία κορυφή *j* που δείχνει στο *i*, που δεν έδειχνε στην προηγούμενη επανάληψη *(s-1)*.

Q(i) < s, σε αντίθετη περίπτωση από το παραπάνω.

Βήμα 0: Αρχικοποίηση

i) Κατανέμουμε τους επεξεργαστές σε κορυφές και κατευθυνόμενα ζευγάρια. Εάν *i <= n*, ο επεξεργαστής P_i κατανέμεται στην κορυφή *i*. Αλλιώς εάν *i > n*, ο επεξεργαστής P_i κατανέμεται στο κατευθυνόμενο ζευγάρι $E(i) = \langle i_1,i_2 \rangle$ και ονομάζεται $P_{i1,i2}$.

ii) if *i <= n* then *D0(i) <- i, Q(i) <- 0, s <- 1, s' <- 1*

While s' = s do

Βήμα 1: if $i <= n$ then $D_s(i) <- D_{s-1}(D_{s-1}(i))$

if $D_s(i) != D_{s-1}(i)$ then $Q(D_s(i)) <- s$

Βήμα 2: if $i > n$ then

if $D_s(i_1) = D_{s-1}(i_1)$ then if $D_s(i_2) < D_s(i_1)$

then $D_s(D_s(i_1)) <- D_s(i_2)$

$Q(D_s(D_s(i_1))) <- s$

Σημείωση: Εάν το $D(i_1)$ δεν έχει υποστεί αλλαγή στο Βήμα 1 τότε $P_i=P_{i1,i2}$ ελέγχει αν το i_2 δείχνει σε κάποια μικρότερη κορυφή. Αν ναι, τότε προσπαθεί να προσαρτήσει τη ρίζα στο $D_s(i_2)$. Ταυτόχρονα, όλοι οι επεξεργαστές τη μορφής P_{jk} για τους οποίους ισχύει $D_s(j) = D_s(i_1)$ και $D_s(k) < D_s(i_1)$, προσπαθούν να ανανεώσουν το $D_s(D_s(i_1))$.

Βήμα 3: if $i > n$ then

if $D_s(i) = D_s(D_s(i_1))$ and $Q(D_s(i_1)) < s$

then if $D_s(i_1) != D_s(i_2)$

then $D_s(D_s(i_1)) <- D_s(i_2)$

Σημείωση: $P_i = P_{i1i2}$ ελέγχει πρώτα αν το $D_s(i_1)$ είναι ρίζα. Αν ναι, τσεκάρει χρησιμοποιώντας τον πίνακα Q, εάν πρόκειται για στάσιμη ρίζα. Αν είναι τότε προσπαθεί να το προσαρτήσει σε άλλο δέντρο. Αυτό πραγματοποιείται ταυτόχρονα από όλους τους επεξεργαστές P_{jk} που ισχύει $D_s(j) = D_s(i_1)$.

Βήμα 4: If $i < n$ then $D_s(i) <- D_s(D_s(i))$

Βήμα 5: $s <- s + 1$

If $i < n$ and $Q(i) = s$

then $s' <- s'' + 1$

Σημείωση: Μόλις όλα τα δέντρα γίνουν στάσιμα, έχουμε ότι $Q(i) < s$ για όλα τα i, $1 <= i <= n$ και με αυτό τον τρόπο το s' δεν θα αυξηθεί έχοντας $s' < s$. Αυτό προκαλεί τον αλγόριθμο να τερματιστεί όταν όλα τα δέντρα γίνουν στάσιμα.

3.4 Bader Shiloach Optimization

Στο άρθρο τους με τίτλο "*On the Architectural Requirements for Efficient Execution of Graph Algorithms*" οι *Bader, Cong* και *Feo* προσπαθούν να επιταχύνουν αλγόριθμους γράφων χρησιμοποιώντας αρχιτεκτονικές υλικού με Συμμετρικούς Πολυεπεξεργαστές (Symmetric Multiprocessors - SMPs) και Αρχιτεκτονικές Πολλαπλών Νημάτων (Multithread Architectures - MTAs).

Τα SMPs είναι αρχιτεκτονικές κατά τις οποίες πολλοί επεξεργαστές λειτουργούν σε ένα πραγματικό περιβάλλον κοινής μνήμης σε επίπεδο υλικού, και όχι λογισμικού, και παρουσιάζονται σαν ένα ενιαίο μηχάνημα. Συνήθως χρησιμοποιούνται σε υπολογιστές υψηλών επιδόσεων υπό την μορφή συμπλεγμάτων (clusters), όπου κάθε κόμβος αποτελείται από 2 έως 100 επεξεργαστές.

Τα MTAs είναι επίσης αρχιτεκτονικές πολλαπλών επεξεργαστών με κοινή μνήμη. Όλη η μνήμη είναι προσβάσιμη και απέχει ίσες αποστάσεις από όλους τους επεξεργαστές. Η διαφορά αυτών των αρχιτεκτονικών σε σχέση με τα SMPs, έγκειται στο ότι δεν υπάρχει ούτε τοπική μνήμη σε κάθε επεξεργαστή ούτε caches. Για τα προβλήματα των καθυστερήσεων της μνήμης καθώς και του συγχρονισμού βασίζεται στον παραλληλισμό.

Χρησιμοποιώντας τα παραπάνω συστήματα και αρχιτεκτονικές, ερεύνησαν κατά πόσο είναι δυνατό να χρησιμοποιηθούν στην επιτάχυνση αλγορίθμων γράφων, και συγκεκριμένα σε αλγόριθμους *Κατάταξης Λίστας (List Ranking)* και *Διασυνδεδεμένων Στοιχείων (Connected Components)*. Στην έρευνά τους για τα Διασυνδεμένα Στοιχεία χρησιμοποίησαν τον αλγόριθμο των *Shiloach-Vishkin*. Εμείς θα επικεντρωθούμε στις αλλαγές και στις βελτιστοποιήσεις που έγιναν για τη χρήση σε MTA αρχιτεκτονικές.

Input: 1. A set of m edges (i, j) given in arbitrary order
2. Array $D[1..n]$ with $D[i] = i$

Output: Array $D[1..n]$ with $D[i]$ being the component to which vertex i belongs

begin
  while *true* **do**
    1.**for** $(i, j) \in E$ *in parallel* **do**
      if $D[i]=D[D[i]]$ *and* $D[j]<D[i]$ **then** $D[D[i]] = D[j]$;
    2.**for** $(i, j) \in E$ *in parallel* **do**
      if *i belongs to a star and* $D[j]\neq D[i]$ **then** $D[D[i]] = D[j]$;
    3.**if** *all vertices are in rooted stars* **then** exit;
    for *all i in parallel* **do**
      $D[i] = D[D[i]]$
end

Ο αλγόριθμος των Shiloach-Viskin για την εύρεση των διασυνδεδεμένων στοιχείων.

```
while (graft) {
   graft = 0;
#pragma mta assert parallel
  1. for (i=0; i<2*m; i++) {
       u = E[i].v1;
       v = E[i].v2;
       if (D[u]<D[v] && D[v]==D[D[v]]) {
         D[D[v]] = D[u];
         graft = 1;
       }
     }
#pragma mta assert parallel
  2. for(i=0; i<n; i++)
       while (D[i] != D[D[i]]) D[i]=D[D[i]];
}
```

Ο αλγόριθμος των SV για χρήση σε MTA αρχιτεκτονικές. E είναι η λίστα των ακμών, με κάθε στοιχείο να έχει δύο πεδία, v_1 και v_2 τα οποία αναπαριστούν τα δύο άκρα.

Η εκτέλεση του αλγορίθμου σε περιβάλλον MTA διαφέρει από την αρχική του περιγραφή. Τα δέντρα χωρίζονται σε υπερκόμβους σε κάθε επανάληψη, ώστε το βήμα δύο του αρχικού αλγορίθμου, όπως φαίνεται στο αριστερά σχήμα, να μπορεί να παραληφθεί. Πλέον δεν χρειάζεται να ελέγχουμε αν κάποιος κόμβος ανήκει σε ένα σχηματισμό αστεριού, το οποίο περιελάβανε ένα μεγάλο αριθμό υπολογισμών και προσπελάσεων στη μνήμη.

Κεφάλαιο 4

Σχεδίαση Αρχιτεκτονικής

4.1 Αρχικός Αλγόριθμος

Η σχεδίαση του αλγορίθμου βασίστηκε στην υλοποίηση που έχει γίνει στη σουίτα *GAP Benchmark Suite[8],* η οποία έχει σχεδιαστεί με στόχο να βοηθήσει στην έρευνα επεξεργασίας γράφων τυποποιώντας τις αξιολογήσεις των πιο γνωστών και πολυχρησιμοποιημένων αλγορίθμων. Η αρχική υλοποίηση που παρουσιάζεται στην παραπάνω σουίτα αναφορών είναι η ακόλουθη:

```
1.  pvector<NodeID> ShiloachVishkin(const Graph &g) {
2.    pvector<NodeID> comp(g.num_nodes());
3.    for (NodeID n=0; n < g.num_nodes(); n++)
4.      comp[n] = n;
5.    bool change = true;
6.    int num_iter = 0;
7.    while (change) {
8.      change = false;
9.      num_iter++;
10.     for (NodeID u=0; u < g.num_nodes(); u++) {
11.       NodeID comp_u = comp[u];
12.       for (NodeID v : g.out_neigh(u)) {
13.         NodeID comp_v = comp[v];
14.         if ((comp_u < comp_v) && (comp_v == comp[comp_v])) {
15.           change = true;
16.           comp[comp_v] = comp_u;
17.         }
18.       }
19.     }
20.     for (NodeID n=0; n < g.num_nodes(); n++) {
21.       while (comp[n] != comp[comp[n]]) {
22.         comp[n] = comp[comp[n]];
23.       }
24.     }
25.   }
26.   return comp;
27. }
```

4.2 Αλγόριθμος FPGA

Ο αλγόριθμος υλοποιήθηκε σε γλώσσα προγραμματισμού C++, και βελτιστοποιήθηκε σε επίπεδο εντολών με την χρήση pragmas που αναφέρθηκαν παραπάνω. Βασικό βήμα για τη βελτίωση του αλγορίθμου και τη δυνατότητα μεταγλώττισης του παραπάνω αλγορίθμου μέσω του HLS είναι η μετατροπή των δυναμικών δομών μνήμης (vectors) σε στατικές δομές μνήμης (arrays). Το HLS χρειάζεται να γνωρίζει, όπως προείπαμε, κατά τη διάρκεια της μεταγλώττισης το μέγεθος των δομών ώστε να μπορεί να δεσμεύσει τα απαραίτητα resources από

την πλατφόρμα του FPGA. Επίσης, γνωρίζουμε πως κάθε δομή, στην περίπτωση μας οι πίνακες, δεν μπορεί να χρησιμοποιηθεί ταυτόχρονα ως έξοδος και ως είσοδος του προγράμματος. Αυτά θα πρέπει να είναι διακριτά και σε καμία περίπτωση δεν μπορούν να συνδυαστούν μεταξύ τους. Οπότε υποχρεωτικά, κάποιες από τις παραμέτρους των συναρτήσεων λειτουργούν ως είσοδοι και κάποιοι ως έξοδοι. Έχοντας υπόψιν λοιπόν τους παραπάνω περιορισμούς στη μνήμη, δοκιμάσαμε αρκετές υλοποιήσεις του αλγόριθμου μέχρι να φθάσουμε στο βέλτιστο χωρικό αποτέλεσμα.

Βάση των παραπάνω ο αλγόριθμος λαμβάνει, τελικά την ακόλουθη μορφή:

```
1.  #include "shiloach.h"
2.  #include "sds_lib.h"
3.
4.  void ShiloachVishkin_HW(ID comp[NODES], int g[ROWS * COLS]) {
5.      ID *compBUFFER;
6.      compBUFFER = (ID *)sds_alloc(NODES*sizeof(ID));
7.      Create_HW(compBUFFER);
8.      bool flag = true;
9.      short iterations=0;
10.     while (flag) {
11.         flag = false;
12.         iterations++;
13.         Inner_HW(compBUFFER, comp, g, flag);
14.         if (!flag) break;
15.         CopyComp_HW(comp, compBUFFER);
16.     }
17.     sds_free(compBUFFER);
18. }
19.
20. void Create_HW(ID comp[NODES]) {
21. #pragma HLS INLINE
22. Loop_Create: for (int i = 0; i < NODES; i++) {
23.                 #pragma HLS PIPELINE rewind
24.                 #pragma HLS UNROLL factor=20
25.                 comp[i] = i;
26.             }
27. }
28.
29. void CopyComp_HW(ID comp_in[NODES], ID comp_out[NODES]) {
30. #pragma HLS ALLOCATION instances=CopyComp_HW limit=1 function
31. #pragma HLS INLINE
32. Loop_CopyCOMP: for (int i = 0; i < NODES; i++) {
33.                 #pragma HLS PIPELINE rewind
34.                 #pragma HLS UNROLL factor=20
35.                 comp_out[i] = comp_in[i];
36.             }
37. }
38.
39. void Inner_HW(ID comp_in[NODES], ID comp_out[NODES], int graph[ROWS * COLS],
     bool &flag) {
40.         #pragma HLS ALLOCATION instances=Inner_HW limit=1 function
41.         ID compBUFF[NODES];
42.         #pragma HLS RESOURCE variable=compBUFF core=RAM_2P
43. Loop_CopyIN: for (int i = 0; i < NODES; i++) {
44.                 #pragma HLS PIPELINE rewind
45.                 #pragma HLS UNROLL factor=10
46.                 compBUFF[i] = comp_in[i];
47.             }
48.
```

```
Loop_Inner1:    for (int u = 0; u < NODES; u++) {
                    unsigned int comp_u = compBUFF[u];
Loop_Inner2:        for (int v = 0; v < COLS; v++) {
                        #pragma HLS PIPELINE rewind
                        if (graph[(u * COLS) + v] != -1) {
                            unsigned int comp_v = compBUFF[graph[(u * COLS) + v]];
                            if ((comp_u < comp_v) && (comp_v == compBUFF[comp_v])) {
                                compBUFF[comp_v] = comp_u;
                                flag = true;
                            }
                        }
                    }
                }

Loop_Check: for (int n = 0; n < NODES; n++) {
                #pragma HLS PIPELINE rewind
                if (compBUFF[n] != compBUFF[compBUFF[n]]) {
                    compBUFF[n] = compBUFF[compBUFF[n]];
                }
            }

Loop_CopyOUT: for (int i = 0; i < NODES; i++) {
                #pragma HLS PIPELINE rewind
                #pragma HLS UNROLL factor=10
                comp_out[i] = compBUFF[i];
            }
}
```

Συγκριτικά με την αρχική υλοποίηση, αναγκαστήκαμε να προσθέσουμε μερικά επιπλέον στάδια στον αλγόριθμο και αυτό έγινε λόγω του περιορισμού των πόρων του συστήματος. Το πιο απλό θα ήταν να μεταφέρουμε αυτούσιο των κώδικα αλλά δυστυχώς λόγω των παραπάνω περιορισμών που αναφέραμε αυτό δεν είναι εφικτό. Οπότε λειτουργήσαμε ως εξής:

1. Χωρίσαμε τη συνάρτηση Shiloach-Viskin, σε μικρότερες διακριτές συναρτήσεις, `Create_HW`, `Inner_HW`, `CopyComp_HW`.
2. Η `Create_HW` είναι υπεύθυνη για την αρχικοποίηση του εσωτερικού πίνακα *compBUFFER*.
3. Η `Inner_HW` πραγματοποιεί όλους τους βασικούς υπολογισμούς του αλγορίθμου. Λόγω όμως του περιορισμού με τις εισόδους/εξόδους που αναφέραμε παραπάνω, είναι υποχρεωμένη να έχει και αυτή δύο μεθόδους αντιγραφής πίνακα. Η πρώτη μέθοδος είναι για την αντιγραφή της εισόδου σε μία εσωτερική δομή της συνάρτησης και τέλος της αντιγραφής αυτής της εσωτερικής δομής στην έξοδο.
4. Τέλος, η `CopyComp_HW` αντιγράφει τον εσωτερικό πίνακα της συνάρτησης Shiloach-Viskin στην έξοδο του FPGA.

Αναλυτικότερα, όπως φάνηκε στην περιγραφή και υλοποίηση του αλγορίθμου, το αρχικό βήμα είναι η δημιουργία ενός μονοδιάστατου πίνακα *compBUFFER* με μέγεθος όσο είναι και το πλήθος των ακμών του γράφου, και αρχικοποίηση αυτού, όπως φαίνεται στη μέθοδο ***CREATE_HW***. Στη συνέχεια έχουμε την κεντρική δομή του αλγορίθμου όπου ελέγχουμε αν όλα τα δέντρα είναι στάσιμα και περιγράφεται

στη μέθοδο ***INNER_HW***. Τέλος, στο label ***Loop_Check*** της προηγούμενης μεθόδου έχουμε την τελική έξοδο/στάδιο του αλγορίθμου όπου πλέον κάθε στοιχείο, στον αρχικό μονοδιάστατο πίνακα, *compBUFFER[i]* είναι το στοιχείο στο οποίο ανήκει η ακμή *i*.

4.3 Βελτιστοποίηση με τη χρήση pragmas

Στη μελέτη που πραγματοποιήσαμε, χρησιμοποιήσαμε και εξετάσαμε τη χρήση των ***PIPELINE, UNROLL, LOOP_FLATTEN*** *pragmas* στους βρόγχους όπου πραγματοποιούνται οι κύριες διεργασίες του αλγορίθμου και καταλήξαμε στην χρήση κυρίως ***PIPELINE*** και ***UNROLL****,* αλλά και άλλων pragmas που θα αναλυθούν αναλυτικά παρακάτω. Με την χρήση των υπόλοιπων *pragmas* σχετικών με τους βρόχους, παρατηρήσαμε πως δεν υπήρξε κάποια βελτίωση στην ταχύτητα εκτέλεσης. Αυτό το γεγονός οφείλεται στον HLS μεταγλωττιστή που όποτε μεταγλωττίζει, ανάλογα και με τις εξαρτήσεις, υλοποιεί αυτούς τους χειρισμούς.

Ξεκινώντας από την αρχή της υλοποίησης σε C++ του αλγορίθμου, βλέπουμε τη χρήση του ***INLINE.*** Όπως προαναφέραμε, χρησιμοποιείται αντί της απλής κλήσης μιας συνάρτησης, δίνοντας έτσι την οδηγία ο compiler να ξαναδομήσει την συνάρτηση στο σημείο που καλείται η συνάρτηση και όχι να πραγματοποιήσει μία πλήρη κλήση της, που απαιτεί μεταφορά δεδομένων. Αυτή η τακτική χρησιμοποιείται για την ενεργοποίηση της βελτιστοποίησης της λογικής μεταξύ των συνόρων των συναρτήσεων, για τη βελτίωση των latencies και (II), και κυρίως σε μικρές και σύντομες συναρτήσεις.

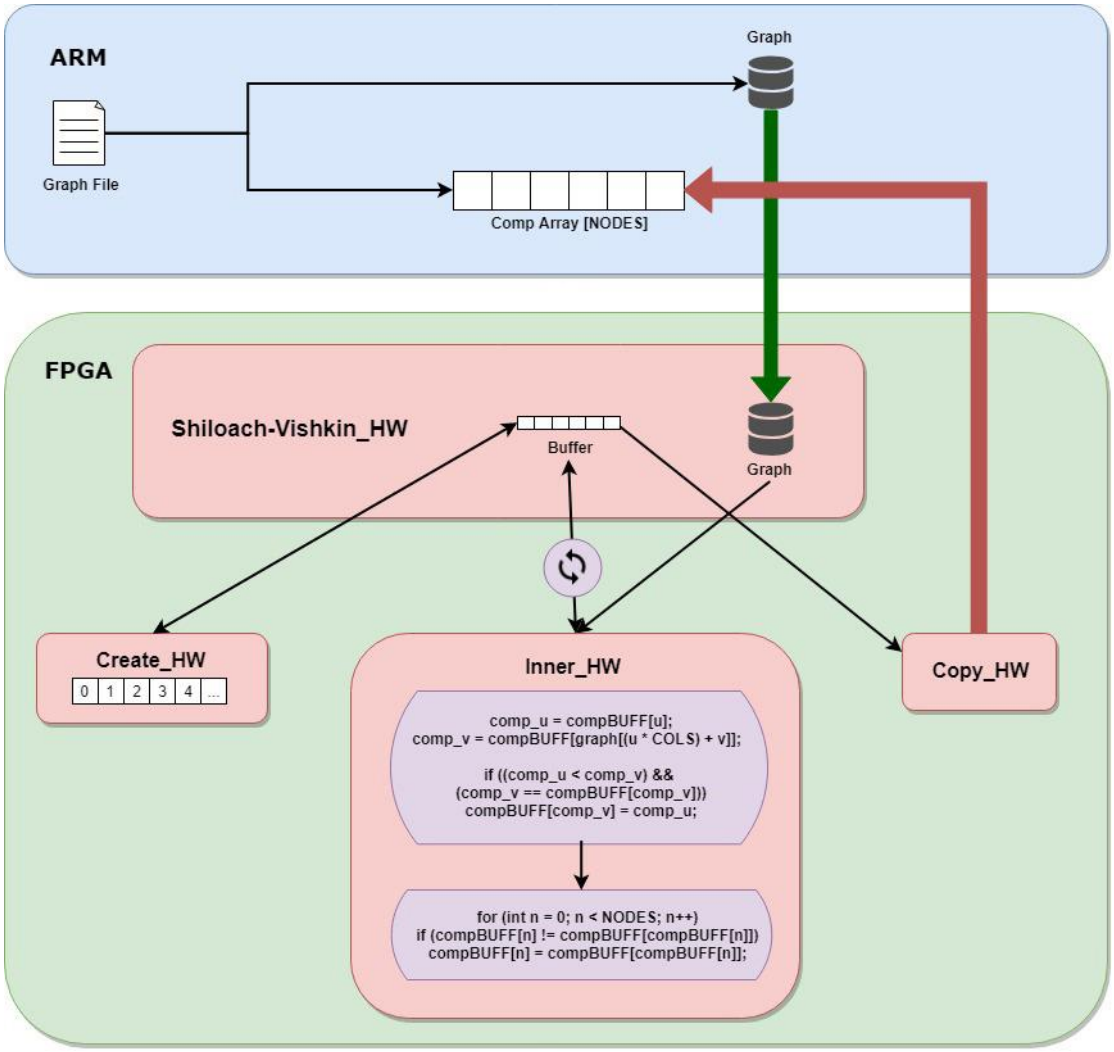
Μετέπειτα, χρησιμοποιώντας το pragma ***PIPELINE*** επιτρέπουμε την εκτέλεση των λειτουργιών σε βρόχο με ταυτόχρονο και παράλληλο τρόπο. Είναι γεγονός ότι μια εντολή που δεν εξαρτάται από προηγούμενες, μπορεί να ολοκληρωθεί πριν από αυτές και να καταλήξουμε με αυτόν τον τρόπο στην αύξηση του throughput. Η επιλογή ***rewind*** ειδοποιεί το HLS πως ο επαναληπτικός βρόγχος επιστρέφει πάντα στην αρχή του για την εκτέλεση της επόμενης εντολής. Έτσι έχουμε το λιγότερο δυνατό (II) μεταξύ διαδοχικών επαναλήψεων του βρόγχου.

Βασικό ρόλο στην παραλληλοποίηση και επιτάχυνση των μεθόδων αντιγραφής και αρχικοποίησης πινάκων, διαδραματίζει το pragma ***UNROLL***. Θα λέγαμε πως «ξετυλίγει» τους βρόγχους σε πολλαπλές παράλληλες εντολές βάση του factor, και με αυτό τον τρόπο παραλληλοποιούνται οι εντολές των βρόγχων. Αυτό είναι πολύ χρήσιμο σε μεθόδους αντιγραφής και αρχικοποίησης, όπως αναφέραμε παραπάνω, γιατί γνωρίζουμε εκ των προτέρων τις τιμές των εντολών καθώς βασίζονται στον iterator του βρόγχου, π.χ. στη μέθοδο ***CREATE_HW*** βλέπουμε το *comp[i]=i*, όπου η τιμή του *comp* εξαρτάται από την τιμή του *i*. Οπότε αν γράφαμε ταυτόχρονα *comp[i+1]=i+1; comp[i+2]=i+2, comp[i+3]=i+3,…* και αλλάζαμε το βήμα του βρόγχου θα είχαμε την παραλληλοποίηση που αναλύσαμε παραπάνω. Στην περίπτωση μας το ***INLINE*** πραγματοποιεί την ίδια διεργασία «ξεδιπλώνοντας» το βρόγχο, αλλά σε επίπεδο hardware. Το βέλτιστο όπως παρατηρεί κάποιος εύκολα

θα ήταν να θέταμε factor ίσο με το μέγεθος των πινάκων σε κάθε περίπτωση, ώστε να εκτελούνταν ολόκληρος ο βρόγχος σε έναν κύκλο ρολογιού. Σε επίπεδο software αυτό γίνεται αυτόματα στους πιο σύγχρονους μεταγλωττιστές, αλλά σε επίπεδο FPGA και hardware είναι αδύνατο για πολύ μεγάλους πίνακες καθώς περιοριζόμαστε από τους πόρους του συστήματος.

Στη συνέχεια βλέπουμε τη χρήση της συνάρτησης ***ALLOCATION*** με επιλογή ***limit=1 function.*** Με αυτό το pragma, δίνουμε εντολή στον μεταγλωττιστή HLS να δημιουργήσει μόνο ένα instance της παραπάνω μεθόδου. Αυτό γίνεται για τον περιορισμό των resources που θα καταλάβει στο FPGA. Στην περίπτωσή μας δεν θέλουμε παραπάνω instances των συγκεκριμένων συναρτήσεων καθώς καλούνται μόνο από ένα σημείο του κώδικά μας αλλά επαναληπτικά σε βρόγχο, οπότε έτσι υποχρεώνουμε το πρόγραμμα πρώτα να τελειώσει την μία επανάληψη μέχρι να ξεκινήσει η επόμενη. Με πολλαπλά instances των συναρτήσεων υπάρχει κίνδυνος να μην έχουμε αρκετή μνήμη στο FPGA, ειδικά όταν μιλάμε για επεξεργασία big data. Αυτό ενέχει βέβαια τον κίνδυνο να αυξήσουμε το Latency της εκτέλεσης των εντολών. Για την καταπολέμηση αυτής της τυχών αύξησης, αλλά και για την ταυτόχρονη όσο το δυνατών επιτάχυνση, χρησιμοποιήσαμε τα παραπάνω ***PIPELINE*** και ***UNROLL***.

Τέλος, έχουμε το pragma ***RESOURCE***. Εδώ χρησιμοποιείται για να δώσει την εντολή στον μεταγλωττιστή να χρησιμοποιείσαι ένα συγκεκριμένο core για την αποθήκευση του πίνακα *compBUFFER*, και πιο συγκεκριμένα τον storage core *RAM_2P*. Ο *RAM_2P* είναι μία RAM με δύο ξεχωριστές θύρες, μία για εγγραφή και μία για ανάγνωση.



Σχήμα 4.1: Shiloach-Vishkin Data flow

Βασισμένοι στην παραπάνω εικόνα μπορούμε να δούμε τη ροή των δεδομένων κατά τη διάρκεια εκτέλεσης του αλγορίθμου των Shiloach-Viskin. Το πρόγραμμα της C++ διαβάζει το αρχείο εισόδου, το οποίο μπορεί να είναι σε μορφή edge list ή graph. Παράλληλα, δημιουργεί δύο στατικές δομές δεδομένων μορφής Array (Πίνακα), το μονοδιάστατο πίνακα *Comp[NODES]* που έχει μέγεθος ίσο με το πλήθος των κόμβων του γράφου, που στην ουσία είναι η έξοδος του αλγορίθμου, και τον πίνακα graph, ο οποίος αντιπροσωπεύει μία λίστα γειτνίασης και πρέπει να μεταφερθεί στο FPGA καθώς αποτελεί την είσοδο. Αφήνοντας το HLS να επιλέξει μόνο του τον τρόπο μεταφοράς των δεδομένων στο FPGA, επιλέγεται το πρωτόκολλο ***AXI4 Memory-Mapped*** που μεταφέρει τα δεδομένα στην BRAM. Αυτό το πρωτόκολλο μεταφοράς είναι το πιο «αργό», καθώς μεταφέρει έως 16384 στοιχεία μόνο, μεγέθους έως και 64 bit και υποθέτει πως τα δεδομένα είναι mapped στη μνήμη και όχι σε sequential θέσεις μνήμης. Αυτός είναι ο default τρόπος που δεσμεύεται η μνήμη σε ένα οποιοδήποτε σύγχρονο υπολογιστικό σύστημα. Για να προσπεράσουμε αυτόν τον περιορισμό αλλά και το bottleneck

που δημιουργεί η επιλογή του ***AXI4 Memory-Mapped*** πρωτοκόλλου, χρησιμοποιήσαμε κάποια pragmas ώστε να δώσουμε τις απαραίτητες οδηγίες στον compiler και να καταλήξουμε να χρησιμοποιηθεί το πρωτόκολλο μεταφοράς δεδομένων ***AXI4-Stream***, για τη σειριακή μεταφορά δεδομένων μέσω ενός δίαυλου με μεγάλο bandwidth. Οι οδηγίες αυτές δόθηκαν σε ένα αρχείο header.

```
1.  //Those are set before compile, based on the input file
2.  static const int NODES = 5000;
3.  static const int ROWS = 5000;
4.  static const int COLS = 10000;
5.  static const int ITERS = 4;
6.
7.  typedef unsigned int ID;
8.
9.  void ShiloachVishkin_HW(ID comp[NODES], int g[ROWS * COLS]);
10. #pragma SDS data mem_attribute(comp:PHYSICAL_CONTIGUOUS)
11. #pragma SDS data access_pattern(comp:SEQUENTIAL)
12. void Create_HW(ID comp[NODES]);
13.
14. #pragma SDS data mem_attribute(comp_in:PHYSICAL_CONTIGUOUS, comp_out:PHYSICA
    L_CONTIGUOUS)
15. #pragma SDS data access_pattern(comp_in:SEQUENTIAL, comp_out:SEQUENTIAL)
16. void CopyComp_HW(ID comp_in[NODES], ID comp_out[NODES]);
17.
18. #pragma SDS data mem_attribute(comp_in:PHYSICAL_CONTIGUOUS, comp_out:PHYSICA
    L_CONTIGUOUS, g:PHYSICAL_CONTIGUOUS)
19. #pragma SDS data access_pattern(comp_in:SEQUENTIAL, comp_out:SEQUENTIAL, g:S
    EQUENTIAL)
20. void Inner_HW(ID comp_in[NODES], ID comp_out[NODES], int g[ROWS * COLS], boo
    l &flag);
```

Αρχικά, δηλώνουμε το πώς είναι αποθηκευμένα τα δεδομένα μας στη μνήμη. Γι' αυτό χρησιμοποιούμε το ***#pragma SDS data mem_attribute***. Η επιλογή ***PHYSICAL_CONTIGUOUS*** δηλώνει ότι ο πίνακας, ή γενικότερα η στατική δομή δεδομένων , για τον οποίο αναφέρεται το #pragma, έχει δεσμευθεί με τη χρήση της εντολής ***sds_alloc***, η οποία είναι μία optimal δέσμευση μνήμης η οποία συμπεριλαμβάνεται στη βιβλιοθήκη ***sds_lib*** του SDSoC για την κατασκευή hardware functions, και είναι φυσική συνεχής μνήμη. Αντίθετα, η χρήση του ***NON_PHYSICAL_CONTIGUOUS*** σημαίνει πως η μνήμη που αντιστοιχεί στο σχετικό πίνακα έχει δεσμευθεί χρησιμοποιώντας το βασικό ***malloc*** της C. Με αυτό τον τρόπο βοηθάμε τον μεταγλωττιστή, δίνοντας κάποιες οδηγίες, να επιλέξει το βέλτιστο μεταφορέα δεδομένων. Εδώ παράλληλα με την επιλογή τη συνεχής μνήμης θα μπορούσαμε να δηλώσουμε και το *cache coherence* της μνήμης. Με τη χρήση της επιλογής ***CACHEABLE*** δηλώνουμε στο μεταγλωττιστή πως πρέπει να διατηρηθεί η συνοχή της κρυφής μνήμης μεταξύ της CPU και του επιταχυντή, για τη μνήμη που αντιστοιχεί στον πίνακα. Το αντίθετο, δηλαδή τη μη ανάγκη συνοχής CPU επιταχυντή, δηλώνουμε με το ***NON_CACHEABLE***. Δοκιμάζοντας αυτές τις επιλογές, δεν παρατηρήθηκε κάποια αλλαγή στην ταχύτητα εκτέλεσης του αλγορίθμου οπότε για χάριν συντομίας παραλήφθηκαν. Default επιλογή είναι το *CACHEABLE*.

Με τη χρήση του ***#pragma SDS data access_pattern***, δηλώνουμε τον τρόπο προσπέλασης των δεδομένων και με την επιλογή ***SEQUENTIAL*** επιλέγουμε την διαδοχική προσπέλαση για τις δομές μας. Για να μπορέσουμε να χρησιμοποιήσουμε το συγκεκριμένο *#pragma* θα πρέπει να έχουμε δεσμεύσει φυσική συνεχόμενη μνήμη και γι' αυτό το λόγο χρησιμοποιήσαμε και το προηγούμενο #pragma παράλληλα με το ***sds_alloc***. Με τη χρήση του ***SEQUENTIAL*** και του ***PHYSICAL_CONTIGUOUS*** παρέχουμε στον μεταγλωττιστή όλα τα απαραίτητα εργαλεία ώστε να επιλέξει τον ταχύτερο και αποδοτικότερο μεταφορέα δεδομένων.

Δηλώνοντας τα παραπάνω, παρατηρούμε πως ο μεταγλωττιστής HLS επιλέγει να χρησιμοποιήσει τον μεταφορέα του SDSoC ***AXIDMA_2D*** , το οποίο είναι ο ταχύτερος μεταφορέας για δυσδιάστατα δεδομένα, όπως ο δυσδιάστατος πίνακας αναπαράστασης του γράφου, αλλά και τον μεταφορέας ***AXIDMA_SIMPLE***, ο οποίος είναι ο αντίστοιχος για μονοδιάστατους πίνακες. Το πρωτόκολλο ***AXI Direct Memory Access (AXIDMA)*** παρέχει άμεση πρόσβαση μνήμης υψηλής ταχύτητας μεταξύ μνήμης και περιφερειακών τύπου ***AXI4-Stream***. Θα μπορούσαμε να έχουμε χρησιμοποιήσει και το ***#pragma SDS data data_mover***, για να δηλώσουμε τον ***data_mover AXIDMA***, αλλά αν δεν έχουν προηγηθεί τα προηγούμενα βήματα δεν θα μπορούσε να εφαρμοστεί. Από τη στιγμή που έχουν γίνει τα απαραίτητα για να υπάρξει φυσική συνεχή μνήμη επιλέγεται αυτόματα από τον μεταγλωττιστή ως η βέλτιστη λύση.

Τέλος, αξίζει να αναφέρουμε πως υπάρχουν και διάφορα άλλα *#pragmas* που θα μπορούσαμε να συμπεριλάβουμε στην υλοποίηση. Ένα από αυτά είναι το ***#pragma SDS data sys_port*** το οποίο καθορίζει την θύρα αναλόγως αν θέλουμε συνοχή με την κρυφή μνήμη. Αν θέλαμε επιλέγουμε ***ACP (S_AXI_ACP)***, αν δεν θέλαμε επιλέγαμε ***AFI*** με θύρες υψηλής απόδοσης ***(S_AXI_HP)***. Βάση παρατηρήσεων είδαμε πως δεν έχουν καμία επίδραση στην ταχύτητα εκτέλεσης του αλγορίθμου οπότε παραλήφθηκαν.

Κεφάλαιο 5

Αξιολόγηση - Αποτελέσματα

5.1 Αρχική εκτίμηση απόδοσης

Ξεκινώντας την αξιολόγηση του αλγορίθμου των Connected Components, πρώτα πρέπει να εξετάσουμε τον αλγόριθμο πριν τις υλοποιήσεις. Παρατηρώντας τον κώδικα από την σουίτα *GAP Benchmarks* αλλά και τον βελτιωμένο αλγόριθμο των *Bader et al*, βλέπουμε ότι χωρίζεται σε τρία στάδια:

1. Στο πρώτο έχουμε την αρχικοποίηση του μονοδιάστατου πίνακα, μεγέθους ίσου με το πλήθος των κορυφών του γράφου.
2. Στο δεύτερο και βασικό στάδιο έχουμε την επεξεργασία του γράφου. Αυτή βασίζεται σε συγκρίσεις τιμών των ακμών του γράφου, του μονοδιάστατου προηγούμενου πίνακα αλλά και σε αναθέσεις τιμών. Αυτό συνεχίζεται μέχρι να γίνουν όλα τα δέντρα στάσιμα.
3. Τέλος στο τρίτο στάδιο, έχουμε τη δημιουργία του πίνακα εξόδου.

Κάθε στάδιο του αλγορίθμου, βασίζεται για τη λειτουργία του και τον υπολογισμό των αποτελεσμάτων στον πίνακα ***comp***, όπως ονομάζεται αυτός στον κώδικα. Αυτός μεταφέρεται σταδιακά και χρησιμοποιείται από όλα τα στάδια της υλοποίησης. Οπότε μπορούμε να καταλάβουμε πως το μέγεθος αυτού του πίνακα, αλλά και του ίδιου το γράφου επηρεάζει και περιορίζει την υλοποίηση σε hardware. Επίσης, η μη δυνατότητα χρήσης τον ίδιων μεταβλητών, εδώ του πίνακα ***comp***, σαν είσοδο και σαν έξοδο ταυτόχρονα, μας υποχρεώνει να έχουμε ανά πάσα στιγμή τουλάχιστον δύο αντίγραφα του πίνακα στη μνήμη, καθώς ο ίδιος πίνακας χρησιμοποιείται ως είσοδος και ως έξοδος αλλεπάλληλα σε όλα τα στάδια της υλοποίησης.

Ένα άλλο σημαντικό κομμάτι στο βασικό στάδιο (2) του αλγορίθμου, είναι ο γράφος graph. Παρατηρούμε ότι σταδιακά, πραγματοποιείται προσπέλαση όλων των στοιχείων του. Συμπεραίνουμε λοιπόν ότι σε κάθε επανάληψη του αλγορίθμου μέχρι να σταματήσει, είναι απαραίτητη η παρουσία όλου του γράφου στη μνήμη, δεδομένο που μας περιορίζει σε πολλές παραμέτρους:

1. Το μέγεθος αυτού του δισδιάστατου πίνακα, είναι πολύ πιθανό να μην χωράει στη μνήμη του FPGA όταν μιλάμε για παροχή δεδομένων big data.
2. Η μη δυνατότητα χωρισμού του πίνακα σε μικρότερους υποπίνακες, όπου να μπορεί να ελεγχθούν ξεχωριστά, δεν μας επιτρέπει να αυξήσουμε τον παραλληλισμό του αλγορίθμου.
3. Η ταχύτητα μεταφοράς και προσπέλασης του πίνακα.

Στη συνέχεια, εξετάζοντας όλα τα στάδια του αλγορίθμου και της υλοποίησης, βλέπουμε πως δεν υπάρχει ούτε ένα υπολογιστικό κομμάτι παρά μόνο συγκρίσεις στοιχείων των πινάκων και αναθέσεις τιμών.

Τέλος, περνώντας στη δική μας υλοποίηση χρειάστηκε να προσθέσουμε επιπλέον στάδια. Όπως αναφέραμε και στην περιγραφή του κώδικά μας, για να μπορέσουμε να έχουμε όσο το δυνατό μικρότερη χρήση των πόρων του συστήματος και να έχουμε τη δυνατότητα να επεξεργαστούμε μεγάλο όγκο δεδομένων, χωρίσαμε τον αλγόριθμο σε επιμέρους υποσυναρτήσεις. Αυτό προκαλεί περισσότερες προσπελάσεις στη μνήμη καθώς και επιπλέον αντιγραφές, που σε μία software υλοποίηση δεν θα χρειαζόντουσαν.

Σε δεύτερο επίπεδο πρέπει να εξετάσουμε τις δυνατότητες που προσφέρουν τα FPGA. Αρχικά ένας χρόνος που πρέπει να ξεπεραστεί, ώστε να έχουμε ταχύτερη εκτέλεση από αυτήν που θα είχαμε στον επεξεργαστή ARM, είναι η μεταφορά των δεδομένων. Συγκρίνοντας software με hardware, το software ξεκινά με προβάδισμα καθώς δεν χρειάζεται να μεταφέρει τα δεδομένα. Το σημείο υπεροχής των FPGAs είναι οι επεξεργαστικές μονάδες. Μπορεί να προσφέρει μεγάλη παραλληλία και ενδείκνυται σε περιπτώσεις μεγάλου υπολογιστικού φόρτου, όταν δηλαδή έχουμε πολλές πράξεις, και όσο το δυνατόν λιγότερες προσπελάσεις μνήμης και πολλές επαναλήψεις.

Βάση όλων των παραπάνω, καταλήγουμε στα εξής συμπεράσματα πριν ακόμη παρατηρήσουμε τα αποτελέσματα:

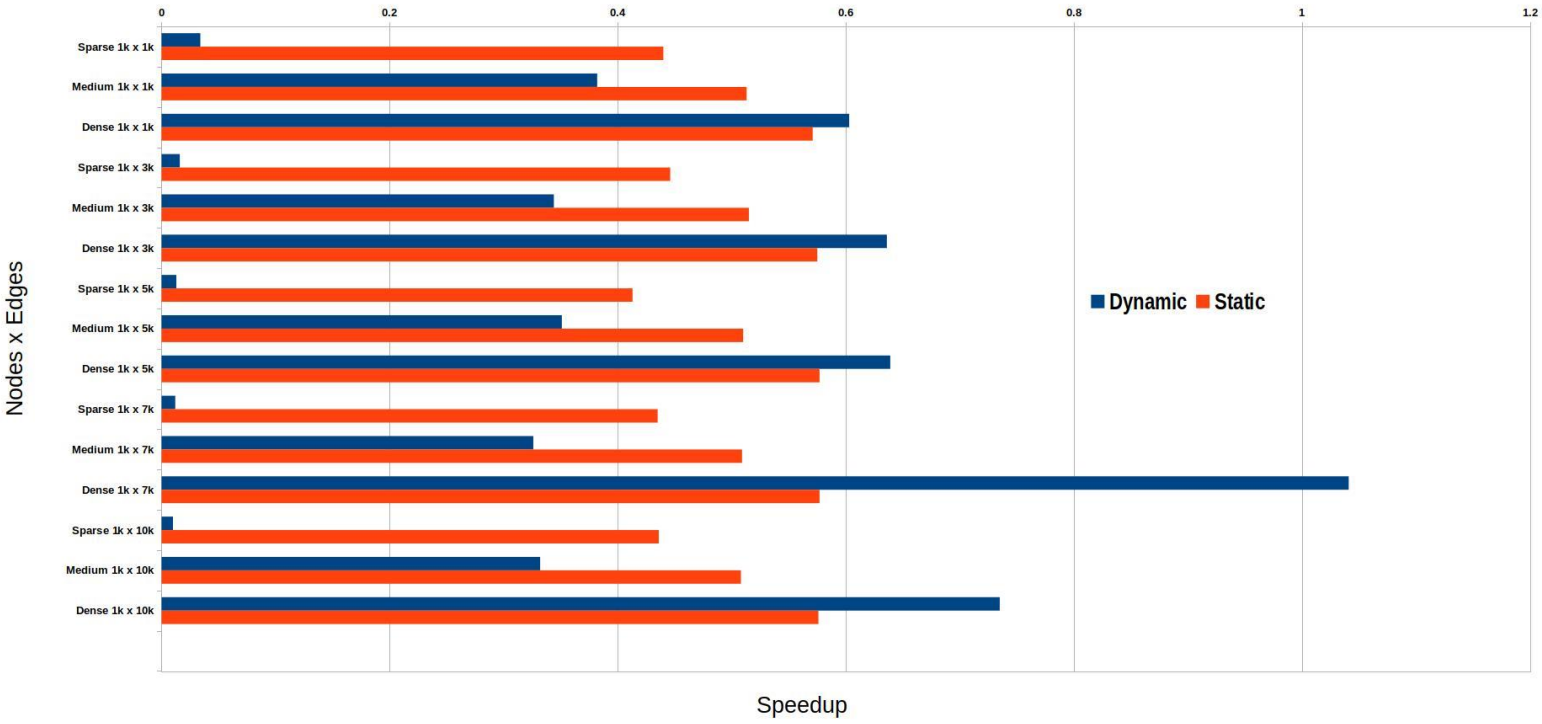
- Όλη η υλοποίηση βασίζεται σε προσπελάσεις μνήμης και μεταφορά δεδομένων. Άρα η ταχύτητα αυτών των λειτουργειών έχει το μεγαλύτερο λόγο στην επιτάχυνση ή όχι του αλγορίθμου.
- Λόγω του FPGA έχουμε επιπλέον μεταφορές δεδομένων για να μεταφερθούν στη μνήμη του, άρα αυξάνουμε αισθητά το χρόνο αν αναλογιστούμε πως επεξεργαζόμαστε big data με εκατοντάδες χιλιάδες στοιχεία.
- Δεν υπάρχουν υπολογισμοί στον αλγόριθμο, το οποίο είναι και το στοιχείο υπεροχής των FPGA.
- Προσθέσαμε περισσότερες προσπελάσεις μνήμης και μεταφορές δεδομένων από το αναμενόμενο, λόγω της περιορισμένης μνήμης του hardware αλλά και του περιορισμού με τις εισόδους-εξόδους των συναρτήσεων.

Αναλογιζόμενοι τα παραπάνω, προβλέπουμε πως δύσκολα θα υπάρξει επιτάχυνση του αλγόριθμου Shiloach-Viskin για τα Connected Components, αλλά και γενικώς για αλγόριθμους γράφων.

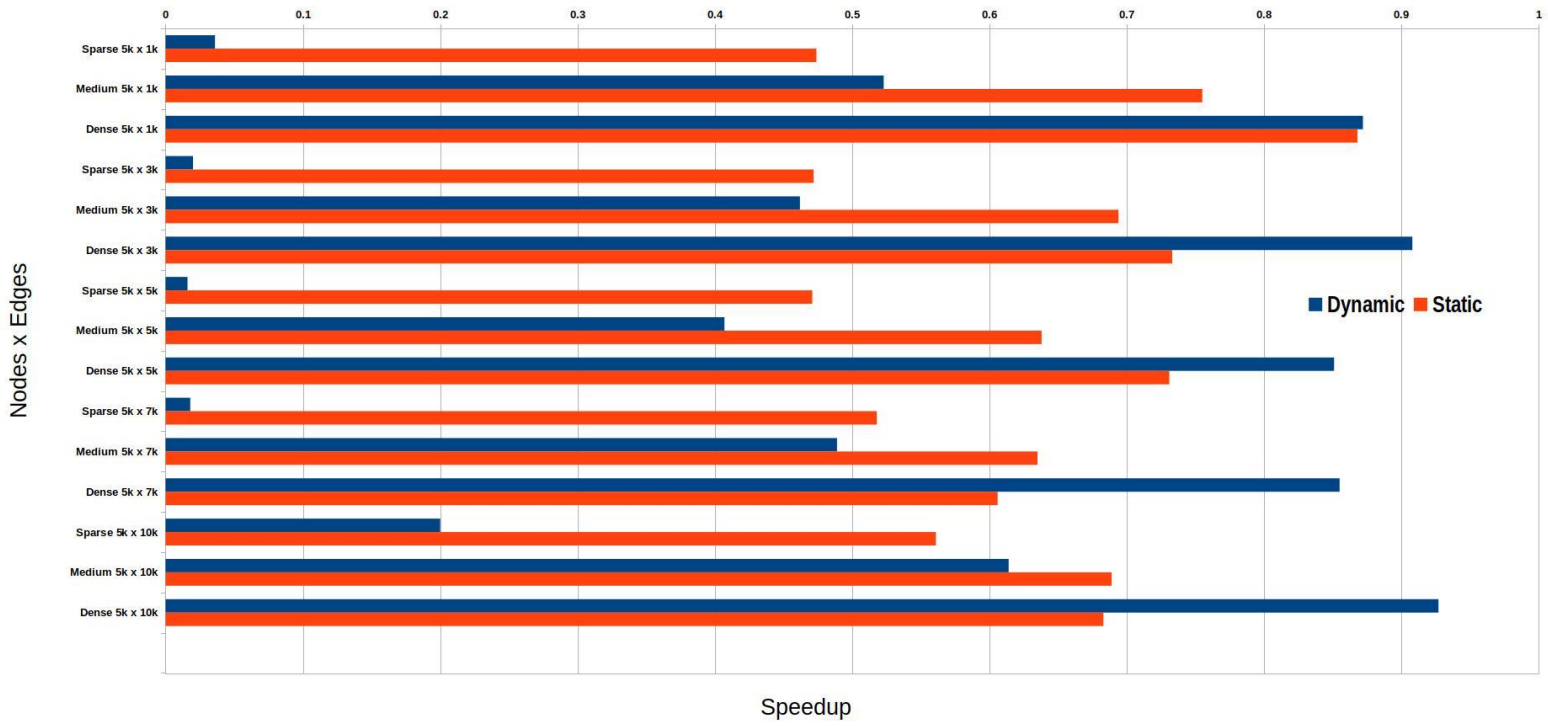
5.2 Αποτελέσματα

Για την υλοποίηση του αλγορίθμου Shiloach-Viskin για την εύρεση των Connected Components έχουμε τα παρακάτω αποτελέσματα. Σημειώνουμε ότι οι κύκλοι ρολογιού μετρήθηκαν σε κώδικα που έτρεξε στον ARM επεξεργαστή της πλατφόρμας (SW Version) και στο FPGA κομμάτι της πλατφόρμας (HW Version). Για να έχουμε σταθερά αποτελέσματα ο αλγόριθμος πραγματοποιούσε δύο επαναλήψεις, όπου παρατηρήθηκε πως ήταν και το μέγιστο που χρειαζόταν για την εύρεση των Connected Components.

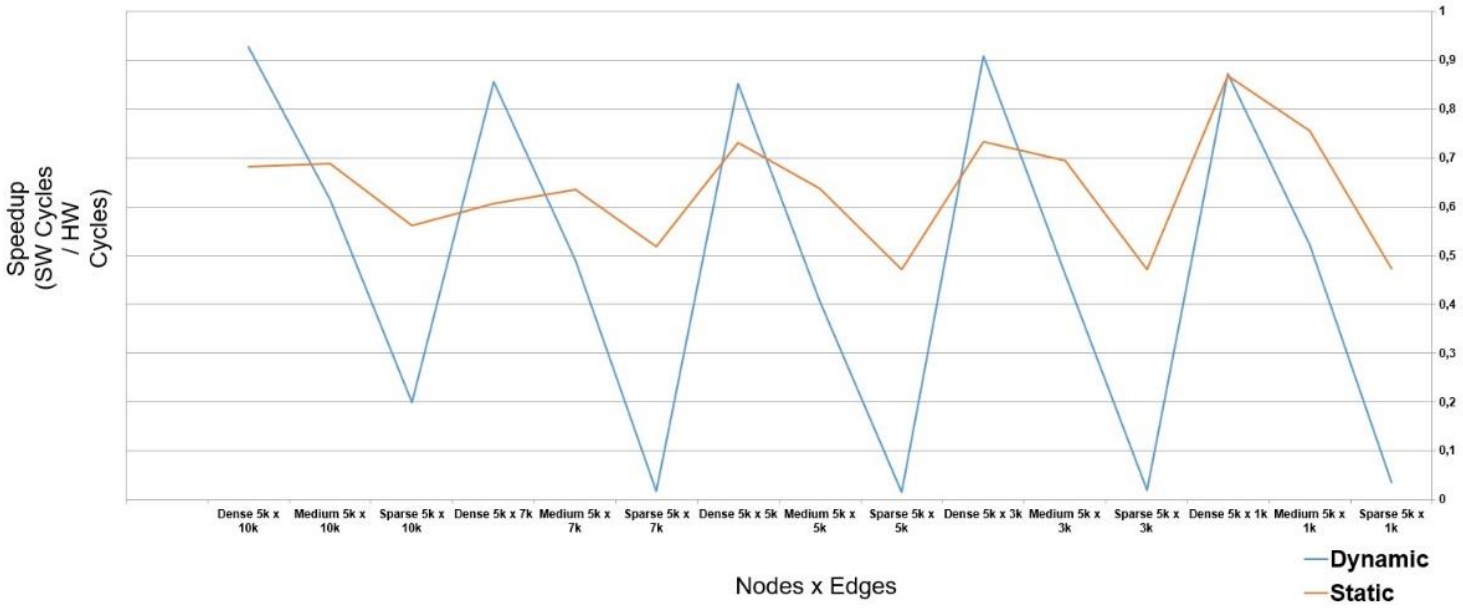
Επιπλέον, για να έχουμε πιο ρεαλιστική σύγκριση ανάμεσα στην hardware υλοποίηση του αλγόριθμου και σε αυτήν που τρέχει στον ARM πυρήνα, χρησιμοποιήθηκαν δύο διαφορετικές εκδόσεις του προγράμματος που θα έτρεχε στον ARM. Η πρώτη χρησιμοποιεί στατικές δομές δεδομένων (πίνακες), όπως ακριβώς και η hardware υλοποίηση, και η δεύτερη χρησιμοποιεί δυναμικές δομές δεδομένων (arrays), η οποία ανταποκρίνεται περισσότερο σε πραγματικές συνθήκες ενός μοντέρνου υπολογιστικού περιβάλλοντος. Στις παρακάτω εικόνες η πρώτη αναφέρεται ως Static και η δεύτερη ως Dynamic.



Σχήμα 5.1: Επιτάχυνση για μικρά datasets



Σχήμα 5.2: Επιτάχυνση για μεγάλα datasets



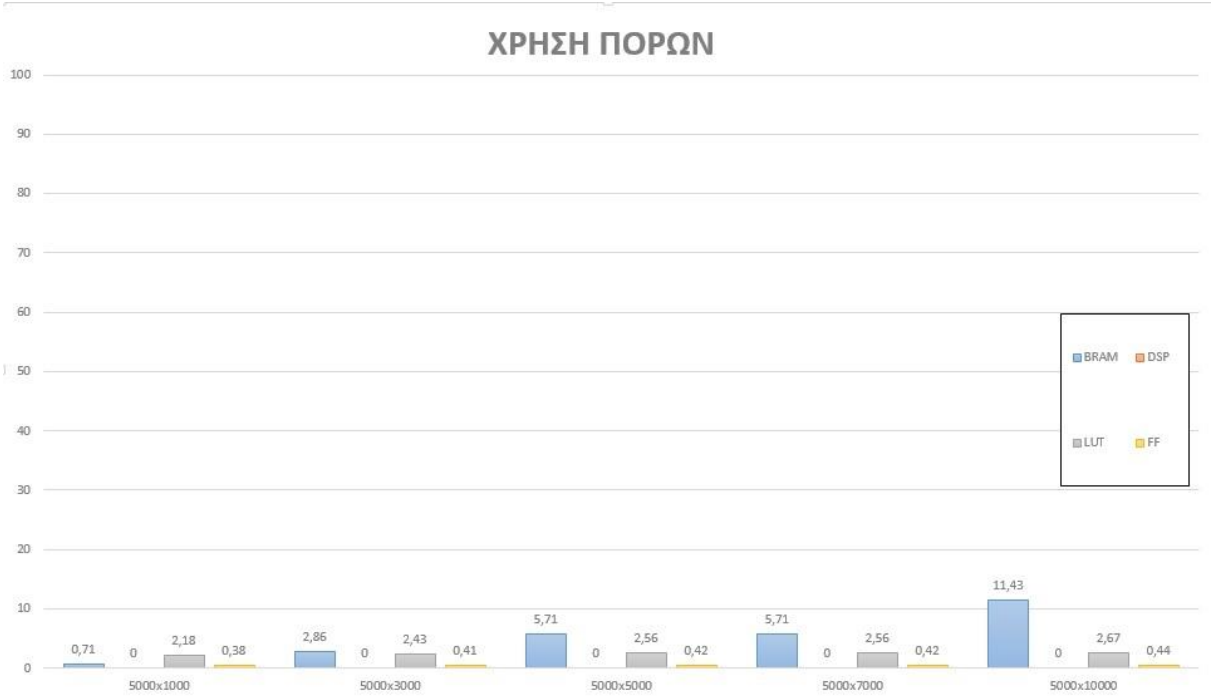
Σχήμα 5.3: Γραμμική απεικόνιση επιτάχυνσης για μεγάλα datasets

Όπως φαίνεται στα παραπάνω γραφήματα, η hardware έκδοση του αλγορίθμου δεν κατάφερε σε καμία των περιπτώσεων να ξεπεράσει σε επιτάχυνση τις software εκδόσεις, όπως ακριβώς είχαμε προβλέψει στο προηγούμενο κεφάλαιο. Η μη δυνατότητα παραλληλοποίησης καθώς και πολλές μεταφορές των δεδομένων από τη μνήμη στο FPGA και πίσω, περιόρισαν αισθητά την ταχύτητα του αλγορίθμου.

Αναλυτικότερα όμως μπορούμε να παρατηρήσουμε τα εξής:

- Όσο αυξάνεται ο όγκος των δεδομένων, τόσο τείνει η hardware υλοποίηση να φθάσει την ταχύτητα εκτέλεσης των software υλοποιήσεων.
- Σε αραιούς πίνακες (sparse) παρατηρούμε πως η δυναμική υλοποίηση είναι σαφώς ανώτερη. Αλλά σε πυκνούς πίνακες (dense), η hardware έκδοση αυξάνει την απόδοσή της αισθητά, χωρίς όμως να καταφέρνει να εξισωθεί με τις υπόλοιπες.
- Η επιτάχυνση, όσο αυξάνουμε τους κόμβους και τις ακμές, τείνει να ακολουθήσει μία γραμμική αύξουσα πορεία.
- Τέλος, παρατηρούμε πως η στατική έκδοση της software υλοποίησης είναι ταχύτερη συγκριτικά με την αντίστοιχη δυναμική έκδοση, όσων αφορά τους πυκνούς πίνακες.

Η απόδοση εξαρτάται και από την πυκνότητα του γράφου, δηλαδή αν έχουμε πολλές ακμές ή αλλιώς από την αναλογία μεταξύ κόμβων και ακμών. Χρησιμοποιήσαμε dataset 1000 και 5000 ακμών, αλλάζοντας τον αριθμό των κόμβων, ώστε να έχουμε ίδιο χρόνο μεταφοράς του dataset. Όσο μεγαλώνουμε τον αριθμό των κόμβων, έχουμε θεωρητικά μεγαλύτερο γράφο και έχοντας σταθερό τον αριθμό των ακμών, έχουμε πιο αραιό γράφο. Σημειώνουμε ότι όσο αυξάνει ο αριθμός των κόμβων, αυξάνεται και ο χρόνος μεταφοράς. Εδώ παρατηρούμε το ρόλο και την εξάρτηση που έχει η απόδοση του αλγορίθμου από την πυκνότητα του γράφου. Γράφοι με μεγαλύτερη πυκνότητα και κατά συνέπεια μεγαλύτερο αριθμό ακμών ανεβάζουν την απόδοση του hardware αλγορίθμου, συγκριτικά πάντα με την αντίστοιχη έκδοση που τρέχει στο software.



Σχήμα 5.4: Χρήση πόρων

Αναλύοντας την χρήση των πόρων του FPGA αντικατοπτρίζεται η φύση του αλγορίθμου, πιο ευκρινώς σε αυτήν την απλή έκδοση. Εφόσον δεν έχουμε παραλληλία, δεν δημιουργούνται παραπάνω κυκλώματα για πράξεις στο FPGA και επόμενα οι πόροι του μένουν ανεκμετάλλευτοι. Η block Ram βλέπουμε ότι είναι αυτή που χρησιμοποιείται παραπάνω από τις άλλες μονάδες του FPGA, καθώς χρειάζεται να περιλαμβάνει τις δομές δεδομένων μας, περιορίζοντας μας στην επίλυση μεγαλύτερων γράφων. Παρόλα αυτά, παρατηρούμε μία πολύ μικρή χρήση των πόρων του συστήματος αναλογικά με τους συνολικούς προσφερόμενους. Αυτό οφείλεται στην υλοποίηση μας, η οποία βασίζεται στο streaming των δεδομένων στην πλατφόρμα του FPGA. Δυστυχώς, δεν μπορέσαμε να ανεβάσουμε το μέγεθος των γράφων που χρησιμοποιήθηκαν γιατί το ίδιο το πρόγραμμα μεταγλώττισης HLS δεν μπορούσε να υπολογίσει την πολυπλοκότητα των παραγόμενων κυκλωμάτων. Ίσως αν είχαμε τη δυνατότητα χρήσης μεγαλύτερων γράφων να είχαμε καλύτερα αποτελέσματα.

Κεφάλαιο 6

Σχετική Έρευνα

6.1 Προσέγγιση αλγορίθμων γράφων με FPGAs

Πραγματοποιώντας έρευνα διαπιστώσαμε πως δεν υπάρχουν κάποια επιστημονικά άρθρα ή δημοσιεύσεις με άμεση σχέση με την παρούσα πτυχιακή, δηλαδή την υλοποίηση του αλγορίθμου των Shiloach-Viskin για την εύρεση των διασυνδεδεμένων στοιχείων. Αντιθέτως, υπάρχουν πολλές δημοσιεύεις σχετικά με την ***Ταμπελοποίηση Συνδεδεμένων Μερών (Connected Components Labeling).***

Η επισήμανση συνδεδεμένων εξαρτημάτων (εναλλακτικά ταμπελοποίηση περιοχών (region labeling), εξαγωγή μπλομπ (blob extraction), ανακάλυψη των μπλομπ (blob discovery), ή εξαγωγή περιοχών (region extraction)) είναι μια αλγοριθμική εφαρμογή της θεωρίας γραφημάτων, όπου υποσύνολα συνδεδεμένων στοιχείων επισημαίνονται με μοναδικό τρόπο με βάση ένα δεδομένο ευρετικό.

Στη τεχνολογία της όρασης υπολογιστών η διαδικασία ταμπελοποίησης συνδεδεμένων μερών επιτρέπει τη κατηγοριοποίηση εικονοστοιχείων μιας ψηφιακής εικόνας σε ξεχωριστές ομάδες. Η ταμπελοποίηση συνήθως εφαρμόζεται σε μια δυαδική εικόνα που είναι αποτέλεσμα από ένα thresholding step. Τα συνδεδεμένα μέρη μπορεί να μετρηθούν, να φιλτραριστούν, και να παρακολουθηθούν.

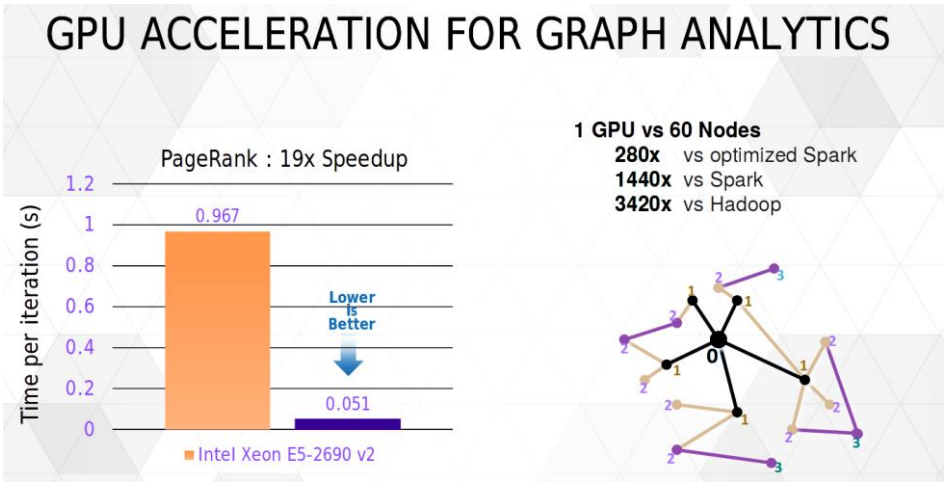
Στη δημοσίευση των Y. Ito και K. Nakano[9] παρουσιάζουν έναν αλγόριθμο υλικού για ταμπελοποίηση συνδεδεμένων μερών με χαμηλό latency που εφαρμόζεται σε δυαδικές εικόνες και σχεδιάσμενος να τρέχει σε FPGA. Όπως αντιμετωπήσαμε αρχικά και εμείς παρόμοιο πρόβλημα με τη μνήμη, έτσι και εδώ είναι δυσκόλο να υλοποιήσουν τη μέθοδο ταμπελοποίησης καθώς χρειάζεται να αποθηκεύουν προσωνινές ταμπέλες για κάθε pixel. Η μικρή εσωτερική μνήμη δεν επιτρέπει να την αποθήκευση των ταμπελών για εικόνες μεγάλης ανάλυσης με αποτέλεσμα να πρέπει να προσφύγεις στη χρήση DRAM, το οποίο είναι κοστοβόρο και σε ταχύτητα αλλά και σε κατανάλωση ενέργειας. Με διάφορες αλλαγές στην υλοποίηση τους καταφέρνουν να περάσουν τα pixels σε σειρά ράστερ με αποτέλεσμα να έχουν χαμηλό latency και να χρησιμοποιούν μικρό κομμάτι της εσωτερικής μνήμης. Βάση της απόδοσης του τελικού αλγορίθμου, καταλήγουν στο συμπέρασμα πως μπορεί να χρησιμοποιηθεί σε ένα σύστημα αναγνώρισης εικόνας, χωρίς όμως να τον συγκρίνουν με αντίστοιχα συστήματα που χρησιμοποιούν επεξεργαστές γενικού σκοπού.

6.2 Χρήση GPUs για υλοποίηση αλγορίθμων γράφων

Οι GPUs είναι άλλος ένας τρόπος επιτάχυνσης αλγορίθμων, με ευρεία ήδη χρήση. Είναι μονάδες επεξεργασίας γραφικών που σε συνεργασία με τις CPU επιταχύνουν εφαρμογές βαθιάς μάθησης, ανάλυσης και μηχανικής. Με πρωτοπόρο την NVIDIA από το 2007, οι επιταχυντές GPU εξοπλίζουν πλέον ενεργειακά αποδοτικά κέντρα δεδομένων σε κρατικά εργαστήρια, πανεπιστήμια, επιχειρήσεις όλων των μεγεθών ανά τον κόσμο. Διαδραματίζουν τεράστιο ρόλο στην επιτάχυνση εφαρμογών σε πλατφόρμες που κυμαίνονται από την τεχνητή νοημοσύνη μέχρι τα αυτοκίνητα, τα drones και τα ρομπότ.

Στη δημοσίευση τους, οι J. Soman, K. Kishore, and P. J. Narayanan[10], επικεντρώνονται στο να βρουν τα διασυνδεδεμένα στοιχεία ενός γράφου με τη χρήση GPU. Η εύρεση των διασυνδεδεμένων στοιχείων ενός γράφου είναι το πρώτο σκέλος σε ένα πλήθος αλγορίθμων γράφων. Ξεκινώντας από τον αλγόριθμο Shiloach-Vishkin και αναλύοντας τη λειτουργία του, καταλήγουν στο συμπέρασμα πως ίσως δεν είναι κατάλληλος για σύγχρονες αρχιτεκτονικές όπως οι CPUs. Ένας από τους λόγους είναι ο μεγάλος αριθμός ακανόνιστων προσπελάσεων στη μνήμη. Βασιζόμενοι όμως σε αυτόν, προτείνουν μία υλοποίηση η οποία επιτυγχάνει μία επιτάχυνση 9 με 12 φορές συγκριτικά με την καλύτερη δυνατή υλοποίηση σε CPU.

Την πεποίθηση ότι οι αλγόριθμοι γράφων ευδοκιμούν στις GPUs, έρχεται να στηρίξει και η παρουσίαση της NVIDIA. Στο συνέδριο Geoint το 2015, ο L. Brown (Solution Architect της NVIDIA), παρουσίασε υλοποιήσεις εφαρμογών στοιχείων ανάλυσης γράφων με GPUs [11]. Ένας από αυτούς ήταν και το Connected Components. Σύγκρινε συνολικά τους αλγόριθμους γράφων, εκτελούμενους αρχικά σε μία GPU και έπειτα σε cluster 60 κόμβων. Η εκτέλεση σε GPU ήταν 3420 φορές ταχύτερη από την εκτέλεση σε Hadoop cluster, 1440 φορές ταχύτερη από την εκτέλεση σε Spark cluster και 280 φορές ταχύτερη από την εκτέλεση σε βελτιωμένη έκδοση σε Spark cluster.



Σχήμα 6.1: Αποδόσεις GPUs συγκριτικά με CPUs

6.3 Σχετική έρευνα για δημιουργία επεξεργαστών ανάλυσης γράφων

Η ανάγκη αποδοτικής υπολογιστικής διαχείρισης των μεγάλων γράφων έχει στρέψει το ενδιαφέρον σε εταιρείες της τεχνολογικής κοινότητας, ώστε να κινηθούν πέρα από τις υπάρχουσες λύσεις σε επεξεργαστές, FPGAs και GPUs. Αναπτύσσουν τεχνολογία για την δημιουργία επεξεργαστών ανάλυσης γράφων.

Η Υπηρεσία Προηγμένων Ερευνητικών Έργων Άμυνας (DARPA) είναι μια υπηρεσία του Υπουργείου Άμυνας των Ηνωμένων Πολιτειών, υπεύθυνη για την ανάπτυξη αναδυόμενων τεχνολογιών για χρήση από τους στρατιωτικούς. Η εταιρεία αυτή αναπτύσσει το πρόγραμμα DARPA HIVE [12] θέλοντας να δημιουργήσει έναν επεξεργαστή ανάλυσης γράφων, ο οποίος μπορεί να επεξεργαστεί ρέοντες γράφους 1000X πιο γρήγορα και σε πολύ χαμηλότερη ισχύ από την τρέχουσα τεχνολογία επεξεργασίας. Έτσι θα δίνεται η δυνατότητα για προηγμένη ανάλυση γράφων και επιλύσεις προκλήσεων σε τομείς όπως η ασφάλεια στον κυβερνοχώρο και η παρακολούθηση της υποδομής. Για την ανάπτυξη του επεξεργαστή η DARPA επέλεξε να συνεργαστεί [13] με τις Intel Corporation (Santa Clara, California), Qualcomm Intelligent Solutions (San Diego, California), Pacific Northwest National Laboratory (Richland, Washington), Georgia Tech (Atlanta, Georgia), και Northrop Grumman (Falls Church, Virginia).

Παράλληλα με την ανάπτυξη υλικού ενός επεξεργαστή HIVE, η DARPA συνεργάζεται με το εργαστήριο MIT Lincoln και την Amazon Web Services (AWS) για να φιλοξενήσει την πρόκληση HIVE Graph Challenge με στόχο την ανάπτυξη ενός συνόλου δεδομένων τρισεκατομμυρίων ακμών. Ο στόχος είναι να επιταχυνθεί η καινοτομία στην ανάλυση γράφων για να ανοίξουν νέες οδοί για την αντιμετώπιση της πρόκλησης της κατανόησης ενός συνεχώς αυξανόμενου χείμαρρου δεδομένων.

Σε παρόμοια λογική με την DARPA κινήθηκε και η ThinCI. Η ThinCI [14] είναι μια οκτώ ετών startup από την Καλιφόρνια. Στο συνέδριο Hot Chips παρουσίασε τον "επεξεργαστή ρέοντων γράφων" (GSP) της εταιρείας. Η ThinCI αναπτύσσει ολοκληρωμένα κυκλώματα για μηχανική εκμάθηση και όραση υπολογιστών και δήλωσε πως είναι έτοιμη να αναπτύξει τον GSP και τον μεταγλωττιστή για ανάπτυξη εφαρμογών γράφων. Αν και στο συνέδριο οι υπόλοιπες εταιρίες ζήτησαν παραπάνω πειστήρια, της αναγνώρισαν ότι ορισμένα βασικά στοιχεία που έχουν σχεδιαστεί στον GSP είναι μοναδικά, καθιστώντας την αρχιτεκτονική του αντάξια του ισχυρισμού της "επόμενης γενιάς".

Κεφάλαιο 7

Συμπεράσματα

7.1 Σύνοψη

Σε αυτήν την διατριβή αναλύσαμε πως εκτελείται ο αλγόριθμος των Shiloach-Vishkin για την εύρεση του αριθμού των Connected Components, τις δομές που χρειάζεται να υλοποιήσει και πως αυτές πρέπει να τις χειριστούμε ώστε να ενσωματωθούν στο FPGA. Γνωρίζοντας τις μικρές καταναλώσεις ενέργειας των FPGAs, συγκριτικά με τις CPUs, προσπαθήσαμε να υλοποιήσουμε μια αποδοτική εφαρμογή του αλγορίθμου των Shiloach-Vishkin. Τα αποτελέσματα δεν ήταν τα επιθυμητά. Χαρακτηριστικό του αλγόριθμου ήταν η έντονη χρήση της μνήμης και η ανάλογα μικρή υπολογιστική χρήση. Από το γεγονός αυτό καθαυτό, εργαζόμασταν αντίθετα στην λογική των FPGAs, της γρήγορης εκτέλεσης υπολογιστικού φόρτου. Εντούτοις, τα συμπεράσματα που βγήκαν δεν ήταν απαγορευτικά.

7.2 Μελλοντική Δουλειά

Τα περιθώρια βελτίωσης στην υπάρχουσα αρχιτεκτονική και υλοποίηση που χρησιμοποιήσαμε είναι ίσως περιορισμένα. Η χρήση του Xilinx ZC702, αναφερόμενοι στους περιορισμένους πόρους, και η μη χρήση της DRAM, έθεσαν τους περιορισμούς. Πρέπει να ερευνηθεί η χρήση της DRAM ώστε να χρησιμοποιείται αποδοτικά στις υλοποιήσεις σε FPGAs. Αλγόριθμοι όπως η εύρεση του πλήθους των Connected Components, που αφορούν μεγάλους γράφους χρειάζεται να μπορούν να κατανεμηθούν σε συστάδες και μάλιστα διαμοιραζόμενης μνήμης. Από την άλλη πλευρά, υπάρχει μεγάλη εξάρτηση δεδομένων και τυχαίες προσπελάσεις μνήμης, που δυσκολεύουν την υλοποίηση με τη χρήση κοινής μνήμης. Βέβαια, υπάρχουν περιθώρια βελτίωσης.

Σύμφωνα μάλιστα με την σχετική δουλειά που αναλύσαμε στο 6ο κεφάλαιο, υπάρχουν περαιτέρω δυνατότητες αποδοτικότερης, περιορισμένες βέβαια, ανάπτυξης εφαρμογών γράφων σε FPGAs. Από την άλλη, οι GPUs, σύμφωνα με την έρευνα μας, δείχνουν πως υπερέχουν σε αποδόσεις συγκριτικά με τα FPGAs, στις εφαρμογές ανάλυσης γράφων. Ιδιαίτερα οι εφαρμογές που βασίζονται στον αλγόριθμο για Connected Components Labeling για την ανάλυση εικόνων, παρουσιάζουν θεαματικά αποτελέσματα και υπάρχει αρκετή έρευνα πάνω σε αυτό το κομμάτι. Οπότε κρίνουμε πως η προσέγγιση αλγόριθμων γράφων με GPUs μπορεί να επιφέρει θετικά αποτελέσματα.

Τέλος, εταιρίες όπως η DARPA και η ThinCI έχουν προβεί στην ανάπτυξη GSP επεξεργαστών, υποσχόμενες να ξεπεράσουν GPUs και CPUs στην ανάπτυξη εφαρμογών γράφων. Ελπίζουμε να δούμε αποτελέσματα αυτών σύντομα. Αν υλοποιηθούν οι GSPs, ικανοποιώντας τα χαρακτηριστικά που τους προλογούν, πιθανότατα στο μέλλον, θα αποτελέσουν την βάση για τις εφαρμογές ανάλυσης γράφων.

Βιβλιογραφία

[1] Understanding FPGA Architecture
https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/devices/confpga-architecture.html

[2] Zynq-7000 All Programmable SoC Data Sheet: Overview
https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000Overview.pd

[3] AXI Reference Guide
https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[4] Introduction to FPGA Design with Vivado High-Level Synthesis
https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpgadesign-hls.pdf

[5] Vivado Design Suite User Guide 902 High-Level Synthesis
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902vivado-high-level-synthesis.pdf

[6] Shiloach, Y. & Vishkin, U. (1980), An O (log n) Parallel Connectivity Algorithm

[7] Bader, D. A., Cong, G., Feo, J. (2005), On the Architectural Requirements for Efficient Execution of Graph Algorithms, *Proceedings of the 2005 International Conference on Parallel Processing (ICPP'05)*

[8] GAP Benchmark Suite
http://gap.cs.berkeley.edu/benchmark.html

[9] Ito, Y., Nakano, K. (2009), Low-Latency Connected Component Labeling Using an FPGA, *International Journal of Foundations of Computer Science*

[10] Soman, J., Kishore, K., Narayanan, P., J. (2010), A fast GPU algorithm for graph connectivity, *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*

[11] NVIDIA - Graph Analytics with GPUs, GEOINT2015
http://www.nvidia.com/content/events/geoInt2015/LBrown_Intro_Graph%20Analytics.pdf

[12] DARPA - Hierarchical Identify Verify Exploit (HIVE)
https://www.darpa.mil/program/hierarchical-identify-verify-exploit

[13] Extracting Insight from the Data Deluge Is a Hard-to-Do Must-Do
https://www.darpa.mil/news-events/2017-06-02

[14] Startup Unveils Graph Processor at Hot Chips
https://www.eetimes.com/document.asp?doc_id=1332176