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

# Accelerating PageRank Graph Algorithms in Reconfigurable Logic

Διπλωματική Εργασία

**ΔΗΜΗΤΡΙΟΣ Ν. ΤΡΙΑΝΤΑΦΥΛΛΟΥ**

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

Εργαστηρίο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Αθήνα, Ιανουάριος 2018

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

# Accelerating PageRank Graph Algorithms in Reconfigurable Logic

## Διπλωματική Εργασία

**ΔΗΜΗΤΡΙΟΣ Ν. ΤΡΙΑΝΤΑΦΥΛΛΟΥ**

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31η Ιανουαρίου 2018.

| | | |
|---|---|---|
| ..... | ..... | ..... |
| Δημήτριος Σούντρης | Κιαμάλ Πεκμεστζή | Γεώργιος Γκούμας |
| Αναπληρωτής Καθηγητής | Καθηγητής | Καθηγητής |

Αθήνα, Ιανουάριος 2018

*(Υπογραφή)*

.....

**Δημήτριος Ν. Τριανταφύλλου**

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, για όλα όσα μου προσέφεραν κατά την διάρκεια φοίτησης στην σχολή.

# Περίληψη

Η θεωρία των γράφων έχει αναγνωριστεί ως ένα από τα πιο χρήσιμα αντικείμενα στην επιστήμη των υπολογιστών. Η προσέγγιση της από την επιστήμη των υπολογιστών γίνεται μέσω αλγορίθμων. Το ενδιαφέρον έγκειται στην εύρεση αποδοτικών αλγορίθμων για την επίλυση σχετικών προβλημάτων.

Οι γράφοι αναλαμβάνουν να αναπαραστήσουν τις σχέσεις μεταξύ των στοιχείων ενός συνόλου. Στην καθημερινότητα μας, ένας γράφος μπορεί να είναι ένα οδικό δίκτυο, ένα δίκτυο υπολογιστών, ένα κοινωνικό δίκτυο ή όποιας μορφής δίκτυο δημιουργείται μέσω μιας σχέσης σε ένα σύνολο. Κύρια χαρακτηριστικά των γράφων είναι ο μεγάλος τους όγκος και ο τυχαίος τρόπος διασύνδεσης τους. Αυτά τα χαρακτηριστικά είναι που μεγενθύνουν τον υπολογιστικό φόρτο και ως επακόλουθο τον χρόνο επίλυσης των αλγορίθμων που τους αφορούν. Γίνεται εύκολα κατανοητό ότι για να επεξεργαστεί και να μας δώσει την πληροφορία που θέλουμε να εξάγουμε ένας γράφος, που αναπαριστά ένα κοινωνικό δίκτυο για παράδειγμα, απαιτείται μεγάλη υπολογιστική ισχύς και αρκετός χρόνος. Όσο όμως η επιστήμη εξελίσσεται, τα δεδομένα και οι συσχετίσεις αυτών πολλαπλασιάζονται. Έτσι κρίνεται αναγκαίο τα υπολογιστικά αυτά προβλήματα να επιλύονται πιο γρήγορα. Η πολυπλοκότητα των αλγορίθμων γράφων και ο υπολογιστικός τους φόρτος καθιστούν την εκτέλεση τους από μονάδες γενικού σκοπού μη αποδοτική. Σε αυτήν την διπλωματική θα διερευνήσουμε την αποδοτικότητα της υλοποίησης τους από ειδικού σκοπού υλικό, FPGA. Πιο συγκεκριμένα θα ασχοληθούμε με τον αλγόριθμο Pagerank. Για την υλοποίηση του αλγορίθμου θα χρησιμοποιήσουμε FPGA της εταιρίας Xilinx και συγκεκριμένα το ZYNQ ZC702. Θα διερευνήσουμε ευνοϊκές τεχνικές και δομές δεδομένων για τον ίδιο τον αλγόριθμο, αλλά και την χρήση του FPGA. Στόχος μας είναι η επιτάχυνση του αλγορίθμου.

## Λέξεις Κλειδιά

Θεωρία γράφων, Pagerank, FPGA, δομές δεδομένων, υπολογιστικός φόρτος, πολυπλοκότητα, παραλληλία, επιτάχυνση, ZYNQ ZC702

# Abstract

Graph theory has been recognized as one of the most useful objects in computer science. Its approach from computer science is through algorithms. The interest lies in finding efficient algorithms to solve relevant problems.

Graphs are used to represent the relationships between the elements of a set. In our everyday life, a graph can be a road network, a computer network, a social network or any form of network that is created through a relationship in a set. The main characteristics of graphs are their large volume and the random way of interconnecting them. These features increase the computational load and therefore, the time to solve the algorithms that concern them. It is easy to understand that in order to process and give us the information we want to export a graph representing a social network, for example, it requires a lot of computational power and enough time. As long as science evolves, these data and associations are multiplied. It is therefore necessary to solve these computational problems more quickly. The complexity of graph algorithms and their computing load make their execution by general purpose units inefficient. In this diploma we will research the efficiency of their implementation by special purpose hardware, FPGA. More specifically, we will deal with the Pagerank algorithm. To implement the algorithm, we will use the Xilinx FPGA and ZYNQ ZC702 in particular. We will explore techniques and data structures suitables for the algorithm itself, as well as the use of the FPGA. Our goal is to accelerate the algorithm.

## Keywords

Graph Theory, Pagerank, FPGA, Data Structures, Computational Load, Complexity, Parallel, Acceleration, ZYNQ ZC702

# Περιεχόμενα

# Κατάλογος σχημάτων

# Κατάλογος πινάκων

# Κεφάλαιο 1

# Εισαγωγή

## 1.1 Αντικείμενο της διπλωματικής

Η αποτελεσματική επεξεργασία γράφων μεγάλης κλίμακας σε κατανεμημένα συστήματα αποτελεί ένα όλο και πιο δημοφιλές θέμα της έρευνας τα τελευταία χρόνια. Διασυνδεδεμένα δεδομένα που μπορούν να διαμορφωθούν ως γράφοι προκύπτουν σε τομείς εφαρμογών, όπως μηχανική μάθηση, σύσταση, αναζήτηση ιστού και ανάλυση κοινωνικών δικτύων. 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, αντιστοίχισης προτύπων, ανάλυσης δικτύων γράφων και διαδρομών γράφων. Έχουν προταθεί και εγκριθεί πολλές ιδέες σε προγραμματισμό υψηλού επιπέδου με κατανεμημένα συστήματα επεξεργασίας γράφων και μεγάλες πλατφόρμες δεδομένων. Παρόλο που έχει γίνει σημαντική δουλειά πειραματικών πλαισίων επεξεργασίας γράφων, δεν έχει βρεθεί βέλτιστη λύση σε αρκετούς αλγορίθμους.

Στην διπλωματική αυτή θα μελετήσουμε την θεωρία των γράφων και συγκεκριμένα θα ασχοληθούμε με τον αλγόριθμο Pagerank. Παράλληλα θα μελετήσουμε και θα αναλύσουμε την χρήση των FPGA, για υλοποίηση αλγορίθμων. Πλεονεκτήματα που αυτά προσφέρουν και θα υλοποιήσουμε έναν επιταχυντή υλικού του αλγορίθμου.

## 1.2 Οργάνωση του τόμου

Η διατριβή αυτή οργανώνεται ως εξής:

1. Στο Κεφάλαιο 2 αναλύουμε το θεωρητικό υπόβαθρο που χρεαζόμαστε σχετικά με τα FPGAs και ευνοϊκές τεχνικές για την χρήση τους.

2. Στο Κεφάλαιο 3 αναφερόμαστε στον αλγόριθμο Pagerank. Αρχίζουμε από την θεωρία των γράφων, μελέτη των δομών δεδομένων για αναπαράσταση του γράφου και επίλυση του αλγορίθμου. Έπειτα δίνουμε την μαθηματική του δομή. Τέλος αναλύουμε τον πολλαπλασιασμό αραιού πίνακα με διάνυσμα.

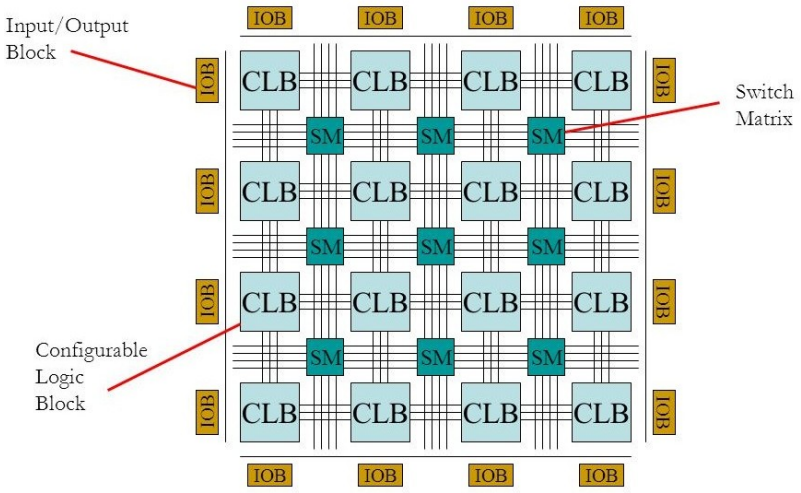
3. Στο Κεφάλαιο 4 αναλύουμε την σχεδίαση της αρχιτεκτονικής του αλγορίθμου στο υλικό.
4. Στο Κεφάλαιο 5 παραθέτουμε τα αποτελέσματα και αξιολογούμε τους επιταχυντές υλικού που υλοποιήσαμε.
5. Στο Κεφάλαιο 6 ερευνούμε δημοσιεύσεις σχετικές με την υλοποίηση αλγορίθμων γράφων σε FPGAs και άλλες τεχνολογίες.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



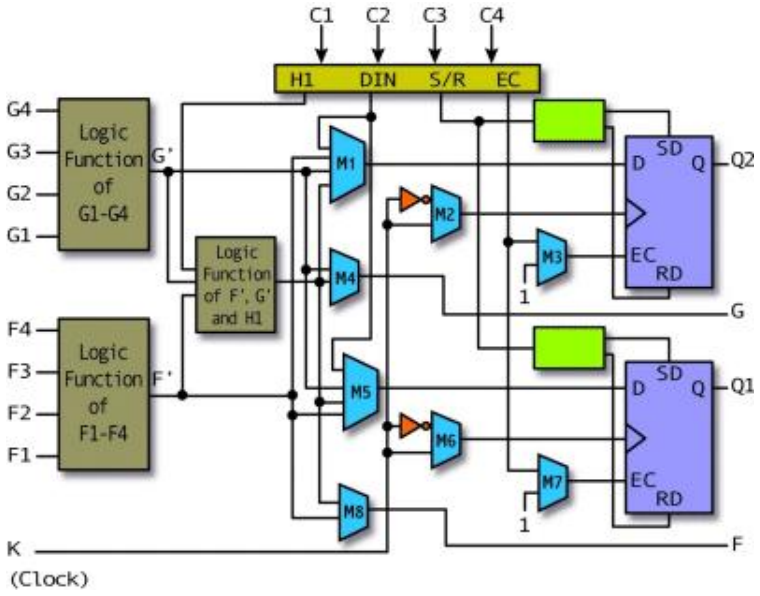

Σχήμα 2.1: Βασική δομή FPGA

## 2.2 Δομικά στοιχεία των FPGAs

Η βασική δομή ενός FPGA [2] περιλαμβάνει τα εξής στοιχεία:

• **Configurable Logic Blocks (CLB)**

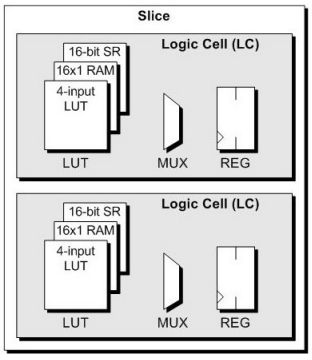
Τα μπλοκ αυτά περιέχουν την λογική για τα FPGA και δημιουργούν μία μικρή μηχανή καταστάσεων, όπως φαίνεται στο παρακάτω σχήμα. Περιέχουν πίνακες αναζήτησης (LUTs), Flip-Flops , πολυπλέκτες για την δρομολόγηση της λογικής εντός του μπλοκ και προς εξωτερικούς πόρους. Οι πολυπλέκτες επιτρέπουν επίσης την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




Σχήμα 2.2: Δομή CLB

• **Slices**

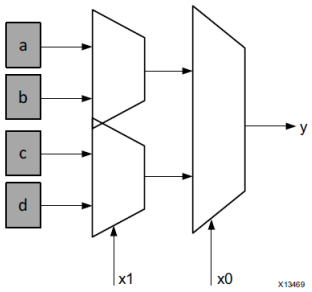
Υπομονάδα των CLBs. Στα FPGA της Xilinx δύο ή τέσσερα slices συνθέτουν ένα CLB.




Σχήμα 2.3: Δομή slice

• **Look-up tables (LUTs)**

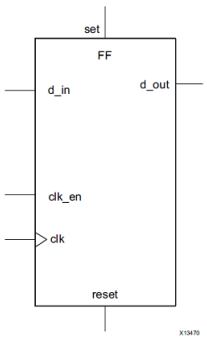
Στοιχεία που είναι ικανά να υλοποιήσουν οποιαδήποτε λογική λειτουργία. Ουσιαστικά, αυτό το στοιχείο είναι ένας πίνακας αλήθειας στον οποίο οι διάφοροι συνδυασμοί των εισόδων που εφαρμόζονται, υλοποιούν διαφορετικές λειτουργίες για να δώσουν τιμές εξόδου. Λόγω της μεγάλης ευελιξίας τους, δύναται να χρησιμοποιούνται 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 συτήματος καθώς και να αποθηκεύουν το 1bit για παραπάνω από έναν παλμό ρολογιού. Συστάδα αυτών υλοποιούν τους καταχωρητές.

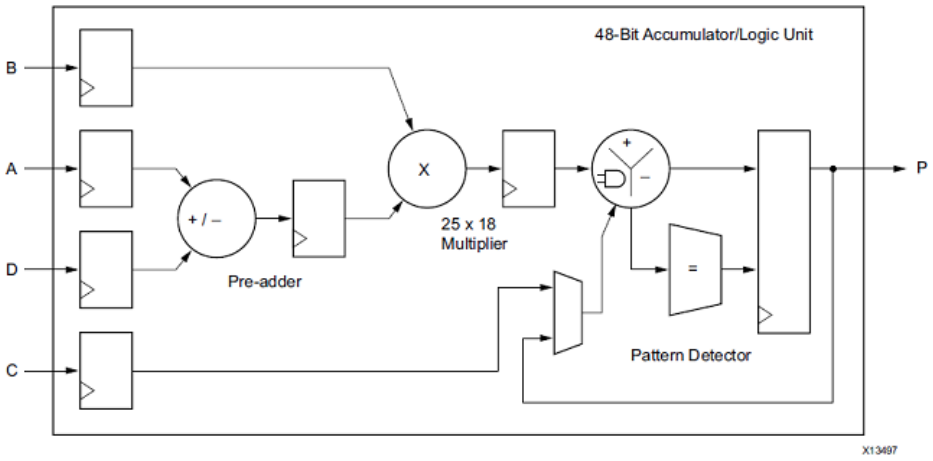



Σχήμα 2.5: Δομή Flip-Flop

• **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

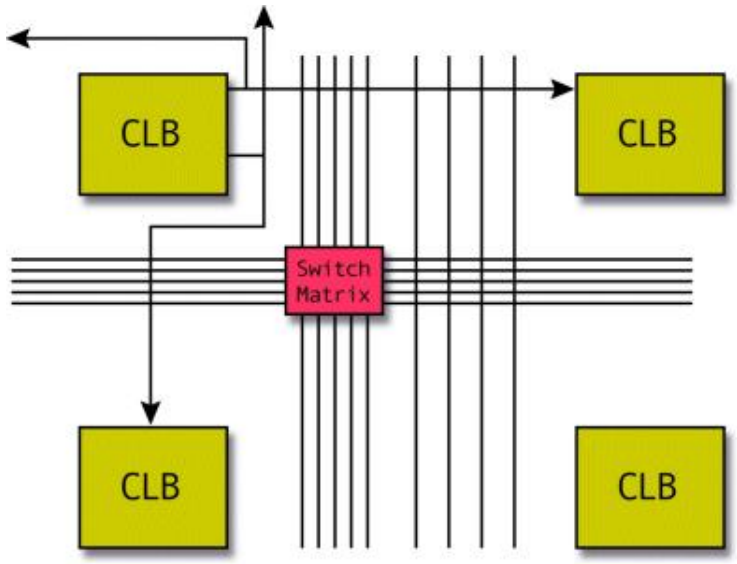
έναν πολλαπλασιαστή που συνδέται στην τελική μονάδα πρόσθεσης / αφαίρεσης / συσσώρευσης. Αυτή η αλυσίδα επιτρέπει να υλοποιηθεί σε ένα μόνο DSP48, συναρτήσεις της μορφής: $P = Bx(A + D) + C$ και να επωμιστεί μεγάλο υπολογιστικό φόρτο.




Σχήμα 2.6: Δομή DSP48

• **Προγραμματιζόμενες Διασυνδέσεις**

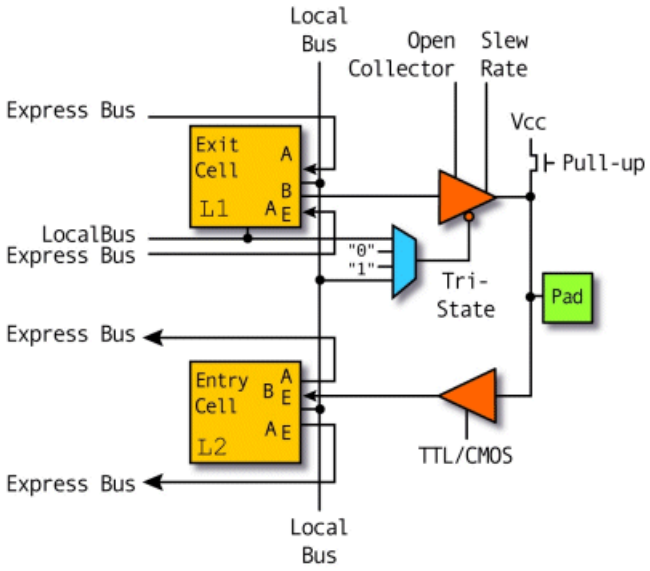
Είναι γραμμές (καλώδια) που διασυνδέουν τα στοιχεία μεταξύ τους χωρίς καθυστέρηση. Χρησιμοποιούνται και ως δίαυλοι μέσα στο chip. Τρανζίστορ χρησιμοποιούνται για να συνδεθούν ή να αποσυνδεθούν διαφορετικές γραμμές. Επίσης υπάρχουν προγραμματιζόμενοι πίνακες αλλαγής (Switch Matrices - SM ), όπως φαίνεται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




Σχήμα 2.7: Προγραμματιζόμενη διασύνδεση

• **Μπλοκ Εισόδου/Εξόδου (IOBs)**

Τα μπλοκ αυτά μεταφέρουν τα δεδομένα εντός και εκτός του FPGA. Περιέχουν έναν buffer εισόδου, έναν buffer εξόδου τριών καταστάσεων και ανοικτού συλλέκτη έξοδο. 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)**

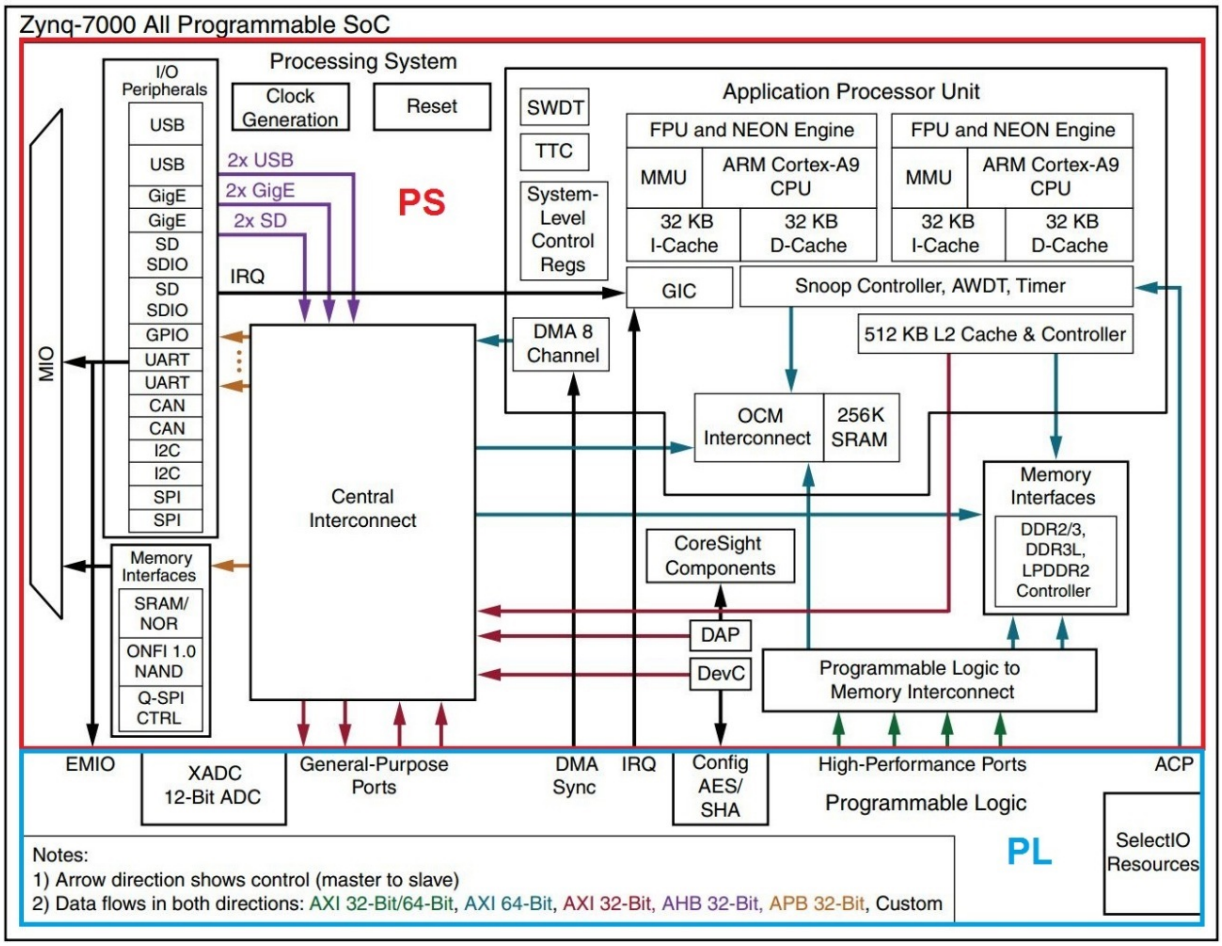
Είναι μία δύο-θυρών μνήμη τυχαίας προσπέλασης (RAM), ενσωματωμένη στο FPGA, που προσφέρει την δυνατότητα αποθήκευσης σχετικά μεγάλου όγκου δεδομένων. Η διασύνδεση της μέσω δύο θυρών επιτρέπει την παράλληλη, σε ίδιο κύκλο ρολογιού πρόσβαση σε διαφορετικές τοποθεσίες μνήμης.

• **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 γενειά των Zynq-7000 All Programmable SoC περιλαμβάνει ετερογενείς υπολογιστικές πλατφόρμες που συνδιάζουν 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 μηκός 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 [3]

### 2.3.1 Σύστημα Επεξερασίας PS

Το τμήμα του Συστήματος Επεξεργασίας (PS) [3] 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 512 kB δεύτερου επιπέδου, οκτώ δρόμων συσχέτισης και μία μνήμη OCM (On-Chip Memory) 256 kB 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 γενειάς 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 όπως φαίνονται και στην εικόνα 2.9 περιγράφονται κάτωθι.

- **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), χρησιμοποεί πρωτόκολλο επικοινωνίας AXI (Advanced eXtensible Interface) [4]. 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) εώς 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.

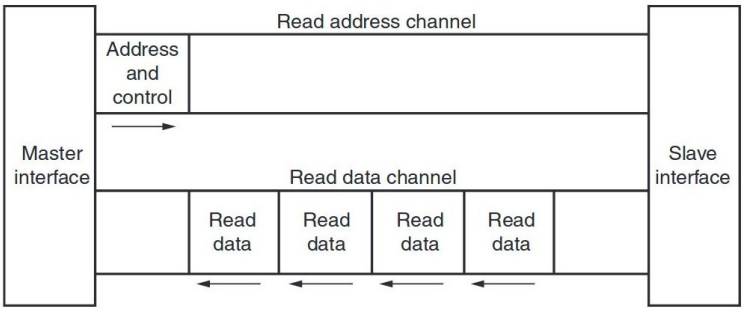
• **AXI4-Stream**

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

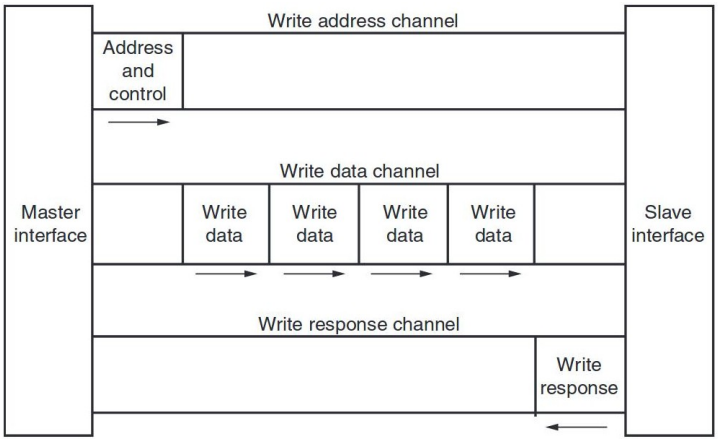
Οι προδιαγραφές AXI περιγράφουν την διεπαφή μίας μονάδας AXI master και μίας AXI slave, που αντιπροσωπεύουν IP cores και ανταλάσσουν πληροφορίες μεταξύ τους. Αμφότερες οι διεπαφές τύπου AXI4 και AXI4-Lite περιέχουν πέντε κανάλια.

- Κανάλι Ανάγνωσης Διέυθυνσης
- Κανάλι Εγγραφής Διέυθυνσης
- 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 εώς 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.

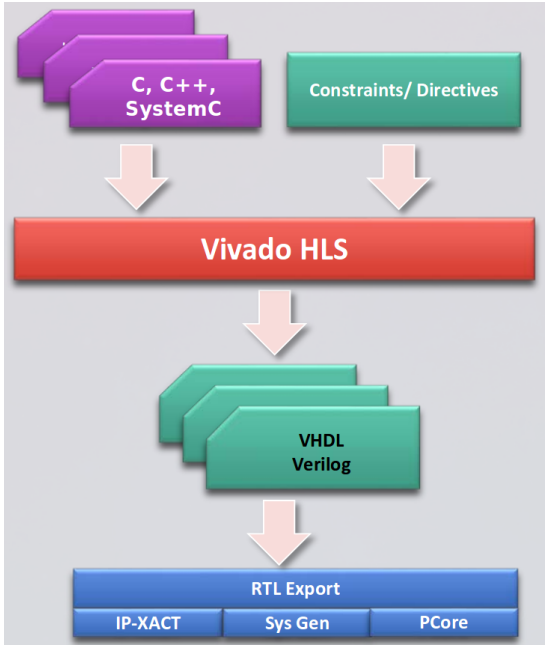
- **General Purpose Ports.** Δίαυλοι δεδομένων 32-bit επικοινωνίας. Η οικογένεια Zynq-7000 περιέχει τέσσερις τέτοιους διαύλους. Οι δύο από αυτούς έχουν ως master το PS και ως slave το PL και οι άλλοι δύο αντίστροφα. Επιτρέποντας έτσι και στα δύο μέρη να διευθετήσουν ή να ξεκινήσουν μια μεταφορά δεδομένων 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 εώς 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. Κατα αυτόν τον τρόπο ο επεξεργαστής ενημερώνεται άμεσα για αλλαγές που πραγματοποιεί το PL στις τιμές των δεδομένων.Η οικογένεια Zynq-7000 περιέχει έναν τέτοιο διαύλο, 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.



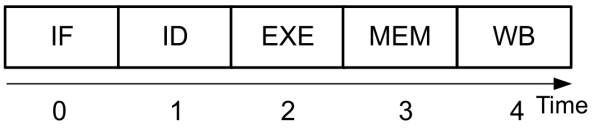

Σχήμα 2.12: Vivado HLS

### 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 επηρρεάζει άμεσα το τρόπο που λειτουγεί ο μεταλωτιστής στις δύο περιπτώσεις. Στην περίπτωση του επεξεργαστή, η υπολογιστική αρχιτεκτονική είναι συγκεκριμένη και σκοπός του μεταγλωτιστή είναι 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 [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 .

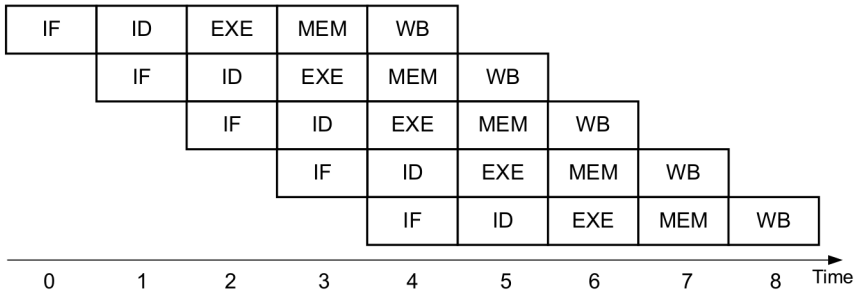



Σχήμα 2.13: Στάδια Εκτέλεσης Εντολών Επεξεργαστή

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

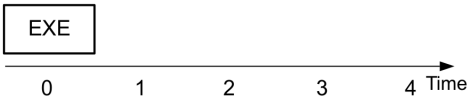
- **Instruction Fetch (IF).** Φόρτωση εντολής από την μνήμη.
- **Instruction Decode (ID).** Αποκωδικοποίηση της εντολής, καθορισμός της διαδικασίας και των ορισμάτων της.
- **Execution (EXE).** Εκτέλεση της εντολής σε διαθέσιμο υλικό, συγκερκριμένα στην Αριθμητική Λογική Μονάδα (ALU) ή στην Μονάδα Κινητής 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 επικάλυψη. 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 pipeline (διοχέτευση). Ακόμα και αν ο μεταγλωττιστής αποφάσιζε ότι πολλά στάδια EXE μπορούν να εκτελεστούν ταυτόχρονα, η δομή των σετ εντολών δεν θα το επιτρέπε. Αποτέλεσμα αυτών είναι ότι στον επεξεργαστή, μέχρι μία εντολή μπορεί να ολοκληρώνεται ανά κύκλο ρολογιού.



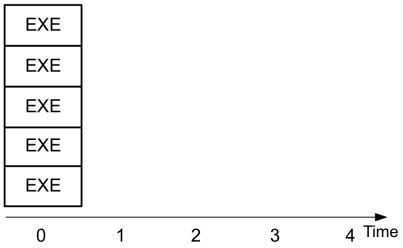

Σχήμα 2.14: Επεξεργαστής με Πολλαπλές Μονάδες Εκτέλεσης Σταδίων

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.

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

**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.

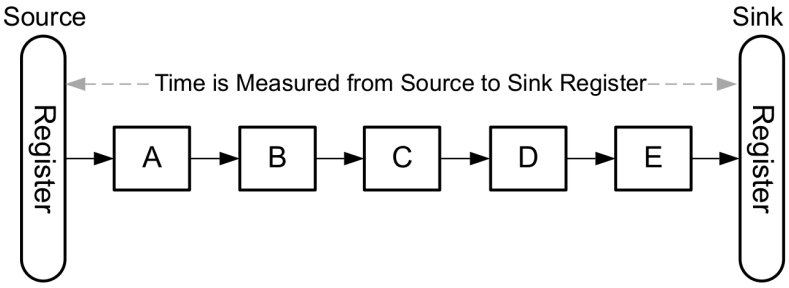
**Latency και Pipelining**

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 την 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 την 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.

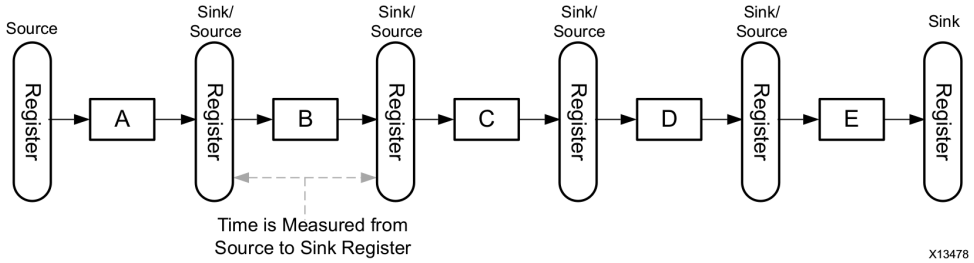
Κατά την υλοποίηση του υλικού στο FPGA, οι εντολές δομούνται σε ένα κύκλωμα. Κάθε εντολή αντιστοιχεί σε ένα μονοπάτι - υποκύκλωμα. Αυτά τα μονοπάτια ορίζονται ανάμεσα σε έναν καταχωρητή προέλευσης και σε έναν καταχωρητή προορισμού (Sink Register). Ο χρόνος μετάδοσης του σήματος σε αυτά τα μονοπάτια καθορίζει 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

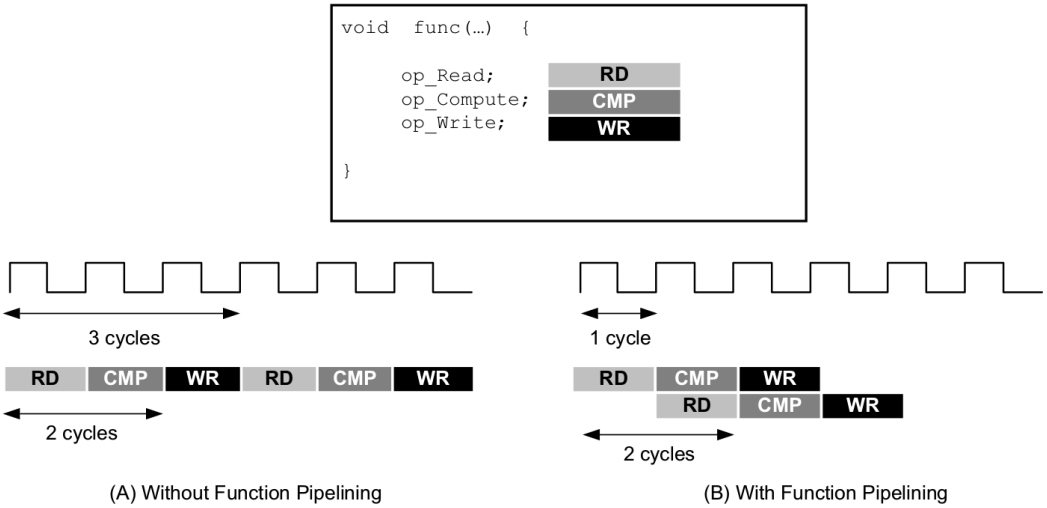
Το Pipelining είναι μια τεχνική ψηφιακού σχεδιασμού που επιτρέπει στον σχεδιαστή να αποφεύγει εξαρτήσεις δεδομένων και να αυξάνει το επίπεδο παραλληλισμού σε μια υλοποίηση υλικού του αλγορίθμου. Η εξάρτηση δεδομένων στην αρχική υλοποίηση του λογισμικού διατηρείται για λειτουργική ισοδυναμία, αλλά το 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, προστίθονται ενδίαμεσοι 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.



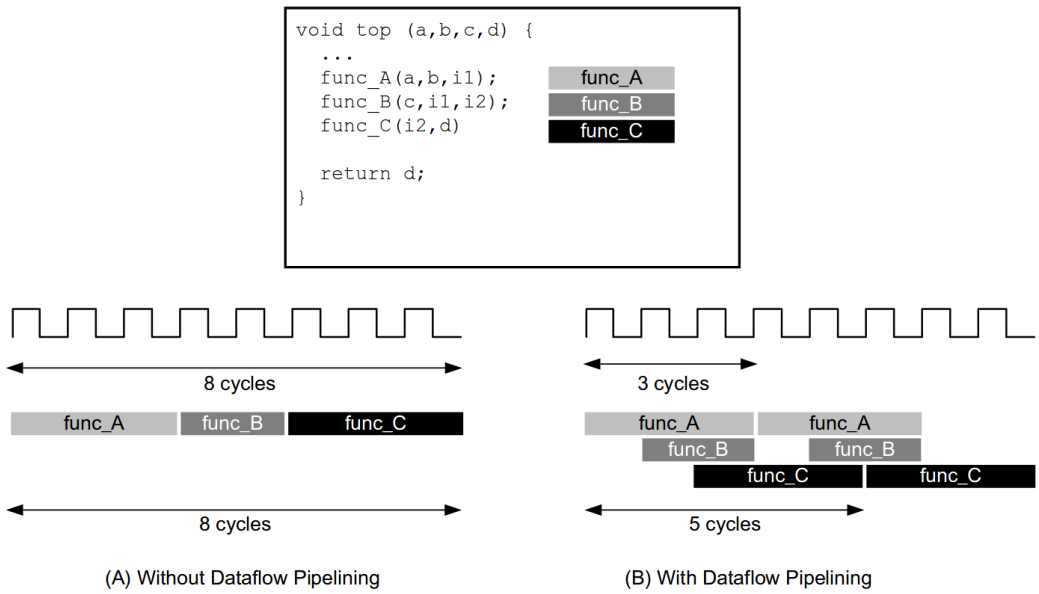

Σχήμα 2.19: Pipeline

### 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 προγραμ-

ματιζόμενη 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 10 ns μεταξύ των δειγμάτων εισόδου, ενώ το κύκλωμα στο σχήμα 2.18 απαιτεί μόνο 2 ns 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 κα-

λύτερα 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 τεχνολογίας 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 εφαρμόσει τη μνήμη για έναν πίνακα με διαφορετικούς τρόπους. Ανάλογα με τον υπολογισμό στον αλγόριθμο, 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 |
| 1: Αρχικές Βελτιστοποιήσεις | • Καθορισμός Διεπαφών<br>• Καθορισμός επαναλήψεων βρόχων |
| 2: Pipeline | • Pipeline και Dataflow |
| 3: Βελτιστοποίηση δομών | • Διαμέριση μνημών και θυρών<br>• Αφαίρεση ψευδών εξαρτήσεων |
| 4: Μείωση Latency | • Προαιρετικός καθορισμός απαιτήσεων του Latency |
| 6: Βελτιστοποίηση του χώρου | • Προαιρετική ανακτήση 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 διυεθετούμε 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 (II) επιτρέποντας την ταυτόχρονη εκτέλεση λειτουργιών εντός ενός βρόχου ή μιας λειτουργίας.
- **DATAFLOW** Επιτρέπει το pipeline σε επιπέδο 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 latency 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

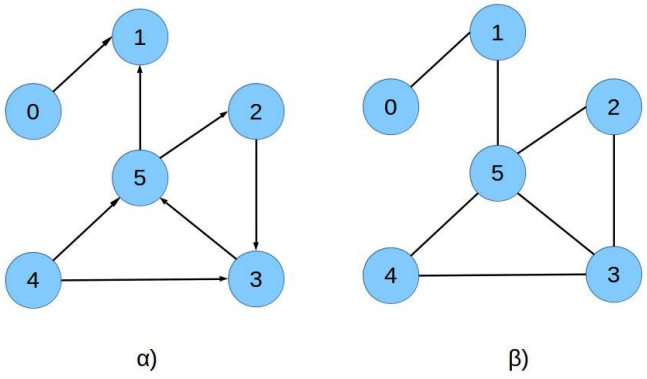
# Pagerank

## 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. 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 ζέυγος $G = (V, E)$, που αποτελείται από ένα σύνολο κόμβων $V = \{1, ..., N\}$ 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

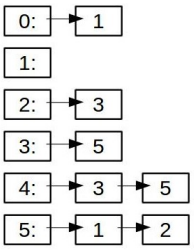
Οι μέθοδοι που μπορούν να αναπαρασταθούν οι γράφοι είναι αρκετοί. Οι πιο σύνηθεις είναι με χρήση πίνακα γειτνίασης, με λίστα γειτνίασης και με μη διατεταγμένη ακολουθία ακμών. Παρακάτω θα χρησιμοποιήσουμε το παράδειγμα του σχήματος 3.1 και συγκεκριμένα τον κατευθυνόμενο γράφο α) για την περιγραφή των 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 $ij$. Επόμενως το παραδειγμά 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

Στην παρούσα διατριβή θα χρησιμοποιήσουμε την μη διατεταγμένη λίστα ακμών. Όπως αναφέρθηκε εκτενέστερα και στο δεύτερο κεφάλαιο, για την επιτάχυνση αλγορίθμων με χρήση FPGA απαιτείται προσεκτικός χειρισμός, όσον αφορά το μέγεθος των δεδομένων και πως αυτά μεταφέρονται στο FPGA. Επιβάλλεται τα δεδομένα που χρησιμοποιεί ο αλγόριθμος να έχουν όσον το δυνατόν μικρότερο μέγεθος. Διερευνώντας τις παραπάνω μεθόδους αναπαράστασης των γράφων, κρίθηκε σωστή η χρήση της μη διατεταγμένης λίστας ακμών. Αρχικά το μέγεθος του πίνακα γειτνίασης ήταν απαγορευτικό. Έπειτα εξετάζοντας την λίστα γειτνίασης κρίθηκε ότι αν και φαίνεται να έχει το μικρότερο μέγεθος δεδομένων, τελικά χρειάζεται επιπρόσθετες πληροφορίες για να δομηθεί και να μεταφερθεί στο FPGA. Η απλότητα της λίστας ακμών ευνοεί στην χρήση των FPGAs , όπως και το μέγεθος των δεδομένων, αφού δεν περιέχει περιττές πληροφορίες, όπως για παράδειγμα ο μεγάλος όγκος μηδενικών στον πίνακα γειτνίασης. Η απλότητα έγκειται στο γεγονός ότι είναι δυνατόν να επεξεργάζεται η λίστα σειριακά και επομένως να μεταφέρεται σειριακά στο FPGA ξεκινώντας άμεσα των αλγόριθμο.

## 3.2 Ο αλγόριθμος Pagerank

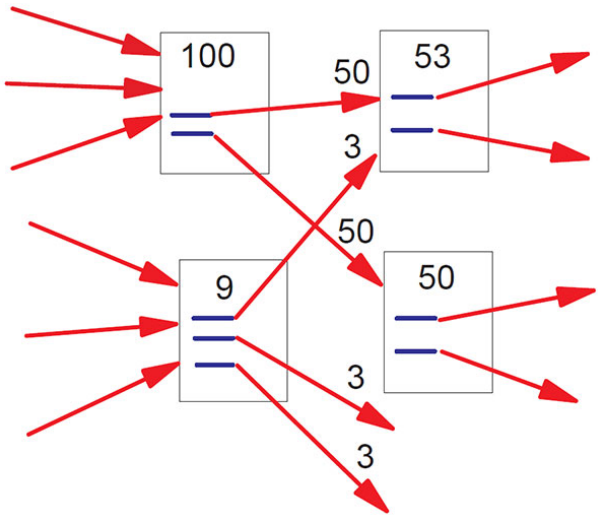
Ο αλγόριθμος Pagerank [7] χρησιμοποιείται για την ταξινόμηση ιστοτόπων σε μηχανές αναζήτησης. Το PageRank είναι ένας τρόπος μέτρησης της σημασίας των σελίδων του ιστότοπου. Σύμφωνα με την Google, το PageRank υπολογίζει τον αριθμό και την ποιότητα των συνδέσεων σε μια σελίδα για να καθορίσει μια πρώτη εκτίμηση για το πόσο σημαντική είναι η ιστοσελίδα. Η υποκείμενη υπόθεση είναι ότι οι πιο σημαντικοί ιστότοποι ενδέχεται να λαμβάνουν περισσότερες συνδέσεις από άλλους ιστότοπους. Δεν είναι ο μόνος αλγόριθμος που χρησιμοποιείται από την Google για την ταξινόμηση των αποτελεσμάτων της μηχανής αναζήτησης, αλλά είναι ο πρώτος αλγόριθμος που χρησιμοποίησε η εταιρεία και είναι ο πιο γνωστός.

### 3.2.1 Μαθηματική Μορφή

Αντιμετωπίζουμε τις συνδέσεις μεταξύ των ιστοσελίδων ως έναν κατευθυνόμενο γράφο $G = (V, E)$. Όπου $V$ θεωρούμε το σύνολο των κόμβων που αποτελείται από $N$ ιστοσελίδες. Όπου $E$ το σύνολο των κατευθυνόμενων ακμών $(i, j)$, όπου υπάρχουν αν η ιστοσελίδα $i$ έχει υπερσύνδεσμο στην ιστοσελίδα $j$. Ορίζουμε ως $u$ μία ιστοσελίδα. Ως $F_u$ το σύνολο των ιστοσελίδων που δείχνει η ιστοσελίδα $u$ και ως $B_u$ το σύνολο των ιστοσελίδων που δείχνουν στην ιστοσελίδα $u$. Επίσης ορίζουμε ως $N_u = |F_u|$ τον αριθμό των εξερχόμενων συνδέσεων από την ισοσελίδα $u$ και ως $c$ έναν παράγοντα που χρησιμοποιείται για κανονικοποίηση, ώστε το άθροισμα των ποσών να είναι σταθερό. Να σημειώσουμε ότι το $N_u$ αναφέρεται ως $degree(u)$.

Στην παρακάτω σχέση έχουμε την απλοποιημένη μορφή του Pagerank.

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v} \tag{3.1}$$




Σχήμα 3.5: Απλοποιημένος υπολογισμός του Pagerank

Η τιμή του Pagerank μιας σελίδας κατανέμεται ομοιόμορφα στους εμπλεκόμενους δεσμούς για να συμβάλλει στις τάξεις των σελίδων στις οποίες αναφέρονται. Σημειώνουμε ότι το c υπάρχει επειδή υπάρχουν αρκετές σελίδες που δεν έχουν εμπρός συνδέσμους, με αποτέλεσμα να χάνεται το βάρος τους από τον αλγόριθμο και το άθροιμα των όλων να μειώνεται. Η εξίσωση είναι αναδρομική, αλλά μπορεί να υπολογιστεί ξεκινώντας με οποιαδήποτε σειρά τάξεων και ερμηνεύοντας τον υπολογισμό μέχρι να συγκλίνει. Το σχήμα 3.6 δείχνει τη διάδοση του βαθμού από ένα ζευγάρι σελίδων σε ένα άλλο.

Μπορούμε να θέσουμε το ζήτημα με άλλο τρόπο. Ορίζουμε $A$ έναν τετραγωνικό πίνακα με τις σειρές και τις στήλες να αντιστοιχούν σε ιστοσελίδες. Έστω $A_{u,v} = \frac{1}{N_u}$ αν υπάρχει άκρο από το u στο v και $A_{u,v} = 0$ αν όχι. Εάν αντιμετωπίσουμε το Pagerank ως διάνυσμα των ιστοσελίδων, τότε έχουμε $R = cAR$. Επομένως, το R είναι ένα ιδιοδιάνυσμα του $A$ με ιδιοτιμή $c$. Στην πραγματικότητα, θέλουμε το κυρίαρχο ιδιοδιάνυσμα του $A$.

Υπάρχει ένα μικρό πρόβλημα σε αυτή την απλοποιημένη λειτουργία κατάταξης. Έχουμε για παράδειγμα δύο ιστοσελίδες που δείχνουν η μία στην άλλη, αλλά σε καμία άλλη σελίδα. Υποθέτουμε ότι υπάρχει κάποια ιστοσελίδα που δείχνει σε μία από αυτές. Στη συνέχεια, κατά τη διάρκεια της επανάληψης, αυτός ο βρόχος θα συσσωρεύσει βαθμό αλλά ποτέ δεν θα διανέμει, αφού δεν υπάρχουν εξερχόμενοι σύνδεσμοι. Ο βρόχος αποτελεί ένα είδος παγίδας το οποίο αποκαλείται rank sink. Για να ξεπεραστεί αυτό το πρόβλημα ο αλγόριθμος τροποποιείται όπως φαίνεται παρακάτω. Ορίζουμε ως $E(u)$ ένα διάνυσμα ιστοσελίδων που ανιστοιχεί σε μία πηγή βαθμού Pagerank. Στη συνέχεια, ο βαθμός PageRank ενός συνόλου ιστοσελίδων είναι μια ανάθεση, R, στις ιστοσελίδες που ικανοποιούν την παρακάτω εξίσωση έτσι ώστε το $c$ να είναι μέγιστο και $||R||_1 = 1$.

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v} + cE(u) \tag{3.2}$$

### Μοντέλο τυχαίου surfer

Ο ορισμός του PageRank παραπάνω έχει μια άλλη διαισθητική βάση σε τυχαίες διασχίσεις στους γράφους. Η απλή έκδοση αντιστοιχεί στην κατανομή μόνιμης πιθανότητας μίας τυχαίας διάσχισης του γράφου του ιστού. Δυστυχώς, αυτό μπορεί να θεωρηθεί ως μοντελοποίηση της συμπεριφοράς ενός τυχαίου surfer. Ο τυχαίος surfer συνεχίζει να μπαίνει σε διαδοχικούς συνδέσμους. Ωστόσο, εάν ένας πραγματικός διαδικτυακός surfer μπει σε ένα μικρό βρόχο ιστοσελίδων, είναι απίθανο να συνεχίσει να βρίσκεται στο βρόχο αυτό για πάντα. Αντί αυτού, ο surfer θα μεταβεί σε κάποια άλλη σελίδα. Ο πρόσθετος παράγοντας $E$ μπορεί να θεωρηθεί ως ένας τρόπος μοντελοποίησης της συμπεριφοράς όπου συχνά ο surfer μεταπηδά σε μια τυχαία σελίδα που επιλέγεται με βάση τη διανομή στην $E$. Το $E$ συνηθίζεται να δίνεται σαν παράμετρος από τον χρήστη.

### Αλγόριθμοι PageRank της Google

Ο αρχικός αλγόριθμος όπως περιγράφηκε από τους Page και Brin [8].

$$R(u) = (1-d) + d \sum_{v \in B_u} \frac{R(v)}{N_v} \tag{3.3}$$

Ως $d$ έχει οριστεί ένας παράγοντας απόσβεσης. Η επιλογή της τιμής του $d$ είναι εμπειρική και στις περισσότερες περιπτώσεις προτείνεται $d = 0.85$ από τους Brin και Page [8]. Βλέπουμε ότι το $c$ της σχέσης 3.2 αντικαταστάθηκε με $d$, καθώς και το $E$ εκφράζεται συναρτήσει του $d$.

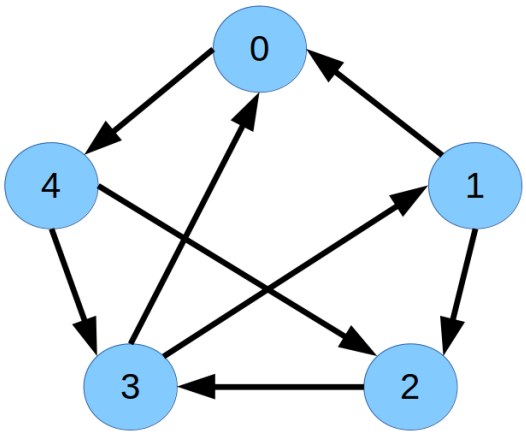
Η δεύτερη έκδοση του αλγορίθμου από τους Page και Brin.

$$R(u) = \frac{(1-d)}{N} + d \sum_{v \in B_u} \frac{R(v)}{N_v} \tag{3.4}$$

Οι δύο εκδοχές του αλγορίθμου δεν διαφέρουν θεμελιωδώς μεταξύ τους. Ο δεύτερος αλγόριθμος απλώς προσαρμόζει το $(1-d)/N$ για να αντικαταστήσει το $(1-d)$. Όσον αφορά το μοντέλο του τυχαίου Surfer, η δεύτερη έκδοση PageRank μιας ιστοσελίδας υποδηλώνει την πραγματική πιθανότητα ενός τυχαίου surfer να φτάσει σε αυτήν τη σελίδα αφού μπει πρώτα σε πολλούς άλλους συνδέσμους.

### 3.2.2 Επίλυση παραδείγματος

Ας πάρουμε για παράδειγμα τον παρακάτω τυχαίο γράφο.




Σχήμα 3.6: Τυχαίος γράφος

Αρχικά βλέπουμε ότι ο γράφος έχει $N = 5$ κόμβους και πως για καθέ κόμβο έχουμε τους εξής αριθμούς εξερχόμενων γειτόνων: $N_0 = 1, N_1 = 2, N_2 = 1, N_3 = 2, N_4 = 2$ Επίσης $\frac{(1-d)}{N} = \frac{(1-0.85)}{5} = 0.03$

Μεχρί να έχουμε $err < 0.00000001$
Ξεκινώντας θέτουμε για κάθε κόμβο:

$$R_0(u) = \frac{1}{N} = \frac{1}{5} = 0.2 \tag{3.5}$$

Στην 1η επανάληψη έχουμε:

$$R_1(0) = 0.03 + d \cdot (\frac{R_0(1)}{N_1} + \frac{R_0(3)}{N_3}) = 0.2 \tag{3.6}$$

$$R_1(1) = 0.03 + d \cdot (\frac{R_0(3)}{N_3}) = 0.115 \tag{3.7}$$

$$R_1(2) = 0.03 + d \cdot (\frac{R_0(1)}{N_1} + \frac{R_0(4)}{N_4}) = 0.2 \tag{3.8}$$

$$R_1(3) = 0.03 + d \cdot (\frac{R_0(2)}{N_2} + \frac{R_0(4)}{N_4}) = 0.285 \tag{3.9}$$

$$R_1(4) = 0.03 + d \cdot (\frac{R_0(0)}{N_0}) = 0.2 \tag{3.10}$$

Έπειτα ελέχγουμε την σύγκλιση:

$$err_1 = \sum_{u \in V} |R_1(u) - R_0(u)| = 0 + 0.085 + 0 + 0.085 + 0 = 0.17 \tag{3.11}$$

$$err_1 > 0.00000001$$

Όμοια στην 2η επανάληψη έχουμε:

$$R_2(0) = 0.03 + d \cdot (\frac{R_1(1)}{N_1} + \frac{R_1(3)}{N_3}) = 0.2 \tag{3.12}$$

$$R_2(1) = 0.03 + d \cdot (\frac{R_1(3)}{N_3}) = 0.151125 \tag{3.13}$$

$$R_2(2) = 0.03 + d \cdot (\frac{R_1(1)}{N_1} + \frac{R_1(4)}{N_4}) = 0.121375 \tag{3.14}$$

$$R_2(3) = 0.03 + d \cdot (\frac{R_1(2)}{N_2} + \frac{R_1(4)}{N_4}) = 0.285 \tag{3.15}$$

$$R_2(4) = 0.03 + d \cdot (\frac{R_1(0)}{N_0}) = 0.2 \tag{3.16}$$

Έπειτα ελέχγουμε την σύγκλιση:

$$err_2 = \sum_{u \in V} |R_2(u) - R_1(u)| = 0.11475 \tag{3.17}$$

$$err_2 > 0.00000001$$

Βλέπουμε ήδη πως διαμορφώνονται οι βαθμοί Pagerank των κόμβων. Το σφάλμα της σύγκλισης, ήδη από την πρώτη επανάληψη μειώθηκε αισθητά. Συνεχίζουμε ομοίως μέχρι να επιτύχουμε σύγκλιση, όποτε και τους τελικούς βαθμούς Pagerank του κάθε κόμβου.

## 3.3 Πολλαπλασιασμός αραιού πίνακα με διάνυσμα (SpMV)

Οι αραιοί πίνακες είναι πίνακες που τα περισσότερα στοιχεία τους ισούνται με μηδέν. Στην τεχνολογική κοινότητα έχουν γίνει πολλές προσπάθειες αποδοτικότερης εκτέλεσης εφαρμογών αυτών. Συγκεκριμένα ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα (Sparse Matrix Vector Multiplication - SpMV) είναι από τους πιο σημαντικούς υπολογιστικούς πυρήνες [9]. Το pagerank αποτελεί μια βασική εφαρμογή που προσεγγίζεται με SpMV. Στην παράγραφο 3.1.2 αναλύσαμε πως αναπαριστούμε απλοϊκά τους γράφους. Σε επόμενο βήμα δείξαμε πως ο πίνακας γειτνίασης του Pagerank περιέχει τα βάρη, δηλαδή το ποσοστό του βαθμού που συνεισφέρει ένας κόμβος στους γείτονές του. Αναγνωρίζοντας το πρόβλημα των ιδιοτιμών του Pagerank, χρειάζεται να εκτελείται αποδοτικά ο πολλαπλασιασμός του αραιού πίνακα γειτνίασης με το διάνυσμα των βαθμών του Pagerank.

Στον SpMV έχουμε τον υπολογιστικό πυρήνα $y = Ax$. Όπου $A$ είναι ένας αραιός πίνακας, τα διανύσματα εισόδου $x$ και εξόδου $y$ είναι πυκνά (με λίγα ή χωρίς μηδενικά). Για την επίλυση του SpMV έχουν δημιουργηθεί αρκετές δομές πινάκων και αντίστοιχοι αλγόριθμοι. Ενδεικτικές των δομών είναι οι COO, CSR και ELL.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | | 1 | | 2 | |
| 1 | | 3 | 4 | | |
| 2 | 5 | | 7 | | |
| 3 | | 8 | | | |
| 4 | | | 6 | 4 | 9 |

Πίνακας 3.1: Παράδειγμα αραιού πίνακα

### COO - Coordinate Format

Σε αυτήν την δομή έχουμε:

- Τον δείκτη της σειράς (row)
- Τον δείκτη της στήλης (column)
- Την τιμή του κελιού του πίνακα (value)

Έτσι ο πίνακας 3.1 δομείται όπως φαίνεται κάτωθι.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| row | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 4 | 4 | 4 |
| column | 1 | 3 | 1 | 2 | 0 | 2 | 1 | 2 | 3 | 4 |
| value | 1 | 2 | 3 | 4 | 5 | 7 | 8 | 6 | 4 | 9 |

Πίνακας 3.2: COO Format

Ο αντίστοιχος αλγόριθμος αυτής της δομής διαμορφώνεται όπως φαίνεται παρακάτω. Όπου $n$ είναι ο αριθμός των στηλών, και όπου $nnz$ ο αριθμός των μη μηδενικών κελιών.

```
for (int i=0; i<n; ++i)
  y[i] = 0.0;
for (int i=0; i<nnz; ++i)
  y[row[i]] += val[i]*x[col[i]];
```

Listing 3.1: SpVM - COO

### CSR - Compressed Sparse Row format

Σε αυτήν την δομή έχουμε:

- Τον αύξοντα αριθμό του τελευταίου μη μεδενικού στοχείου της σειράς +1, (row offset)
- Τον δείκτη της στήλης (column)
- Την τιμή του κελιού του πίνακα (value)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| row offset | 0 | 2 | 4 | 6 | 7 | 10 | | | |
| column | 1 | 3 | 1 | 2 | 0 | 2 | 1 | 2 | 3 | 4 |
| value | 1 | 2 | 3 | 4 | 5 | 7 | 8 | 6 | 4 | 9 |

Πίνακας 3.3: CSR Format

Ο αντίστοιχος αλγόριθμος αυτής της δομής:

```
for (int i=0; i<n; ++i) {
  y[i] = 0.0;
for (int j=row_off[i]; j<row_off[i+1]; ++j)
  y[i] += val[j]*x[col[j]];
}
```

Listing 3.2: SpVM - CSR

### ELL format

Σε αυτήν την δομή έχουμε:

- Τον δείκτη της στήλης (column)
- Την τιμή του κελιού του πίνακα (value) Εδώ ορίζουμε $M$ τον μέγιστο αριθμό μη μηδενικών στοιχείων που υπάρχουν σε κάθε σειρά του πίνακα

column

| | | |
|---|---|---|
| 1 | 3 | * |
| 1 | 2 | * |
| 0 | 2 | * |
| 1 | * | * |
| 2 | 3 | 4 |

value

| | | |
|---|---|---|
| 1 | 2 | * |
| 3 | 4 | * |
| 5 | 7 | * |
| 8 | * | * |
| 6 | 4 | 9 |

Πίνακας 3.4: ELL Format

Ο αντίστοιχος αλγόριθμος αυτής της δομής:

```
for (int i=0; i<n; ++i) {
  y[i] = 0.0;
  for (int j=0; j<max_row; ++j) {
    jj = j + max_row*i;
    c = col[jj]
    if ((c >= 0) && (c < n))
      y[i] += val[jj] * x[c];
    }
}
```

Listing 3.3: SpVM - ELL

### 3.3.1 Διερεύνηση χρήσης του SpMV στην υλοποίηση του Pagerank σε FPGA

Μελετώντας την χρήση των δομών που συνοδεύουν τον SpMV, για την αποδοτικότερη επίλυση του αλγορίθμου Pagerank, παρουσιάστηκαν κάποιες δυσκολίες. Στην περίπτωση των γράφων, για κάθε ακμή θα έχουμε κάποια τιμή (ανάλογα την εφαρμογή) στο αντίστοιχο κελί του πίνακα value, με βάση τις συντεταγμένες. Συγκεκριμένα, σύμφωνα με την θεωρία του Pagerank, στον πίνακα value θα είχαμε την τιμή $\frac{1}{N_u}$, όπως αναφέρουμε στην παράγραφο 3.2.1. Αυτή η τιμή, είναι το ποσοστό του βαθμού κάθε κόμβου που μοιράζει ισόποσα στους γειτονές του. Για κάθε γειτονικό κόμβο θα αποθηκεύαμε την ίδια τιμή, επομένως θα σπαταλούσαμε περισσότερη μνήμη, έχοντας αντίτυπα και ειδικότερα 32b float αριθμούς. Όπως αναφέραμε και στο προηγούμενο κεφάλαιο, το FPGA μας περιορίζει όσον αφορά το χώρο αποθήκευσης, αλλα και μεταφοράς των δεδομένων. Επίσης όλες οι παραπάνω δομές προυποθέτουν την ταξινόμηση των γράφων, που είναι χρονοβόρα διαδικασία και θα πρέπει να πραγματοποιηθούν εκτός των FPGAs.

Ερευνώντας την COO δομή, βλέπουμε ότι ομοιάζει με την μη διατεταγμένη λίστα ακμών. Αρχικά και οι δυο περιέχουν τις συντεταγμένες ακμών. Η διαφορά είναι ότι η COO δομή έχει τις ακμές ταξινομημένες και τους αντιστοιχεί ένα αντίτυπο του $\frac{1}{N_u}$ στον πίνακα value. Για να μην έχουμε πολλαπλά αντίτυπα, προτιμήσαμε να χρησιμοποιούμε μια δομή $degree$, που περιέχει τον αριθμό των εξερχόμενων ακμών κάθε κόμβου $N_u$. Η δομή $degree$ θα περιγραφεί αναλυτικότερα στο επόμενο κεφάλαιο. Επίσης όπως προαναφέραμε για

να δημιουργηθεί αυτή η δομή χρειάζεται ταξινόμηση του γράφου. Ο γράφος περιέχεται σε ένα αρχείο σε μορφή συνδέμων-ακμών. Η αποδοτική ταξινόμηση του γράφου για να το μετατρέψουμε σε επιθυμητή δομή είναι προς διερεύνηση. Είναι γεγονός πως ένας ταξινομημένος γράφος, κατά την εκτέλεση του Pagerank θα δημιουργήσει πολλές λιγότερες αστοχίες μνήμης από ότι ένας μη ταξινομημένος. Επίσης για να επιτύχουμε παραλληλία πολλές φορές είναι χρήσιμο να έχουμε αντίτυπα δεδομένων. Τα FPGAs όμως, λόγω του χρόνου μεταφοράς δεδομένων και του περιορισμένου χώρου αποθήκευσης, μας κατευθύνουν στην μη χρήση αυτών.

# Κεφάλαιο 4

# Σχεδίαση Αρχιτεκτονικής

Η σχεδίαση του αλγορίθμου PageRank υλοποιήθηκε με δύο τρόπους. Αμφότερες οι υλοποιήσεις επιλύονται ακμοκεντρικά [6]. Σε πρώτο βήμα υλοποιήθηκε μία απλή αρχιτεκτονική σχεδιάζοντας ένα ip core. Η επιτάχυνση σε αυτήν την περίπτωση στηρίχτηκε στους πολλαπλούς υπολογιστικούς πόρους του FPGA και της δυνατότητας παραλληλοποίησης σε επίπεδο εντολών. Έπειτα υλοποιήθηκε μία πιο σύνθετη έκδοση, με διαχωρισμό των δεδομένων του γράφου και χρήση τεχνικής scatter-gather, καθώς και παραλληλοποίηση σε επίπεδο συναρτήσεων, εν δυνάμει ip cores. Ουσιαστικά και οι δύο υλοποιήσεις περιέχουν την τεχνική scatter-gather, καθώς πρώτα διατρέχεται ο γράφος και μοιράζονται (scatter) οι τιμές του pagerank των κόμβων και μετά συλλέγονται (gather).

## 4.1 Single core Αρχιτεκτονική

Σε αυτή την υλοποίηση ο αλγοριθμος δομήθηκε όπως φαίνεται παρακάτω. Αρχικά προσπελάστηκε το αρχείο του γράφου και σε κάθε κόμβο αντιστοιχήσαμε έναν δείκτη, δημιουργώντας παράλληλα το σύνολο ακμών edge_set. Όπου $N$ είναι ο αριθμός των κόμβων, όπου $E$ ο αριθμός των ακμών και όπου $degree(v)$ ο αριθμός των εξερχόμενων ακμών του κόμβου $v$.

Για την κατανόηση της υλοποίησης ας αναλύσουμε επιγραμματικά πως εκτελείται ο αλγόριθμος. Αρχικά μεταφέρουμε τις δομές $edge\_set$ και $degree$ στο FPGA, λειτουργία που δεν χρειάζεται σε εκτέλεση στον ARM. Δημιουργούμε την δομή $giving\_pr$ κάθε κόμβου, που είναι το μοιραζόμενο pagerank του κάθε κόμβου και προκύπτει από την ανάγνωση και διαίρεση των $pr, degree$. Προχωρήσαμε σε αυτήν την δημιουργία, ώστε να μην επαναλαμβάνονται οι ίδιες πράξεις, ειδικά τώρα που αφορούν ανάγνωση και διαίρεση με 32bit $float$ αριθμούς. Ακολούθως, διατρέχεται το σύνολο των κόμβων, ενημερώνοντας το $sum\_pr$ των κόμβων προορισμού και δημιουργώντας το αθροιστικό μέρος "$\Sigma$" του αλγορίθμου. Έπειτα αθροίζεται το σταθερό πρώτο μέρος, του pagerank κάθε κόμβου, και υπολογίζεται η απόκλιση. Τέλος όταν συγκλίνει ο αλγόριθμος, κανονικοποιούνται οι τιμές pagerank και επιστρέφονται στον ARM επεξεργαστή. Η κανονικοποίηση γίνεται για την επίλυση του προβλήματος rank sink που αναλύθηκε παραπάνω.

**Algorithm 1** PageRank single-core

1: **procedure** $calculate\_pagerank(edge\_set[E], degree[N], pr\_out[N], conv)$
2: **for** each vertex $v$ **do**
3: initiliaze $pr[v] = \frac{1}{N}$
4: **while** not converged
5: **for** each vertex $v$ **do**
6: $giving\_pr[v] = \frac{pr[v]}{degree[v]}$
7: Calculate $\Sigma$ factor:
8: **for** each edge $(src, dest)$ **do**
9: $sum\_pr[dest] += giving\_pr[src]$
10: **for** each vertex $v$ **do**
11: $pr[v] = \frac{1-d}{N} + d * sum\_pr[v]$
12: Normalize pr:
13: **for** each vertex $v$ **do**
14: $pr\_out[v] = \frac{pr[v]}{sum\_of\_prs}$

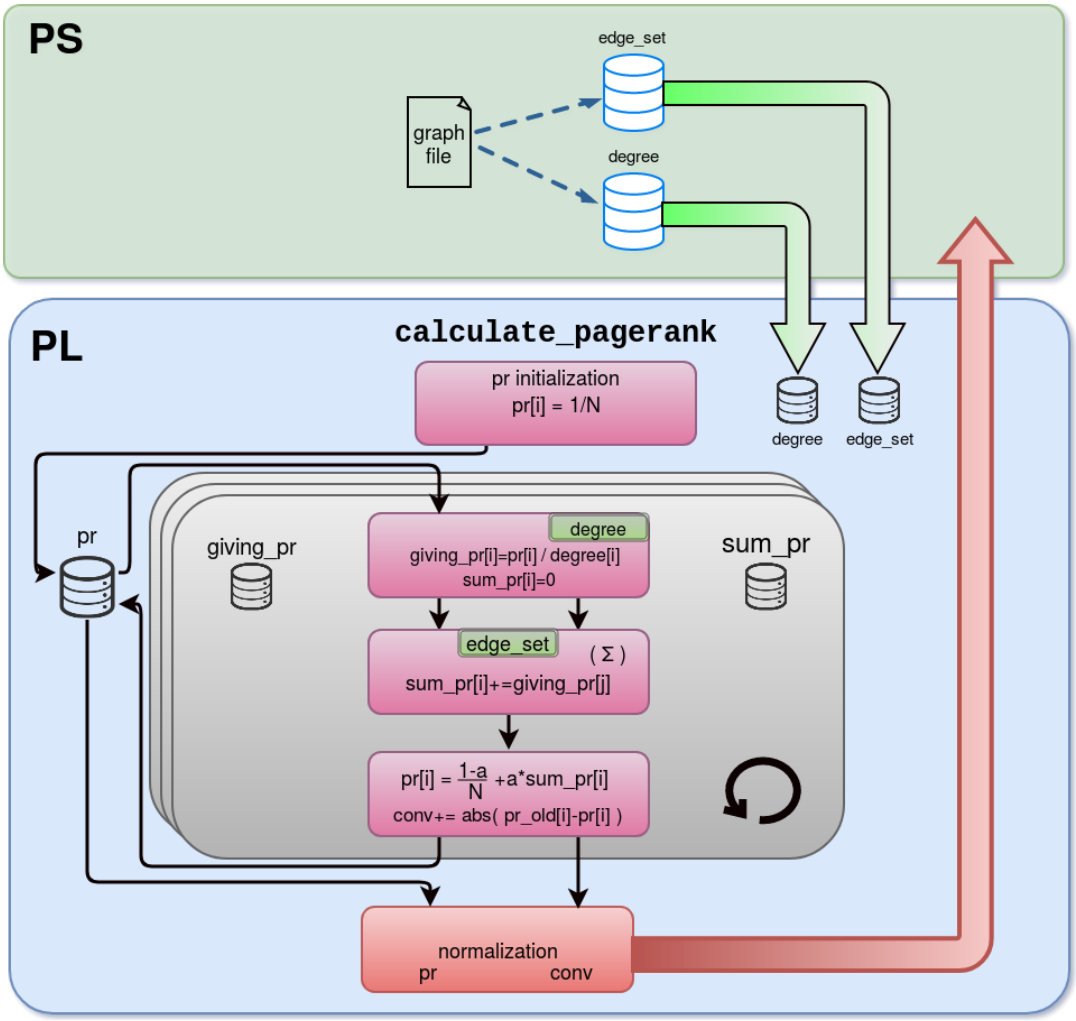
Αυτόν τον αλγόριθμο τον υλοποιήσαμε σε C++ με την χρήση κάποιων pragmas, αρχικά σε επίπεδο εντολών, όπως φαίνεται και στον κώδικα παρακάτω. Χειριζόμενοι τους βρόχους με $pipeline$, $unroll$, $loop\_flatten$, καταλήξαμε στην χρήση κυρίως $pipeline$. Με την χρήση των υπόλοιπων pragmas σχετικών με τους βρόχους, δεν είχαμε κάποια επίπτωση στην ταχύτητα εκτέλεσης. Αυτό το γεγονός οφείλεται στον HLS μεταγλωττιστή που όποτε μεταγλωττίζει, ανάλογα και με τις εξαρτήσεις, υλοποιεί αυτούς τους χειρισμούς. Όταν χρησιμοποιούμε $pipeline$ επιτρέπουμε την εκτέλεση των λειτουργιών σε βρόχο με ταυτόχρονο τρόπο. Είναι γεγονός ότι μια εντολή που δεν εξαρτάται από προηγούμενες, μπορεί να ολοκληρωθεί πριν από αυτές.

```
20 void calculate_pagerank(unsigned short int edge_set[E], unsigned short int
       degree[N], float pr_out[N], float & conv, int & counter) {
21
22   float pr[N], sum_pr[N], giving_pr[N], _edge_set[E], temp, temp_conv = 1,
       temp_add, _convergence = CONVERGENCE*N, _alpha = 0.85, initial_pr = 1.0/N,
       sum_of_prs = 0, stand_part = (1.0-_alpha)/N;
23   int temp_src, temp_dest, hw_counter = 0;
24
25   for (int i = 0; i < E; i++) {
26 #pragma HLS PIPELINE
27     _edge_set[i] = edge_set[i];
28   }
29
30   for (int i = 0; i < N; i++) {
31 #pragma HLS PIPELINE
32     pr[i] = initial_pr;
33   }
```

```
34   for (int l = 0; l < 40; l++) {
35     if (temp_conv > _convergence) {
36       temp_conv = 0;
37       LOAD_LOOP:
38       for (int i = 0; i < N; i++) {
39 #pragma HLS loop_tripcount min = 1 max = 1000
40 #pragma HLS PIPELINE
41         giving_pr[i] = pr[i] / degree[i];
42         sum_pr[i] = 0;
43       }
44       LOOP_2:
45       for (int j = 0; j < E; j = j + 2) {
46 #pragma HLS loop_tripcount min = 1 max = 32000
47 #pragma HLS PIPELINE
48         temp_src = _edge_set[j];
49         temp_dest = _edge_set[j + 1];
50         temp_add = sum_pr[temp_dest] + giving_pr[temp_src];
51         sum_pr[temp_dest] = temp_add;
52       }
53       temp_conv = 0;
54       sum_of_prs = 0;
55
56       LOOP_3:
57       for (int i = 0; i < N; i++) {
58 #pragma HLS loop_tripcount min = 1 max = 1000
59 #pragma HLS PIPELINE
60         temp = sum_pr[i] * _alpha + stand_part;
61         temp_conv += abs(pr[i] - temp);
62         pr[i] = temp;
63         sum_of_prs += temp;
64       }
65       hw_counter++;
66     }
67   }
68   counter = hw_counter;
69   conv = temp_conv;
70
71   LOOP_4:
72   for (int i = 0; i < N; i++) {
73 #pragma HLS loop_tripcount min = 1 max = 1000
74 #pragma HLS PIPELINE
75     pr_out[i] = pr[i] / sum_of_prs;
76   }
77 }
```

Listing 4.1: Single-core υλοποίηση της calculate_pagerank

Για να καταστήσουμε δυνατή την υλοποίηση πρέπει να καθορίσουμε πως θα μεταφερθούν τα δεδομένα μας στο FPGA και που θα αποθηκευτούν, για να επιτύχουμε και καλύτερη απόδοση. Στο σχήμα 4.1 βλέπουμε πως υλοποιείται συνολικά ο αλγόριθμος, καθώς και οι δομές δεδομένων αυτού.




Σχήμα 4.1: sigle-core Pagerank

Όπως φαίνεται και στην παραπάνω εικόνα, το σύστημα επεξεργασίας διαβάζει το αρχείο του γράφου και δημιουργεί δύο δομές. Οι δύο αυτές δομές είναι οι πίνακες edge_set (σύνολο ακμών) και degree (αριθμός εξερχόμενων συνδέσεων κάθε κόμβου) και πρέπει να μεταφερθούν στο FPGA. Χωρίς να δώσουμε οδηγία για την μεταφορά, ο compiler αναλύοντας τον κώδικα επιλέγει το πρωτοκόλλο AXI4 Memory-Mapped που μεταφέρει τα δεδομένα στην BRAM. Αυτό το πρωτόκολλο μας περιορίζει καθώς μπορεί να μεταφέρει εως 16384 στοιχεία μεγέθους εώς και 64 bits. Για να ξεπεράσουμε αυτόν τον περιορισμό μπορούμε είτε να χρησιμοποιήσουμε $zero\_copy$ όπου μεταφέρει τα δεδομένα στην DRAM είτε να χρησιμοποιήσουμε το πρωτόκολλο AXI4-Stream. Το $zero\_copy$ όπου δοκιμάστηκε, μας επιβάρυνε χρονικά. Επιλέξαμε λοιπόν να χρησιμοποιήσουμε το πρωτόκολλο AXI4-Stream για την μεταφορά του edge_set. Οι οδηγίες δόθηκαν σε ένα header αρχείο.

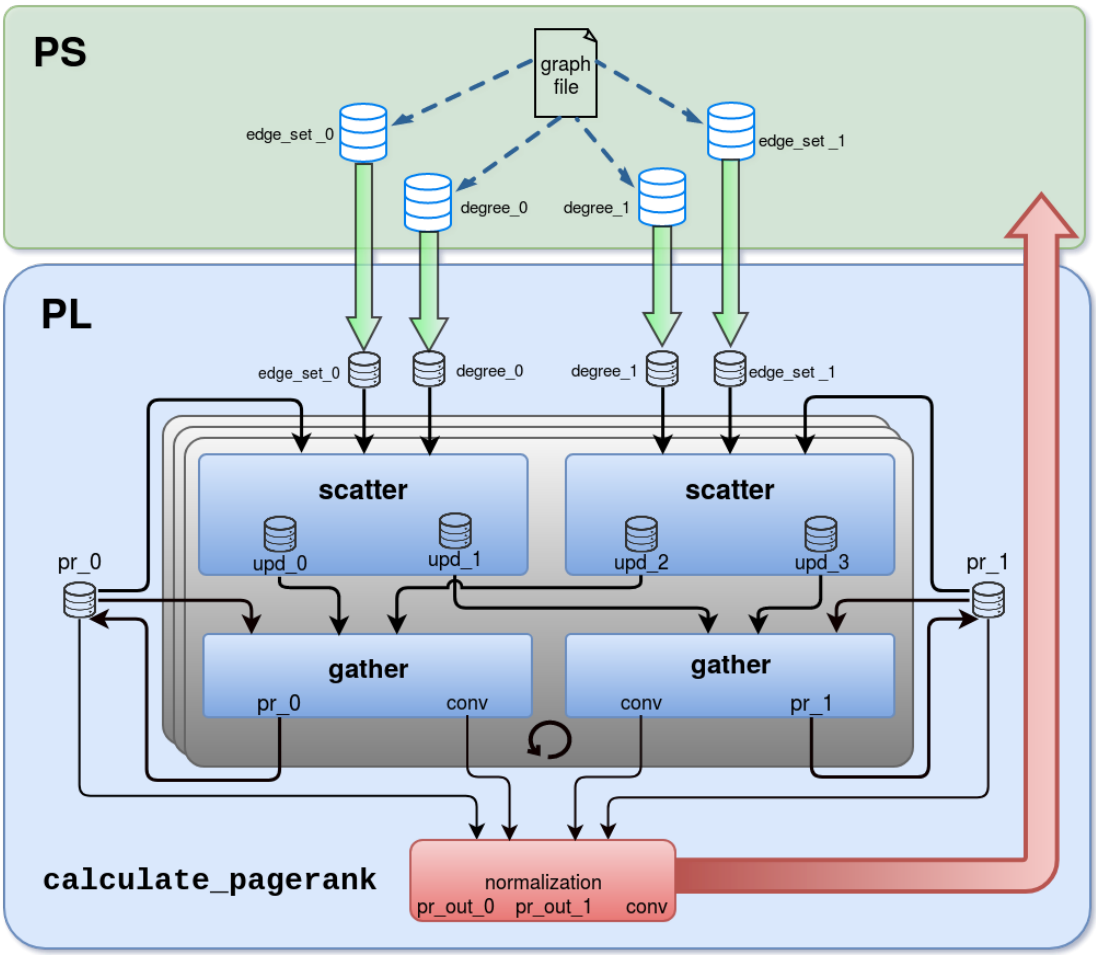
```
#define N 1000
#define E 2*32000
const float CONVERGENCE = 0.00000001;

#pragma SDS data access_pattern(edge_set:SEQUENTIAL, degree:SEQUENTIAL,
    pr_out:SEQUENTIAL|NON_CACHEABLE)
#pragma SDS data copy(degree[0:N])
#pragma SDS data mem_attribute( edge_set:PHYSICAL_CONTIGUOUS|NON_CACHEABLE,
    degree:PHYSICAL_CONTIGUOUS|NON_CACHEABLE, pr_out:PHYSICAL_CONTIGUOUS)
#pragma SDS data data\_mover(edge_set:AXIDMA_SIMPLE, degree:AXIDMA_SIMPLE,
      pr_out:AXIDMA_SIMPLE)
#pragma SDS data sys_port(edge_set:AFI, degree:AFI, pr_out:AFI)
void calculate_pagerank(unsigned short int edge_set[E], unsigned short int
    degree[N], float pr_out[N], float &conv, int &counter );
```

Listing 4.2: Αρχείο single-core header

Με την χρήση του #pragma SDS data access_pattern, δηλώνουμε τον τρόπο με τον οποίο προσπελάζονται τα δεδομένα και με την επιλογή SEQUENTIAL επιλέγουμε διαδοχικά για τις δομές μας. Με το #pragma SDS data copy(degree[0:N]) αντιγράφουμε τον πίνακα degree στην BRAM. Με αυτόν τον τρόπο το degree δύναται να έχει μέχρι 16384 στοιχεία. Για το λόγο αυτό, σε γράφους με περισσότερους κόμβους και άρα και μεγαλύτερους πίνακες degree δεν χρησιμοποιούμε data copy. Αντίθετα κάνουμε stream σε μια μεταβλητή του FPGA και έχουμε μόνο περιορισμό το συνολικό μέγεθος της BRAM. Υλοποιώντας και τις δύο τεχνικές, data copy και stream, δεν είχαμε διαφορά στον χρόνο μεταφοράς. Έπειτα με το #pragma SDS data mem_attribute δηλώνουμε πως έχουν αποθηκευτεί τα δεδομένα στο σύστημα επεξεργασίας. Το PHYSICAL_CONTIGUOUS σημαίνει ότι η μνήμη που αντιστοιχεί στο σχετικό πίνακα έχει κατανεμηθεί χρησιμοποιώντας το sds_alloc και είναι φυσική συνεχής μνήμη, ενώ το NON_PHYSICAL_CONTIGUOUS σημαίνει ότι η μνήμη που αντιστοιχεί στο σχετικό πίνακα έχει κατανεμηθεί χρησιμοποιώντας το malloc. Αυτό βοηθά τον μεταγλωττιστή SDSoC να επιλέξει τον βέλτιστο μεταφορέα δεδομένων. Το CACHEABLE σημαίνει ότι ο μεταγλωττιστής πρέπει να διατηρεί τη συνοχή της κρυφής μνήμης μεταξύ της CPU και του επιταχυντή για τη μνήμη που αντιστοιχεί στον πίνακα. Αντίθετα το NON_CACHEABLE σημαίνει ότι ο μεταγλωττιστής δεν χρειάζεται να εξασφαλίσει τη συνοχή της μνήμης για τον συγκεκριμένο πίνακα. Με την χρήση του #pragma SDS data data_mover, επιλέγουμε τον μεταφορέα δεδομένων που μπορεί να είναι AXIFIFO, AXIDMA_SG, ή AXIDMA_SIMPLE. Το AXI Direct Memory Access (AXIDMA) παρέχει άμεση πρόσβαση μνήμης υψηλής ταχύτητας μεταξύ μνήμης και περιφερειακών τύπου AXI4-Stream. Τέλος με το #pragma SDS data sys_port καθορίζουμε την θύρα αναλόγως αν θέλουμε συνοχή με την κρυφή μνήμη. Αν θέλουμε επιλέγουμε ACP (S_AXI_ACP), αν δεν θέλουμε επιλέγουμε AFI με θύρες υψηλής απόδοσης (S_AXI_HP).

## 4.2 Dual core Αρχιτεκτονική

Στην προσπάθεια παραλληλοποίησης του αλγορίθμου καταλήξαμε στην παρακάτω υλοποίηση. Αρχικά, κατά την ανάγνωση του αρχείου του γράφου δημιουργούνται δύο σύνολα edge_set_0 και edge_set_1. Οι ακμές διαχωρίζονται σε δύο σύνολα με βάση τον κόμβο προέλευσης. Στο πρώτο σύνολο έχουμε τις ακμές με κόμβο προέλευσης στο διάστημα (0, $N/2$) και στο δεύτερο με κόμβο προέλευσης στο διάστημα ($N/2+1, N$). Σε καθένα σύνολο αντιστοιχεί και ένας πίνακας degree που αφορά τους κόμβους του διαστήματος του. Αλλάζουμε τον αρχικό αλγόριθμο υλοποιώντας μέσα στην top-level συνάρτηση μικρότερες συναρτήσεις, ip-cores. Χωρίζουμε τον αλγόριθμο σε δύο στάδια scatter και gather. Σε κάθε στάδιο υλοποιούνται από δύο ip-cores, μονάδες που μπορούν να εκτελεστούν παράλληλα. Κατά αυτόν τον τρόπο έχουμε δύο ip-cores για scatter και δύο για gather. Τονίζουμε ότι εκτελούνται παράλληλα τα ip-cores του ίδιου σταδίου. Έχουμε ως αποτέλεσμα να εκτελείται παράλληλα το scatter στα δύο του ip-cores και ακολούθως να εκτελείται παράλληλα στα δύο του ip-cores το gather. Στις δομές δεδομένων έχουν προστεθεί και τα updates. Τα updates αφορούν τους κόμβους προορισμού και συνθέτουν τον όρο του αθροίσματος $\Sigma$ του αλγορίθμου (βλέπε σχέση (3.4)).




Σχήμα 4.2: Dual core Pagerank

```
1  void calculate_pagerank(unsigned short int edge_set_0[EDGES_0], unsigned short
       int edge_set_1[EDGES_1], float pr_out_0[sources_per_part], float pr_out_1[
       sources_per_part], unsigned short int degree_0[sources_per_part], unsigned
       short int degree_1[sources_per_part], float & conv, int & hw_counter) {
2
3          float updates_0[sources_per_part], updates_1[sources_per_part],
       updates_2[sources_per_part], updates_3[sources_per_part];
4          float pr_0[sources_per_part], pr_1[sources_per_part],conver[partitions
       ], _convergence = N * CONVERGENCE, _conv = 1;
5          float sum_of_prs = 0,initial_pr = 1.0 / N;
6          int _hw_counter = 0;
7          unsigned short int _edge_set_0[EDGES_0], _edge_set_1[EDGES_1],
       _degree_0[sources_per_part], _degree_1[sources_per_part];
8
9          for (int i = 0; i < sources_per_part; i++) {
10 #pragma HLS pipeline
11             pr_0[i] = initial_pr;
12             pr_1[i] = initial_pr;
13         }
14
15         for (int i = 0; i < EDGES_0; i++) {
16 #pragma HLS PIPELINE
17             _edge_set_0[i] = edge_set_0[i];
18         }
19
20         for (int i = 0; i < EDGES_1; i++) {
21 #pragma HLS PIPELINE
22             _edge_set_1[i] = edge_set_1[i];
23         }
24
25         for (int i = 0; i < sources_per_part; i++) {
26 #pragma HLS PIPELINE
27             _degree_0[i] = degree_0[i];
28             _degree_1[i] = degree_1[i];
29         }
30
31         for (int k = 0; k < 40; k++) {
32             if (_conv > _convergence) {
33                 scatter(_edge_set_0, pr_0, _degree_0, updates_0, updates_1);
34                 scatter1(_edge_set_1, pr_1, _degree_1, updates_2, updates_3);
35
36                 gather(updates_0, updates_2, pr_0, conver[0]);
37                 gather1(updates_1, updates_3, pr_1, conver[1]);
38
39                 _conv = conver[0] + conver[1];
40                 _hw_counter++;
41             }
42         }
43         conv = _conv / N;
44         hw_counter = _hw_counter;
```

```
45         for (unsigned short int j = 0; j < sources_per_part; j++) {
46 #pragma HLS loop_tripcount min = 1 max = 500
47 #pragma HLS PIPELINE
48             sum_of_prs += pr_0[j] + pr_1[j];
49             for (int i = 0; i < sources_per_part; i++) {
50 #pragma HLS loop_tripcount min = 1 max = 500
51 #pragma HLS PIPELINE
52                 pr_out_0[i] = pr_0[i] / sum_of_prs;
53                 pr_out_1[i] = pr_1[i] / sum_of_prs;
54             }
55         }
```

Listing 4.3: Παραλληλοποιημένη υλοποίηση της calculate_pagerank

Σε κάθε συνάρτηση scatter εισέρχεται ένα συνόλο ακμών και δημιουργούνται δύο σύνολα update. Σύμφωνα με τον κόμβο προορισμού, δηλαδή τον κόμβο που δέχεται το μοιραζόμενο pagerank, τα σύνολα update αφορούν τους κόμβους είτε στο διάστημα (0, $N/2$), είτε στο διάστημα ($N/2+1, N$). Τα σύνολα update δημιουργούν τον αθροιστικό όρο "$\Sigma$" του αλγορίθμου. Κατά αυτόν τον τρόπο, όταν τελειώσει η εκτέλεση των δύο scatter, θα έχει δημιουργηθεί από την πρώτη τα $update\_0, update\_1$ και από την δεύτερη τα $update\_2, update\_3$. Τα $update\_0, update\_2$ αφορούν τους κόμβους του διαστήματος (0, $N/2$) και τα $update\_1, update\_3$ τους κόμβους του διαστήματος ($N/2+1, N$). Στην πρώτη φάση αυτής της υλοποίησης για να διαπιστώσουμε στην scatter ποιο update πρέπει να ενημερώσει, σύμφωνα με τον κόμβο προορισμού, χρησιμοποιούμε σύγκριση με το $/2$. Σε αυτήν την περίπτωση αν είχαμε χωρίσει σε περισσότερα από δύο σύνολα το σύνολο των κόμβων, θα χρειαζόμασταν διαίρεση για την επιλογή του update. Αυτή η διαδικασία σε κάθε ακμή, εκτός από τον συνολικό χρόνο που δαπανούσε, μεγάλωνε και το critical path στο fpga. Για να ξεπεράσουμε την δυσκολία αυτή, μέσα στην κάθε scatter δημιουργήσαμε μια ενιαία δομή $\_update$ και δεν χρειάζοταν επιλογή συνόλου κόμβων. Αυτή η δομή αφού ενημερώνοταν, στο τέλος του αλγορίθμου, μοιράζοταν στις δύο δομές $update$. Τα update δεν υπήρχαν στην single core έκδοση. Γίνεται κατανοητό ότι τα update είναι μια ενδιάμεση δομή που παρεμβάλαμε ώστε να επιτρέψουμε την παραλληλία.

```
1  void scatter(unsigned short int edge_set[EDGES_0], float pr[sources_per_part],
       unsigned short int degree[sources_per_part], float update_0[
       sources_per_part], float update_1[sources_per_part]) {
2      unsigned short int temp_dest, temp_src;
3      float _update[2 * sources_per_part];
4
5      float _pr[sources_per_part];
6      PR_LOAD_sc:
7          for (unsigned short int i = 0; i < sources_per_part; i++) {
8  #pragma HLS loop_tripcount min = 1 max = 500
9  #pragma HLS PIPELINE
10             _pr[i] = pr[i] / degree[i];
11             _update[i] = 0;
```

```
12              _update[i + sources_per_part] = 0;
13          }
14      SCATTER_FOR:
15          for (unsigned short int i = 0; i < EDGES_0; i = i + 2) {
16  #pragma HLS loop_tripcount min = 1 max = 16063
17  #pragma HLS PIPELINE
18              temp_src = edge_set[i];
19              temp_dest = edge_set[i + 1];
20              _update[temp_dest] += _pr[temp_src];
21          }
22      for (unsigned short int i = 0; i < sources_per_part; i++) {
23  #pragma HLS pipeline
24          update_0[i] = _update[i];
25          update_1[i] = _update[i + sources_per_part];
26      }
27  }
```

Listing 4.4: Συνάρτηση scatter

Ο ρόλος των δύο gather είναι σχετικά απλός. Παίρνουν η κάθε μία τα update που τους αναλογούν, η πρώτη τα $update\_0, update\_2$ και η δεύτερη τα $update\_1, update\_3$. Τα αθροίζουν με τον σταθερό όρο και υπολογίζουν την απόκλιση στο δικό τους μέρος. Έπειτα επιστρέφουν την νέα τιμή pagerank των κόμβων και την απόκλιση του μέρους τους.

```
1   void gather(float updates_0[sources_per_part], float updates_1[sources_per_part
        ], float pr[sources_per_part], float & conv) {
2   #pragma HLS inline off
3
4       float temp_conv = 0,_alpha = 0.85, stand_part = (1.0 - _alpha) / N,
        temp_old, temp_new, temp_update;
5
6       GATHER_LOOP:
7           for (unsigned short int i = 0; i < sources_per_part; i++) {
8   #pragma HLS loop_tripcount min = 1 max = 500
9   # pragma HLS PIPELINE
10              temp_old = pr[i];
11              temp_update = updates_0[i] + updates_1[i];
12              temp_new = stand_part + alpha * temp_update;
13              temp_conv += abs(temp_new - temp_old);
14              pr[i] = temp_new;
15          }
16
17      conv = temp_conv;
18  }
```

Listing 4.5: Συνάρτηση gather

Μετά το τέλος και των συναρτήσεων gather, βρισκόμενοι στην calculate_pagerank, αθροίζονται οι επιμέρους αποκλίσεις και ελέγχεται εάν συνέκλινε ο αλγόριθμος. Σε περίπτωση που δεν έχουμε σύγκλιση συνεχίζεται ο βρόχος των επαναλήψεων, μέχρι να συγκλίσει ή να φτάσει έναν μέγιστο αριθμό που εμείς ορίζουμε. Εάν έχουμε σύγγλιση πριν το μέγιστο αριθμό επαναλήψεων διακόπτεται ο βρόχος και έχουμε το στάδιο της κανονικοποίησης και μεταφοράς- επιστροφής από το fpga στον ARM επεξεργαστή. Η κανονικοποίηση χρειάζεται γιατί όπως αναφέραμε και στην περιγραφή του αλγορίθμου, έχουμε το φαινόμενο rank sink, όπου κόμβοι δέχονται pagerank αλλά δεν μοιράζουν, μεταβάλλοντας το συνολικό άθροισμα.

Σχετικά με την μεταφορά και την αποθήκευση των δομών των δεδομένων, καταλήξαμε να κινηθούμε όπως και στην single core έκδοση. Όλοι οι πίνακες μεταφέρονται με AXI4-Stream, με access_pattern SEQUENTIAL, έχοντας κατανεμηθεί στη μνήμη με sds_alloc. Χρησιμοποιούν data_mover AXIDMA και θύρες υψηλής απόδοσης S_AXI_HP καθώς δεν χρειάζονται συνοχή με την κρυφή μνήμη. Η διαφορά όμως βρίσκεται στο γεγονός ότι τώρα οι δομές $edge\_set$ και $degree$ έχουν μοιραστεί σε $edge\_set\_0, edge\_set\_1$ και $degree\_0, degree\_1$, χρησιμοποιώντας τέσσερα κανάλια μεταφοράς αντί για δύο, για την μεταφορά τους. Αυτή η αξιοποίηση περισσότερων καναλιών θα αξιολογηθεί στο επόμενο κεφάλαιο.

```
1  #define N 1000
2  #define E 32000
3  #define partitions 2
4  #define sources_per_part 500
5  #define EDGES_0 2*15937
6  #define EDGES_1 2*16063
7
8  #pragma SDS data access_pattern(edge_set_0:SEQUENTIAL,edge_set_1:SEQUENTIAL,
       pr_out_0:SEQUENTIAL,pr_out_1:SEQUENTIAL,degree_0:SEQUENTIAL,degree_1:
       SEQUENTIAL)
9  #pragma SDS data mem_attribute(edge_set_0:PHYSICAL_CONTIGUOUS|NON_CACHEABLE,
       edge_set_1:PHYSICAL_CONTIGUOUS|NON_CACHEABLE,pr_out_0:PHYSICAL_CONTIGUOUS|
       NON_CACHEABLE,pr_out_1:PHYSICAL_CONTIGUOUS|NON_CACHEABLE,degree_0:
       PHYSICAL_CONTIGUOUS|NON_CACHEABLE,degree_1:PHYSICAL_CONTIGUOUS|
       NON_CACHEABLE)
10 #pragma SDS data data\_mover(edge_set_0:AXIDMA_SIMPLE,edge_set_1:AXIDMA_SIMPLE,
       pr_out_0:AXIDMA_SIMPLE,pr_out_1:AXIDMA_SIMPLE, degree_0:AXIDMA_SIMPLE,
       degree_1:AXIDMA_SIMPLE)
11 #pragma SDS data sys_port(edge_set_0:AFI,edge_set_1:AFI, pr_out_0:AFI,pr_out_1:
       AFI, degree_0:AFI,degree_1:AFI)
12
13 void calculate_pagerank(unsigned short int edge_set_0[EDGES_0],unsigned short
       int edge_set_1[EDGES_1],float pr_out_0[sources_per_part],float pr_out_1[
       sources_per_part],unsigned short int degree_0[sources_per_part],unsigned
       short int degree_1[sources_per_part],float &conv,int &hw_counter);
14
```

```
15 void scatter(unsigned short int edge_set[EDGES_0],float pr[sources_per_part],
       unsigned short int degree[sources_per_part],float update_0[sources_per_part
       ],float update_1[sources_per_part]);
16
17 void scatter1(unsigned short int edge_set[EDGES_1],float pr[sources_per_part],
       unsigned short int  degree[sources_per_part],float update_0[
       sources_per_part],float update_1[sources_per_part]);
18
19 void gather(float updates_0[sources_per_part], float updates_1[sources_per_part
       ], float pr[sources_per_part], float &conv);
20
21 void gather1(float updates_0[sources_per_part], float updates_1[
       sources_per_part], float pr[sources_per_part], float &conv);
```

Listing 4.6: Αρχείο header παραλληλοποιημένου PageRank

# Κεφάλαιο 5

# Αξιολόγηση

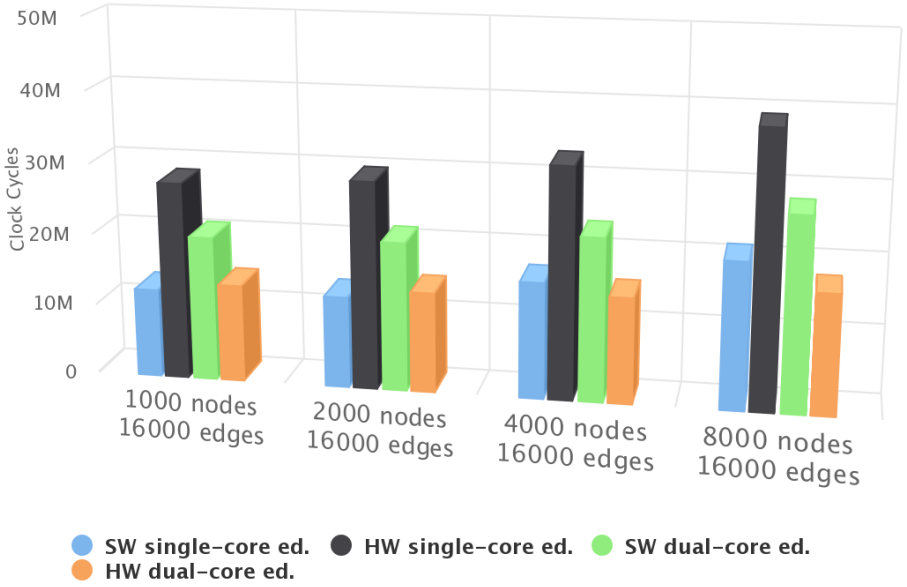
Αρχικά είναι θεμιτό να εξετάσουμε τον Αλγόρθιμο Pagerank πριν τις υλοποιήσεις. Βλέποντας την σχέση 3.4, παρατηρούμε ότι έχουμε δύο όρους, τον πρώτο που είναι σταθερός και τον δεύτερο με το σύμβολο της άθροισης. Και στις δύο υλοποιήσεις πρώτα δημιουργούμε τον δεύτερο όρο και έπειτα προσθέτουμε τον σταθερό πρώτο όρο. Όπως φαίνεται και στα σχήματα 3.6 και 3.7, έχουμε δημιουργήσει δομές δεδομένων για τον όρο του αθροιστή για κάθε κόμβο το $sum\_pr$ και το $giving\_pr$ για το Pagerank που μοιράζει κάθε κόμβος στους γείτονές του. Για την δημιουργία του αθροιστικού αυτού δεύτερου μέρους έχουμε τυχαίες προσπελάσεις μνήμης. Συγκεκριμένα προσπελάζοντας κάθε σύνδεσμο (ακμή γράφου), προσθέτουμε στον αθροιστή του κόμβου προορισμού το μέρος του Pagerank που μοιράζει ο κόμβος προέλευσης. Έτσι αν για παράδειγμα έχουμε σε σειρά δύο συνδέσμους (1,15111) και (14000,8) θα πρέπει αρχικά να διαβάσουμε το $giving\_pr[1]$ και να το προσθέσουμε στο υπάρχον $sum\_pr[15111]$ και έπειτα να διαβάσουμε το $giving\_pr[14000]$ και να το προσθέσουμε στο υπάρχον $sum\_pr[8]$. Εκτός από τις αλλεπάλληλες τυχαίες προσπελάσεις μνήμης, που προκαλούν μεγάλες καθυστερήσεις λόγω των αποτυχιών της μνήμης (misses), εχουμε και εξάρτηση μεταξύ των δεδομένων. Το $sum\_pr$ κάθε κόμβου ενημερώνεται από όλους τους γείτονες που δείχνουν σε αυτόν. Αυτό το γεγονός μας περιορίζει στο θέμα της παραλληλίας.

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 σημειό υπεροχής των FPGAs είναι οι επεξεργαστικές μονάδες. Μπορεί να προσφέρει μεγάλη παραλληλία και ενδείκνυται σε περιπτώσεις μεγάλου υπολογιστικού φόρτου, όταν δηλαδή έχουμε πολλές πράξεις, όσο το δυνατόν λίγοτερες προσπελάσεις μνήμης και πολλές επαναλήψεις.

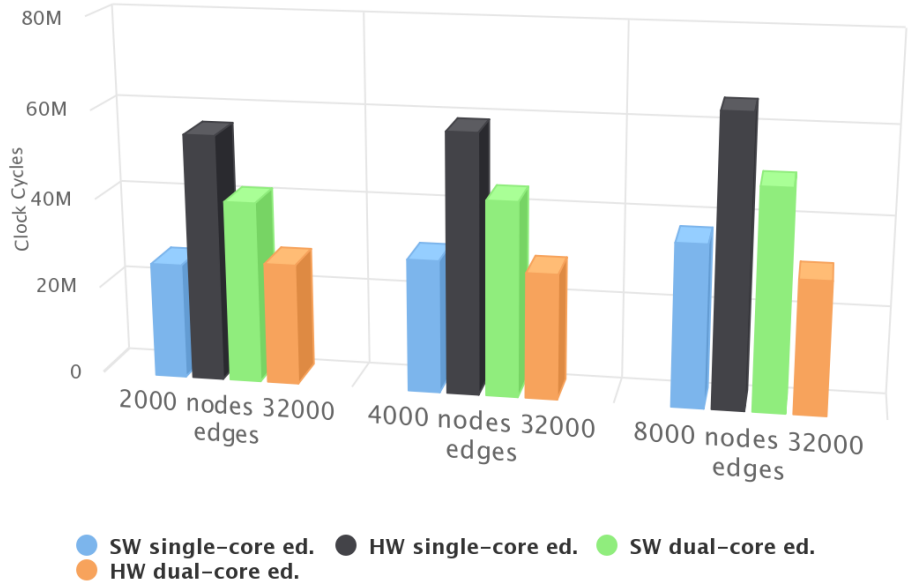
Αναλογιζόμενοι τα παραπάνω προβλέπουμε την δυσκολία επιτάχυνσης του αλγόριθμου Pagerank και άλλων αλγορίθμων γράφων με την χρήση FPGAs.

## 5.1 Αποτελέσματα

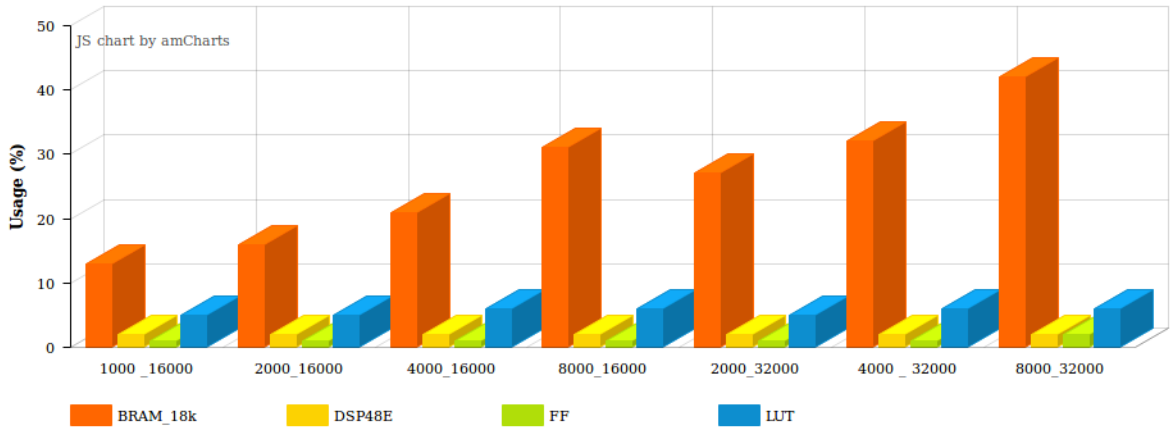
Για τις υλοποιήσεις του αλγορίθμου Pagerank σε FPGA που αναφέραμε στο προηγούμενο κεφάλαιο, έχουμε τα παρακάτω αποτελέσματα. Σημειώνουμε ότι οι κύκλοι ρολογιού μετρήθηκαν για συγκρίση σε λογισμικό (SW) και υλικό (HW). Επίσης για να έχουν σαφέστερο νόημα οι συγκρίσεις, ο αλγόριθμος παρότι έλεγχε την σύγκλιση, σε όλες τις εκτελέσεις θέσαμε να γίνονται 40 επαναλήψεις του αλγορίθμου.



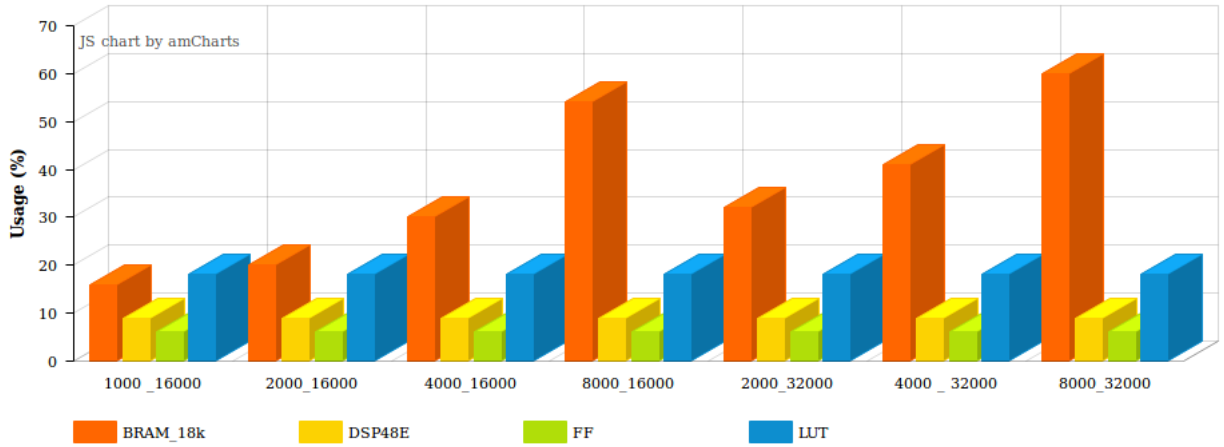

Σχήμα 5.1: Αποδόσεις των δύο υλοποιήσεων σε dataset 16000 συνδέσμων



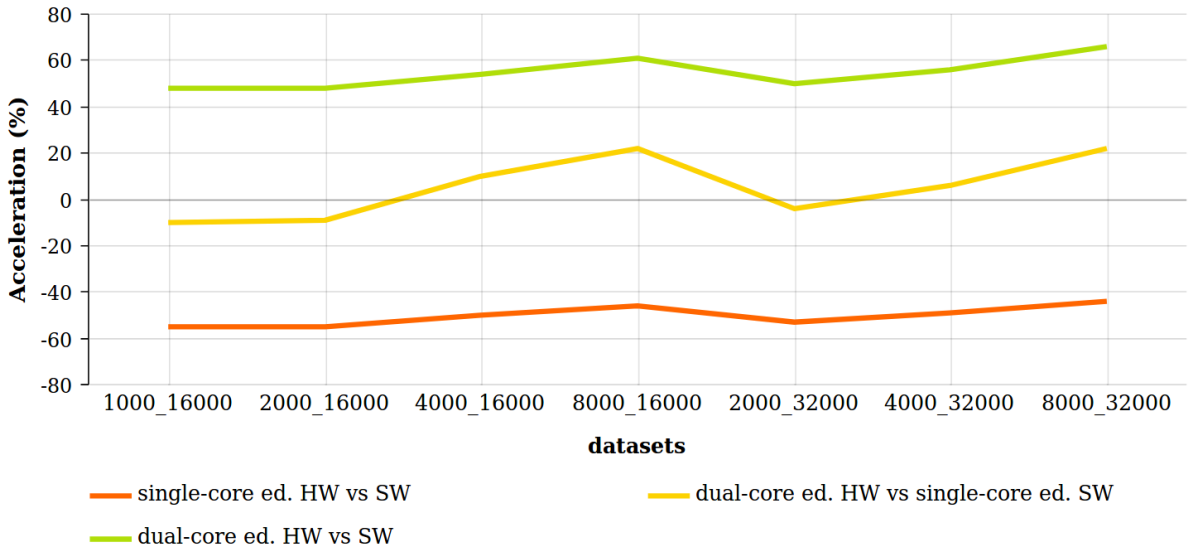

Σχήμα 5.2: Αποδόσεις των δύο υλοποιήσεων σε dataset 32000 συνδέσμων




Σχήμα 5.3: Χρήση πόρων FPGA στην single core ed.




Σχήμα 5.4: Χρήση πόρων FPGA στην dual core ed.




Σχήμα 5.5: Αποτελέσματα Επιτάχυνσης

### 5.1.1 Σχετικά με την Single core Αρχιτεκτονική

Όπως φαίνεται κάθαρα στα παραπάνω γραφήματα των σχημάτων 5.1 και 5.2, η single-core έκδοση ήταν πάντα περίπου δύο φορές πιο αργή όταν εκτελούνταν σε HW από όταν εκτελούνταν στον ARM επεξεργαστή (SW). Η υλοποίηση single-core δοκιμάστηκε με πολλούς τρόπους και βελτιώθηκε όσο ήταν δυνατόν. Μια τροποποίηση που δεν περιλαμβάνεται είναι η χρησιμοποίηση περισσότερων καναλιών για την μεταφορά των edge_set και degree. Αυτή η τροποποίηση θα είχε αντίκτυπο στην επιτάχυνση του αλγορίθμου και έχει χρησιμοποιηθεί στην dual core έκδοση.

Η αδυναμία της single-core υλοποίησης να επιταχύνει τον αλγόριθμο ή έστω ακόμα να σημειώσει ίδιους χρόνους σε software και hardware, κρίνεται λογική. Δεν μπόρεσε να ξεπεράσει το χρόνο μεταφοράς των δεδομένων και να σημειώσει επιτάχυνση. Όπως προαναφέραμε η δομή της δεν έχει παραλληλοποιηθεί, αλλά έχουν γίνει βελτιστοποιήσεις που προσφέρουν τα FPGAs όπως pipeline, loop unroll, σε επίπεδο εντολών και βρόχων. Χωρίς τις βελτιστοποιήσεις, τα αποτελέσματα του υλικού ήταν περίπου οκτώ φορές πιο αργά. Ο λόγος που αναλύουμε αυτήν την απλή έκδοση είναι το γεγονός ότι έχει απλούστερο αλγόριθμο, λιγότερο υπολογιστικό φόρτο και όπως φαίνεται στα παραπάνω γραφήματα είναι 1,3-1,6 φορές πιο γρήγορη στον ARM επεξεργαστή από την dual-core έκδοση. Εδώ να τονίσουμε ότι η dual-core έκδοση στον ARM εκτελείται σειρακά, δεν έχουμε την παραλληλία που έχουμε στο υλικό.

Αναλύοντας την χρήση των πόρων του FPGA αντικατοπτρίζεται η φύση του αλγορίθμου, πιο ευκρινώς σε αυτήν την απλή έκδοση. Εφόσον δεν έχουμε παραλληλία, δεν δημιουργουργούνται παραπάνω κυκλώματα για πράξεις στο FPGA και επόμενα οι πόροι του μένουν ανεκμετάλευτοι. 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. Όπως φαίνεται στα παραπάνω διαγράμματα, χρησιμοποιήσαμε μέχρι μόνο το 42% της block Ram στην απλή έκδοση. Αυτό έγινε ώστε να έχουμε πλήρη αντιστοιχία στην σύγκριση με την παραλληλοποιημένη έκδοση.

Η απόδοση εξαρτάται και από την πυκνότητα του γράφου, δηλαδή αν έχουμε πολλές ακμές ή αλλιώς από την αναλογία μεταξύ κόμβων και ακμών. Αρχικά χρησιμοποιήσαμε dataset 16000 και 32000 συνδέσμων, 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 της δομής degree. Αμφότερες στις περιπτώσεις των 16000 και 32000 συνδέσμων, όσο αυξάνοταν ο αριθμός των ακμών είχαμε μια μικρή αύξηση στην απόδοση του αλγορίθμου. Συγκερκριμένα το υλικό, όπως φαίνεται και στο γράφημα του σχήματος 5.5, στην περίπτωση των 16000 συνδέσμων ενώ αυξάνοταν οι κόμβοι από 1000 εώς 8000,το υλικό ήταν από 55% εώς 46% πιο αργό από τον ARM επεξεργαστή. Στην περίπτωση των 32000 συνδέσμων ενώ αυξάνοταν οι κόμβοι από 2000 εώς 8000,

το υλικό ήταν από 53% εώς 44% πιο αργό. Βλέπουμε ότι παρότι αυξάνοταν ο χρόνος μεταφοράς, το υλικό επιταχύνοταν. Αυτό οφείλεται στον αρχικό υπολογισμό των $giving\_pr$ που με το pipeline υπολογίζοταν πιο γρήγορα στο υλικό και επόμενα αντιστάθμιζοταν η καθυστέρηση που προκαλούσε η μεταφορά μεγαλύτερου όγκου δεδομένων της δομής degree. Ένα ακόμα βοηθητικό στην βελτίωση της απόδοσης είναι ο καλύτερος χειρισμός των αστοχιών της μνήμης από το υλικό.

### 5.1.2 Σχετικά με την Dual core Αρχιτεκτονική

Για την dual core έκδοση χρειάστηκε να μετατρέψουμε αρκετά τον αρχικό αλγόριθμο. Συγκρίνοντας την εκτέλεση στον ARM και στο FPGA και βλέποντας το σχήμα 5.5, το FPGA είναι από 48% εώς 66% ταχύτερο. Συγκρίνοντας όμως με την single core έκδοση, εκτελούμενη στον ARM επεξεργαστή, έχουμε επιτάχυνση από -10% εώς 22%. Έχουμε λοιπόν μία υλοποίηση που έχοντας τροποποιήσει την απλή έκδοση, έχουμε στον ARM 30-65% καθυστέρηση συγκριτικά με την πρώτη. Καταφέρνει όμως, όσον αφορά την εκτέλεση στο FPGA να είναι 100-130% ταχύτερη.

Αυτή η επιτάχυνση που επιτυγχάνεται, ιδίως συγκριτικά με την single core έκδοση, οφείλεται σε συγκεκριμένες τεχνικές που ακολουθήσαμε. Αρχικά όπως αναφέραμε παράπανω, εισερχόμενοι στην top-level function $calculate\_pagerank$ μεταφέρονται τα δυο σύνολα ακμών $edge\_set$ και τα δύο σύνολα $degree$. Τα σύνολα αυτά μεταφέρονται δύο φορές πιο γρήγορα από ότι στην περίπτωση της single core έκδοσης, καθώς μεταφέρονται με τέσσερα κανάλια, αντί για δύο. Έπειτα έχουμε και παραλληλοποίηση. Καθώς μετά την μεταφορά, ξεκινούν ταυτόχρονα τα δύο ip-cores $scatter$ και αμέσως μετά εκτελούνται και τα ip-cores $gather$. Αναλύσαμε παραπάνω όμως, ότι την δομή $update$ την δημιουργήσαμε και την παρεμβάλαμε για να επιτύχουμε παραλληλία. Αυτή η τεχνική όμως επιβάλλει περαιτέρω χρήση της block Ram, καθώς και παραπάνω λειτουργίες της μνήμης. Ακόμα και για την επιστροφή των δεδομένων της συνάρτησης, δηλαδή το Pagerank των κόμβων, έχουν χρησιμοποιηθεί δύο κανάλια αντί για ένα που έχουμε στην απλή έκδοση.

Αναλύοντας τα σχήματα 5.1, 5.2 βλέπουμε μεγαλύτερη χρήση των πόρων του FPGA συγκριτικά με την single core έκδοση, αλλά έχοντας πάλι εντονότερη τη χρήση της block RAM. Όπως εξηγήσαμε, ο ίδιος ο αλγόριθμος επιβάλλει την έντονη χρήση της BRAM και δεν ευνοεί την παραλληλία υπολογιστικών πράξεων. Ακόμα, στα ίδια dataset, συγκρίνοντας με την απλή έκδοση, βλέπουμε μεγαλύτερη χρήση της block RAM, που οφείλεται στις δομές update.

Είναι σημαντικό σε αυτό το σημείο να συμπληρώσουμε, την αξιολογήση αυτή και με την προσπάθεια χρήσης της DRAM. Προσπαθήσαμε να κινηθούμε στην λογική ότι αποθηκεύουμε στην πιο γρήγορη μνήμη (Block RAM), τα δεδομένα που προσπελάσονται κατά τυχαίο τρόπο, καθώς και στην πιο αργή μνήμη (DRAM), τα δεδομένα που προσπελάσονται με σειριακό τρόπο. Κατά αυτόν τον τρόπο θα μπορούσαμε να χειριστούμε καλύτερα τις αστοχίες μνήμης και θα μπορούσαμε να εκμεταλλευτούμε τον χώρο της DRAM, δίνοντας έτσι την δυνατότητα να επιλύσουμε μεγαλύτερους γράφους και dataset. Ακόμα

και στην σειριακή ανάγνωση της DRAM, είχαμε αρκετά μεγάλες καθυστερήσεις, που μας απαγόρευσαν την χρησιμοποίηση της. Ιδανικά η δομή edge_set, που προσπελάσεται σειριακά, χωρίς αστοχίες μνήμης, θα θέλαμε να βρίσκεται στην DRAM, καθώς καταναλώνει μεγάλο μέρος της BRAM.

# Κεφάλαιο 6

# Σχετική έρευνα

## 6.1 Προσέγγιση αλγορίθμων γράφων με FPGAs

Στην δημοσίευση των S. Zhou, C. Chelmis, V. K. Prasanna [10], πραγματεύονται την βελτιστοποίηση της απόδοσης της μνήμης για την υλοποίηση του Pagerank σε FPGA. Έχουν κινηθεί με παρόμοιο τρόπο με εμάς. Προσεγγίζουν τον αλγόριθμο με scatter - gather, καθώς και χειρίζονται παρόμοια τον διαμοιρασμό των dataset των συνδέσμων. Έχουν κινηθεί όμως δαφορετικά στην αποθήκευση των δομών, καθώς έχουν αποθηκεύσει στην DRAM τις δομές edge_set και update. Στην περίπτωση που χρησιμοποιούν παραλληλοποίηση σε βαθμό δύο, όπως εμείς, έχουν δύο δομές update, σε αντίθεση με εμάς που έχουμε τέσσερεις. Επίσης για να ελαχιστοποιήσουν, τις αστοχίες της μνήμης DRAM στην ενημέρωση των update, ταξινόμησαν την δομή edge_set με βάση τον κόμβο προορισμού. Τα αποτελέσματα που δίνουν συγκρίνουν με την αρχική τους υλοποίηση και δίνουν μια βελτίωση εώς 70%. Έτσι δεν έχουμε σαφές συμπέρασμα για τους χρόνους που χρειάζοταν εκτέλεση της υλοποίησης τους. Επίσης είμαστε αρκετά σκεπτικοί τόσο για την εκτενή χρήση της DRAM, όσο και για τον χρόνο που απαιτεί η ταξινόμηση του συνόλου των ακμών.

Οι S. Mcgettrick, D. Geraghty,C. McElroy [11] στην προσπάθεια τους να υλοποιήσουν το Pagerank σε FPGA, αρχικά διερεύνησαν την απόδοση του SMVM (Sparse Matrix Vector Multi- plication) σε FPGA για πίνακες διαδικτύου. Αξιολόγησαν την απόδοση δύο αρχιτεκτονικών SMVM βασισμένων σε FPGA που σχεδιάστηκαν αρχικά για προβλήματα πεπερασμένων στοιχείων σε ένα Virtex 5 FPGA. Η μια βασίστηκε στην δομή ELL που αναπτύξαμε στην παράγραφο 3.3 και η δεύτερη αποθήκευε τον πίνακα σε μπλοκ (υποπίνακες), η οποία είχε καλύτερες αποδόσεις. Αρχικά έδειξαν ότι η αναδιάταξη RCM (Reverse Cuthill McKee) [13], είχε θετική επίδραση, αλλά δεν ήταν η κατάλληλη προτείνοντας διερεύνηση για πιο ταιριαστή μέθοδο αναδιάταξης. Στη συνέχεια, διερεύνησαν την δυνατότητα υπέρβασης των γενικού σκοπού επεξεργαστών, χρησιμοποιώντας παραλληλισμό των μονάδων επεξεργασίας, δείχνοντας ότι το SMVM που βασίζεται σε FPGA μπορεί να αποδώσει καλύτερα από το SMVM σε έναν επεξεργαστή. Τελικά πέτυχαν επιτάχυνση τριών φορών, που μάλιστα θεωρούν ότι μπορεί να γίνει μεγαλύτερη. Καθώς σε όλη την διε-

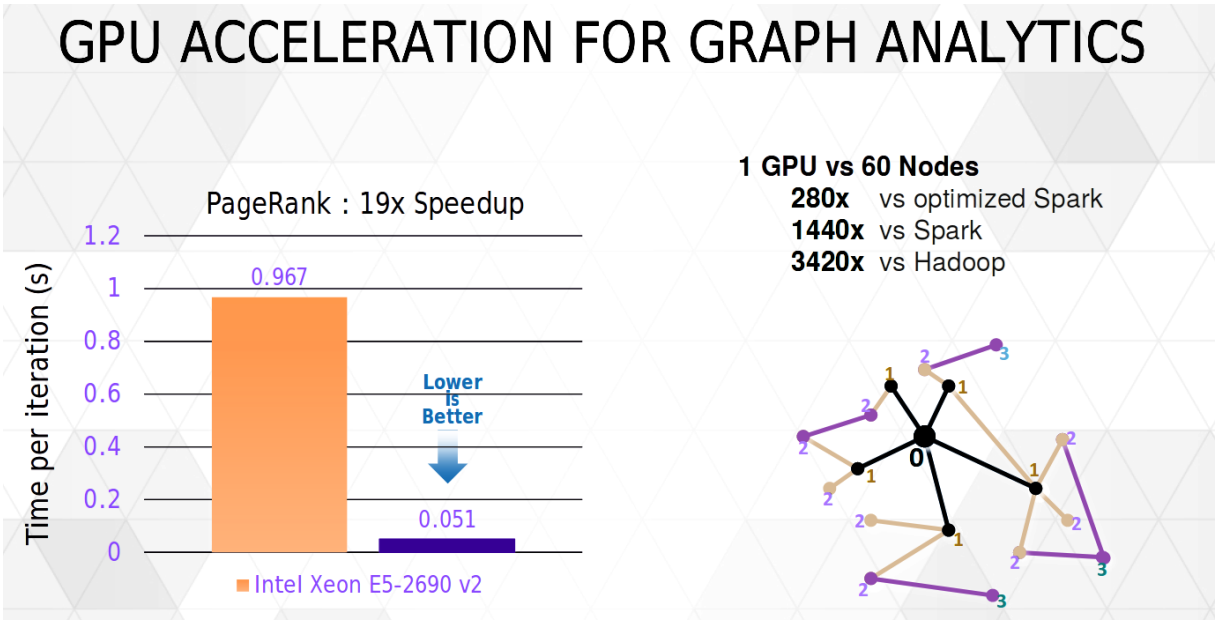
ξαγωγή της έρευνας χρησιμοποίησαν διπλής ακρίβειας κινητής υποδιαστολής μεταβλητές και στην περίπτωση του Pagerank δεν χρειάζεται αυτή η ακρίβεια.

Οι ίδιοι, έπειτα την έρευνα που προαναφέραμε , προχώρησαν στην υλοποίηση του Pagerank σε FPGA [12]. Χρησιμοποίησαν τους SMVM που ανάλυσαν στην προηγούμενη δημοσίευση τους με κάποιες τροποποιήσεις. Μετέτρεψαν τον πίνακα του διαδικτύου με τα βάρη του Pagerank, όπως αναφέραμε στην παράγραφο 3.3, σε ένα πίνακα - διάνυσμα για εξοικονόμηση μνήμης. Έπειτα προχώρησαν στην σχεδίαση υλικού, όπου συμπεριέλαβαν τρία SMVM ip cores. Υλοποίησαν αποδοτικούς τρόπους για την χρήση της μνήμης DRAM. Χρησιμοποίησαν τέσσερα ανεξάρτητα DDR κανάλια μνήμης, καθώς και τους σχετικούς χειριστές μνήμης. Όσον αφορά το κομμάτι της εκτέλεσης χρησιμοποίησαν την τεχνική blocking, φορτώνοντας υποπίνακες σε buffers από την DRAM, καθώς και ενδιάμεσους buffers για την αποθήκευση πίσω στην DRAM. Τελικά πέτυχαν επιτάχυνση εώς 2.5 φορές συγκρίνοντας την υλοποίηση τους σε ένα Virtex-5 και σε ένα επεξεργαστή Pentium Xeon (Woodcrest).

## 6.2 Χρήση GPUs για υλοποίηση αλγορίθμων γράφων

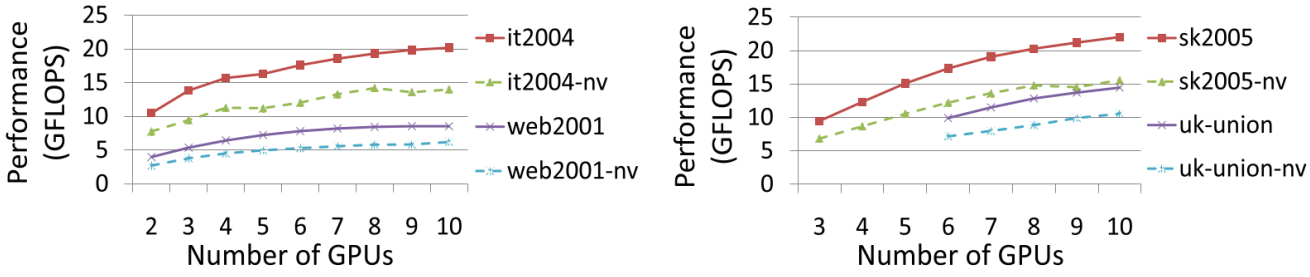
Οι GPUs είναι άλλος ένας τρόπος επιτάχυνσης αλγορίθμων, με ευρεία ήδη χρήση. Είναι μονάδες επεξεργασίας γραφικών που σε συναργασία με CPU επιταχύνουν εφαρμογές βαθιάς μάθησης, ανάλυσης και μηχανικής. Με πρωτοπόρο την NVIDIA από το 2007, οι επιταχυντές GPU εξοπλίζουν πλέον ενεργειακά αποδοτικά κέντρα 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.

Στο συνέδριο Geoint το 2015, ο L. Brown (Solution Architect της NVIDIA), παρουσίασε υλοποιήσεις εφαρμογών στοιχείων ανάλυσης γράφων με GPUs [14]. Συγκεκριμένα για το Pagerank όπως φαίνεται και στο παρακάτω σχήμα 6.1, κατέδειξε πως εκτελείται 19 φορές πιο γρήγορα στην NVIDIA K40 GPU από όταν εκτελείται σε έναν επεξεργαστή Intel Xeon E5-2690 v2. Στην συνέχεια 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[14]

Την πεποίθηση ότι το Pagerank και άλλοι αλγόριθμοι γράφων ευδοκιμούν στις GPUs, έρχεται να στηρίξει και η δημοσίευση [15]. Ερευνώντας με βάση πάλι το SpMV, τις δομές του και τις δυνατότητες αναδιάταξης αυτών συγκρίνει συνολικά την εκτέλεση τους σε GPUs. Έπειτα συγκρίνει, εκτελώντας σε μία GPU, τους βελτιστοποιημένους του kernels για SpMV με αυτούς των NVIDIA's SpMV βιβλιοθήκης, Baskaran και Bordawekar και με εκτελώντας σε μία CPU. Σε αυτήν τη σύγκριση οι βελτιστοποιημένοι kernel του ήταν πάντα καλύτεροι. Συγκεκριμένα για το Pagerank οι υλοποιήσεις του σε μια GPU συγκριτικά με μία CPU ήταν 18 με 32 φορές ταχύτερες. Μεγάλο ενδιαφέρον έχει όμως η υλοποίηση του Pagereank σε framework πολλών GPUs και σύγκριση με αντίστοιχο της NVIDIA.




(a) it-2004 and web-2001

(b) sk-2005 and uk-union

Σχήμα 6.2: Επεκτασιμότητα του multi-GPU Pagerank, σύγκριση με τον HYB (NVIDIA) [15]

Όπως βλέπουμε και στο σχήμα 6.2 στην δυγκεκριμένη δημοσίευση επιτυγχάνεται ταχύτητα εώς 20 GFLOPs και είναι πάντα ταχύτερη από τον kernel HYB της NVIDIA. Να σημειωθεί ότι στην μέχρι τώρα έρευνα μας, δεν έχουμε συναντήσει υλοποίηση του Pagerank σε FPGA να ξεπερνάει το 1,1 GFLOPs.

## 6.3 Σχετκή 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 [16] 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 [17] 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 ακμών τρισεκατομμυρίων. 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 [18] είναι μια πέντε ετών startup στην 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 μεταγλωτιστή 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 Pagerank, 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 Pagerank. Δεν καταφέραμε να έχουμε τα επιθυμητά αποτελέσματα. 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 συμπερασμάτα που βγήκαν δεν ήταν απαγορευτικά.

## 7.2 Μελλοντική Δουλειά

Τα περιθώρια βελτίωσης στην υπάρχουσα λογική που κινήθηκαμε στην dual core υλοποίηση 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 το Pagerank, που αφορούν μεγάλους γράφους χρειάζεται να μπορούν να κατανεμηθούν σε συστάδες και μάλιστα διαμοιραζόμενης μνήμης. Καθώς έχουν εξαρτήσεις δεδομένων και δεν επιτρέπουν εύκολη, χωρίς εμπόδια παραλληλία. Σύμφωνα μάλιστα με την σχετική δουλειά που αναλύσαμε στο 6ο κεφάλαιο, υπάρχουν περαιτέρω δυνατότητες αποδοτικότερης ανάπτυξης εφαρμογών γράφων σε FPGAs, με υπάρχοντες ήδη υλοποιήσεις.

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 SpMV έχουν αρκετές δυνατότητες επιτάχυνσης σε GPUs. Ο SpMV έχει μελετηθεί αρκετά σε GPUs και έχουν αναπτυχθεί kernels με κορυφαίες επιδόσεις. 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, CPUs στην ανάπτυξη εφαρμογών γράφων. Ελπίζουμε να δούμε αποτελέσματα αυτών σύντομα. Αν υλοποιηθούν οι GSPs, ικανοποιώντας τα χαρακτηριστικά που τους προλογούν, πιθανότα στο μέλλον, θα αποτελέσουν την βάση για τις εφαρμογές ανάλυσης γράφων.

# Βιβλιογραφία


[1] Vivado Design Suite User Guide 902 High-Level Synthesis
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902-vivado-high-level-synthesis.pdf

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

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

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

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

[6] A. Roy, I.Mihailovic, W. Zwanpoel, "X-Stream: Edge-centric Graph Processing using Streaming Partitions"

[7] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web"

[8] S. Brin, L. Page, "The anatomy of a large-scale hypertextual web search engine" in Computer Networks and ISDN Systems1998;30:107–117.

[9] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, N. Koziris, "Understanding the Perfomance of Sparse Matrix-Vector Multiplication"

[10] S. Zhou, C. Chelmis, V. K. Prasanna, "Optimizing memory performance for FPGA implementation of pagerank"

[11] S. McGettrick, D. Geraghty, C. McElroy, "Towards an FPGA Solver for the PageRank Eigenvector Problem"

[12] S. McGettrick, D. Geraghty, C. McElroy, "An FPGA Architecture for the Pagerank Eigenvector problem"

[13] RCM - Reverse Cuthill McKee Ordering
http://people.sc.fsu.edu/∼jburkardt/m_src/rcm/rcm.html

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

[15] X. Yang, S. Parthasarathy, P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs: Implications on Graph Mining"

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

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

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