Ατομική Διπλωματική Εργασία

Maxeler MaxCompiler
FPGA Based Hardware Acceleration

Χαράλαμπος Χαραλάμπους

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Δεκέμβριος 2013

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Maxeler MaxCompiler
FPGA Based Hardware Acceleration

Χαράλαμπος Χαραλάμπους

Επιβλέπων Καθηγητής

Pedro Trancoso

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Δεκέμβριος 2013

ΕΥΧΑΡΙΣΤΙΕΣ

Με την ολοκλήρωση της διπλωματικής μου θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή μου κ. Pedro Trancoso που μου έδωσε την ευκαιρία να ασχοληθώ με τον συγκεκριμένο θέμα ως επίσης και για την στήριξη και για τις πολύτιμες συμβουλές του καθ' όλη την διάρκεια διεκπεραίωσης της εργασίας μου.

Επίσης θα ήθελα να ευχαριστήσω θερμά τον κ. Frederico Pratas (Research Scientist at Intel Barcelona Research Center in Spain) για την πολύτιμη συνεχή βοήθεια και συμβουλές του τις οποίες μπορώ να χαρακτηρίσω ως καθοριστικές για την επιτυχή ολοκλήρωση της εργασίας μου.

Ακόμη θα ήθελα να ευχαριστήσω τον διδακτορικό φοιτητή Ανδρέα Διαβαστό του οποίου οι συμβουλές οποιαδήποτε στιγμή χρειάστηκα ήταν αρκετά βοηθητικές.

Ιδιαίτερες ευχαριστίες στον κ. Παττίχη που αποδέχτηκε να είναι ο δεύτερος εξεταστής στην παρουσίαση της ατομικής διπλωματικής μου εργασίας.

Τέλος θα ήταν αμέλεια μου να μην ευχαριστήσω την οικογένεια και τους φίλους μου για την συνεχή στήριξη τους κατά την διάρκεια εκπόνησης της εργασίας μου.

Περίληψη

Στην επιστήμη της πληροφορικής χρησιμοποιούνται πολλές τεχνικές επιτάχυνσης και βελτίωσης των επιδόσεων εφαρμογών. Η παράλληλη επεξεργασία είναι ο τομέας στον οποίο ανατίθεται το μεγαλύτερο βάρος για την ανεύρεση και επιτυχή υλοποίηση τεχνικών για να επιτευχθούν. Μία απ' αυτές τις μεθόδους είναι η χρήση του υλικού και ειδικότερα μέσω της χρήσης των FPGAs (Field Programmable Gate Arrays).

Στη παρούσα διπλωματική εργασία έχει γίνει μελέτη και ανάλυση της χρήσης των FPGAs σε υπολογιστικά συστήματα και η προτροπή αξιοποίησης τους ιδιαίτερα για βελτιστοποίηση των επιδόσεων σε επερωτήματα βάσεων δεδομένων καθώς και άλλου είδους εφαρμογές.

Μελετήθηκε και αναλύθηκε η νέα καινοτομία χρήσης και αξιοποίησης των FPGAs μέσω ενός νέου εργαλείου του MaxCompiler του οποίου οι λειτουργίες και δυνατότητες περιγράφονται αναλυτικά στην εργασία αυτή. Επίσης εκτός της αναλυτικής περιγραφής του εργαλείου περιγράφονται τα πλεονεκτήματα και μειονεκτήματα του. Η διαφορετικότητα του εστιάζεται στο γεγονός ότι μπορούμε να διαμορφώσουμε και να προγραμματίσουμε ένα FPGA με την χρήση μιας υψηλού επιπέδου γλώσσας προγραμματισμού αντί της χρήσης των μέχρι τώρα γνωστών γλωσσών περιγραφής υλικού.

Τέλος επιλέγηκε η εφαρμογή (επερώτημα) TPC-H Query 6 για μετατροπή του σε υλοποίηση MaxCompiler, γίνονται πειραματικές μετρήσεις και συγκρίσεις επιδόσεων μεταξύ της υλοποίησης αυτής και της αρχικής ενώ επίσης δίνεται έμφαση στην περιγραφή της διαδικασίας μετατροπής της εφαρμογής για καλύτερη εμπέδωση, αντίληψη και εξοικείωσης με το εργαλείο.

Πίνακας περιεχόμενων

Κατάλογος Σχημάτων

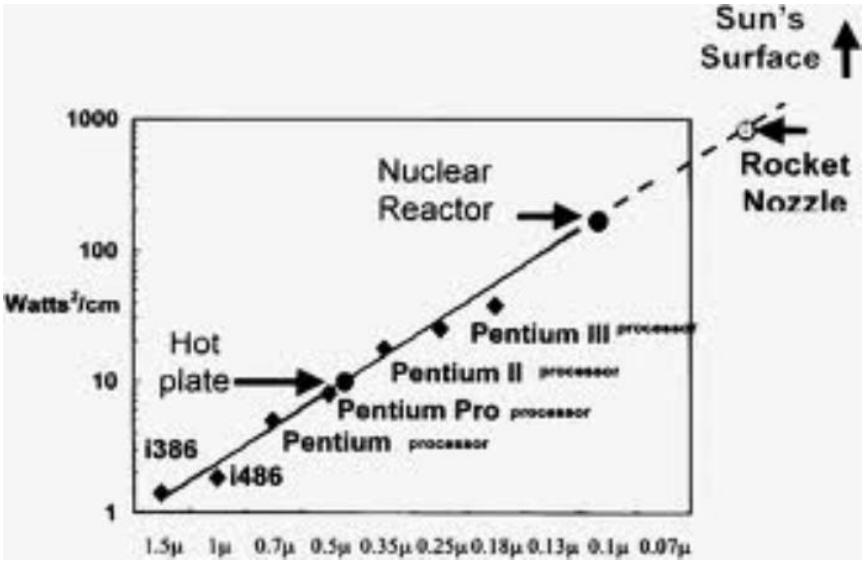
Κατάλογος Πινάκων

Κεφάλαιο 1 Εισαγωγή

1.1 Παράλληλη Επεξεργασία

Παράλληλη επεξεργασία είναι η ταυτόχρονη εκτέλεση δύο ή και περισσοτέρων διαφορετικών ροών εκτέλεσης κατά το ίδιο χρονικό διάστημα. Παράλληλο σύστημα έχουμε όταν συνυπάρχουν πολλαπλά αντίτυπα μονάδων υλικού (Hardware) στο ίδιο υπολογιστικό σύστημα τα οποία έχουν τη δυνατότητα να παράγουν έργο ταυτόχρονα.

Η παράλληλη επεξεργασία μεταμορφώθηκε από την ιδέα στην πράξη όταν η συνηθισμένη μέθοδος αύξησης των επιδόσεων και μείωσης του χρόνου εκτέλεσης εφαρμογών στους υπολογιστές, η αύξηση της συχνότητας των επεξεργαστών, δεν μπορούσε πλέον να συνεχιστεί με την ίδια συχνότητα όπως γινόταν μέχρι το πρόσφατο παρελθόν. Από την δημιουργία των ηλεκτρονικών υπολογιστών μέχρι και πριν από λίγα χρόνια η βελτίωση των επιδόσεων στηριζόταν αποκλειστικά στην συνεχή αύξηση της συχνότητας του επεξεργαστή. Η μέθοδος αυτή έφτασε όμως στα “όρια” της (power wall) αφού θα οδηγούσε σε τεράστια κατανάλωση ενέργειας ως επίσης και ραγδαίας αύξησης της θερμοκρασίας των συστημάτων σε επίπεδα που δεν θα ήταν ανεχτά ούτε καν για τα υλικά που είναι φτιαγμένοι οι ηλεκτρονικές υπολογιστές, πόσο μάλλον για τους χρήστες. Επίσης τα πιο πάνω προβλήματα θα δημιουργούσαν την ανάγκη για ακριβά συστήματα ψύξης και θα μείωναν την διάρκεια της μπαταρίας σε κινητές και ενσωματωμένες συσκευές και μάλιστα στις μικρότερες από αυτές όπως για παράδειγμα στα smart phones λόγω μεγέθους δεν υπάρχει χώρος για ανεμιστήρες ψύξης.



Σχήμα 1.1:CPU's Power Wall[1]

Η λύση του πιο πάνω προβλήματος και νέα φιλοσοφία για βελτίωση επιδόσεων με λιγότερη κατανάλωση ενέργειας και χαμηλότερες θερμοκρασίες είναι η παράλληλη επεξεργασία και οι πολυπύρηνοι επεξεργαστές. Η παράλληλη επεξεργασία[2][3] λειτουργεί με βασική αρχή ότι μεγάλα υπολογιστικά προβλήματα κατά πάσα πιθανότητα θα μπορούν να διαιρεθούν σε μικρότερα τα οποία θα μπορούν να επιλύονται ταυτόχρονα. Η παράλληλη επεξεργασία παρουσιάζεται σε αρκετές διαφορετικές μορφές[4] όπως:

- Παραλληλισμός σε Επίπεδο Δεδομένων (Bit Level Parallelism) που είναι η ταυτόχρονη εκτέλεση πράξεων σε ομοειδή δεδομένα, σε ένα επεξεργαστή ή και επεξεργαστή ειδικού σκοπού (π.χ. bit-level παραλληλισμός, λειτουργίες επεξεργασίας γραφικών, FPGAs, ASICs).
- Παραλληλισμός σε Επίπεδο Εντολών (Instruction Level Parallelism, ILP) που είναι η ταυτόχρονη εντολών ενός προγράμματος (κρυφή μνήμη (cache), προ-προσκόμιση, διοχέτευση εντολών και πράξεων, υπερβαθμωτή (superscalar) εκτέλεση, πρόβλεψη διακλάδωσης(branch prediction)).
- Παραλληλισμός σε Επίπεδο Λογικού Πυρήνα (Multithreading) που είναι η ταυτόχρονη εκτέλεση νημάτων (ελαφρών διεργασιών) σε ένα επεξεργαστή
- Παραλληλισμός σε Επίπεδο Φυσικού Πυρήνα (Multicore) που είναι η ενσωμάτωση περισσοτέρων πλήρων επεξεργαστικών μονάδων(CPU) στο ίδιο ολοκληρωμένο κύκλωμα οι οποίες συνδέονται σε εσωτερικό διαυλο και μοιράζονται υψηλότερα επίπεδα κρυφής μνήμης (cache).
- Παραλληλισμός σε Επίπεδο Λειτουργιών (Multitasking) που είναι η ταυτόχρονη εκτέλεση λειτουργιών (διεργασιών) σε ένα υπολογιστή (π.χ.

χρήση συνεπεξεργαστή (co-processor), άμεση πρόσβαση μνήμης (DMA), I/O, κάρτες γραφικών)υπό τον έλεγχο του λειτουργικού συστήματος.

- Παραλληλισμός σε Επίπεδο Υπολογιστή/Υπολογισμου (Multiprocessing/ Multicomputing) που είναι η ταυτόχρονη εκτέλεση ενός ή διαφορετικών προγραμμάτων σε πολλαπλούς επεξεργαστές ή πλήρεις υπολογιστές.
- Παραλληλισμός σε Επίπεδο Δικτύου και Διαδικτύου (Network/ Distributed/ Computing) που είναι η ταυτόχρονη εκτέλεση διαφορετικών (συνήθως) προγραμμάτων σε πολύ χαλαρά συνδεδεμένους (loosely coupled) υπολογιστές. Σε τέτοιο επίπεδο το κύριο ζητούμενο δεν είναι η ταχύτητα αλλά η κατανομή των πόρων, των δεδομένων και των εργασιών και η μεγαλύτερη διεκπεραιωτή ικανότητα (throughput).

Η οδός της παράλληλης επεξεργασίας θα έφερνε νέες προκλήσεις στους προγραμματιστές όπως τον εντοπισμό του κομματιού του κώδικα ενός σειριακού προγράμματος όπου μπορεί να εφαρμοστεί και κατά πόσο θα αποδώσει ο παραλληλισμός ως επίσης και το πώς θα συνεργαστούν και θα επικοινωνήσουν οι διάφορες υπολογιστικές μονάδες ώστε να παράξουν τα επιθυμητά αποτελέσματα χωρίς σφάλματα καθώς και την ανάγκη για νέα προγραμματιστικά μοντέλα, νέες γλώσσες προγραμματισμού, νέες αρχιτεκτονικές, νέες μεθοδολογίες καθώς και εκμετάλλευσης και αποδοτικής αξιοποίησης του υλικού (hardware) ώστε να επιτευχθεί η μέγιστη δυνατή επεξεργασία στον ελάχιστο δυνατό χρόνο.

1.2 Χρήση του υλικού για εκτέλεση εφαρμογών

Με τον όρο εκτέλεση εφαρμογών με τη βοήθεια του υλικού εννοούμαι την εκτέλεση κάποιων συναρτήσεων, λειτουργιών ή/και κομματιού ενός προγράμματος σε οποιαδήποτε άλλη μονάδα επεξεργασίας (κομμάτι του υλικού) εκτός της κεντρικής μονάδας επεξεργασίας (CPU). Η εκτέλεση εφαρμογών με την βοήθεια του υλικού γίνεται με σκοπό να αποφορτίσουμε την κεντρική μονάδα επεξεργασίας (CPU) από διεργασίες οι οποίες μπορούν να εκτελεστούν αλλού και να εκμεταλλευτούμε αυτό το “κενό” της για να της αναθέσουμε άλλες διεργασίες οι οποίες τους είναι απαραίτητη η εκτέλεση σ’ αυτήν. Επίσης μπορεί να μας προσφέρει οφέλη όπως την κατανάλωση λιγότερης ενέργειας απ’ ότι μια ολοκληρωμένη εκτέλεση στην κεντρική μονάδα επεξεργασίας. Ιδανικό είναι το σενάριο να επιτύχουμε τα πιο πάνω και παράλληλα να πετύχουμε και βελτίωση της απόδοσης της εκτέλεσης μιας εφαρμογής με την συνεργασία υλικού και λογισμικού.

Τέτοια κομμάτια υλικού που μπορούν να εκτελέσουν κομμάτια μιας εφαρμογής μπορεί να είναι κάρτες γραφικών, GPUs, FPGAs κ.α. και όταν τους αναθέτουμε κάτι τέτοιο συνηθίζετε να τους αναφέρουμε ως συνεπεξεργαστές.

1.3 Κίνητρο Ατομικής Διπλωματικής Εργασίας

Η παράλληλη επεξεργασία και η προσπάθεια βελτίωσης αποδόσεων στις μέρες μας δεν στηρίζετε απλά στους πολυπύρηνους επεξεργαστές και μόνο αλλά έχει απλώσει τα πλοκάμια στη χρήση των καρτών γραφικών, των GPU ακόμη και του Hardware (υλικού). Το ερώτημα κατά πόσο μια υλοποίηση με την εκμετάλλευση του υλικού μπορεί να επιφέρει βελτίωση επιδόσεων αποτελεί το κύριο κίνητρο της διπλωματικής μου εργασίας. Η απάντηση στο ερώτημα εξαρτάται από πολλές παραμέτρους και δεν μπορεί να απαντηθεί με ένα ναι η ένα όχι. Πρέπει να μελετηθούν και να ληφθούν υπόψη για ποιου τύπου προγράμματα μπορεί να επέλθει η βελτίωση (αν μπορεί), για ποιας πολυπλοκότητας προγράμματα, αν μπορεί να βελτιωθεί η επίδοση επερωτημάτων (queries) μιας βάσης δεδομένων καθώς επίσης και για ποιους όγκους δεδομένων και τις διαφορές που μπορεί να προκύψουν στις επιδόσεις λαμβάνοντας σαν παραμέτρους τα πιο πάνω. Επιπλέον τι μπορεί να μας προσφέρει η εκμετάλλευση του υλικού με την χρήση των FPGAs και παράλληλα η μελέτη των δυνατοτήτων του νέου εργαλείου της Maxeler του MaxCompiler.

1.4 Σκοπός και αναμενόμενα αποτελέσματα

Η παρούσα διπλωματική εργασία έχει εκπονηθεί με σκοπό τη μελέτη και τρόπους εκμετάλλευσης και διαμόρφωσης των FPGAS ώστε να επιτύχουμε επιτάχυνση στην εκτέλεση εφαρμογών και επερωτημάτων σε βάσεις δεδομένων.

Το θεωρητικό μέρος της εργασίας είναι η επικέντρωση κυρίως στην περιγραφή του MaxCompiler, της αρχιτεκτονικής , του τρόπου λειτουργίας και των δυνατοτήτων του. Για το πρακτικό μέρος της εργασίας μου θα πρέπει να αναγνωριστεί το κομμάτι κώδικα το οποίο εκτελείται πιο εντατικά σε ένα σειριακό πρόγραμμα και η μετατροπή αυτού σε FPGA διαμόρφωση (configuration) με την βοήθεια μιας νέας τεχνολογίας επιτάχυνσης που μας προσφέρει η Maxeler Technologies τον Maxeler MaxCompiler ώστε να επιμορφωθούμε σχετικά, να εξοικειωθούμε και να εξάγουμε συμπεράσματα για την τεχνολογία αυτή.

Όσον αφορά τα αναμενόμενα αποτελέσματα αναμένουμε την μη παρουσία βελτίωσης ή ακόμη και χειρότερων επιδόσεων σε απλά, γρήγορα (όσον αφορά χρόνους) και για μικρό όγκο δεδομένων σειριακά προγράμματα αφού ίσως η διαδικασία (χρόνος) μεταφοράς δεδομένων προς το FPGA, η εκτέλεση των

υπολογισμών σε αυτό και η επιστροφή των αποτελεσμάτων από το FPGA θα είναι απλά μια χρονοβόρα διαδικασία (σε σχέση με το σειριακό) που θα εξάγει τα ίδια αποτελέσματα. Για προβλήματα μεγαλύτερης πολυπλοκότητας που δεν τρέχουν σε "μηδαμινούς" χρόνους αλλά και με μεγάλο όγκο δεδομένων αναμένουμε ότι μια υλοποίηση στον MaxCompiler θα μπορεί να επιφέρει βελτίωση και επιτάχυνση καθώς και άλλα οφέλη όπως λιγότερη κατανάλωση ενέργειας και η μείωση του φόρτου εργασίας του επεξεργαστή.

1.5 Οργάνωση και Μεθοδολογία

Για την επιτυχή διεκπεραίωση της εργασίας μας θα έπρεπε να ακολουθηθούν κάποια στάδια και βήματα τα οποία έπρεπε να θέσουμε εξ αρχής και να ακολουθήσουμε. Σε πρώτο στάδιο ήταν η γενική επιμόρφωση μας γύρω από το θέμα της παράλληλης επεξεργασίας ώστε να κτίσουμε το υπόβαθρο μας για να μπορούμε να εμβαθύνουμε ακολούθως σε κάποιο πιο εδικό τομέα της. Ακολούθως ξεκίνησε η διαδικασίας μελέτης δημοσιεύσεων σχετικά με τους επιταχυντές υλικού και κυρίως των FPGAs, από τι αποτελούνται, πως μπορούμε να τα χρησιμοποιήσουμε ή/και προγραμματίσουμε, τι διαφορετικό μπορούν να μας προσφέρουν και πώς μπορούν να αξιοποιηθούν στο μέγιστο βαθμό. Έχει μελετηθεί επίσης η χρήση τους για βελτίωση επερωτημάτων για βάσεις δεδομένων όπως και για άλλου είδους εφαρμογών. Το επόμενο βήμα που ήταν και αρκετά σημαντικό για την εργασία μου ήταν η μελέτη του εργαλείου που ασχολείται ο MaxCompiler ο οποίος είναι μια νέα τεχνολογία που μας προσφέρει η Maxeler Technologies για αξιοποίηση των FPGAs με ένα διαφορετικό τρόπο από τους μέχρι σήμερα γνωστούς. Ξεκίνησα με την γενικότερη μελέτη πληροφοριών γύρω από το εργαλείο ξεκινώντας με πληροφορίες από τη σελίδα της εταιρείας και ακολούθως εμβάθυνα ψάχνοντας για δημοσιεύσεις στις οποίες έχει χρησιμοποιηθεί και τις μελέτησα για να αντιλήφθώ όσο το δυνατό περισσότερα γι' αυτό. Αφού είχα μελετήσει ξεκίνησα την δική μου πρακτική πάνω στο εργαλείο χρησιμοποιώντας κυρίως τα φροντιστήρια που έχουν ετοιμαστεί για τον MaxCompiler, τα οποία περιέχουν αρκετές βοηθητικές πληροφορίες για την χρησιμοποίηση του καθώς και διαφόρων ειδών παραδείγματα από τα απλούστερα που μπορεί να υπάρξουν μέχρι πιο σύνθετα, τα οποία παρατηρούσα, μελετούσα, μεταγλώττούσα, έτρεχα και έκανα τις δίκες μου μετατροπές σε αυτά. Στα φροντιστήρια του MaxCompiler και στα παραδείγματα που περιέχουν αυτά υπάρχουν επίσης αρκετές λεπτομέρειες για τα κομμάτια που πρέπει να περιέχει ένα ολοκληρωμένο πρόγραμμα υλοποιημένο στον MaxCompiler, πώς μπορούν να τροποποιηθούν ανάλογα με τις ανάγκες του προγράμματος μας ως επίσης και το ρόλο και λειτουργίες του κάθε κομματιού. Ακολούθως επέλεξα μια σειριακή εφαρμογή

και συγκεκριμένα το επερώτημα TPC-H Query6 (Q6), την ανάλυσα και ακολούθως την μετέτρεψα σε υλοποίηση για MaxCompiler έκανα κάποιες πειραματικές μετρήσεις και εξήγαγα τα συμπεράσματα μου. Τέλος περιγράφω επίσης το τι μπορεί να γίνει ως μελλοντική δουλειά της εργασίας μου που αποτελείται από έξι κεφάλαια και η δομή της αναλύεται στο υποκεφάλαιο 1.6.

1.6 Δομή της Εργασίας

Η ατομική διπλωματική μου εργασία χωρίζεται σε έξι κεφάλαια μέσα από τα οποία περιγράφεται η συνολική δουλειά της εργασίας μας. Κάθε κεφάλαιο περιγράφει και ένα μέρος της εργασίας όπως αυτά έχουν αποφασιστεί να χωριστούν από τον γράφων. Στο Κεφάλαιο 1 κάναμε μια σύντομη εισαγωγή της εργασίας, αναφέροντας μεταξύ άλλων το κίνητρο, το σκοπό καθώς και τη μεθοδολογία που ακολουθήθηκε. Στο Κεφάλαιο 2 περιγράφονται και αναλύονται επιστημονικά άρθρα και δουλειές που έχουν γίνει και σχετίζονται με το θέμα μας, κυρίως για την αξιοποίηση των FPGAs και για εργασίες που έγιναν με την χρήση του εργαλείου που ασχολούμαστε δηλαδή του MaxCompiler. Στο Κεφάλαιο 3 γίνεται μια σύντομη περιγραφή για τους επιταχυντές υλικού, τι μπορεί να είναι ένας επιταχυντής υλικού και του πως μπορεί να αξιοποιηθεί το υλικό των ηλεκτρονικών υπολογιστών ώστε να επωφεληθούμε από τη χρήση του όσον αφορά κυρίως επιδόσεις καθώς επίσης και τα πλεονεκτήματα του, μέθοδοι χρήσης του, πως συνεργάζεται με το λογισμικό. Στο κεφάλαιο αυτό γίνεται περισσότερη εμβάθυνση και περιγραφή των Field Programmable Gate Arrays (FPGAs) στα οποία επικεντρώνεται και η όλη μας εργασία. Στο Κεφάλαιο 4 περιγράφεται αναλυτικά ο MaxCompiler που είναι και το εργαλείο στο οποίο εμβαθύνει η εργασία μας και στο κεφάλαιο αυτό περιγράφονται οι βασικές του λειτουργίες, πως μπορεί να χρησιμοποιηθεί, πως μπορεί να αξιοποιηθεί για βελτίωση επιδόσεων, από τι αποτελείται καθώς επίσης και τα πλεονεκτήματα και τα μειονεκτήματα του. Το Κεφάλαιο 5 είναι το κεφάλαιο στο οποίο παρουσιάζω το πώς εγώ έχω χρησιμοποιήσει τον MaxCompiler παίρνοντας μια σειριακή εφαρμογή την οποία θα μετατρέψω σε υλοποίηση για τον MaxCompiler. Κύριος σκοπός σε αυτό το κεφάλαιο είναι να δείξω την διαδικασία του πως μπορεί αυτό να γίνει, να γίνουν πειραματικές μετρήσεις και συγκρίσεις μεταξύ των υλοποιήσεων και βάση των αποτελεσμάτων να μπορούν να εξαγχθούν συμπεράσματα. Επίσης περιγράφεται η προσπάθεια βελτίωσης της επίδοση της εφαρμογής ,οι συγκρίσεις των αποτελεσμάτων μας και παρουσιάζονται οι σχετικές γραφικές παραστάσεις. Στο τελευταίο Κεφάλαιο 6 της διπλωματικής μου εργασίας θα περιγραφούν τα συμπεράσματα της συνολικής μας δουλείας καθώς επίσης και η μελλοντική εργασία που μπορεί να γίνει περί του θέματος.

Κεφάλαιο 2 Σχετική Δουλειά

2.1 Ανάλυση Σχετικών Εργασιών για FPGAs

Για να είμαστε σε θέση να επιτύχουμε τους στόχους που θέσαμε για την εργασία μας ήταν απαραίτητο να ψάξουμε, να μελετήσουμε και να αναλύσουμε επιστημονικά άρθρα και σχετικές με την εργασία μας δουλειές που έχουν γίνει ώστε να εμβαθύνουμε τις γνώσεις μας γύρω από το αντικείμενο και να αντιληφθούμε σε καλύτερο βαθμό τη χρήση, λειτουργία και την ορθή αξιοποίηση του.

Ξεκινώντας θέλησα να μελετήσω γενικότερα περί της χρήσης των FPGAs και αξιοποίησης τους για βελτίωση επιδόσεων. Ψάχνοντας είχα προσέξει ότι γίνεται μια προσπάθεια γενικώς για αξιοποιήσει τους στο τομέα των βάσεων δεδομένων και ότι εστιάζεται ιδιαιτέρα στην βελτίωση των επιδόσεων επερωτημάτων σε αυτές. Οι λόγοι που γίνεται αυτή η προσπάθεια μπορούν να γίνουν εύκολα αντιληπτοί από κάποιον σχετικό με το χώρο της πληροφορικής, καθώς η εξέλιξη της τεχνολογίας και της επιστήμης μας μπορεί να παρομοιαστεί σαν ένα “ασταμάτητο” τρένο στη πάροδο των χρόνων που πάντα λες εδώ θα σταματήσει και όλο και κάτι ετοιμάζει για να σε εκπλήξει. Η εξέλιξη αυτή έχει σαν αποτέλεσμα οι βάσεις δεδομένων στις μέρες μας να έχουν τεράστιο όγκο δεδομένων αποθηκευμένο που φθάνει και ξεπερνά ακόμη και τα Terabytes, τα οποία πριν από μερικά χρόνια υπήρχαν απλά σαν ορολογία, που έχει σαν αποτέλεσμα ότι τα ερωτήματα στις μέρες μας θα πρέπει να επεξεργαστούν αυτούς τους τεράστιους όγκους ώστε να εξάγουν τα αποτελέσματα τους. Όπως όλοι γνωρίζουμε “το γοργόν και χάριν έχει” άρα γιατί όχι και στις επιδόσεις των επερωτημάτων βάσεων δεδομένων; Με αυτό το σκεπτικό καθώς και την ανάγκη για γρηγορότερη επεξεργασία τεράστιων όγκων δεδομένων και εξαγωγής αποτελεσμάτων ξεκίνησε και συνεχίζεται η προσπάθεια με διάφορες τεχνικές παράλληλης επεξεργασίας να επιτευχθεί στο μέγιστο δυνατό βαθμό η βελτίωση αυτή.

Ανάμεσα στις τεχνικές και μεθόδους που ακολουθήθηκαν συμπεριλαμβάνεται κα η χρήση των FPGAs και σε αυτή εστιάζονται και οι δημοσιεύσεις τις οποίες έχω μελετήσει.

Στα άρθρα[5] που επέλεξα προς ανάλυση βρήκα αρκετές χρήσιμες πληροφορίες γύρω από τα FPGAs και το πώς μπορούν να χρησιμοποιηθούν. Τα FPGAs είναι στη ουσία ολοκληρωμένα επαναπρογραμματζόμενα κυκλώματα τα οποία μπορούν να υλοποιήσουν οποιαδήποτε λογική πράξη. Στα πλεονεκτήματα συμπεριλαμβάνονται η χαμηλή καθυστέρηση και ο υψηλός βαθμός παραλληλοποίησεις τους. Είναι ευέλικτα όσον αφορά τον προγραμματισμό τους και μπορούν αξιοποιήσουν όλους τους διαθέσιμους τους πόρους. Για να περιοριστούν τα FPGAs όσον αφορά τα όρια τους πρωταρχικό και πιο σύνηθες είναι η εξάντληση των διαθέσιμων lookup tables που διαθέτουν. Άρα ο αριθμός των lookup tables σε ένα FPGA chip είναι σημαντικός παράγοντας για να το περιορίσει όσον αφορά την πολυπλοκότητα του κυκλώματος που επιθυμούμαι να προγραμματιστεί. Επιπλέον ενδεχόμενοι περιορισμοί και όρια των δυνατοτήτων των FPGAs είναι οι απαιτήσεις σε μνήμη, η διαθεσιμότητα επιπλέον λειτουργικότητας καθώς και το ευρος ζώνης του μέσου διασύνδεσης. Οι συχνότητες ρολογιού των FPGAs δεν μπορεί να συγκριθεί με αυτές των γενικού σκοπού κεντρικών μονάδων επεξεργασίας (CPUs) αφού συνήθως είναι γύρω στα 100MHZ ενώ των CPUs στα μερικά GHZ (~<=3) παρόλα αυτά όμως μπορεί να συμπεριφερθεί στα θετικά του γιατί αν αξιοποιηθούν σωστά τα πλεονεκτήματα του FPGA και έχουμε μια ανάλογη ή και καλύτερη επίδοση της εκτέλεσης σε σχέση με το CPU τότε σημαίνει ότι μέσω του FPGA καταφέραμε να εκτελέσουμε τις ίδιες λειτουργίες σε πανομοιότυπο χρόνο με λιγότερη κατανάλωση ενέργειας που υπολογίζεται στις μέρες μας σαν μια αρκετά σημαντική παράμετρος. Τα FPGAs προγραμματίζονται με τις γλώσσες περιγραφής υλικού (Hardware Description Languages-HDL) με πιο γνωστές τις VHDL και Verilog. Με τον κώδικα των γλωσσών περιγραφής υλικού και επιπρόσθετα εργαλεία λογισμικού μπορούμε να συνθέσουμε τον σχεδιασμό λογικών ολοκληρωμένων κυκλωμάτων και ακολούθως να φορτώσουμε το σχεδιασμό αυτό πάνω στο FPGA chip. Μαζί με τα εργαλεία αυτά προσφέρεται συνήθως και υψηλής ακρίβειας προσομοιωτές για να ελέγχετε ο σχεδιασμός πριν την πραγματική του φόρτωση και εκτέλεση στα FPGAs. Για να υπάρξει προοπτική της χρήσης των FPGAs στης βάσεις δεδομένων θα πρέπει να χρησιμοποιηθούν για επεξεργασία streams δεδομένων. Έτσι η ροή δεδομένων σε ένα τέτοιο σύστημα μεταφράζεται άμεσα στην φυσική καλωδίωση για την πλευρά του υλικού,

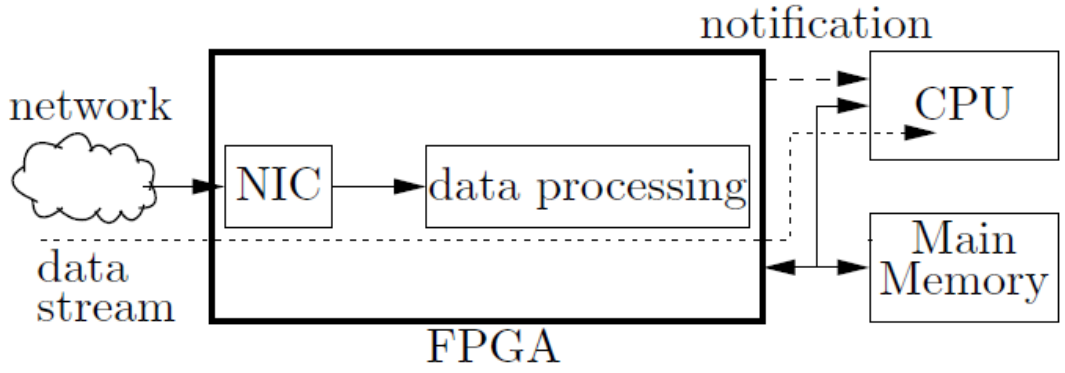
Προτείνετε επίσης για επερωτήματα βάσεων δεδομένων όπως εκτελούνται ασύγχρονα στο FPGA ούτως ώστε να μπορεί να αξιοποιηθεί ο μέγιστος δυνατός παραλληλισμός και για να προσφέρει κάτι διαφορετικό σε σχέση με τον τρόπο που λειτουργούν οι γενικού σκοπού επεξεργαστές η ο οποίοι ανά κύκλο εκτελούν μια και

μόνο εντολή. Στην ασύγχρονη εκτέλεση του FPGA σε κάθε κομμάτι του πλάνου (έχοντας εικόνα ενός διαγράμματος δεδομένων) όπου μπορεί να εκτελείτε κάποια λειτουργία εάν υπάρχει διαθέσιμο δεδομένο τότε θα εκτελείτε. Αυτό μας δίνει την δυνατότητα να εκτελούμε περισσότερες από μία εντολές ανά κύκλο στο FPGA και να παράξουμε έτσι και γρηγορότερα τα αποτελέσματα μας. Από μια αυστηρά σύγχρονη εκτέλεση στο FPGA δεν μπορούμε να περιμένουμε βελτίωση καθώς σε ένα απλό επερώτημα όπως αυτό που παρουσιάζει το άρθρο χρειάζεται πέντε κύκλους ώστε να εξάγει ένα αποτέλεσμα φυλάγοντας τα ενδιάμεσα αποτελέσματα σε κάθε κύκλο στους flip-flop καταχωρητές ενώ η ασύγχρονη εκτέλεση σε ένα κύκλο μπορεί να εξάγει το αποτέλεσμα. Ο λόγος που συμβαίνει αυτό με τις δύο εκτελέσεις είναι διότι όταν ένας operator εκτέλεση την λειτουργία του στην σύγχρονη εκτέλεση για ένα δεδομένο, φυλάσσει το αποτέλεσμα του στον καταχωρείτε εξόδου του και ακολούθως παραμένει ανενεργός μέχρι την έναρξη του επόμενου κύκλου όπου θα παραλάβει τα δεδομένα από τον καταχωριτή εισόδου του και να επαναλάβει την διαδικασία. Αυτό γίνετε για λόγους συγχρονισμού αλλά ο “νεκρός” αυτός χρόνος επηρεάζει κατά πολύ την απόδοση.

Επίσης ένα από τα πλεονεκτήματα που μας προσφέρουν τα FPGAs στην επεξεργασία δεδομένων είναι η παράλληλη πρόσβαση στην ίδια του τη μνήμη αφού έχει αρκετούς και διανεμημένους στην πλακέτα του πόρους διαφεύγοντας έτσι από το von Neumann bottleneck (γνωστό και ως φράγμα μνήμης) με το οποίο οι υπολογιστές με κλασσική αρχιτεκτονική ταλαιπωρούνται τα μέγιστα. Τα δεδομένα στα FPGAs ανακτώνται από την μνήμη στέλλοντας τους την διεύθυνση της μνήμης όπου βρίσκεται ένα τεμάχιο δεδομένων και μετά την ολοκλήρωση της δουλειάς του επιστρέφει πίσω τα δεδομένα(αποτελέσματα) που παράγει. Οι διανεμημένοι πόροι που αναφέραμε είναι οι flip-flop καταχωρητές και οι block-RAM (BRAMs). Ειδικότερη χρήση των δυνατοτήτων αυτών ενός FPGA γίνεται με την υλοποίηση της CAM (Content-addressable memory) που μπορεί να γίνει προσβάσιμη από την τιμή των δεδομένων παρά από την ρητή τους διεύθυνση μνήμης. Οι CAM μνήμες χρησιμοποιούνται κυρίως στην επεξεργασία δικτύων και ιδιαίτερα στην υλοποίηση των πινάκων δρομολόγησης ή/και στο ταίριασμα μοτίβων στους firewalls.

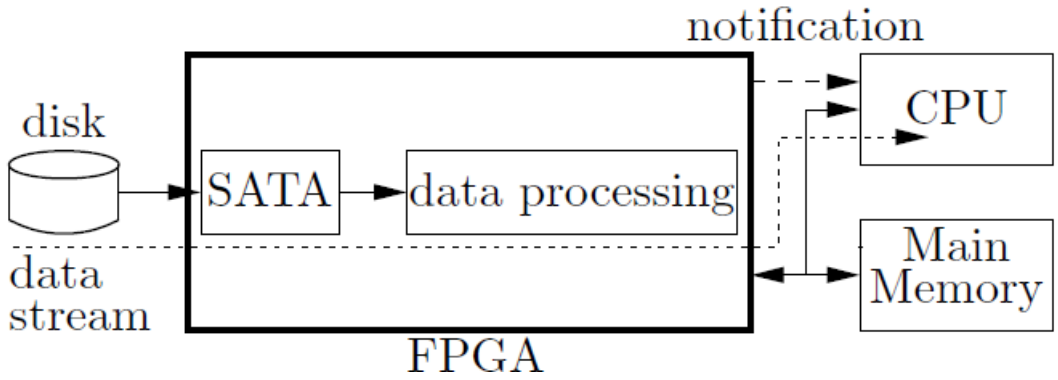
Η χρησιμοποίηση των FPGA στα συστήματα βάσεων δεδομένων (DBMS) μπορεί να γίνει με τρείς διαφορετικές αρχιτεκτονικές που στην κάθε μια μπορεί να προσφέρει κάτι διαφορετικό και η κάθε μια απ’ αυτές ασφαλώς έχει τα δικά της πλεονεκτήματα και μειονεκτήματα. Ένα FPGA λοιπόν μπορεί να σε τοποθετηθεί ένα σύστημα διαχείρισης βάσης δεδομένων μεταξύ της διεπαφής δικτύου (NIC) και της κεντρικής μονάδας επεξεργασίας (CPU) στην περίπτωση που έχουμε λειτουργίες ευαίσθητες

στην επεξεργασία των streams δεδομένων. Με αυτό τον τρόπο το FPGA χρησιμοποιείται για το φιλτράρισμα και την συνάθροιση και ο φόρτος εργασίας του CPU μειώνεται αισθητά και ταυτοχρόνως αυξάνεται η εξωτερικός φόρτωση.



Σχήμα 2.1: FPGA τοποθετημένο μεταξύ διεπαφής δικτύου (NIC) και CPU[5]

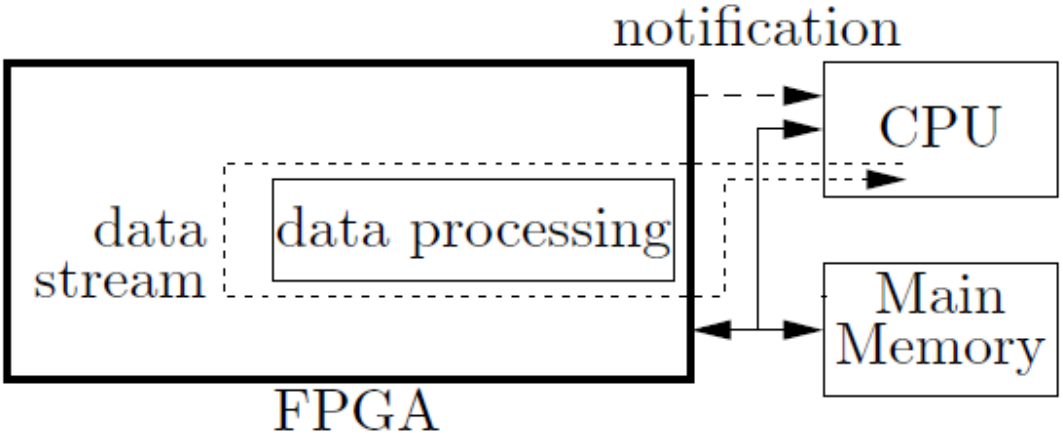
Παρομοίως τρόπος χρήσης του FPGA αλλά αντί για διεπαφή δικτύου έχουμε δίσκο ως την πηγή των δεδομένων στην οποία έχει πρόσβαση και προ-φιλτράρει τα δεδομένα το FPGA είναι η δεύτερη αρχιτεκτονική και απεικονίζετε στο Σχήμα 2.2: FPGA τοποθετημένο μεταξύ δίσκου και CPUΣχήμα 2.2 που είναι το πρότυπο για βάσεις δεδομένων των οποίων τα δεδομένα χρειάζονται ουσιαστική σάρωση. Και σε αυτή την αρχιτεκτονική το FPGA βοηθά στην μείωση του φόρτου εργασίας του CPU



Σχήμα 2.2: FPGA τοποθετημένο μεταξύ δίσκου και CPU[5]

Η τρίτη περίπτωση για τη των FPGAs στις βάσεις δεδομένων είναι η χρήση τους σαν συνεπεξεργαστές του CPU. Με αυτή την προσέγγιση μπορούμε να αποφορτίσουμε το CPU από διεργασίες εντατικού υπολογισμού και να τις αναθέσουμε στο FPGA το οποίο σε τέτοιες περιπτώσεις είναι κατά πάσα πιθανότητα συνδεδεμένο με ένα PCI δίαυλο διασύνδεσης. Η επικοινωνία μεταξύ CPU και FPGA γίνεται μέσω κοινής μνήμης και ειδοποιήσεων μέσω διακοπών. Η αρχιτεκτονική αυτή μπορεί εύκολα να ενσωματωθεί σε ένα παραδοσιακό σύστημα, όμως η μεταφορά δεδομένων από και προς στο FPGA από την κύρια μνήμη την οποία μοιράζονται με τους επεξεργαστές

μπορεί να επιφέρει μεγάλο κόστος όσον αφορά τη καθυστέρηση και ιδιαίτερα για μεγάλο όγκο δεδομένων μπορεί να επιδεινώσει το “memory bottleneck” των παραδοσιακών συστημάτων κάτι που συνεπάγεται ότι η χρήση του FPGA σε τέτοια περίπτωση χάνει ένα από τα κύρια της πλεονεκτήματα. Λύσεις γύρω από το πρόβλημα αυτό υπάρχουν φυσικά και μπορούν να υλοποιηθούν. Μια λύση είναι η βαθιά διασωληνωμένη εκτέλεση που να μας παρέχει ακόμη και βελτίωση των επιδόσεων.



Σχήμα 2.3: Το FPGA σαν συνεπεξεργαστής[5]

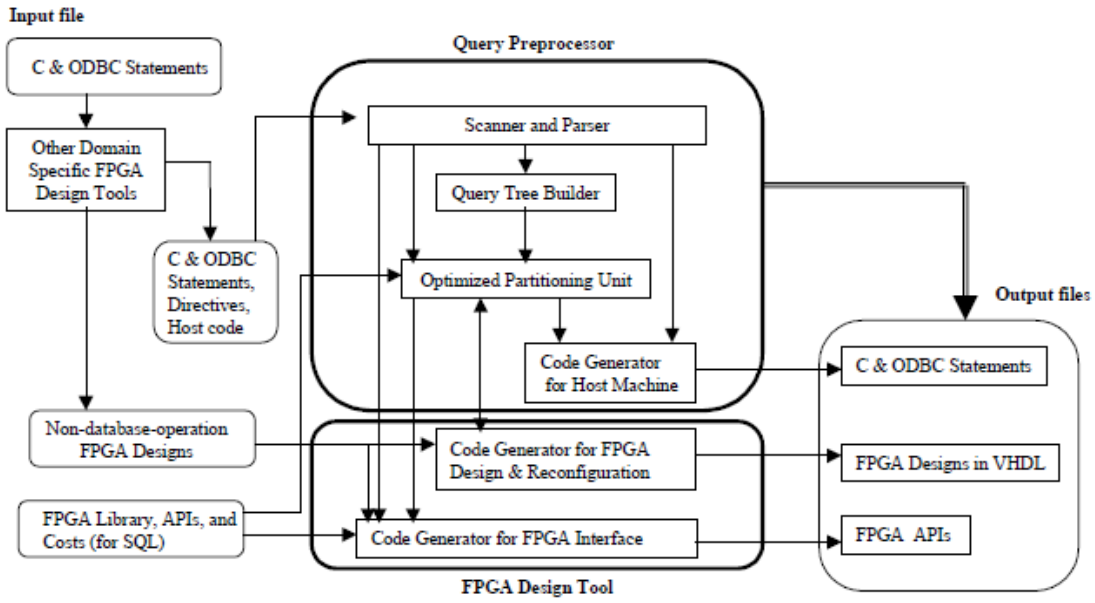
Τέλος προτείνεται να ληφθούν σοβαρότερα υπόψη η χρήση των FPGAs στις βάσεις δεδομένων από την κοινότητα τους ενώ στον αντίποδα η κοινότητα που ασχολείται με το υλικό έχει κάνει αρκετή δουλειά και έχει αναπτύξει αρκετές μεθόδους τις οποίες τα FPGAs μπορούν να επιφέρουν όφελος στις βάσεις δεδομένων. Επίσης σημειώνεται ότι ο παραλληλισμός που προσφέρουν είναι παρόμοιας φύσεως με αυτόν των GPUs δηλαδή SIMD(Single Instruction Multiple Data – Μια Εντολή Πολλά Δεδομένα) και ότι υλοποιήσεις υλικού έχουν προταθεί στον τομέα εξόρυξης δεδομένων φέρνοντας παράδειγμα υλοποίησης του αλγόριθμου Apriori σε FPGA βασισμένο σε συστολικές συστοιχίες (πίνακες).

Τα FPGAs χρησιμοποιούνται με επιτυχία ως συνεπεξεργαστές για διάφορους τύπους εφαρμογών των οποίων μπορούν να επιτύχουν επιτάχυνση και κυρίως εφαρμογών που δεν περιέχουν μεγάλο αριθμό υπολογισμών κινητής διαστολής. Επίσης έγινε προσπάθεια αξιοποίησης τους για επεξεργασία επερωτημάτων ανίχνευσης κειμένων και ταίριασμα εικόνας σε σχεσιακές βάσεις δεδομένων αποτελούμενες από κείμενα και εικόνες [6]. Έγινε η χρήση δύο αλγορίθμων για ανίχνευση κειμένου του KMP αλγόριθμου και του αλγόριθμου βίαιης παράλληλης σύγκρισης. Ο KMP αν και αποδέχτηκε καλύτερος στην σειριακή εκτέλεση η υλοποίηση του σε FPGA βρίσκει δύο σοβαρά κωλύματα που ήταν ο δύσκολος έλεγχος του κυκλώματος του λόγω

πολυπλοκότητας καθώς επίσης και δυσκολίες του FPGA να εισάγει εξωτερικά δεδομένα σε κάθε κύκλο ρολογιού λόγω των εξαρτήσεων μεταξύ των δεδομένων κάθε προηγούμενου υπολογισμού με τον αμέσως επόμενο του και αυτό έχει ως αποτέλεσμα να μπορεί να εισάγει νέα δεδομένα κάθε 3 κύκλους ρολογιού. Σε αντίθεση με τον KMP ο αλγόριθμος “βίαιης” παράλληλης σύγκρισης στην δική του υλοποίηση μπορεί να εισάγει νέο χαρακτήρα σε κάθε κύκλο ρολογιού και να και να επεξεργάζεται 4 αντίγραφα της υλοποίησης του ανά κύκλο ρολογιού και έτσι θεωρήθηκε ως ανώτερος του KMP Και επιλέγει για την υλοποίηση. Σε μετρήσεις που έγιναν έδειξαν ότι η σειριακή εκτέλεση για 22Kbytes κείμενο “case insensitive” πήρε 1,5 δευτερόλεπτο για μετατροπή της περίπτωσης και 0,01 για το ταίριασμα ενώ η εκτέλεση της ίδιας διαδικασίας στην FPGA σχεδίαση πηρέ λιγότερο από 0,01 δευτερόλεπτα και για τους δύο υπολογισμούς κάτι που συνεπάγεται και σε επιτάχυνση μεγαλύτερη του 150. Για το ταίριασμα εικόνας ο χρήστης καθορίζει μια περιοχή 128 x 128 από μια εικόνα 320 x 240 και χρησιμοποιεί αυτή την περιοχή ως πρότυπο για να το συγκρίνει με όλες τις εικόνες της βάσης δεδομένων. Στις γκρίζων αποχρώσεων εικόνες γίνεται προεπεξεργασία και οι εικονοστοιχίες (pixels) τους μετατρέπονται σε δυαδικές τιμές για απλοποίηση και επιτάχυνση της διαδικασίας ταιριάσματος. Το 128 x 128 πρότυπο μας κινείται στο παράθυρο ελέγχου για αναντιστοιχίες (mismatches) οι οποίες ελέγχονται μέσω πύλης XOR που με αποτέλεσμα ένα συνεπάγεται αναντιστοιχία αφού πρότυπο και εικόνα σύγκρισης έχουν σε αυτό το στάδιο δυαδική μορφή. Η εκτέλεση της διαδικασίας στον επεξεργαστή παίρνει χρόνο 8,48 δευτερολέπτων για εννέα εικόνες και η εκτέλεση σε μία υπολογιστική μονάδα FPGA 3,97 δευτερόλεπτα κάτι το οποίο συνεπάγεται επιτάχυνση ίση με 2,17. Η χρησιμοποίηση περισσότερων της μίας υπολογιστικής μονάδας πάνω στο FPGA θα επιφέρει περεταίρω βελτίωση και η χρήση περισσοτέρων του ενός FPGAs ακόμη ψηλότερη επιτάχυνση. Συγκεκριμένα με τέσσερις υπολογιστικές μονάδες σε ένα FPGA θα έχουμε επιτάχυνση κοντά στο 5 και με 4 FPGAs μπορεί να ξεπεράσει και το 10.

Επίσης προτείνεται αλγόριθμος πέραν του αυθεντικού JOIN για να επιτευχθεί επιτάχυνση καθώς ο αυθεντικός αλγόριθμος με πολυπλοκότητα O(M log N) δεν μας επιτρέπει κάτι τέτοιο. Ο σχεδιασμός που προτείνεται είναι οι διασωλήνωση συγκρίσεων όπου το κάθε αποτέλεσμα σύγκρισης θα καθορίζει τον τελεστή για να σταλθεί στο επόμενο στάδιο σύγκρισης και με αυτό τον τρόπο για 40MHZ σχεδίαση θα μπορούν 40 εκατομμύρια ακέραιων αριθμών να επεξεργάζονται σε κάθε δευτερόλεπτο ενώ ο επεξεργαστής 600 MHz Pentium III που χρησιμοποιείται στην κανονική υλοποίηση θα χρειάζεται 30 δευτερόλεπτα ώστε να εκτελέσει τον ανάλογο

αριθμό τέτοιων λειτουργιών. Ως αποτέλεσμα αυτού συνεπάγεται τεράστια βελτίωση της επίδοσης και της JOIN λειτουργίας.



Σχήμα 2.4: FPGA σχεδίαση[6]

Τελικό στάδιο αυτής της δουλείας ήταν η επιτάχυνση της προ-επεξεργασίας των επερωτημάτων με την μετατροπή τους από SQL σε γλώσσα προγραμματισμού C και ο διαχωρισμός των ενσωματωμένων SQL καταστάσεων (δηλώσεων) και η μετατροπή τους σε δύο μέρη. Το ένα μέρος για τους πόρους του FPGA και το άλλο για την back-end μηχανή της βάσης δεδομένων. Ο λόγος που συμβαίνει το πιο πάνω είναι ο πεπερασμένος αριθμός των πόρων του FPGA αφού πιθανόν να μην μπορεί να αγκαλιάσει όλες τις λειτουργίες των επερωτημάτων στη κάρτα του. Τα βήματα της προ-επεξεργασίας των επερωτημάτων απαιτεί τα ακόλουθα βήματα. Αρχικά ο προ-επεξεργαστής θα απομονώσει τις καταστάσεις από μια ενσωματωμένη SQL εφαρμογή και ακολούθως θα μετατρέψει τα επερωτήματα σε δέντρο επερωτημάτων (query tree) και να τα μετασχηματίσει σε βελτιστοποιημένα δέντρα. Στην συνέχεια ο προ-επεξεργαστης αναγνωρίζει τα «υπο-δέντρα» που μπορούν να επιταχυνθούν με FPGA υπολογισμούς. Κάθε τέτοιο υπό-δεντρο θα αντικατασταθεί με κώδικα γλώσσας προγραμματισμού C που θα ελέγχεται από τον FPGA συνεπεξεργαστή και το υπόλοιπο κομμάτι του δέντρου θα παραμείνει σε κώδικα SQL. Τέλος, μετά την μεταγλώττιση ο τελικός κώδικας θα μπορεί να εκτελεστεί. Το FPGA θα αναλάβει την εκτέλεση των υπο-δέντρων με κώδικα C (αρκετές από τις οποίες θα είναι παράλληλες λειτουργιές για τα υπό-δεντρα που δεσμεύτηκαν στην FPGA πλακέτα) και το SQL κομμάτι θα εκτελεστεί από back-end μηχανή της βάσης. Σημαντικότατο πλεονέκτημα των FPGAs είναι ότι μπορούν να επιταχυνθούν λειτουργίες με την

χρήση του για τις οποίες οι ίδιες οι βάσεις δεδομένων και ότι τις αποτελεί δεν μπορούν να κάνουν απολύτως τίποτα για να το επιτύχουν αυτό.

Επιτάχυνση με την βοήθεια των FPGAs έχει επιτευχθεί επίσης στην εκτέλεση επερωτημάτων αναλυτικών λειτουργιών(analytics query operations) και συστημάτων υποστήριξης λήψης αποφάσεων [7].Μάλιστα σε αρκετά ικανοποιητικό βαθμό επιτάχυνσης (6,2x) καθώς επίσης και 94% εξοικονόμηση της χρήσης του CPU εν σύγκριση με το κόστος της βασικής εκτέλεσης. Αυτό επιτυγχάνετε μέσω της προσπάθειας εκτέλεσης της μεταφοράς των δεδομένων με το μέγιστο δυνατό εύρος ζώνης του PCIe (επετεύχθη 2,7 GB/s data rate) μέσω τριών τεχνικών που χρησιμοποιήθηκαν που με την πρώτη καθορίστηκαν οι DMA διευθύνσεις για όλα τα δεδομένα της εργασίας, με την δεύτερη μια σειρά από ουρές μεταξύ του Service στρώματος και της λογικής της εφαρμογής εφαρμόστηκαν για να επιτρέπουν στα DMA αιτήματα να μπαίνουν στην ουρά κάτι που μεγιστοποιεί την χρήση των DMA μηχανών και τρίτη ήταν τα πολλαπλά buffers εισόδου εξόδου που επιτρέπουν το διπλό buffering κάτι το οποίο σχεδόν εξαφανίζει την καθυστέρηση μεταφοράς μεταξύ host και FPGA. Επίσης ανατέθηκαν δύο από τις πιο εντατικές συναρτήσεις του CPU στο FPGA (αποσυμπίεση και υποστήριξη εκτίμησης)

2.2 Ανάλυση σχετικών εργασιών για MaxCompiler

Απαραίτητο θεωρήθηκε επιπλέον η ανεύρεση και ανάλυση σχετικών εργασιών και δημοσιεύσεων γύρω από το εργαλείο της εργασίας μας τον MaxCompiler. Μελετήθηκαν αρχικά κάποιες γενικότερες πληροφορίες από την ιστοσελίδα της εταιρείας που παράγει το εργαλείο [8] καθώς επίσης και κάποια άρθρα που αναρτηθήκαν επίσης για περιγραφή του εργαλείου και των λειτουργιών του και πήραμε μια πρώτη ιδέα για το πώς μπορεί να χρησιμοποιηθεί κα τι μπορεί να μας προσφέρει[9][10]. Επίσης για το εργαλείο ετοιμάστηκαν κάποια φροντιστήρια αρκετά επιμορφωτικά τα οποία επίσης μελετήθηκαν για να προσαρμοστούμε με την χρήση του MaxCompiler μέσω διαφόρων παραδειγμάτων και εργασιών [11] [12] [13].

Δεν υπάρχει ακόμη πληθώρα δουλειών και αναρτήσεων σχετικών με τον MaxCompiler αλλά στην αναζήτηση που έκανα βρήκα ικανοποιητικό αριθμό ικανό για να εμπλουτίσω τις γνώσεις μου.

Έχει μελετηθεί άρθρο [14]για την πρόβλεψη του καιρού στο οποίο παρατηρήθηκε 50 φορές συντομότερος χρόνος από τον χρόνο εκτέλεσης σε ένα Intel i3 based PC 3,09 GHz και 4GB RAM.

Στην έρευνα αυτή παρουσιάζεται η αποδοτική προσέγγιση για λύση ενός προβλήματος για μεγάλο όγκο δεδομένων με την χρήση FPGA επιταχυντή βασισμένο στην dataflow αρχιτεκτονική της Maxeler Technologies. Η επιτάχυνση επιτέυχθηκέ με Max2 γενιάς κάρτα και σημαντικοί πόροι για την ακρίβεια ήταν τα flip-flops και εικάζει ότι με τις κάρτες νεότερων γενεών οι οποίες θα έχουν και περισσότερους διαθέσιμους πόρους θα μπορεί να επιτευχθέι μεγαλύτερη ακρίβεια.

Σε κάποια άλλη σχετική δουλειά [15] επιταχύνθηκε η μέθοδος Lattice-Boltzman και μειώθηκε επίσης η κατανάλωση ενέργειας που απαιτείτο από τον αλγόριθμο. Η Lattice-Boltzman μέθοδος είναι μια διακριτή μέθοδος που προσομοιώνει την ροή υγρών, και η μέθοδος αυτή παρουσιάζει τα υγρά σαν πλασματικά σωματίδια και μελετώντας την δυναμική των σωματιδίων αυτών μοντελοποιεί την ροή υγρού σε μακροσκοπικό επίπεδο. Τα κλειδιά της επιτυχής αυτής βελτίωσης ήταν ο μεγάλος όγκος δεδομένων και η επαναχρησιμοποίηση τους, η ανοχή στην καθυστέρηση για τον συγκεκριμένο αλγόριθμο, το πλεονέκτημα WORM (write once, run many) που μας δίνει ο προγραμματισμός της DataFlow αρχιτεκτονικής καθώς επίσης ότι ο περισσότερος χρόνος εκτέλεσης της εφαρμογής σπαταλείται σε βρόγχους των οποίων η επιτάχυνση με DataFlow αρχιτεκτονική είναι ευκολότερη και φυσικά στην ορθή υλοποίηση από την πλευρά του προγραμματιστή. Περιγράφεται ότι η βασική διάφορα των συστημάτων ροής ελέγχου και των συστημάτων ροής δεδομένων πηγάζει από το ότι τα προγράμματα ροής δεδομένων μεταγλωττίζονται σε χαμηλότερο επίπεδο από αυτό του επεξεργαστή σε επίπεδο υλικού με λογικές πύλες, τρανζίστορς και καλώδια και το γεγονός αυτό επιφέρει επιτάχυνση, μείωση κατανάλωση ενέργειας, μείωση του κόστους και του μεγέθους. Επίσης παραδείγματα WORM εφαρμογών βρίσκουμε να χρησιμοποιούνται στην γεωφυσική με επιταχύνσεις μεταξύ 20x-40x, σε τραπεζικές εργασίες με επιταχύνσεις 200x εως και 1000x, σε εφαρμογές παρακολούθησης και ελέγχου και εξόρυξης δεδομένων (π.χ. Η Google κάνει εξόρυξη δεδομένων από τα κοινωνικά δίκτυα). Γίνεται και σε αυτή την εργασία χρήση Max2 γενιάς Maxeler card. Για να επιτευχθεί η ακρίβεια που επιθυμεί τροποποιήθηκε δεδομένα σε μορφή dfeFloat(12,52) αντί για την κλασσική για double τιμές dfeFloat(11,53). Συγκρίθηκε η Maxeler υλοποίηση με την εκτέλεση του αλγορίθμου σε ένα Intel i5 650 3,2 GHz και παρατηρήθηκαν για οποιοδήποτε αριθμό επαναλήψεων (από 1 εώς 10000) καλύτερη επίδοση της Maxeler υλοποίησης.

Τέλος μελετήθηκε μια διατριβή μεταπτυχιακού που ασχολήθηκε με την χρήση FPGA επιταχυντή σε βάσεις δεδομένων με την βοήθεια του εργαλείου της Maxeler και Max3 γενιάς Maxeler card [16]. Επεξηγεί αναλυτικά τα στοιχεία της υλοποίησης του και της μεθοδολογίας που ακολουθείται και παρουσιάζει αναλυτικά τα αποτελέσματα του.

Κεφάλαιο 3 Hardware accelerators

3.1 Hardware Acceleration

Στη Πληροφορική υπάρχουν και χρησιμοποιούνται πολλές τεχνικές για βελτίωση των επιδόσεων. Μία εξ αυτών είναι και το hardware acceleration (επιτάχυνση υλικού) που είναι η χρήση του υλικού των υπολογιστών για την εκτέλεση λειτουργιών οι οποίες να μπορούν να εκτελεστούν γρηγορότερα απ ’ότι στο λογισμικό (software).

Hardware accelerator (επιταχυντής υλικού) ονομάζεται το κομμάτι του υλικού (όχι ο επεξεργαστής) που επιτυγχάνει την επιτάχυνση σε μια διαφορετική μονάδα από αυτή του επεξεργαστή (CPU).

Πολλοί hardware accelerators (επιταχυντές υλικού) χτίζονται πάνω σε FPGA (Field Programmable Gate Array) chips. Πάνω σ’ αυτό τον τύπο θα ασχοληθεί και η δική μου εργασία.

3.2 Hardware Acceleration Methods

Ο πρωταρχικός στόχος της επιτάχυνσης υλικού (hardware acceleration) είναι να αυξηθεί η ταχύτητα με την οποία τα δεδομένα μπορούν να υποβληθούν σε επεξεργασία με τη χρήση προσαρμοσμένου υλικού ειδικά σχεδιασμένου να εκτελεί

συγκεκριμένες λειτουργίες (ρουτίνες). Με αυτόν τον τρόπο το λογισμικό (software) μπορεί να επιταχυνθεί με δύο τρόπους.

Το πρώτο πλεονέκτημα είναι ότι ο επεξεργαστής (CPU) είναι σε θέση να επεξεργάζεται και άλλα δεδομένα καθώς οι αναγκαίοι υπολογισμοί για την επιταχυνόμενη ρουτίνα (διαδικασία) θα εκφορτωθούν για εκτέλεση στον συνεπεξεργαστή (π.χ. FPGA) και αυτό κάνει τον επεξεργαστή να φαίνεται ουσιαστικά "ελεύθερος" κατά την διάρκεια του υπολογισμού. Οι μόνες στιγμές που ο επεξεργαστής συμμετέχει στον υπολογισμό είναι κατά τη διάρκεια της σύστασης (set up) του συνεπεξεργαστή για να ξεκινήσει τους υπολογισμούς και κατά τη διάρκεια που λαμβάνει τα αποτελέσματα απ' αυτόν. Εάν ο χρόνος που απαιτείται για την επικοινωνία με τον συνεπεξεργαστή είναι μικρότερος (λιγότερο δαπανηρή διαδικασία) από τον χρόνο που απαιτεί η πραγματική εκτέλεση στον επεξεργαστή τότε συνεπάγεται ότι μπορεί να επιτευχθεί επιτάχυνση (speedup).

Το δεύτερο πιθανό πλεονέκτημα (κέρδος) είναι όταν ο επιταχυντής υλικού έχει δομηθεί (διαμορφωθεί) με τέτοιο τρόπο ώστε να εκτελέσει τους υπολογισμούς σε συντομότερο χρόνο απ' ότι το λογισμικό (software – CPU). Σε αυτή την περίπτωση πρέπει ο χρόνος εκτέλεσης στον επιταχυντή και ο χρόνος που απαιτείται για την επικοινωνία μαζί να είναι λιγότερος από το χρόνο που απαιτείται για εκτέλεση η υλοποίηση στο λογισμικό (software implementation) ο ίδιος αλγόριθμος. Εάν ισχύουν τα πιο πάνω τότε θα παρουσιαστούν βελτιωμένες επιδόσεις ανεξαρτήτως αν ο επεξεργαστής επεξεργάζεται δεδομένα παράλληλα με τον συνεπεξεργαστή ή όχι.

Ιδεατή είναι η περίπτωση να πληρούνται και οι δύο προϋποθέσεις ούτως ώστε ο επεξεργαστής να μπορεί να επεξεργάζεται δεδομένα παράλληλα (ταυτόχρονα) με τον συνεπεξεργαστή (hardware) και η υλοποίηση στο υλικό να είναι γρηγορότερη απ' αυτή στο λογισμικό. Παρόλα αυτά δεν είναι απαραίτητο η υλοποίηση στο hardware να είναι γρηγορότερη απ' του software αρκεί το overhead για να ξεκινήσουν οι υπολογισμοί να είναι λιγότερο δαπανηρό απ' ότι η ίδια η εκτέλεση στον επεξεργαστή και ότι κατά την διάρκεια των υπολογισμών στον συνεπεξεργαστή να μπορούν να ανατεθούν ανεξάρτητες (από τα αποτελέσματα του επιταχυντή) εργασίες στον επεξεργαστή για να επωφεληθούμε το "ελεύθερο" του CPU.[17]

3.2.1 Μεταφορά Δεδομένων στη Επιτάχυνση Υλικού

Ένα από τα κύρια bottlenecks στη επιτάχυνση υλικού είναι η μεταφορά δεδομένων (Data Transfer in Hardware Accelaration). Για να εξασφαλίσουμε μια γρήγορη μεταφορά δεδομένων ο επιταχυντής υλικού πρέπει να είναι στενά συνδεδεμένος (closely coupled) με τον κύριο επεξεργαστή. Για παράδειγμα αν ένας επιταχυντής

είναι στο ίδιο die με τον επεξεργαστή τότε θα είναι ικανός να μεταφέρει τα δεδομένα μεταξύ τους (επεξεργαστή και επιταχυντή) γρηγορότερα απ' ότι εάν είναι συνδεδεμένος με USB.

Ανάμεσα στις διεπαφές για μεταφορά δεδομένων από και προς στους επιταχυντές υλικού συμπεριλαμβάνονται το USB, το Ethernet, η σειριακή θύρα και το PCI. Μπορεί να χρησιμοποιηθεί οποιαδήποτε τέτοια διεπαφή με μόνη προϋπόθεση να μπορεί να ταιριάξει πάνω στο FPGA. Στην επιλογή της διεπαφής αυτής θα πρέπει να ληφθεί υπόψη σαν κύρια παράμετρος η καθυστέρηση (latency) στην επικοινωνία αυτής καθώς και το εύρος ζώνης (bandwidth) της. Το εύρος ζώνης μιας διεπαφής εννοούμε τον ρυθμό μεταφοράς δεδομένων (rate of data transfer) της. Η σημαντικότητα των δύο αυτών παραμέτρων σημαίνεται στο γεγονός ότι όσο περισσότερο καθυστερήσουν τα δεδομένα να μεταφερθούν στο FPGA τόσο περισσότερο θα καθυστερήσει να ξεκινήσει και η επεξεργασία τους καθώς και η επιστροφή των αποτελεσμάτων.

3.2.2 Άμεση Πρόσβαση Μνήμης (Direct Memory Access)

Η άμεση πρόσβαση της μνήμης (DMA) επιτρέπει η πρόσβαση της μνήμης να γίνετε ανεξάρτητα από την κεντρική μονάδα επεξεργασίας (CPU), και αυτό μας επιτρέπει υψηλότερη ταχύτητα μεταφοράς των blocks δεδομένων.

Με την χρήση του DMA ο επεξεργαστή απλώς θα αρχικοποιήσει την διαδικασία (DMA Transfer) για μεταφορά και ακολούθως ο DMA controller θα χειριστεί τη μεταφορά των δεδομένων ενόσω ταυτόχρονα ο επεξεργαστή θα ασχολείται με την εκτέλεση κάποιας άλλης λειτουργίας. Η διαδικασία αυτή προσφέρει γρηγορότερη μεταφορά των δεδομένων απ' ότι να αναθέσουμε στον επεξεργαστή να αντιγράψει αυτός κάθε κομμάτι δεδομένων από μια πηγή σε ένα προορισμό (source-to-destination).

3.2.3 Κοινές Στρατηγικές Επιτάχυνσης

Η κύρια στρατηγική για την επιτυχή επιτάχυνση του υλικού είναι η μετατροπή των χρονικών υπολογισμών σε χωρικούς υπολογισμούς. Για να επιτευχθεί το πιο πάνω θα πρέπει να επεκτείνουμε τον αλγόριθμο σε πολλαπλούς παράλληλους υπολογισμούς. Για τον υπολογισμών των τιμών συνήθως χρησιμοποιούμε πίνακες (συστοιχίες)(arrays). Για παράδειγμα μια συστολική συστοιχία αποτελείται από πολυάριθμες μονάδες επεξεργασίας δεδομένων συνδεδεμένες με ένα πλέγμα συρμάτων. Τα δεδομένα μεταδίδονται στη μια ή και στις δύο πλευρές της συστοιχίας και αποθηκεύονται σε μια επεξεργασία δεδομένων όπου εκτελείται κάποια λειτουργία πάνω στις τιμές. Αφού ολοκληρωθεί η πιο πάνω λειτουργία/ές τα

αποτελέσματα μεταδίδονται πίσω με παρόμοιο τρόπο. Τέτοιου είδους συστοιχίες μας προσφέρουν αρκετά υψηλό παραλληλισμό στους υπολογισμούς (highly parallel computations) αφού κάθε μονάδα επεξεργασίας δεδομένων μπορεί να λειτουργεί παράλληλα και ταυτόχρονα με όλες τις υπόλοιπες (independently & concurrently) και συχνότερη τους γίνεται στους πολλαπλασιασμούς.

Η διαδικασία παραλληλοποίησης ενός αλγορίθμου κατά πάσα πιθανότητα σχετίζεται και εξαρτάται απ' αυτόν, γι' αυτό η γενική προσέγγιση είναι να μελετήσουμε και να καθορίσουμε ποια κομμάτια του αλγόριθμου μπορούν να εκτελεστούν παράλληλα. Μια ειδική υπολογιστική μονάδα δημιουργείται για κάθε κομμάτι που μπορεί να τρέξει παράλληλα και αφού έχουν ολοκληρωθεί όλα τα επιμέρους τμήματα τα αποτελέσματα αυτών συνδυάζονται για να παράξουν το τελικό αποτέλεσμα. Η πιο πάνω στρατηγική είναι ευρέως γνωστή ως "διαίρει και βασίλευε" ("divide and conquer") και αυτό που κάνει είναι να διαιρεί μια λειτουργία (διαδικασία) σε μικρότερες και ακολούθως να συνδυάζει τα αποτελέσματα αυτών ώστε να εξάγει το τελικό αποτέλεσμα της αρχικής λειτουργίας.

Μια κοινή μέθοδος για να δημιουργήσουμε ένα επιταχυντή υλικού είναι να δημιουργήσαμε ένα "σύστημα σ' ένα προγραμματιζόμενο chip" ("system on a programmable chip "ή (SOPC)). Ένα SOPC αποτελείται από ένα soft-processor ο οποίος φορτώνετε στο επαναρυθμιζόμενο FPGA. Ακολούθως θα του προσθέσουμε τις προσαρμοσμένες εντολές και αυτό επιτρέπει την εύκολη ενσωμάτωση του υλικού με ένα επεξεργαστή και προσδίδει υψηλό βαθμό προσαρμογής(high-degree of customization).

3.2.4 Hardware-Software Co-design

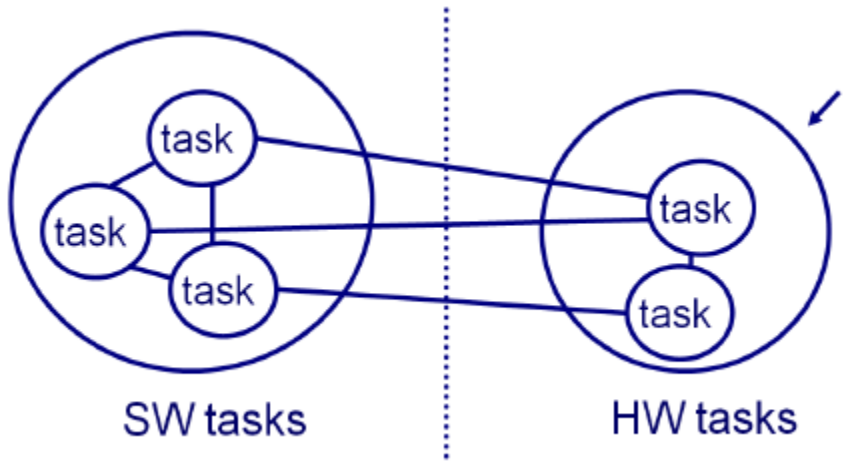
Για να επιτύχουμε το στόχο του διαχωρισμού υλικού-λοσγιμικού θα πρέπει να καθορίσουμε και να αποφασίσουμε ποια κομμάτια του κώδικα είναι καταλληλότερα για το υλικό και ποια κομμάτια του κώδικα είναι καταλληλότερα για το λογισμικό. Στην επιλογή μας αυτή θα πρέπει να ληφθούν υπόψη παράγοντες όπως το κόστος, η αποδοτικότητα και η ταχύτητα.

Όταν θέλουμε θα επιταχύνουμε μια ρουτίνα (κομμάτι) ενός προγράμματος επιθυμούμε να ελαχιστοποιήσουμε το bus traffic της διαδικασίας σε υλικό και λογισμικό καθώς και να της μεγιστοποιήσουμε τον συγχρονισμό υλικού-λογισμικού. Με αυτό τον τρόπο η χρόνος εκτέλεσης της ρουτίνας αυτής θα ελαχιστοποιηθεί.

Για να αποφασίσουμε όσο το δυνατό καλύτερα πώς θα χωρίσουμε τις λειτουργίες σε υλικό και λογισμικό θα πρέπει να κάνουμε profiling την αρχική υλοποίηση του

λογισμικού. Με το να χρησιμοποιήσουμε profiling θα μπορούμε να αναγνωρίσουμε ποια κομμάτια του λογισμικού μπορούν να υλοποιηθούν στο υλικό και να καθορίσουμε ποια απ' αυτά μπορούν να μας προσφέρουν τα περισσότερα οφέλη.

Το profiling μας προσφέρει μια καθολική εικόνα για το πώς μπορεί το έργο να διασπαστεί σε διάφορες δραστηριότητες (tasks). Οι δραστηριότητες αυτές μπορούν να αναπαρασταθούν σε ένα γράφημα σαν κόμβοι σαν κόμβοι όπου οι ακμές τους αντιπροσωπεύουν την επικοινωνία μεταξύ τους. [18][19][20]



Σχήμα 3.1: Hardware-Software Partitioning Graph[17]

Σε ένα ολοκληρωμένο τέτοιο γράφημα οι ακμές θα έχουν κάποια βάρη, η τιμή των οποίων αντιπροσωπεύει το κόστος επικοινωνίας (communication overhead) μεταξύ των κόμβων που συνδέει η ακμή αυτή.

3.2.5 Διεπαφή PCI (PCI Interface)

Ο δίαυλος PCI (Peripheral Component Interconnect bus) παρέχει μια μέθοδο για τη μεταφορά δεδομένων μεταξύ της πλευράς του λογισμικού της εφαρμογής και του επιταχυντή υλικού.

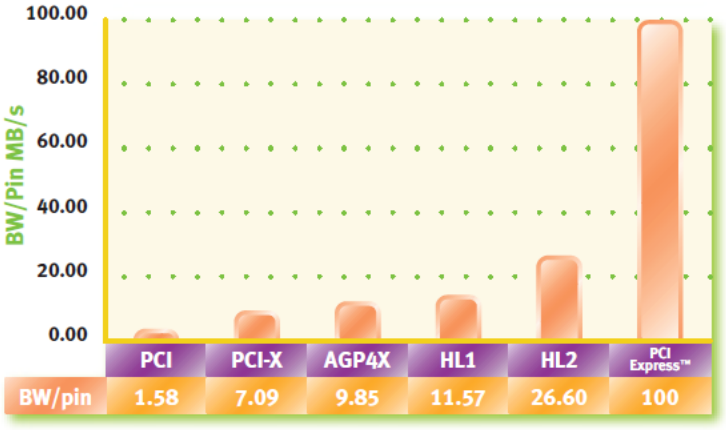
3.2.5.1 PCI Express

Το PCI Express[21] είναι μια υψηλής απόδοσης εξέλιξη του ευρέως γνωστού PCI bus και δημιουργήθηκε μέσα από την ανάγκη για υψηλότερο εύρος ζώνης εισόδου/εξόδου (I/O bandwidth) σε σχέση με τα ήδη υπάρχοντα PCI. Αποτελείτε από δύο χαμηλής τάσης, διαφορετικών κατευθύνσεων ζεύγη σημάτων (1 μετάδοσης και 1 παραλαβής). Το ενσωματωμένο του ρολόι χρησιμοποιεί 8b/10b κωδικοποίηση για την επίτευξη πολύ υψηλών ρυθμών δεδομένων. Η αρχική του συχνότητα είναι 2,5 Gb/second/direction δηλαδή 2,5Gb ανά δευτερόλεπτο ανά κατεύθυνση και αυτό αναμένεται ότι με τα πλεονεκτήματα της τεχνολογία σιλικόνης θα αυξηθεί στα

10Gb/s/direction (δηλαδή το πρακτικό μέγιστο για σήματα στο χαλκό). Επίσης παρέχει 200MB/s κανάλι επικοινωνίας που είναι περίπου το διπλάσιο του ρυθμού δεδομένων ενός κλασικού PCI.

Το τι μας προσφέρει το PCI Express είναι μεγαλύτερη μνήμη σε σχέση με τους προκάτοχους του, χαμηλό κόστος, είναι συμβατό με όλα τα λειτουργικά συστήματα. Επίσης επεκτάσιμη επίδοση μέσω συχνότητας, υψηλό εύρος ζώνης ανά Pin, χαμηλό overhead και χαμηλή καθυστέρηση. Υποστηρίζει πολλαπλούς τύπους πλατφορμών σύνδεσης (chip-to-chip, board-to-board). Στα προηγμένα χαρακτηριστικά του συμπεριλαμβάνονται η κατανόηση διαφορετικών τύπων δεδομένων, η διαχείριση ενέργειας, η ποιότητα υπηρεσίας (Quality of Service) καθώς και ακεραιότητα δεδομένων και διαχείριση σφαλμάτων.

Το PCI Express μας προσφέρει μια γενικού σκοπού διασύνδεση εισόδου/εξόδου (I/O Interconnection) για ένα ευρύ φάσμα υπολογιστικών και επικοινωνιακών πλατφόρμων στο παρών αλλά και στο μέλλον.



PCI @ 32b x 33MHz and 84 pins, PCI-X @ 64b x 133MHz and 150 pins, AGP4X @ 32b x 4x66MHz and 108 pins, Intel® Hub Architecture 1 @ 8b x 4x66MHz and 23 pins; Intel Hub Architecture 2 @ 16b x 8x66MHz and 40 pins; PCI Express™ @ 8b/direction x 2.5Gb/s/direction and 40 pins.

Σχήμα 3.2:Comparing PCI Express™'s bandwidth per pin with other buses.[21]

3.3 FPGA (Field Programmable Gate Array)

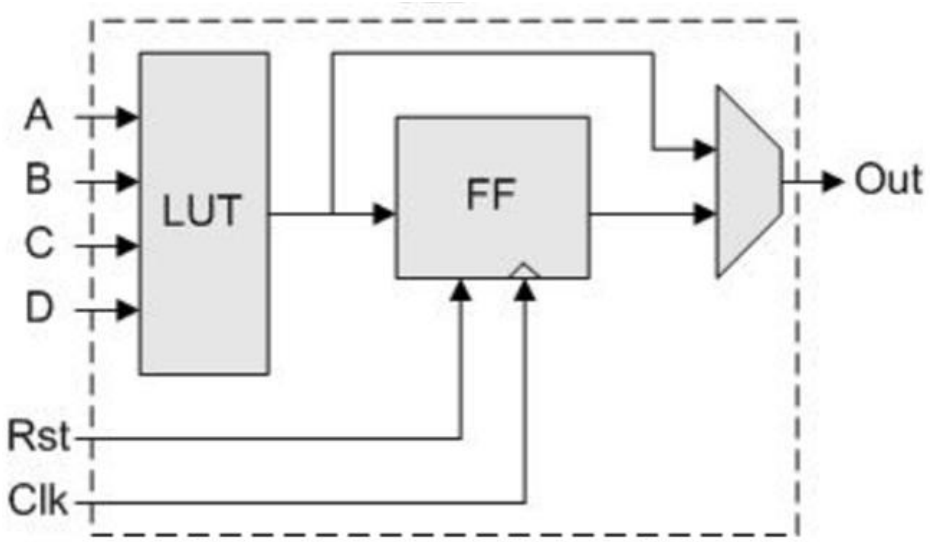
FPGA [5](Field Programmable Gate Array) είναι ένα ολοκληρωμένο κύκλωμα (συνδεδεμένων λογικών πυλών) σχεδιασμένο για να ρυθμιστεί από ένα σχεδιαστή και η διαμόρφωση αυτή καθορίζεται χρησιμοποιώντας μια γλώσσα περιγραφή υλικού (Hardware Description Languages,HDL). Επίσης μπορεί να υλοποιήσει οποιαδήποτε

λογική λειτουργία που μπορεί ένα ASIC[22] (Application-specific integrated circuit) εκτελεί. Το πλεονέκτημα των FPGAs έναντι των ASICs είναι το χαμηλότερο τους κόστος ως επίσης και δυνατότητα επαναπρογραμματισμού τους που μας προσφέρει μεγαλύτερη ευελιξία και την ευκαιρία για ευκολότερη και γρηγορότερη ανάπτυξη ενός κυκλώματος.

Τα FPGAs περιέχουν λογικά προγραμματιζόμενα μέρη που ονομάζονται “logic blocks” και μια ιεραρχία από επαναρυθμιζώμενες διασυνδέσεις που επιτρέπουν στα blocks να είναι ενωμένα μεταξύ τους κάπως σαν λογικές πύλες που μπορούν να διασυνδεθούν με διαφορετικούς σχηματισμούς. Τα logic blocks μπορούν να ρυθμιστούν ούτως ώστε να εκτελούν συνδυαστικές λειτουργίες ή απλές λογικές πύλες όπως AND και XOR. Στα περισσότερα FPGAs τα logic blocks επιπλέον περιέχουν στοιχεία μνήμης τα οποία μπορεί να είναι απλά flip-flops (registers) ή περισσότερο πλήρη μπλοκ μνήμης(BRAM). Τα flip-flops (ή καταχωρητές) είναι ένας τύπος μνήμης (on-chip memory) που μπορεί να αποθηκεύσει ένα bit. Για αποθήκευση περισσότερων bits θα πρέπει να συνδυαστούν τέτοιες εξόδου ενός bit και μπορούν να αποθηκευτούν στην block RAM (BRAM on-chip memory) που είναι επίσης μνήμη πάνω στο FPGA. Αρκετά συχνά τα FPGA chips είναι εξοπλισμένα με κάποια επιπλέον τυπική λειτουργικότητα όπως για παράδειγμα να έχει τους δικούς του πλήρεις πυρήνες επεξεργαστών.

3.3.1 Logic blocks in FPGA

Το πιο βασικό στοιχείο ενός FPGA είναι τα logic blocks. Ένα logic block αποτελείται από ένα lookup table (LUT) συνδεδεμένο με ένα πολυπλέκτη (multiplexer) ο οποίος επιλέγει μεταξύ της εξόδου του LUT και ενός Flip-Flop (FF) που είναι συνδεδεμένο με την έξοδο του LUT με σκοπό να επιτρέπει την αποθήκευση τιμών σαν καταχωρητές (registers). Με αυτό το σχεδιασμό μπορούν να υλοποιηθούν αυθαίρετες συναρτήσεις οι οποίες περιορίζονται μόνο από το μέγεθος των lookup table στο FPGA. Τα logic blocks μπορούν να ρυθμιστούν επίσης ούτως ώστε να εκτελούν συνδυαστικές λειτουργίες ή απλές λογικές πύλες όπως παραδείγματος χάρη AND και XOR. Τα logic blocks αυτά συνδέονται μεταξύ τους για να παρέχουν lookup tables οποιουδήποτε μεγέθους ή διαμόρφωσης απαιτείται. Τα look-up tables (LUTs) μπορούν να προγραμματιστούν ώστε να εφαρμόσουν οποιαδήποτε λογική συνάρτηση με N εισόδους (όπως π.χ. στο Σχήμα 3.3: Logic Block in FPGA N=4 (συνήθως μεταξύ 4 εώς 6) και εισόδοι A,B,C,D).



Σχήμα 3.3: Logic Block in FPGA[23]

Τα logic blocks είναι συνδεδεμένα μεταξύ τους σε τοπικές ομάδες οι οποίες ονομάζονται logic block clusters. Μέσα σε ένα logic block cluster τα logic blocks που το αποτελούν είναι πλήρη συνδεδεμένα το ένα με το άλλο μεταξύ τους.

3.3.2 Προγραμματισμός FPGA

Ο κώδικας με τον οποίο προγραμματίζεται το FPGA γράφεται σε γλώσσες περιγραφής υλικού (Hardware Description Languages, HDL) (VHDL, AHDL, Verilog). Παρόλα αυτά στη δική μου εργασία θα χρησιμοποιήσω μια νέα τεχνολογία μεταγλωττιστή από την Maxeler τον MaxCompiler ο οποίος επιτρέπει να προγραμματίζω και να αναδιαμορφώνω το FPGA μέσω μιας γλώσσας προγραμματισμού υψηλού επιπέδου παρόμοιας σύνταξης με την JAVA. Έτσι θα μπορώ να επωφεληθώ τα πλεονεκτήματα του data flow computing (ροής δεδομένων) χωρίς να αντιμετωπίσω την πολύπλοκη και χρονοβόρα γλώσσα περιγραφής υλικού (VHDL).

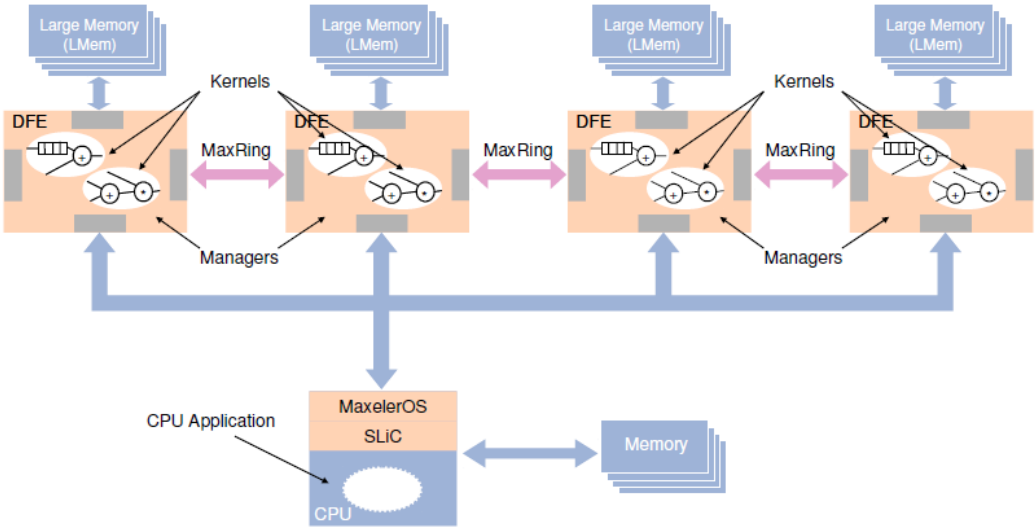
3.3.3 Πλεονεκτήματα υλοποίησης σε FPGA

Σε αντίθεση με το λογισμικό, όταν ένας αλγόριθμος υλοποιείται πάνω σε FPGA η διαδικασία μπορεί να παραλληλοποιηθεί σε πολλούς ανεξάρτητους υπολογισμούς που μπορούν να εκτελούνται ταυτόχρονα. Η ικανότητα των επιταχυντών υλικού να παραλληλίζουν με τέτοιο τρόπο ένα αλγόριθμο είναι που τους κάνει να έχουν τόσο δελεαστική προοπτική. Τα FPGAs ήδη χρησιμοποιούνται με επιτυχία στην επεξεργασία σήματος και εικόνας αφού υπερτερούν σε αυτούς τους τομείς κυρίως λόγω της χαμηλής τους καθυστέρησης (latency) τους και του υψηλού βαθμού παραλληλισμού που μπορούν να προσφέρουν.

Κεφάλαιο 4 Maxeler MaxCompiler

4.1 Εισαγωγή στον Maxeler MaxCompiler

Ο Maxeler MaxCompiler[8][9] είναι ένα σύστημα μεταγλωττιστή για επιτάχυνση υλικού χρησιμοποιώντας FPGAs (ή αλλιώς Dataflow Engines - DFEs). Η αρχιτεκτονική ενός Maxeler hardware acceleration συστήματος είναι εξοπλισμένη με ένα ή περισσότερα FPGAs επισυναπτόμενα σε ένα σετ από μνήμες και συνδεδεμένα με την κεντρική μονάδα επεξεργασίας (CPU) μέσω PCI Express καναλιών.



Σχήμα 4.1 Maxeler dataflow system architecture[11]

Σε ένα τέτοιο σύστημα τα FPGAs (DFEs) είναι συνδεδεμένα μεταξύ τους μέσω μίας υψηλού εύρους ζώνης MaxRing διασύνδεσης (high-bandwidth interconnect). Η MaxRing διασύνδεση επιτρέπει στις εφαρμογές να επεκταθούν γραμμικά με πολλαπλά FPGAs στο σύστημα ενώ υποστηρίζει και πλήρη επικάλυψη επικοινωνίας και των υπολογισμών

Ο Maxeler Java Compiler [16] είναι ένας εξειδικευμένος μεταγλωττιστής που έχει την δυνατότητα να μεταγλωττίζει από μια γλώσσα προγραμματισμού υψηλού επιπέδου όπως η Java σε περιγραφή υλικού Maxeler ροής σχεδιασμού (Maxeler design flow). Ο αλγόριθμος που θέλουμε να επιταχύνουμε πρέπει να γραφτεί σε κώδικα MaxJ (ειδική βιβλιοθήκη/επέκταση της Java). Ο MaxCompiler μεταφράζει (μεταγλωττίζει) αυτό τον MaxJ κώδικα σε κώδικα VHDL, και ακολούθως ο VHDL κώδικας αυτός με τη σειρά του θα μεταγλωττιστεί σε περιγραφή υλικού από την συνηθισμένη Xilinx ροή σχεδιασμού (regular Xilinx design flow). Για ευκολία της διαδικασίας συγγραφής του MaxJ κώδικα μας παρέχεται η ελαφρώς τροποποιημένη για τον MaxCompiler έκδοση του Eclipse IDE(Integrated Development Environment, Ενσωματωμένο Περιβάλλον Ανάπτυξης) το MaxIDE.

Για την επιτάχυνση ενός προγράμματος (εφαρμογής, αλγόριθμου) χρειάζεται ο προγραμματισμός 3 κομματιών του προγράμματος και ο MaxCompiler υποστηρίζει τα εργαλεία και για τα τρία όπως φαίνονται πιο κάτω αριστερά το κομμάτι και δεξιά το εργαλείο.

- Οι πυρήνες (Kernels) (Kernel Compiler)
- Διαμόρφωση διαχειριστή (Manager configuration) (Manager Compiler)

- Η ίδια η εφαρμογή (Host application) (MaxelerOS)

Οι πυρήνες και η διαμόρφωση του διαχειριστή γράφονται σε παρόμοια έγκυρη σύνταξη με την Java (έχουν κατάληξη αρχείου .maxj) και μεταγλωττίζονται σε χαμηλού επιπέδου περιγραφή υλικού για προσομοίωση (simulation) του σχεδιασμού ή/και σε πραγματικό υλικό με FPGA διαμόρφωση. Στην πραγματικότητα ο κώδικας αυτός είναι ένας απλοποιημένος (αφού είναι γραμμένος σε υψηλού επιπέδου γλώσσα προγραμματισμού) και σύντομος τρόπος περιγραφής υλικού και μπορεί να περιγράψει με ακρίβεια την συμπεριφορά stream υπολογισμών, δεν είναι όμως περιγραφή επιπέδου μεταφοράς καταχωρητών RTL(Register Transfer Level) όπως η VHDL και η Verilog. Τα συστατικά του υλικού που θα επιταχύνουν τις λειτουργίες (συναρτήσεις) του λογισμικού στο FPGA είναι αυτά που αναφέρουμε εμείς ως πυρήνες και περισσότεροι του ενός μπορούν να τρέχουν παράλληλα στον επιταχυντή και τα streams δεδομένων να μεταφέρονται από, προς και μεταξύ αυτών.

Η MaxJ είναι μια επέκταση (extension) της γλώσσας προγραμματισμού Java για τον MaxCompiler και γράφοντας κώδικα MaxJ μπορούμε να περιγράψουμε τον dataflow επιταχυντή. Τα αρχεία πηγαίου κώδικα της MaxJ έχουν κατάληξη αρχείου .maxj για να διακρίνονται από τα αρχεία πηγαίου κώδικα της JAVA.

Στην εργασία μου επικεντρώνωμαι και έχω ασχοληθεί κυρίως με την έκδοση 2012.2

4.1.1 MaxCompiler Kernels (Πυρήνες)

Ο Maxeler Kernel Compiler είναι ένας μεταγλωττιστής υλικού και αυτού συνεπάγεται ότι περιγράφει τους υπολογισμούς ενός προγράμματος δομικά (χωρική εκτέλεση) αντί να καθορίσει μια σειρά (sequence) από εντολές στον επεξεργαστή (χρονική εκτέλεση). Επίσης ο Kernel Compiler είναι μια αποτελεσματικά βιβλιοθήκη λογισμικού JAVA που μπορεί να δημιουργεί γραφήματα πυρήνων και να τα εκτελεί.

Ένας πυρήνας ουσιαστικά είναι ένα μονής κατεύθυνσης γράφημα ροής δεδομένων αποτελούμενο από κόμβους οι όποιοι είναι κόμβοι που εκτελούν κάποια λειτουργία /υπολογισμό, κόμβοι πολυπλεξίας (multiplexers) ή απλά κόμβοι που περιέχουν μια τιμή. Επίσης υπάρχουν κόμβοι είσοδοι/εξόδου (I/O) δεδομένων από και προς τους διαχειριστές (managers) ή/και από την εφαρμογή. Αναλυτικότερα οι τύποι των κόμβων που μπορούν να περιλαμβάνονται σε ένα γράφημα πυρήνα και το πώς απεικονίζονται:

- Οι κόμβοι υπολογισμού (computation nodes) οι οποίοι εκτελούν αριθμητικές και λογικές λειτουργίες (π.χ. πρόσθεση(+), πολλαπλασιασμό(*), συγκρίσεις

(>,<),λογικές εκφράσεις (& (and), |(or) κτλ) καθώς επίσης και casting για τη μετατροπή του τύπου μιας μεταβλητής (π.χ. από float σε integer ή/και το αντίθετο).

Σχήμα 4.2:Computation Node[9]

- Οι κόμβοι τιμών (Value nodes) οι οποίοι παρέχουν την τιμή κάποιας παραμέτρου η οποία είναι είτε μια σταθερά ή μια τιμή που έχει καθοριστεί από την εφαρμογή (host application) κατά τη διάρκεια της εκτέλεσης της.

Σχήμα 4.3:Value Node[9]

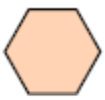
- Οι stream offset κόμβοι οι οποίοι μας επιτρέπουν την πρόσβαση στα στοιχεία των stream δεδομένων σε διαφορετική θέση από την τρέχουσα.

Σχήμα 4.4:Stream offset node[9]

- Οι κόμβοι πολυπλεξίας (multiplexer nodes) για να παίρνουν αποφάσεις.

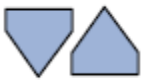
Σχήμα 4.5: Multiplexer Node[9]

- Οι κόμβοι μετρητών (counter nodes) που χρησιμοποιούνται για να κατευθύνουν τον έλεγχο ροής με τη πάροδο του χρόνου, για παράδειγμα κρατούν την παρούσα θέση ενός stream για υπολογισμούς σε ένα συγκεκριμένο πεδίο τιμών χωρίς να ξεφεύγει από τα όρια..



Σχήμα 4.6: Counter Nodes[9]

- Οι κόμβοι εισόδου/εξόδου (I/O nodes) συνδέουν τον πυρήνα με τον διαχειριστή και μας βοηθούν να μεταδίδουμε δεδομένα μέσα και έξω από το πυρήνα.

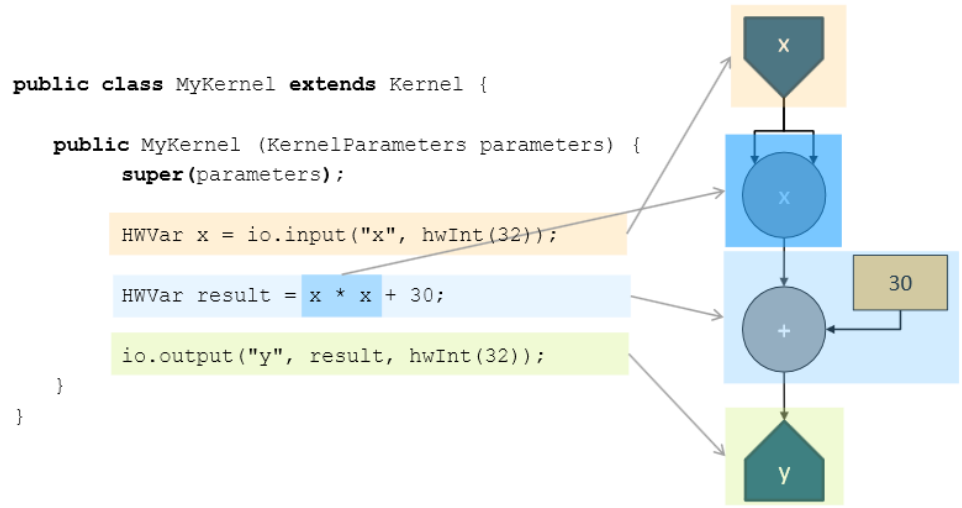


Σχήμα 4.7: I/O Nodes[9]

Ο κώδικας για τους πυρήνες (kernels) του MaxCompiler όπως έχω προαναφέρει γράφεται σε MaxJ(επέκταση της JAVA με όμοια σύνταξη) και είναι το κομμάτι της υλοποίησης μιας εφαρμογής στον MaxCompiler που θα μεταφράσει το κομμάτι του κώδικα της αρχικής εφαρμογής που επιλέξαμε να μετατρέψουμε. Για παράδειγμα αν επιλέξαμε ένα επαναλαμβανόμενο βρόγχο από την εφαρμογή μας για μετατροπή θα πρέπει η λειτουργίες αυτού του κομματιού δηλαδή η ρουτίνα που επαναλαμβάνεται σε κάθε εκτέλεση να μετατραπούν από την αρχική γλώσσα της εφαρμογής (C,C++,Fortran κ.α.) σε κώδικα για τον MaxCompiler και συγκεκριμένα για τον kernel.

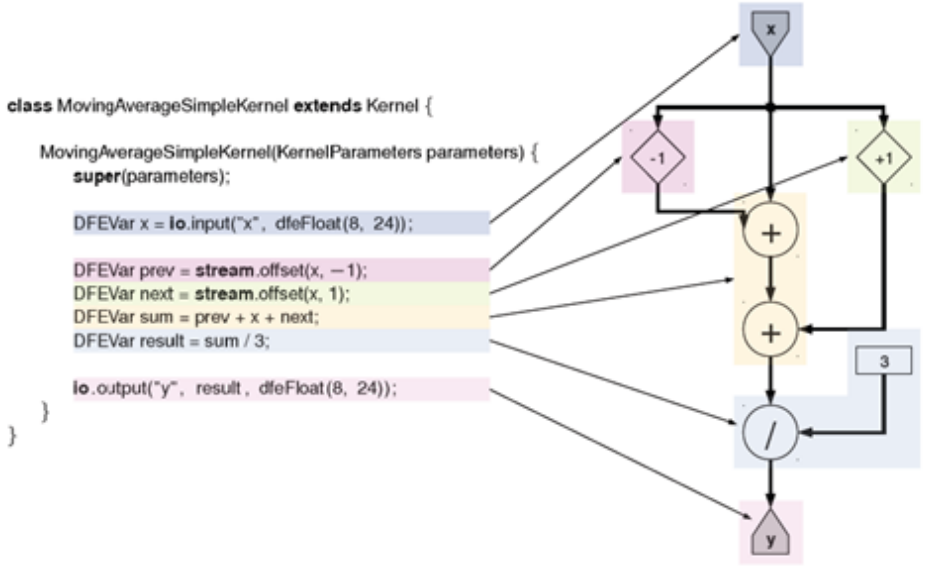
Ακολούθως ο κώδικας αυτός μεταφράζετε σε μια ροή δεδομένων (data flow) αποτελούμενο από κόμβους όπως έχω προαναφέρει πιο πάνω.

Ένα απλό παράδειγμα κώδικα ο οποίος παίρνει σαν είσοδο ένα πίνακα από ακέραιους αριθμούς και κάθε στοιχείο του πίνακα αυτού πολλαπλασιάζεται με τον εαυτό του και του προσθέτετε η σταθερή τιμή 30 απεικονίζετε στο πιο κάτω στο Σχήμα 4.8. Σαν έξοδος επιστρέφετε ένας πίνακας του ίδιου μεγέθους με τα αντίστοιχα αποτελέσματα για κάθε στοιχείο στη ανάλογη θέση. (π.χ. `output[i]=input[i]* input[i] +30`).



Σχήμα 4.8: Κώδικας για ένα απλό Kernel και το αντίστοιχο γράφημα πυρήνα[24]

Ακόμη ένα παράδειγμα ενός κώδικα για Kernel ο οποίος υπολογίζει το moving average (κινητό μέσο όρο) των στοιχείων ενός πίνακα και επιστρέφει ένα πινάκα με τα αποτελέσματα παρουσιάζεται πιο κάτω στο Σχήμα 4.9. Ο πίνακας που εξάγεται σαν αποτέλεσμα του κώδικα έχει σαν στοιχεία του το moving average των στοιχείων του αρχικού πίνακα στην ανάλογη θέση (π.χ. `output[i]=(input[i-1]+input[i]+input[i+1])/3`).

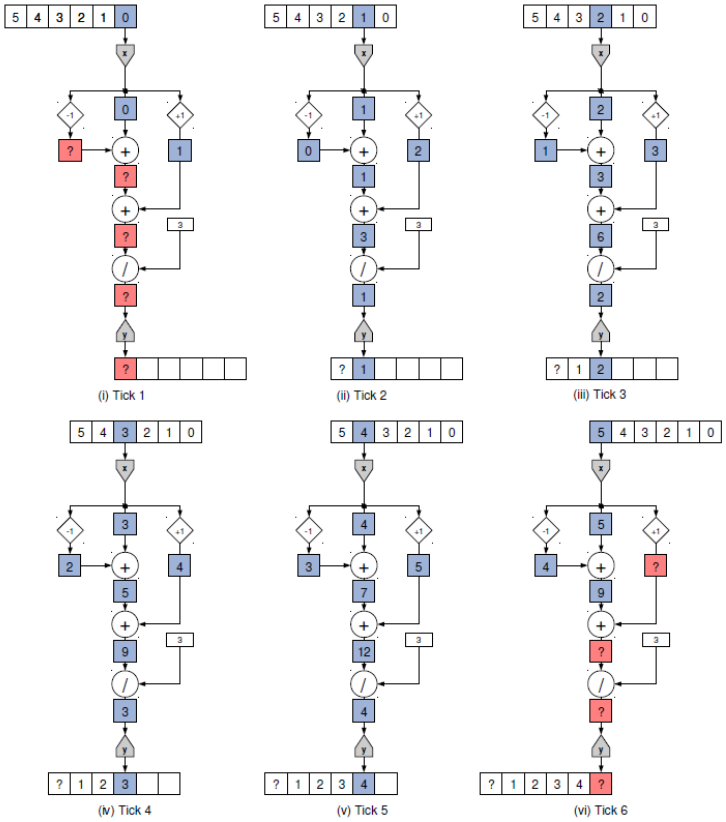


Σχήμα 4.9:Source code for a simple moving average Kernel and the corresponding Kernel graph[11]

Έτσι με την βοήθεια των πιο πάνω απεικονίσεων γίνετε και πιο αντιληπτή η χρήση και λειτουργία των κόμβων που αποτελούν ένα γράφημα πυρήνα.

Τέτοια γραφήματα έχουμε την δυνατότητα να παράξουμε και για τις δικές μας εφαρμογές με την βοήθεια μιας εντολής (`maxRenderGraph`) που μπορούμε να εκτελέσουμε στο terminal της μηχανής.

Παράδειγμα εκτέλεσης του Moving Average Kernel απεικονίζεται στο μαζί με το εισαγόμενο και το εξαγόμενο stream.



Σχήμα 4.10: Παράδειμγα εκτέλεσης Moving Average Kernel [11]

4.1.2 MaxCompiler Managers (Διαχειριστές)

Οι διαχειριστές στον MaxCompiler είναι υπεύθυνοι για την οργάνωση των δεδομένων εισόδου και εξόδου στους πυρήνες. Στην ουσία ένας διαχειριστής (manager) τυλίγει και έχει υπό την επίβλεψη του τους πυρήνες (kernels) των όποιων οργανώνει και καθοδηγεί την ροή δεδομένων μεταξύ τους. Ανάμεσα στις λειτουργίες και αρμοδιότητες που πρέπει να ανατεθούν σε ένα διαχειριστή συμπεριλαμβάνονται τα διαμορφωμένα από το χρήστη εισαγόμενα και εξαγόμενα (I/O) streams από και προς τον επεξεργαστή (CPU), της DMA του CPU, σε άλλες DFEs (DataFlow Engines ή FPGAs) στον κόμβο και στην off-chip Large Memory (εκτός τσίπ μεγάλη μνήμη). Το εκτελέσιμο πρόγραμμα του διαχειριστή (π.χ. manager.class) εκτελείται κατά τη διάρκεια της εκτέλεσης από το MaxelerOS και τρέχει εν μέρει στην κεντρική μονάδα επεξεργασίας (CPU) και εν μέρει στο DFE (FPGA) αυτό καθ’ αυτό.

Οι διαχειριστές παρέχουν μια JAVA διεπαφή προγραμματισμού εφαρμογών (Application programming interface – API) για ρύθμιση της συνδεσιμότητας μεταξύ των πυρήνων (Kernels) και εξωτερικών εισόδων/εξόδων (I/O) καθώς και για έλεγχο της διαδικασίας χτισίματος (μεταγλώττισης). Ο MaxCompiler παρέχει προ-διαμορφωμένους διαχειριστές (Managers) με ποιο συνηθισμένο τον Standard Manager (χρησιμοποιείτε ως πρότυπο), καθώς επίσης και τον Manager Compiler ο οποίος μας δίνει τη δυνατότητα να υλοποιήσουμε τους δικού μας σύνθετους διαχειριστές. Οι σύνθετοι αυτοί διαχειριστές μπορούν να σχεδιαστούν με πολλαπλούς πυρήνες και με πιο πολύπλοκη αλληλεπίδραση μεταξύ των πυρήνων και των πόρων εισόδου/εξόδου (I/O resources).

Σε αντίθεση ο Standard Manager που μας παρέχει ο MaxCompiler είναι ένας γενικού σκοπού διαχειριστής ο οποίος μπορεί α υποστηρίξει μόνο ένα πυρήνα σε κάθε FPGA (DFE), εξωτερικές LMem (Large Memory) διεπαφές, σύνδεση μεταξύ πολλαπλών dataflow engines (FPGAs) καθώς επίσης και συνδέσμους στην κεντρική μονάδα επεξεργασίας (εισόδου/εξόδου). Ο Standard Manager κατασκευάζετε με ένα EngineParameters αντικείμενο. Το EngineParameters αντικείμενο περιέχει πληροφορίες οι οποίες περνιούνται με τη βοήθεια του MaxIDE (4.3),ενώ επίσης μπορούμε να προσθέσουμε επιπλέον παραμέτρους στο αντικείμενο.

```
Manager m = new Manager(new EngineParameters(args));
```

Ο διαχειριστής Standard Manager μπορεί να ενσωματώσει μόνο ένα πυρήνα (Kernel) όπως αναφέραμε και τον θέτει χρησιμοποιώντας την setKernel μέθοδο:

void `setKernel(Kernel k)` όπου k το όνομα του kernel όπως δηλώθηκε

Επίσης υπάρχει η μέθοδος makeKernelParamrters της οποίας η έκδοση στον Standard Manager δεν χρειάζεται να πάρει το όνομα του Kernel σαν String παράμετρο αφού είναι μοναδικός και μπορεί να χρησιμοποιηθεί ως εξής:

public `KernelParameters makeKernelParameters()` εκτός εάν ο ίδιος ο διαχειριστής στέλλει παραμέτρους στον Kernel παραδείγματος χάρη μια σταθερά που δηλώνετε σ' αυτόν τότε θα καλεστεί ως:
`makeKernelParameters(mCONSTANT)` ενώ για τους πιο σύνθετους Custom Managers είναι απαραίτητη παράμετρος αφού μπορούν να διαχειριστούν περισσότερους Kernels: `makeKernelParameters(String KernelName)` όπου π.χ `KernelName="nameKernel"`.

Μπορούμε επίσης να αλλάξουμε την προκαθορισμένη συχνότητα του ρολογιού (που είναι 75MHZ) του FPGA μας καλώντας τη μέθοδο:

```
public void setClockFrequency(int clock_frequency)
```

Η προκαθορισμένη μπορεί να χρησιμοποιηθεί ρητά με την χρήση της σταθεράς `DEFAULT_CLOCK_FREQUENCY`.

Ακόμη υπάρχει η συνάρτηση setIO η οποία μας επιτρέπει να συνδέουμε τα εισαγόμενα και εξαγόμενα streams στον Kernel στους I/O πόρους που ενεργοποιούνται από την διαχειριστή.

```
void setIO(Manager.IOType.iotype) & void setIO(IOLink ..links)
```

Η αρχική έκδοση της συνάρτησης επιτρέπει όλες τις εισόδους και εξόδους να τεθούν μαζί. Για παράδειγμα όλοι οι σύνδεσμοι να είναι στην κεντρική μονάδα επεξεργασίας(CPU):

```
Manager m = new Manager(new EngineParameters(args));
m.setIO(IOType.ALLCPU)
```

Οι σύνδεσμοι για κάθε stream εισόδου εξόδου στον Kernel μπορεί να καθοριστούν μεμονωμένα χρησιμοποιώντας μια λίστα από IOLinks. Ένα IOLink δηλώνεται χρησιμοποιώντας το όνομα του stream και τον αντίστοιχο τύπο του συνδέσμου.

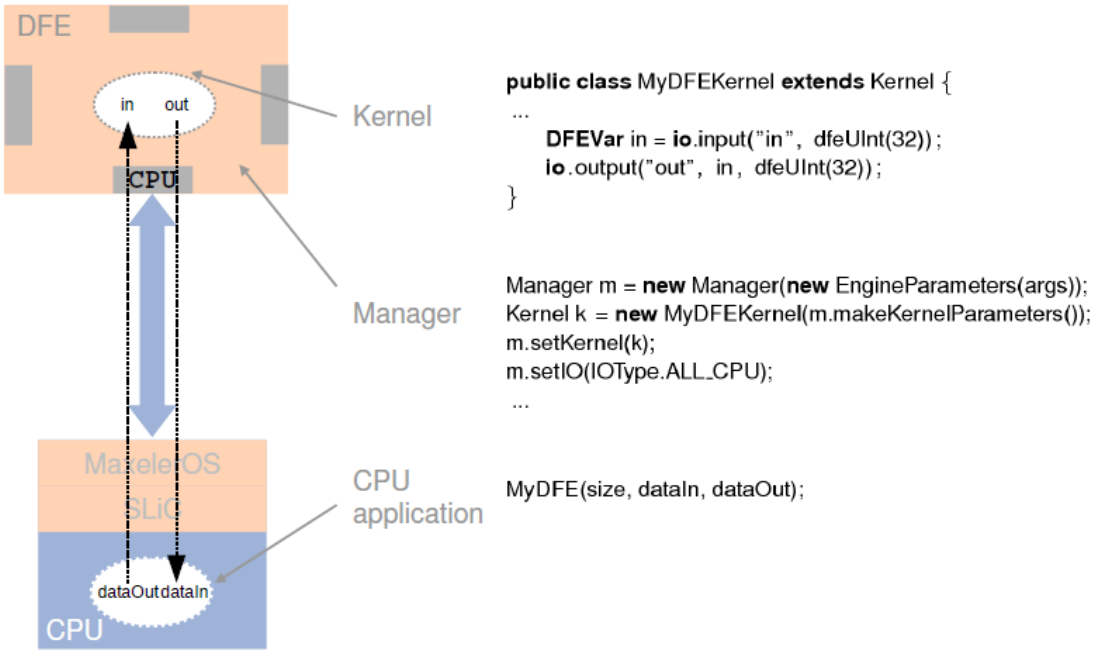
```
IOLink link(String io_name, IOLink.IODestination iotype)
```

Όπου το iotype στην πιο πάνω εντολή μπορεί να είναι το “CPU” δηλαδή το stream συνδέεται με την κεντρική μονάδα επεξεργασίας, “MAXRING_A” ή “MAXRING_B” όπου συνδέει το stream σε ένα από τους δύο διαθέτουν διπλής κατεύθυνσης MaxRing συνδέσμους που διαθέτουν οι MAX3 κάρτες, “LMEM_LINEAR_1D” που συνδέει το stream στην LMEM με ένα παραγωγό γραμμικών διευθύνσεων και τέλος “LMEM_BLOCKED_3D” που συνδέει το stream με ένα παραγωγό τρισδιάστατων 3D διευθύνσεων.

Απλό παράδειγμα χρήσης IOLinks για ένα εισαγόμενο και ένα εξαγόμενο stream που τίθενται και τα δύο από το CPU:

```
mymanager.setIO( link(“inStream”,IODestination.CPU),
                 link(“outStream”,IODestination.CPU));
```

Εναλλακτικά με την χρήση της αρχικής έκδοσης της συνάρτησης setIO το πιο κάτω Σχήμα 4.11 απεικονίζεται το πώς ο διαχειριστής, ο πυρήνας και η κεντρική μονάδα επεξεργασίας αλληλεπιδρούν με ένα εισαγόμενο και ένα εξαγόμενο stream που και τα δύο τίθενται από την κεντρική μονάδα επεξεργασίας.



Σχήμα 4.11: All CPU links on a Standard Manager[11]

Επίσης στο πιο πάνω Σχήμα 4.11 μπορούμε να δούμε και το αντίστοιχο κομμάτι κώδικα του καθενός. Ένας πολύ απλός πυρήνας ο οποίος το μόνο που κάνει είναι να παίρνει μια είσοδο τύπου dfeUint(32) που είναι ο τύπος που χρησιμοποιείται για απρόσημους αριθμούς 32bits αρά και τεσσάρων Bytes και να την επιστρέφει με τις δύο αυτές εντολές του. Η είσοδος και η έξοδος αυτή έχουν τεθεί από την κεντρική μονάδα επεξεργασίας και αυτό περιγράφεται με μια απλή εντολή στον κώδικα του διαχειριστή (`m.setIO(IOType.ALL_CPU)`. Οι υπόλοιπες εντολές που εκτελούνται στην κλάση του διαχειριστή είναι η δήλωση του ίδιου, του πυρήνα που διαχειρίζεται τον οποίο ακολούθως θέτει και ρητά. Τέλος εμφανίζεται η κλήση (`MyDFE(size, dataIn, dataOut);`) από την εφαρμογή για να εκτελεστεί η πιο πάνω διαδικασία όπου `dataIn` είναι το stream(πίνακας) δεδομένων που εισάγεται και `dataOut` το stream όπου θα εισαχθούν τα αποτελέσματα που στην πραγματικότητα θα είναι αντιγραφή του πίνακα εισόδου. Οι δύο αυτοί πίνακες έχουν μέγεθος ίσο με size το οποίο περνιέται επίσης σαν παράμετρος και ο MaxCompiler αντιλαμβάνεται το μέγεθος των πινάκων καθώς και το πόσο στοιχεία θα εισάγει στον και πόσα θα εξάγει από τον πυρήνα του. Η παράμετρος size όπως και επιπρόσθετες παράμετροι

στην κλήση μπορούν να χρησιμοποιηθούν και με άλλους τρόπους αναλόγως του τι ακριβώς επιθυμούμε να κάνουμε.

```
package simple;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.IOType;

class SimpleManager {
    public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        Manager manager = new Manager(params);
        Kernel kernel = new SimpleKernel(manager.makeKernelParameters());
        manager.setKernel(kernel);
        manager.setIO(IOType.ALL_CPU);
        manager.createSLiCinterface();
        manager.build();
    }
}
```

Σχήμα 4.12: Ολοκληρωμένο παράδειγμα κώδικα ενός απλού διαχειριστή

Στο Σχήμα 4.12 παρουσιάζεται ο ολοκληρωμένος πηγαίος κώδικας της κλάσης ενός απλού διαχειριστή όπου γίνεται αναφορά στο πακέτο που ανήκει, γίνονται οι απαραίτητες εισαγωγές και εκτελούνται η λειτουργίες(συναρτήσεις) που έχουμε αναφέρει. Δεν έχουμε αναφέρει ακόμη για τις δύο τελευταίες γραμμές του κώδικα. Η προτελευταία γραμμή (`manager.creatSLiCinterface();`) είναι μια απλή κλήση συνάρτησης η οποία χτίζει με τις προεπιλεγμένες ρυθμίσεις την SLiC διεπαφή για τον κώδικα της κεντρική μονάδας επεξεργασίας (builds the default SLiC interface for CPU code), και η τελευταία γραμμή (`manager.buid();`) καλούμε την μέθοδο αυτή της κλάσης του Standard Manager η οποία τρέχει όλα τα απαραίτητα βήματα για το κτίσιμο της dataflow μηχανής (DFE/FPGA), όπως την κλήση διάφορων back-end εργαλείων.

Η κλήση της `createSLiCinterface()` όπως προαναφέραμε παράγει μια απλή διεπαφή μηχανής. Πιο περίπλοκες διεπαφές μηχανής μπορούν να περιγραφούν από τον προγραμματιστή και μια DFE(FPGA) μπορεί να έχει περισσότερες από μία διεπαφές μηχανής για να διαχειρίζεται διαφορετικά μοντέλα χρήσης.

Όταν οι πυρήνες (Kernels) και ο διαχειριστής (Manager) μεταγλωττιστούν με τον MaxCompiler παράγονται συναρτήσεις λογισμικού για κάθε διεπαφή μηχανής και συμπεριλαμβάνονται στο .max αρχείο. Αυτές συναρτήσεις διεπαφής μηχανής μπορούν να καλεστούν από τον κώδικα της κεντρική μονάδας επεξεργασίας, δηλαδή από εφαρμογή, ο οποίος μπορεί να γραφτεί συνήθως σε C/C++ αλλά και σε άλλες γλώσσες όπως οι Fortran,Python,MATLAB και Excel μέσω SLiC Skins διεπαφή. Το

ενσωματωμένο περιβάλλον ανάπτυξης MaxIDE επιτρέπει στον χρήστη να προγραμματίζει τον διαχειριστή στο ίδιο περιβάλλον μαζί με την εφαρμογή και τους πυρήνες, Το πρόγραμμα του διαχειριστή είναι αυτό που δηλώνει και διαμορφώνει τους πυρήνες και τον διαχειριστή και ακολούθως ο Manager Compiler μετατρέπει την διαμόρφωση αυτή στο αντίστοιχο .max αρχείο.

4.1.3 Εφαρμογή CPU στον MaxCompiler

Μια εφαρμογή για να μπορεί να μετατραπεί σε MaxCompiler υλοποίηση και να επεξεργαστεί στο MaxIDE περιβάλλον, για βελτίωση των επιδόσεων της συνήθως θα πρέπει να είναι γραμμένη σε γλώσσα προγραμματισμού C/C++. Η εφαρμογή του CPU “κάθεται” στην κορυφή του SLiC και του MaxelerOS. Με την βοήθεια του MaxIDE που μπορεί αυτόματα να διαχειριστεί ένα πλαίσιο C για τον κώδικα του CPU.

```
const int N = 80;
float x[80], y[80];
for (int i=0; i<N; ++i)
    x[i] = 10.0 * rand() / RAND_MAX;

// This SLiC function is generated automatically by MaxCompiler.
MovingAverage(N, x, y);
```

Σχήμα 4.13: Κομμάτι κώδικα της εφαρμογής με την κλήση της SLiC συνάρτησης MovingAverage[10]

Στην εφαρμογή καλούνται συναρτήσεις για τη φόρτωση της FPGA (DFE) διαμόρφωσης και την αρχικοποίηση των DMA Transactions (Direct Memory Access) για μεταφορά δεδομένων από την μνήμη της εφαρμογής στην μνήμη (RAM) του επιταχυντή και το αντίστροφο. Επίσης μέσω της εφαρμογής θέτονται τα streams δεδομένων που θα περαστούν στον διαχειριστή, καθώς επίσης και η εκτέλεση των πυρήνων. Αφού καταχωρηθούν όλες οι απαιτούμενες ρυθμίσεις (μνήμης κ.α.) τότε το FPGA θα ξεκινήσει την διαδικασία του υπολογισμού. Στην ουσία εκείνη η μία και μόνο γραμμή κώδικα (`MovingAverage(N,x,y)`) χωρίζεται σε τρεις κύριες φάσεις. Η αρχική φάση είναι η ροή των ακατέργαστων δεδομένων (`x`) από την εφαρμογή στη μνήμη του επιταχυντή (FPGA) μέσω της PCIExpress διασύνδεσης. Σε δεύτερη φάση είναι η εκτέλεση των υπολογισμών όπου τα ακατέργαστα δεδομένα που αποστάληκαν από την εφαρμογή στη μνήμη του επιταχυντή μεταφέρονται ακολούθως στους πυρήνες για τους απαραίτητους υπολογισμούς και επιστρέφουν τα αποτελέσματα πίσω στην μνήμη ή καλυτέρα προετοιμάζουν το εξαγόμενο stream (`y`) που θα επιστρέψουν με το τέλος της εκτέλεσης στην εφαρμογή. Τρίτη και τελευταία φάση της διαδικασίας αυτής είναι η επιστροφή των αποτελεσμάτων (εξαγόμενου stream) πίσω στην εφαρμογή μας και πάλι με τη χρήση της διασύνδεσης

PICExpress. Η παράμετρος N καθορίζει το μέγεθος των streams (x,y) ενώ ο επιταχυντής μας το αντιλαμβάνεται επίσης σαν την παράμετρο για το σε πόσα στοιχεία θα εκτελέσει υπολογισμούς κα πόσα θα επιστρέψει. Η παράμετρος N μπορεί να χρησιμοποιηθεί και με άλλους τρόπους που θα αναλυθούν αργότερα και δεν μας δεσμεύουν πάντοτε τα όσα έχουμε προαναφέρει που ισχύουν ως προεπιλογή και ισχύουν επίσης και στο παράδειγμα του σχήματος Σχήμα 4.13 που αναλύσαμε.

Απόσπασμα ενός κώδικα εφαρμογής προσαρμοσμένο για τον MaxCompiler παρατηρείτε και στο Σχήμα 4.13. Σε αυτό το σημείο θα ήθελα να κάνω μια μικρή σύγκριση του αποσπάσματος του κώδικα αυτού (Version 2012.1) με την παλαιότερη έκδοση του MaxCompiler που παρουσιάζεται στο Σχήμα 4.14

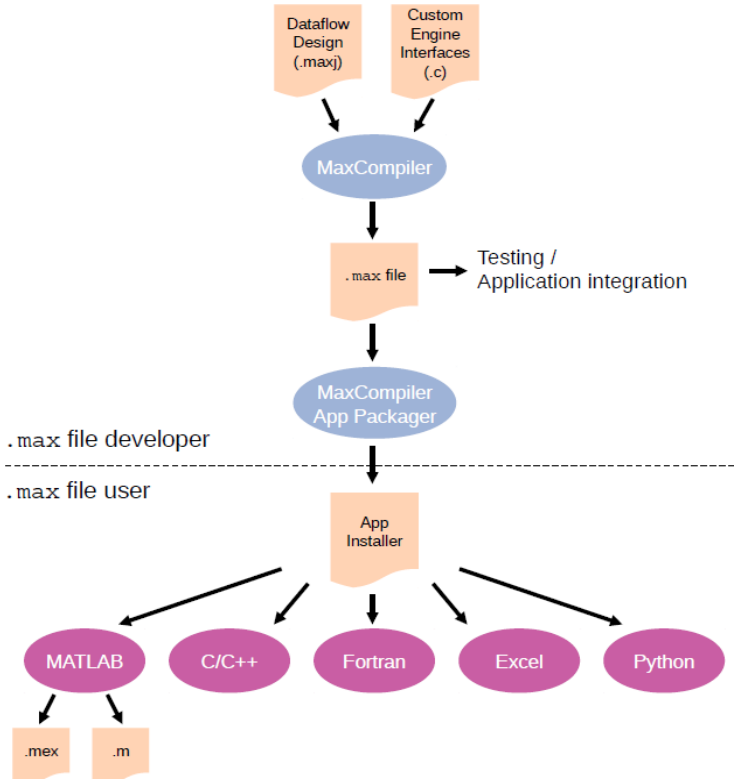
```
// Set up memory address generators on the FPGA to
// stream the X and Y arrays to/from the kernel.
int64_t datasize = n*sizeof(float) / BURST_SIZE;
max_memory_stream_set_access_linear(dramctrl, "cmd_x", size);
max_memory_stream_set_start_address(dramctrl, "x", addr_x);
max_memory_stream_set_enable(dramctrl, "x", 1);

max_memory_stream_set_access_linear(dramctrl, "cmd_y", size);
max_memory_stream_set_start_address(dramctrl, "y", addr_y);
max_memory_stream_set_enable(dramctrl, "y", 1);

max_memory_commit_setting(device, dramctrl, FPGA_A);

// Trigger computation on the FPGA. The Kernel will run until
// it has processed n items, then the function will return.
max_run(device,
    max_runfor("MAVKernel", n),
    max_end()
);
```

Σχήμα 4.14: Κομμάτι κώδικα εφαρμογής για Moving Average σε παλαιότερη έκδοση MaxCompiler[9]

Παρατηρώντας τα δύο σχήματα (Σχήμα 4.13, Σχήμα 4.14) μπορούμε με ευκολία να διακρίνουμε τις διαφορές και το πόσο ποιο αφαιρετική είναι η νέα έκδοση του MaxCompiler ευκολύνοντας έτσι την δουλειά του προγραμματιστή.

Σ' ένα Maxeler MPC(Maximum Performance Computing) σύστημα οι επεξεργαστές (CPUs) έχουν τον έλεγχο και οδηγούν τους υπολογισμούς στους επιταχυντές (DFEs). Η διεπαφή SLiC (Simple Live CPU) είναι μία Maxeler διεπαφή προγραμματισμού εφαρμογών (API) για αλληλεπίδραση μεταξύ επεξεργαστή και επιταχυντή (CPU-DFE integration) και στο απλούστερο της επίπεδο οι υπολογισμοί του DFE μπορούν να προστεθούν σε μια εφαρμογή με μια απλή κλήση εφαρμογής, ενώ για πιο λεπτούς χειρισμούς προσφέρει μια "(action"-based) διεπαφή ενεργειών. Οι κλήσεις της SLiC διεπαφής παράγονται αυτόματα από τα αντίστοιχα κομμάτια του DFE

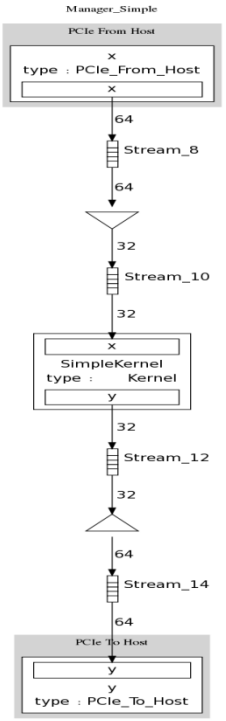
προγράμματος. Επίσης επιτρέπει στα DFE προγράμματα να διαμοιράζονται (shared) και να επαναχρησιμοποιούνται σε βιβλιοθήκες και ρυθμίσεις εφαρμογών όπως οι MATLAB, Excel ενώ τα SLiC Skins παρέχουν δεσμεύσεις σε μια εμβέλεια από γλώσσες προγραμματισμού όπως οι C/C++, Fortran, Python και MATLAB. Τα Skins αυτά επιτρέπουν στο χρήστη των περιβαλλόντων που προαναφέραμε να έχουν πρόσβαση σε DFEs χωρίς να χρειάζεται να μάθουν να τα προγραμματίζουν προκειμένου να ξεκινήσουν.



Σχήμα 4.15: Ανάπτυξη .max αρχείου και διανομή του σε πολλαπλά skins

Το .max αρχείο είναι το “εκτελέσιμο” των DFE.

Ο MaxCompiler μας δίνει τη δυνατότητα να εξάγουμε επίσης γραφήματα των διαχειριστών όπως και με τους πυρήνες με την χρήση της εντολή `maxRenderGraph`. Παράδειγμα τέτοιου γραφήματος παρουσιάζετε στο πιο κάτω Σχήμα 4.16



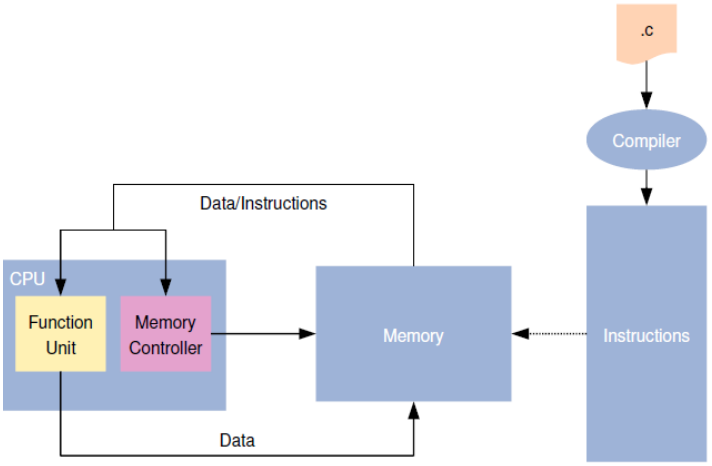
Σχήμα 4.16: Παράδειγμα γραφήματος διαχειριστή

Στο πιο πάνω Σχήμα 4.16 παρουσιάζεται ένας πολύ απλός διαχειριστής με ένα εισερχόμενο και ένα εξερχόμενο stream και παρατηρείτε η μεταφορά αυτών στον πυρήνα του και ακολούθως η εξαγωγή των αποτελεσμάτων απ’ αυτόν και η επιστροφή τους στο κυρίως πρόγραμμα.

4.2 Αρχιτεκτονική MaxCompiler

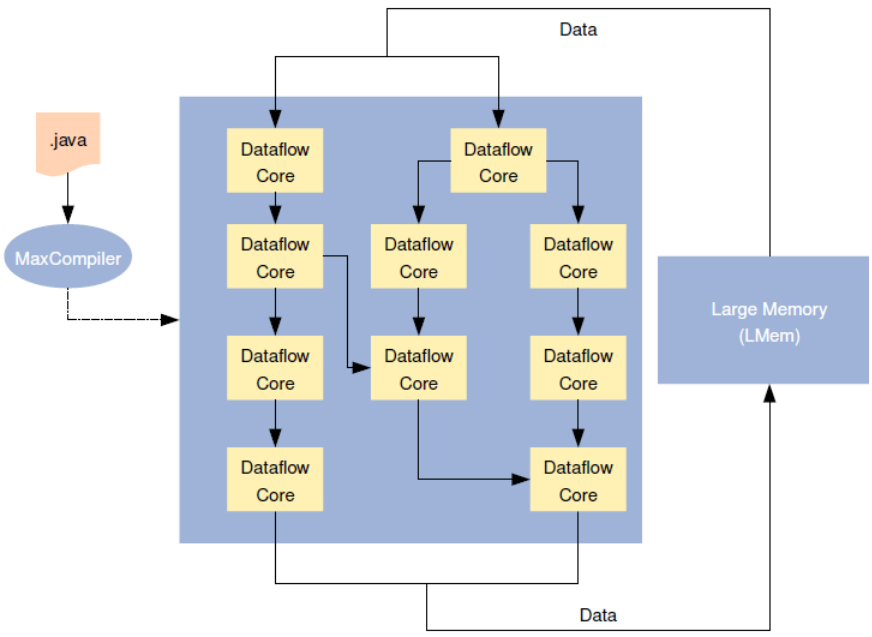
Η αρχιτεκτονική του MaxCompiler είναι σχεδιασμένη με τέτοιο τρόπο για να διαχειρίζεται “dataflow” εφαρμογές. Η κύρια διαφορά των “dataflow” εφαρμογών από τις κλασσικές εφαρμογές του λογισμικού (control flow - ροής ελέγχου) είναι ο τρόπος εκτέλεσης των υπολογισμών χωρικά αντί χρονικά έχοντας έτσι την δυνατότητα να εκτελούν πολλούς υπολογισμούς ταυτοχρόνως στους κόμβους που τις αποτελούν κυρίως με την βοήθεια διασωλήνωσης. Επίσης σημαντική διαφορά είναι ότι στα προγράμματα ροής ελέγχου ακολουθούν μια σειρά εντολών και αναλόγως των περιστάσεων διαβάζουν ή/και γράφουν στην μνήμη κατά την διάρκεια της εκτέλεσης τους, ενώ στον αντίποδα στα προγράμματα ροής δεδομένων streams δεδομένων μεταφέρονται από την μνήμη στην dataflow μηχανή (π.χ. FPGA) και τα δεδομένα αυτά μεταφέρονται από μια υπολογιστική μονάδα σε άλλη μέχρι την ολοκλήρωση της όλης διαδικασίας και ακολούθως τα αποτελέσματα επιστρέφονται και πάλι σε μορφή streams δεδομένων. Η μετατροπή ενός προγράμματος ροής ελέγχου σε ροής δεδομένων παρομοιάζεται σαν η αντικατάσταση ακριβοπληρωμένων ειδικών σε ένα εργοστάσιο με πολλούς απλούς εργάτες που με το να εκτελούν απλές λειτουργίες

όλοι τους παράλληλα μπορούν να μας προσφέρουν καλύτερη απόδοση στην παραγωγή.



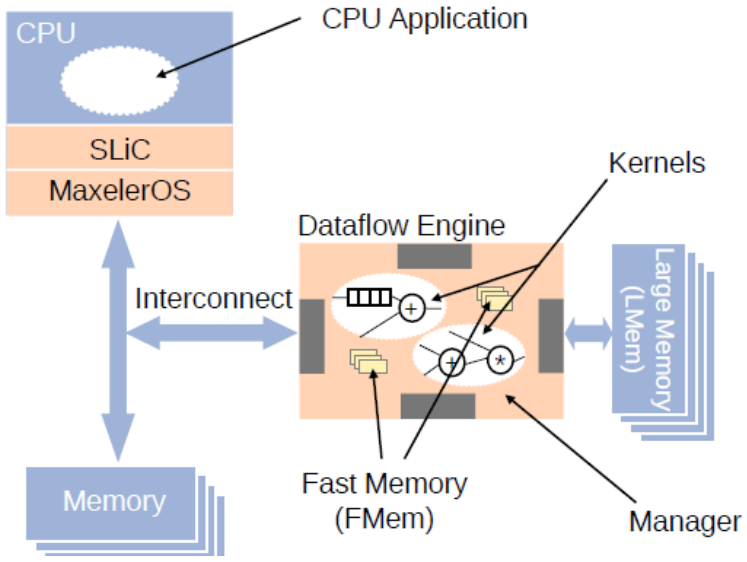
Σχήμα 4.17: Επαναχρησιμοποίηση των υπολογιστικών μονάδων στη πάροδο του χρόνου σε μια κεντρική μονάδα επεξεργασίας (CPU)[11]

Στις εφαρμογές ροής δεδομένων οι εντολές μεταφράζονται σε αριθμητικές λογικές μονάδες συνδεδεμένες μεταξύ τους και σκορπισμένες “στο χώρο”. Αυτό έχει σαν αποτέλεσμα να μην χρειαζόμαστε αποκωδικοποιητή και κρυφή μνήμη εντολών (instruction decode logic & instruction cache) καθώς επίσης και πρόβλεψη διακλάδωσης και δυναμική εκτός σειράς εκτέλεση. Με την εξάλειψη της του δυναμικού ελέγχου ροής όλοι οι διαθέσιμοι πόροι ενός chip μπορούν να αφιερωθούν στην εκτέλεση των υπολογισμών.



Σχήμα 4.18: Πρόγραμμα ροής δεδομένων σε δράση[11]

Η αρχιτεκτονική ενός Maxeler συστήματος επεξεργασίας ροής δεδομένων αποτελείται από τις δικές του τοπικές μνήμες επισυναπτόμενες με μια διασύνδεση στην κεντρική μονάδα επεξεργασίας. Σε κάθε μηχανή ροής δεδομένων (DFE/FPGA) μπορούν να υλοποιηθούν πολλαπλοί πυρήνες οι οποίοι εκτελούν υπολογισμούς σαν γραφήματα ροής δεδομένων μεταξύ του CPU, της DFE και των συνδεδεμένων τους μνήμων. Οι μηχανές αυτέ έχουν δύο τύπους μνήμης την γρήγορη (Fast Memory – FMEM) μεγέθους μερικών megabytes (MBs) για αποθήκευση δεδομένων στη μηχανή (on-chip) με ταχύτητες πρόσβασης terabytes/second και την μεγάλη τους μνήμη (Large Memory – LMEM) μεγέθους μερικών gigabytes (GBs) για αποθήκευση δεδομένων εκτός του chip (off-chip).Το υψηλό εύρος ζώνης και η ευελιξία της FMEM είναι ο κύριος λόγων που οι μηχανές ροής δεδομένων είναι ικανές να επιτύχουν υψηλές επιδόσεις σε πολύπλοκες εφαρμογές. ΟΙ εφαρμογές μπορούν να εκμεταλλευτούν πλήρως την χωρητικότητα της γρήγορης τους μνήμης (FMEM) αυτή και οι υπολογισμοί εξαπλώνονται στη μηχανή (“στο χώρο”) και έτσι τα δεδομένα κρατιούνται πάντα κοντά στους υπολογισμούς σε αντίθεση με την πολυεπίπεδη κρυφή μνήμη των CPUs που υπάρχει η αναλογία όσο μικρότερη τόσο γρηγορότερη και αναλόγως αυτής της αναλογίας σχετίζεται και ποια είναι η πιο κοντινή



Σχήμα 4.19: Αρχιτεκτονική maxeler μηχανής ροής δεδομένων (DFE)[11]

Εκτός από τις μνήμες μιας Maxeler μηχανής ροής δεδομένων για τις οποίες έχουμε πει πρέπει να αναφέρουμε και τα υπόλοιπα κομμάτια της αρχιτεκτονικής τους. Κατ’ αρχάς όπως είχαμε αναφέρει και προηγουμένως μια τέτοια μηχανή μπορεί να προγραμματιστεί με ένα ή και περισσότερους πυρήνες και ένα διαχειριστή και αναφέραμε επίσης ότι οι πυρήνες εκτελούν τους υπολογισμούς και ο διαχειριστής οργανώνει την ροή δεδομένων στη μηχανή και τους πυρήνες της. Αφού υλοποιηθούν

οι πυρήνες και ο διαχειριστής ο MaxCompiler παράγει μια υλοποίηση ροής δεδομένων η οποία μπορεί να καλεστεί από το CPU μέσω της διεπαφής SLiC (Simple Live CPU) η οποία είναι μια αυτόματα παραγόμενη διεπαφή στο πρόγραμμα ροής δεδομένων κάνοντας έτσι ευκολότερη την κλήση της εκτέλεσης των dataflow μηχανών από τις επισυναπτόμενες CPUs. Η SLiC (ονομαζόταν MaxelerRT σε παλαιότερες εκδόσεις) βιβλιοθήκη λειτουργεί σαν ένα περιγραφικό επίπεδο μεταξύ του υλικού και της εφαρμογής και μέσω αυτού επιτυγχάνετε η πρόσβαση της εφαρμογής στο υλικό και η κλήση των API συναρτήσεων του εκτελούν τη δράση στο DFE. Μεταξύ αυτών των δράσεων η αποστολή των streams δεδομένων και ο καθορισμός των παραμέτρων στο DFE. Η διαχείριση του συνολικού συστήματος γίνεται από το MaxelerOS το οποίο συνεργάζεται με λειτουργικό σύστημα Linux και τον διαχειριστή (Manager) της μηχανής και είναι υπεύθυνο για την διαχείριση μεταφοράς δεδομένων και της δυναμικής βελτιστοποίησης κατά την διάρκεια της εκτέλεσης. Η διασύνδεση (interconnect) όπως φαίνεται και στην εικόνα είναι συνήθως PCIExpress υψηλού εύρους ζώνης.

Σε ένα πλήρη Maxeler σύστημα υπερυπολογοστών πολλές μηχανές είναι συνδεδεμένες μεταξύ τους μέσω μιας υψηλού εύρους ζώνης MaxRing διασύνδεσης η οποία επιτρέπει στις εφαρμογές να επεκτείνονται γραμμικά με πολλαπλές DFEs στο σύστημα ενώ υποστηρίζουν και πλήρη επικάλυψη επικοινωνίας και υπολογισμών. (βλ. Σχήμα 4.1)

. Ένα Maxeler σχεδιασμένο σύστημα λοιπόν αποτελείται από CPUs και DFEs. Τα CPUs τρέχουν τα εκτελέσιμα αρχεία ενώ οι DFEs τρέχουν τα αρχεία διαμόρφωσης του υλικού (.max). Τα .max αρχεία φορτώνετε από το κυρίως πρόγραμμα (CPU) και τρέχουν σε μια διαθέσιμη μηχανή ροής δεδομένων (DFE/FPGA). Ο κύκλος ζωής ενός .max αρχείου ακολουθείτε από κάποια βήματα με αρχικό την αρχικοποίηση και ακολούθως την φόρτωση του αρχείου στην DFE (το οποίο μπορεί να διαρκέσει από 100ms εως 1 δευτερόλεπτο), σαν επόμενο βήμα είναι η εκτέλεση των λειτουργιών (στο βήμα αυτό το CPU καλεί συναρτήσεις της SLiC διεπαφής για να εκτελέσουν τη δράση τους πάνω στο DFE) και σαν τελευταίο βήμα είναι η εκφόρτωση του (στο τελευταίο αυτό βήμα η DFE αποδεσμεύετε από το CPU) και τέλος η αποδεσμευση του. Υπάρχουν τρία είδη της SLiC διεπαφής (Basic Static, Advance Static και Advanced Dynamic) και στην Basic Static το .max αρχείο φορτώνετε με την πρώτη κλήση μιας SLiC συνάρτησης και απελευθερώνετε όταν το κυριώς πρόγραμμα τερματίσει, Επίσης δίνεται η δυνατότητα χρήσης πολλαπλών .max αρχείων σε πολλαπλές μηχανές ροής δεδομένων/ Στην Basic Static SLiC διεπαφή κάθε .max τρέχει σε διαφορετική DFE.

Οι διαφορές των τριών ειδών (επιπέδων) SLiC διεπαφών είναι ότι, στην Basic Static αρκεί μια και μόνο κλήση συνάρτησης για να τρέξει το DFE χρησιμοποιώντας στατικές λειτουργίες για την λεπτομερή δήλωση του .max αρχείου. Στην Advanced Static επιτρέπετε ο έλεγχος της φόρτωσης στο DFE καθορίζοντας πολλαπλές πιο σύνθετες λειτουργίες και βελτιστοποίηση της συνεργασίας CPU και DFE. Τέλος στην Advanced Dynamic επιτρέπεται η πλήρη βελτιστοποίηση των λειτουργιών και ο λεπτομερής έλεγχος της κατανομής, δέσμευσης και αποδέσμευσης όλων των πόρων της ροής δεδομένων.

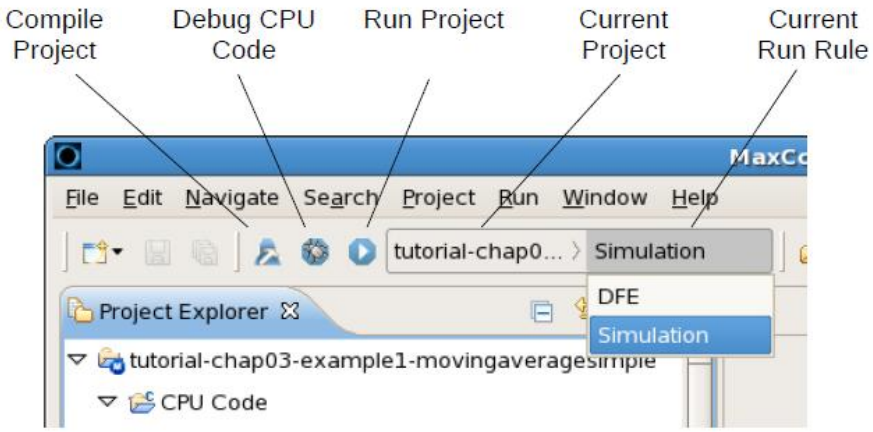
4.3 MaxIDE

Ο MaxCompiler διαθέτει το δικό του ενσωματωμένο περιβάλλον ανάπτυξης (Integrated Development Environment) το MaxIDE. Ο MaxIDE είναι μια ελαφρώς τροποποιημένη έκδοση του EclipseIDE ειδικά σχεδιασμένη για τον MaxCompiler. Μεταξύ άλλων το MaxIDE περιλαμβάνει ότι ακριβώς περιλαμβάνουν συνήθως τέτοια περιβάλλοντα όπως συντάκτη πηγαίου κώδικα, εργαλεία αυτοματοποίησης (π.χ. αυτόματη μεταγλώττιση κώδικα, παρουσίαση συντακτικών λαθών) καθώς επίσης και ένα ελεγκτή/παρατηρητη (debugger). Στα επιπρόσθετα του συμπεριλαμβάνετε η δυνατότητα συγγραφής MaxJ κώδικα που είναι η γλώσσα προγραμματισμού για τους πυρήνες και διαχειριστές στον MaxCompiler. Με την εκκίνηση του MaxIDE γίνεται η επιλογή του χώρου εργασίας (workspace) μας.

Σχήμα 4.20: Επιλογή χώρου εργασίας

Ακολούθως εμφανίζεται η σελίδα καλωσορίσματος και μας δίνει επιλογές όπως την δημιουργία καινούργιου MaxCompiler Project ή την εισαγωγή (import) των projects με τα παραδείγματα και τις εργασίες των φροντιστηρίων.

Μέσω του MaxIDE μας δίνετε η δυνατότητα να μεταγλωττίσουμε τις υλοποιήσεις μας, να απασφαλματώσουμε (debug) τον κώδικα του κυρίως προγράμματος μας και να τρέξουμε τις υλοποιήσεις μας με πραγματική εκτέλεση στο υλικό ή/και να τα προσομοιώσουμε για να ελέγξουμε την ορθότητας τους πριν μεταγλωττίσουμε για την πραγματικη εκτέλεση της οποίας η μεταγλώττιση μπορεί να διαρκέσει από 35 λεπτά μέχρι και περισσότερο της μιας ώρας και όλα αυτά με ιδιαίτερη ευκολία αφού όλα τα πιο πάνω μπορούν να γίνουν απλά με το πάτημα ενός κουμπιού.



Σχήμα 4.21: MaxIDE buttons for building and running report[11]

Το πλήκτρο Compile Project χτίζει (δημιουργεί) το .max αρχείο είτε για την προσομοίωση είτε για πραγματική εκτέλεση αναλόγως της επιλογής Run Rule ενώ επίσης μεταγλωττίζει και το πηγαίο κώδικα του κυρίως προγράμματος. Οι έξοδοι των μεταγλωττίσεων και των εκτελέσεων εμφανίζονται στην ενσωματωμένη κονσόλα του MaxIDE καθώς επίσης και τυχόν προειδοποιήσεις (warnings) και σφάλματα (errors) της υλοποίησης.

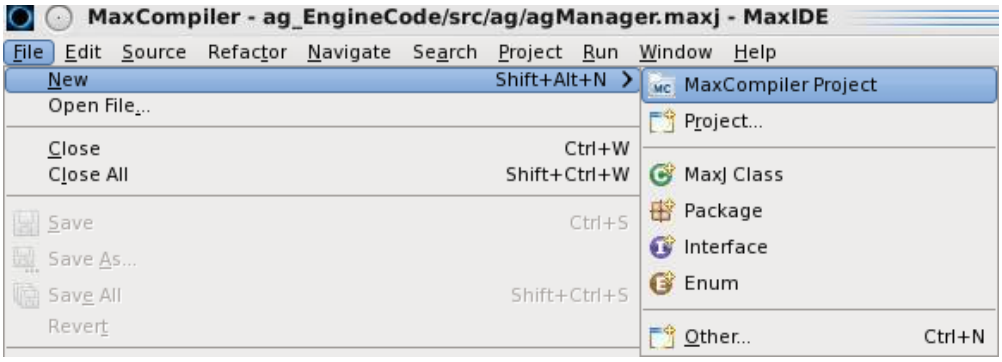
Τα MaxCompiler Projects έχουν την δυνατότητα να μεταγλωττίζονται και να εκτελούνται εκτός του MaxIDE με την βοήθεια εντολών που μπορούν να εκτελεστούν στο terminal. Οι δυνατές αυτές εντολές στην τελευταία έκδοση του MaxCompiler (version 2012.2.1) είναι πέντε ενώ σε παλαιότερες ήταν περισσότερες αποφασίστηκε όπως μερικές από αυτές να συγχωνευτούν για περαιτέρω ευκολία του χρήστη. Οι πέντε αυτές εντολές είναι οι make, make startsim, make run, make stopsim και make runsim. Για να χρησιμοποιήσουμε και να εκτελέσουμε αυτές τις εντολές θα πρέπει να μετακινηθούμε στον κατάλογο του χώρου εργασίας μας όπου υπάρχει το project που μας αφορά. Για την εκτέλεση της εντολής make εκτελούμε:

{currentproject]$> `make RunRules/DFE` ή `make RunRules/Simulation`

αναλόγως για ποιας εκτέλεση επιθυμούμαι να παράξουμαι το .max αρχείο. Αν εκτελέσουμε την εντολή make startsim ξεκινούμε τον προσομοιωτή αν δεν υπάρχει κάποιος ήδη ενεργοποιημένος και απαραίτητα να εκτελεστεί στον κατάλογο RunRules/Simulation αλλιώς εάν εκτελεστεί στον κατάλογο RunRules/DFE ή υπάρχει ήδη ενεργοποιημένος προσομοιωτής η εντολή δεν θα πράξει απολύτως τίποτα. Η εντολή make run μεταγλωττίζει αν χρειάζεται και ακολούθως τρέχει την εφαρμογή μας. Για την εκτέλεση της προσομοίωσης αιτείται προηγουμένως η εκτέλεση της make startsim για την εκκίνηση του προσομοιωτή. Η εντολή make stopsim πρέπει να εκτελεστεί στον κατάλογο προσομοίωσης και σταματά τον ενεργό προσομοιωτή εάν υπάρχει αλλιώς εξάγετε σφάλμα. Τέλος η εντολή make runsim είναι ισοδύναμη με την συγχώνευση των τριών εντολών της προσομοίωσης (δηλαδή make startsim run stopsim).

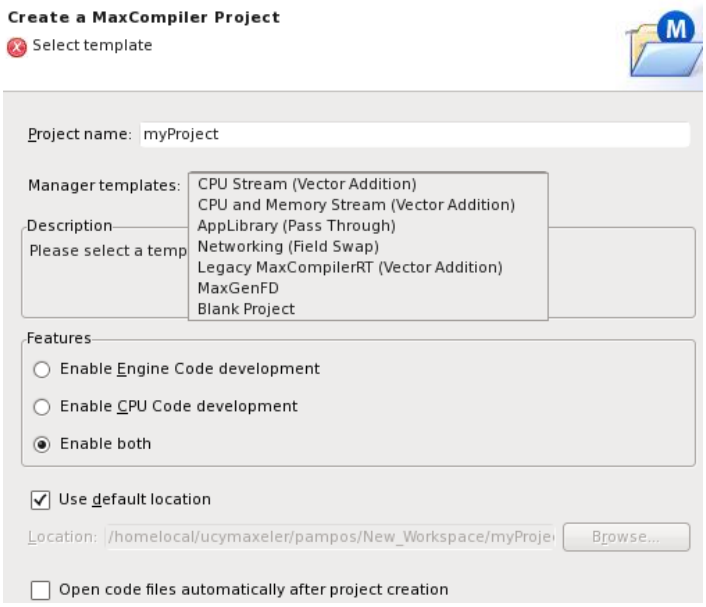
4.4 Δημιουργία ενός MaxCompiler Project

Για την δημιουργίας ενός νέου δικού μας MaxCompiler Project θα χρησιμοποιήσουμε ασφαλώς το MaxIDE και File -> New -> MaxCompiler Project



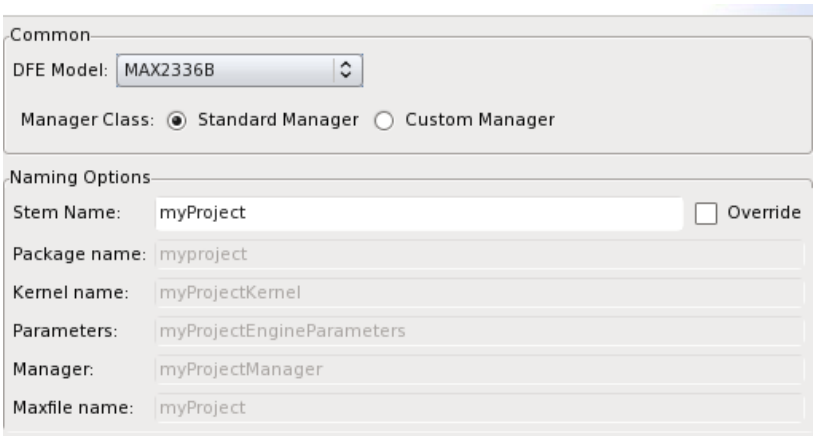
Σχήμα 4.22: Δημιουργία νέου MaxCompiler Project

Αφού πατήσουμε την δημιουργία ενός νέου MaxCompiler Project θα μας εμφανιστεί ένα καινούργιο παράθυρο στο οποίο θα πρέπει να επιλέξουμε το όνομα που θα δώσουμε στο νέο μας Project, να επιλέξουμε ανάμεσα στα διαθέσιμα templates που μας δίνονται σαν επιλογές και κάποιες επιπλέον επιλογές.



Σχήμα 4.23: Ονομασία project και επιλογή template

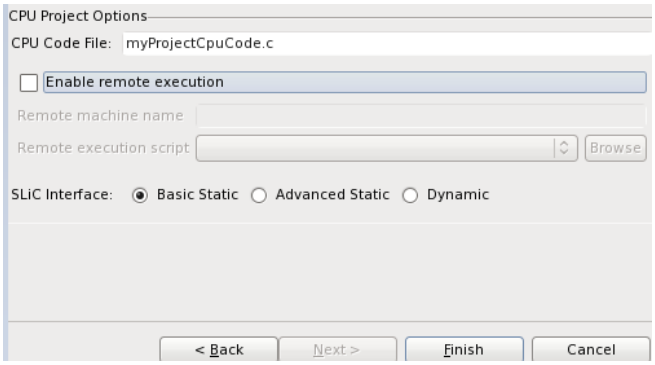
Συνήθως επιλέγουμε το πρώτο template (CPU Stream (Vector Addition)) το οποίο είναι ένα απλό παράδειγμα με παραμέτρους ένα εισαγόμενο και ένα εξαγόμενο stream μαζί με μία σταθερά η οποία προσθέτετε σε όλα τα στοιχεία του εισαγόμενου stream και επιστρέφει τα αποτελέσματα αυτών των πράξεων μέσω του εξαγόμενου. Επιλέγεται συνήθως λόγω της απλότητας του που μας βοηθά να επεκταθούμε πάνω του και να μετατρέψουμε όπως εμείς επιθυμούμε. Προτιμάται από το κενό project (Blank Project) για να αποφεύγουμε τις ούτως οι αλλιώς απαραίτητες διαδικασίες δημιουργίας ενός καινούργιου έργου αφού μας προσφέρονται έτοιμες.



Σχήμα 4.24: Επιλογή DFE model

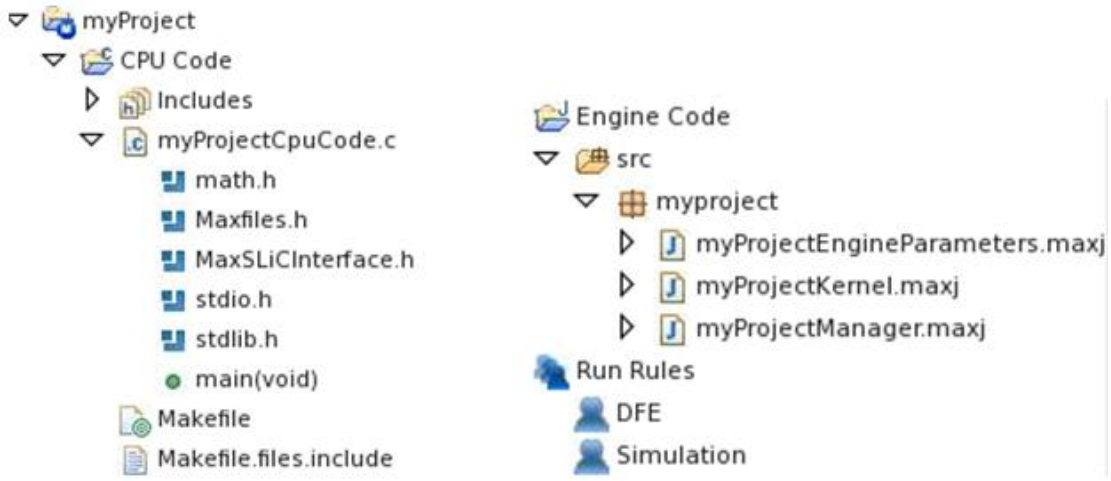
Στο επόμενο παράθυρο γίνετε η επιλογή του μοντέλου της μηχανής ροής δεδομένων (MAX2336B είναι το μοντέλο που διαθέτει η μηχανή που έχουμε δουλέψει) και η επιλογή αυτή θέτεται στα RunRules, εάν η κλάση του διαχειριστή θα είναι η τυποποιημένη διαμόρφωση (Standard Manager) ή αν επιθυμούμε να

δημιουργήσουμε την δική μας (Custom Manager) και οι επιλογές ονομασίας για τα διάφορα κομμάτια της εφαρμογής μας. Στο τελευταίο παράθυρο πριν την ολοκλήρωση της δημιουργίας του νέου μας Project μας δίνεται η επιλογή ονομασίας του αρχείου που θα περιέχει τον κώδικα του κυρίως προγράμματος και η επιλογή της SLiC διεπαφής ανάμεσα στις τρείς επιλογές που αναφέραμε πιο πάνω.



Σχήμα 4.25: Ονομασία αρχείου CPU κώδικα και επιλογή SLiC διεπαφής

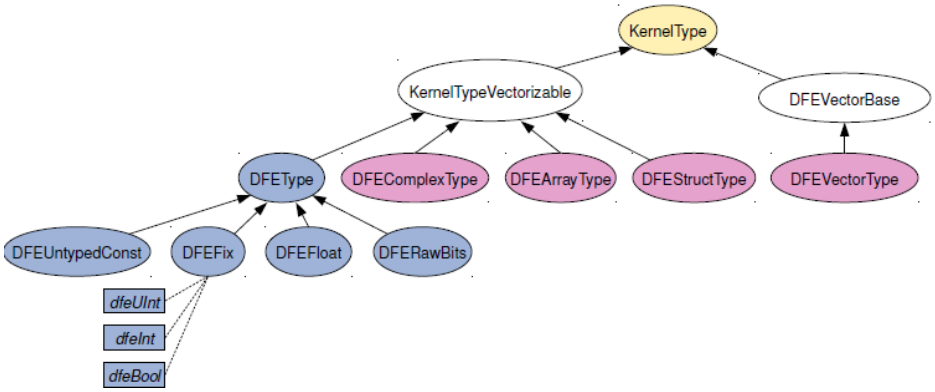
Οι επιλογές για συγγραφή του κώδικα του κυρίως προγράμματος που μας δίνονται είναι ανάμεσα στις γλώσσες προγραμματισμού C και C++. Αφού πατήσουμε Finish θα δημιουργηθούν όλα τα απαραίτητα αρχεία που θα τους εισαχθούν αυτόματα οι απαραίτητες βιβλιοθήκες και θα χωριστούν σε τρείς κατηγορίες. Οι κατηγορίες αυτές είναι οι CPU Code, Engine Code και RunRules στις οποίες υπάγονται αντίστοιχα ο κώδικας του κυρίως προγράμματος στην πρώτη μαζί με το αυτόματα παραγόμενο Makefile, ο διαχειριστής, ο πυρήνας και το αρχείο παραμέτρων μηχανής στην δεύτερη και η κανόνες εκτέλεσης για προσομοίωση και πραγματική εκτέλεση στη τρίτη.



Σχήμα 4.26: Περιεχόμενα του νέου Project

4.5 Τύποι δεδομένων MaxCompiler

Οι μεταβλητές που δηλώνονται σε ένα πρόγραμμα ροής δεδομένων (π.χ kernel) είναι στην ουσία πύλες από τις οποίες streams από αριθμούς περνούν απ' αυτές αντιπροσωπευόμενοι από άσσους και μηδενικά δηλαδή δυαδική μορφή και αναλόγως του μεγέθους του αριθμού καθορίζονται τα bits που χρειάζεται για την αναπαράσταση του. Ο MaxCompiler προσφέρει διαφορετική υλοποίηση γύρω από το θέμα και προσφέρει διάφορα είδη τύπων για δηλώσεις μεταβλητών. Οι μεταβλητές που συνήθως δηλώνονται στον MaxCompiler είναι τα εισαγόμενα και εξαγόμενα streams και ο βασικότερος τύπος που χρησιμοποιεί είναι ο τύπος DFEVar. Κάθε μεταβλητή έχει την δική της εμβέλεια και ακρίβεια. Υπάρχει μια ιεραρχία των τύπων δεδομένων και απεικονίζεται στο πιο κάτω Σχήμα 4.27.



Σχήμα 4.27: Ιεραρχία της κλάσης για τους τύπους δεδομένων[11]

Υπάρχουν οι αρχέγονοι (μπλε) και οι σύνθετοι τύποι (ροζ). Οι πιο συνήθης χρησιμοποιημένοι είναι οι αρχέγονοι μεταξύ των οποίων οι γνωστοί σε όλους μας τύποι float, int και Boolean με διαφορετική όμως μορφή. Ο dfeFloat(int exponent bits, int mantissa bits) μπορεί να αναπαραστήσει float (dfeFloat(i8,24)) και double (dfeFloat(i11,53)) μεταβλητές όπου η πρώτη παράμετρος είναι τα bits για τον εκθέτη και η δεύτερη για το δεκαδικό μέρος. Αθροίζοντας τις 2 αυτές παραμέτρους έχουμε 32 bits = 4 bytes όπως και στον float τύπο και 64 bits = 8 bytes όπως τον double. Επίσης πολύ συχνά χρησιμοποιούνται και οι τύποι dfeInt(int bits) και dfeUInt(int bits) που αναπαριστούν ακέραιους αριθμούς μεγέθους bits της επιλογής μας. Για τους κλασσικούς τύπους Int και unsigned int χρησιμοποιούμαι 32 bits, για long 64 ενώ μπορούμε να χρησιμοποιήσουμε και λιγότερα για μεταβλητές που γνωρίζουμε ότι δεν θα παίρνουν μεγάλες τιμές. Ακόμη υπάρχει και ο τύπος dfeBool() για δυαδικές μεταβλητές.

Όλοι οι πιο πάνω τύποι στην ουσία είναι συντομογραφίες της dfeFixOffset(int num_bits, int offset, SignMode signmode) η οποία μπορεί να

καθορίσει του πιο πάνω τύπους αλλά και παραλλαγές αυτών. Το SignMode μπορεί να είναι SignMode.UNSIGNED ή SignMode.TWOSCOMPLEMENT. Αν το offset είναι 0 έχουμε ακέραιο αριθμό αν είναι αρνητικό έχουμε κλασματικό μέρος και αν έχει θετική τιμή μπορεί να πάρει μεγάλες τιμές όμως όχι ακριβή όσον αφορά τις ακέραιες.

Επίσης ο MaxCompiler μας δίνει την δυνατότητα να κάνουμε casting σε μεταβλητές του αλλάζοντας τους τον τύπο κάτι το οποίο συστήνεται να αποφεύγεται καθώς δεσμεύει από του διαθέσιμους πόρους του υλικού μας. Ένα παράδειγμα casting σε 8 bit απρόσημο ακέραιο DFEVar x_int=x.cast(dfeUint(8));. Ο MaxCompiler σε αντίθεση με άλλες γλώσσες προγραμματισμού σε casting από float σε ακέραιο στρογγυλεύει την τιμή στον πλησιέστερο αντί να αποκόπτει το δεκαδικό μέρος.

Όταν γίνονται πράξεις μεταξύ δυό διαφορετικού τύπου μεταβλητών ο MaxCompiler δεν συμπεραίνει αυτόματα το type cast που πρέπει να κάνει αλλά δίνει σφάλμα μεταγλώττισης και ζητά από τον χρήστη να κάνει ρητά το casting. Επίσης οποιαδήποτε σταθερά χρησιμοποιείτε στη σχεδίαση των πυρήνων χωρίς ρητό τύπο, ο Kernel Compiler την αναθέτει σε τύπο DFEUntypeConst.

Όσον αφορά του σύνθετους τύπους υπάρχουν τέσσερις διαθέσιμοι οι οποίοι απεικονίζονται και στο Σχήμα 4.27 και είναι οι DFEComplexType, DFEArrayType, DFEStruct και DFEVector.

4.6 Προηγμένες λειτουργίες MaxCompiler

4.6.1 Debugging

Οι πιο σημαντικές λειτουργίες του debugging στον MaxCompiler είναι η εκτέλεση του κώδικα του κυρίως προγράμματος βήμα-βήμα και η δυνατότητα συναρτήσεων όπως η debug,printf() η οποία μπορεί να τυπώσει την τιμή μιας μεταβλητής πυρήνα κατά την διάρκεια εκτέλεσης της προσομοίωσης ώστε να μπορεί να γίνετε καλύτερος έλεγχος ορθότητας. Ακόμη δυνατότερη λειτουργία είναι τα simulation watches που μπορούν να κάνουν κάτι ανάλογο αλλά μπορούν να δουν και παλαιότερη τιμή μιας μεταβλητής ενώ επίσης παράγουν και ένα αρχείο παρόμοιου τύπου με την excel όπου φυλάγονται όλες οι τιμές που παρακολουθήθηκαν. Επιπρόσθετα δίνετε και η εντολή maxdebug η οποία μπορεί να εκτελεστεί από την γραμμή εντολών που μπορεί να παράξει παρόμοια γραφήματα με αυτά της maxRenderGraph όμως αυτά είναι αρκετά βοηθητικά στην απασφαλμάτωση του κώδικα.

4.6.2 Πλοήγηση στα Streams δεδομένων

Ο MaxCompiler μας δίνει την δυνατότητα μετακίνησης μέσα στα stream δεδομένων από την προσωρινή μας θέση μπροστά ή πίσω αναλόγως του τι επιθυμούμαι. Ατυτό είναι αρκετά βοηθητικό για συγκρίσεις μεταξύ στοιχείων, για ταξινόμηση καθώς επίσης και για πολλαπλούς υπολογισμούς σε μια εκτέλεση ενός kernel. Υπάρχουν διαθέσιμα τρία είδη offset, το στατικό στο οποίο το μέγεθος είναι προκαθορισμένο πριν την εκτέλεση , τα variables offsets που το μέγεθος τους θέτετε κατά την εκτέλεση αλλά πριν να επεξεργαστούν οποιαδήποτε δεδομένα και τέλος το δυναμικό που θέτετε κατά την διάρκεια της εκτέλεσης και συγκεκριμένα στην διαρκεία επεξεργασίας των δεδομένων.

	LUTs	FFs	BRAM
Static offsets	11,076	13,749	28
Variable offsets	11,172	13,946	29
Dynamic offsets	12,341	14,662	71

Σχήμα 4.28: Σύγκριση των offset σε χρήση πόρων για το ίδιο 9-σημείων φίλτρο

	Static Offsets	Variable Offsets	Dynamic Offsets
Size configurable	At compile-time	Before a stream	tick-by-tick
On-chip resource cost	Low	Moderate	High
Compiler optimizes	Yes	Yes	No

Σχήμα 4.29: Σύγκριση διαφορετικών τύπων stream offset

4.6.3 Έλεγχοι ροής

Μας δίνετε επίσης η δυνατότητα ελέγχου ροής (control flow). Μέσω της control.count.simpleCounter(width) που αναθέτετε σε ένα DFEVar αντικείμενο που είναι στην ουσία ένας απλός μετρητής από το μηδέν μέχρι το width μας δίνετε η δυνατότητα να εφαρμόσουμε εσωτερικούς βρόγχους σε ένα πυρήνα και να επαναλάβουμε κάποια ρουτίνα, απλώς με το να τοποθετήσουμε κάποιες εντολές που θέλουμε να επαναληφθούν κάτω από την εντολή του μετρητή. Επίσης μπορούμε να αναπαραστήσουμε φωλιασμένα loops (βρόγχους) με την βοήθεια των CounterChain αντικειμένων. Για παράδειγμα για μετετράπη του κομματιού κώδικα

```
for ( int i = 0; i < 6; i += 2) {
      for ( int j = 0; j < 2; ++j) {
System.out.println( ” i = ” + i + ” , j = ” + j ) ;

}}
```

θα γίνει ως εξής:

```
CounterChain chain = control.count.makeCounterChain();
DFEVar i = chain.addCounter(maxI, 2);
DFEVar j = chain.addCounter(maxJ, 1);
```

Υπάρχουν και πιο προηγμένοι μετρητές τους οποίους μπορούμε να χρησιμοποιήσουμε αναλόγως των επιθυμιών μας καθορίζοντας παραμέτρους όπως την τιμή εκκίνησης του μετρητή, την μέγιστη τιμή ακόμα και την δυνατή επανάληψη (συνέχεια) εκτέλεσης του βρόγχου αν φθάσει στον μέγιστο αριθμό που θέσαμε εάν το επιθυμούμε, και την κατά πόσο αύξηση του μετρητή σε κάθε επανάληψη.

4.6.4 SLiC ρυθμίσεις πυρήνα (Kernel settings)

Πολλές επιλογές διαμόρφωσης των Kernels μπορούν να τεθούν από την SLiC διεπαφή μηχανής. Μεταξύ αυτών ο αριθμός των ticks (επαναλήψεων του κώδικα τους) που θα εκτελέσουν οι πυρήνες, οι ρυθμίσεις των streams, ο καθορισμός των εισόδων στις μνήμες, οι offset εκφράσεις καθώς επίσης και τα βαθμωτά εισαγόμενα (Scalar inputs) που συνήθως είναι σταθερές τιμές.

Όλα τα πιο πάνω μπορούν να ρυθμιστούν ακολούθως:

```
void setTicks(String blockName, InterfaceParam p)
void setTicks(String blockName, long p)

void setStream(String strmName,CPUTypes type,InterfaceParam p)
void setStream(String streamName, CPUTypes type, long p)

void setMem(String blockName,String memoryName,int index,double valu)
void setMem(String blockName,String memoryName,int index, long value)
setMem(String blkName,String memName,int index,InterfaceParam value)

void setOffset(String block, String offset, InterfaceParam p)
void setOffset(String blockName, String offsetName, long p)

void setScalar(String blockName, String scalar, long value)
void setScalar(String blokName, String scalarName, double val)
void setScalar(String blokName,String scalar,InterfaceParam p)
```

Οι υπερφορτωμένες εκδόσεις των μεθόδων επιτρέπουν η τιμές να τεθούν είτε από Java μεταβλητή είτε από παράμετρο engine interface.

4.6.5 Ασύγχρονη εκτέλεση

Η ασύγχρονη εκτέλεση (Asynchronous execution) που μας προσφέρει ο MaxCompiler μας δίνει την δυνατότητα κατά την διάρκεια της που εκτελούνται λειτουργίες στο DFE να αξιοποιήσουμε αυτό το διάστημα αναθέτοντας άλλες λειτουργίες παράλληλα στο CPU για να έχουμε καλύτερες επιδόσεις. Για να γίνει αυτό πρέπει να καλεστούν οι λεγόμενες non-blocking συναρτήσεις καθώς επίσης και η κλήση της max_nowait (void max_nowait(max_run *run)) για να αγνοήσει ο κώδικας του κυρίως προγράμματος τις εξόδους της εκτέλεσης στο DFE. Υπάρχει επίσης η max_wait η κλήση της οποίας επανασυγχρονίζει την εκτέλεση με το DFE και το κυρίως πρόγραμμα περιμένει την ολοκλήρωση των λειτουργιών του DFE για να συνεχίσει. ΟΙ non-blocking συναρτήσεις είναι οι εξής:

```
max_run_t *MovAvg_nonblock(int32_t param_N, const float *instream_x,
float *outstream_y);

max_run_t *MovAvg_run_nonblock(max_engine_t *engine, MovAvg actions t
*interface actions);

max_run_t *MovAvg_run_group_nonblock(max_group_t *group,
MovAvg_actions_t *interface_actions);

max_run_t *MovAvg_run_array_nonblock(max_engarray_t *engarray,
MovAvg_actions_t *interface_actions[]);
```

Και οι ανάλογες non-blocking συναρτήσεις για Advanced Dynamic:

```
max_run_t* max_run_nonblock(max_engine_t *engine, max_actions_t
                  *actions);

max_run_t* max run_array_nonblock(max_engarray_t *engarray,
                  max_actarray_t *actarray);

max_run_t* max_run_group_nonblock(max_group_t *group, max_actions_t
                  *actions);
```

Παράδειγμα ασύγχρονής εκτέλεσης στην οποία τρέχει ταυτοχρόνως η CPU συνάρτηση του κινητού μέσου όρου ενόσω εκτελείται και η DFE:

```
max_run_t *execStatus = MovAvg_nonblock(size,dataIn, dataOut);
movAvgCPU(size, dataIn, expected);
/_ Other CPU work can be done here. _/
max_wait(execStatus);
```

Και η ανάλογη εκτέλεση για Advanced Static API:

```
max_file_t *mavMaxFile = MovAvg init();
max_engine_t *mavDFE = max_load(mavMaxFile, ”local:*”);
MovAvg_actions_t actions;
actions.param_N = size;
actions.instream_x = dataIn;
actions.outstream_y = dataOut;
max_run_t *execStatus = MovAvg_run_nonblock(mavDFE, &actions);

MovAvgCPU(size, dataIn, expected);
/---- Other CPU work can be done here. ---/

max_wait(execStatus);
max_unload(mavDFE);
```

4.6.6 Custom Manager

Μέσω του Custom Manager μας δίνεται η δυνατότητα να δημιουργούμε τις δικές μας διαμορφώσεις διαχειριστών αναλόγως του πως τον επιθυμούμε και των αναγκών της εφαρμογής που θα υλοποιήσουμε.

Ιδιαίτερο χαρακτηριστικό ενός Custom Manager είναι ότι έχει την δυνατότητα διαχείρισης πολλαπλών πυρήνων κάτω από την αιγίδα του. Τα εισαγόμενα stream σε τέτοιες περιπτώσεις τα πρέπει να διαμοιραστούν στους πυρήνες αναλόγως του πως ο προγραμματιστής επιθυμεί. Custom Manager χρησιμοποιήσαμε και για δύο από τις υλοποιήσεις της εργασίας μας.

Επίσης μας προσφέρεται οι mux, demux, slpit και join μεθόδοι οι οποίες είναι ιδιαίτερα χρήσιμες αν αναλογιστεί κανείς τον περιορισμένο αριθμό επιτρεπόμενων εισαγόμενων streams τότε θα χρησιμοποιήσουμε τέτοιες μεθόδους για να μοιράσουμε κατά την διάρκεια της εκτέλεσης στο DFE αντί πρώτου της έναρξης.

4.7 Ανασκόπηση MaxCompiler

Δυστυχώς οι συνολικές λειτουργίες και δυνατότητες του MaxCompiler δεν μπορούμε να τις εξαντλήσουμε ώστε να περιγραφούν και να αναλυθούν όλες στα πλαίσια της ατομικής μου διπλωματικής εργασίας. Έχουν αναλυθεί τα κυριότερα στοιχεία και οι βασικότερες λειτουργίες του όπως ο καθορισμός παραμέτρων από το κυρίως πρόγραμμα για εκτέλεση, τρόποι εισόδου/εξόδου των δεδομένων, τρόπος συγγραφής των υλοποιήσεων καθώς και κάποιες από τις προηγμένες λειτουργίες που μας προσφέρει το εργαλείο.

Κεφάλαιο 5 Πειραματικές Μετρήσεις και Αποτελέσματα

5.1 Εισαγωγή

Μετά από τη μελέτη και την εκπαίδευση πάνω στο αντικείμενο της εργασίας θα πρέπει να ακολουθήσει το στάδιο της υλοποίησης και προσαρμογής μιας εφαρμογής σε αυτό και οι πειραματικές μου μετρήσεις από τα αποτελέσματα των οποίων θα μπορέσω να εξάγω τα συμπεράσματα μου. Για να κάνω τις πειραματικές μου μετρήσεις θα πρέπει να επιλέξω κάποια έτοιμη σειριακή εφαρμογή και να την μετατρέψουμε σε υλοποίηση για τον MaxCompiler με σκοπό να συγκρίνουμε τους χρόνους εκτέλεσης τους και να ελέγξουμε αν μπορούμε να επιτύχουμε βελτίωση των επιδόσεων και σε ποια επίπεδα μπορεί να φτάσει η βελτίωση αυτή αν μπορέσει να επιτευχτεί. Το σειριακό πρόγραμμα που επέλεξα για τις πειραματικές μου μετρήσεις είναι το TPC-H Q6 (Query-6). Το συγκεκριμένο επερώτημα προέρχεται από το TPC-H Benchmark (που είναι benchmark υποστήριξης αποφάσεων) και έχει επιλεγεί για το λόγο ότι το επερώτημα αυτό αποτελεί μια από τις σημαντικές διαδικασίες που συχνά συναντούμε σε βάσεις δεδομένων, έχει ευρύ βιομηχανικό ενδιαφέρον και ως επίσης γιατί μπορεί να εξετάσει μεγάλους όγκους δεδομένων. Πολλές ομάδες προγραμματιστών καθώς και εταιρείες ανά τον κόσμο προσπαθούν να βελτιώσουν τις επιδόσεις τέτοιων επερωτήσεων και οι κορυφαίες εξ αυτών αναρτώνται και στη σελίδα της TPC (Transaction Processing Performance Council).[25]

5.2 Περιγραφή Αλγορίθμου

Το επερώτημα TPC-H Q6[25] (Forecasting Revenue Change Query (Q6)) που επιλέξαμε υπολογίζει το ποσό αύξησης εισοδήματος που θα είχε προκύψει από την εξάλειψη ορισμένων επιχειρησιακών εκπτώσεων σε μια δεδομένη σειρά ποσοστού και ένα συγκεκριμένο έτος. Μπορεί να χρησιμοποιηθεί ώστε να κοιτάξουμε για τρόπους να αυξήσουμε εισοδήματα. Αναλυτικότερα το επερωτάμε επερώτημα εξετάζει όλα τα lineitems και ψάχνει για αυτά που στάλθηκαν το 1994 με έκπτωση μεταξύ 0,00 και 0,02 και ποσότητα μικρότερη από 24. Τα lineitems βρίσκονται σε ένα αρχείο βάσεων δεδομένων που δίνουμε στο πρόγραμμα μας σαν είσοδο (lineitem.tbl). Κάθε φορά που ισχύουν οι πιο πάνω συνθήκες για κάποιο στοιχείο προσθέτουμε σε ένα άθροισμα (αρχικοποιημένο με μηδέν) το γινόμενο της τιμής και του έτους που έχει αποσταλεί. Τα αποτελέσματα που επιστρέφει η πιο πάνω διαδικασίας είναι το άθροισμα που υπολογίζεται και ένας μετρητής που αυξάνετε κάθε φορά που ισχύουν οι συνθήκες μας. Στα πιο κάτω σχήματα απεικονίζεται η SQL μορφή του επερωτήματος και το επερώτημα σε σειριακό κώδικα γραμμένο στη γλώσσα προγραμματισμού C (5.1 και 5.2 αντίστοιχα).

```
select
        sum( l_extendedprice * l_discount )
        as revenue
from
        lineitem
where
        l_shipdate >= date ' [1994] '
        and l_shipdate < date ' [1995] '
        + interval '1 ' year
        and l_orderkey = o_orderkey
        and l_discount between
        [0.01] - 0.01 and + 0.01
        and l_quantity < [24] ;
```

Σχήμα 5.1:Η SQL μορφή του TPC-H Q6[25]

```
for(i = start; i < end; i++)
{
	if((line_item[2][i]>=DATE1) && (line_item[2][i]<DATE2)
		&& ((line_item[1][i]>(DISCOUNT-0.01)) && (line_item[1][i]<(DISCOUNT+0.01)))
			&& (line_item[3][i]<QUANTITY))
		{
			sum+=line_item[0][i]*line_item[2][i];

			result_count++;
		}

}
```

Σχήμα 5.2:C source code for computation part of TPC-H Q6

Απλώς αναφορικά οι τιμές των σταθερών που παρουσιάζονται στον πιο πανω κώδικα είναι οι εξής:

- DATE1 = 1994
- DATE2 = 1995
- DISCOUNT = 0,01
- QUANTITY = 24

Στην μορφή του επερωτήματος 6 σε κώδικα της γλώσσας προγραμματισμού C ο πίνακας `line_item` στον οποίο θα ανατεθούν τα στοιχεία από το αρχείο lineitem.tbl δηλώνετε σαν δισδιάστατος πίνακας μεγέθους LINEITMEMMX = 10 000000 γραμμών και 4 στηλών (`float line_item[LINEITEMX][4];`)αλλά στην πραγματικότητα οι γραμμές που χρησιμοποιούνται είναι αναλόγως του αριθμού των στοιχείων που περιέχει το αρχείο εισόδου και ο υπόλοιπος πίνακας είναι κενά στοιχεία.

5.3 Απαραίτητο υλικό και λογισμικό

Για να χρησιμοποιήσουμε τον MaxCompiler έπρεπε να έχουμε στην διάθεση μας μια μηχανή με όλα τα απαραίτητα ,όσον αφορά υλικό και λογισμικό, για να κάνουμε τις πειραματικές μας μετρήσεις κάτι που μας έχει προσφέρει η μηχανή larissa.inesc-id.pt η οποία βρίσκεται στο Πανεπιστήμιο INESC-ID στην Λισαβόνα της Πορτογαλίας.

Η Larissa λοιπόν έχει τις ακόλουθες προδιαγραφές:

Hardware	Software
Intel® Core™ i7 3930k @ 3.20GHz	**Linux Kernel 2.6.32-279.14.1.el6.x86_64**
DDR3 4x8GB @ 1,6GHz	**GCC 4.4.6**

Maxeler Data-flow engine MAX2336B with Virtex	Xilinx ISE 13.3
	MaxCompiler and MaxelerOS 2012.2.1

Πίνακας 1: Προδιαγραφές μηχανής

Και οι προδιαγραφές της Maxeler Card (FPGA) Maxeler Data-flow engine MAX2336B with Virtex [26]που χρησιμοποιούμε είναι:

Resource	Number of Resources available
Lookup Tables (LUTs)	207360
Flip-Flops (FFs)	207360
Block RAMs (BRAMs)	324
DSPs	192
DRAM (DRAM bandwidth)	24 GB (28GB/s)

Πίνακας 2: Προδιαγραφές DFE

Η κάρτα που χρησιμοποιούμε είναι μια Max2 γενιάς κάρτα. Επίσης έχει μέγιστη συχνότητα ρολογιού 243MHz αλλά η προεπιλεγμένη της είναι 75MHz. Περιγραφικά τα LUTs είναι μικρές συνδυαστικές μονάδες που ρυθμίζονται για να εκτελούν οποιαδήποτε λογική συνάρτηση, τα Flip-Flops χρησιμοποιούνται σαν καταχωρητές, οι BRAMs είναι χαμηλής καθυστέρησης on-chip μνήμες, τα DSP slices (Digital Signal Processing elements) είναι γρήγορα στοιχεία πολλαπλασιασμού και η DRAM είναι η onboard-μνήμη.

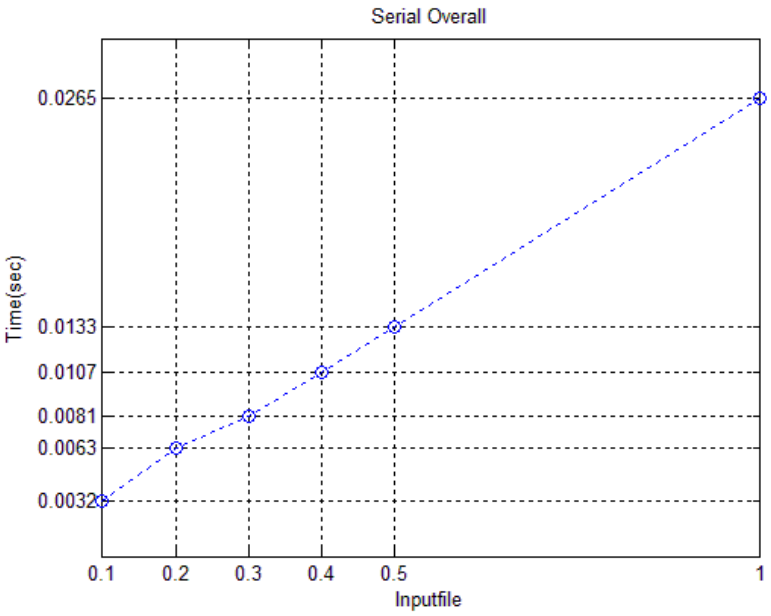
5.4 Πειραματικές μετρήσεις σειριακού προγράμματος

Πήραμε την σειριακή εφαρμογή και επιλέξαμε να την εκτελέσουμε για έξι διαφορετικά αρχεία εισόδου και με μια μικρή μετατροπή στον κώδικα καταμετρήσαμε την διάρκεια εκτέλεσης του κομματιού του κώδικα που μας ενδιαφέρει. Στον πιο κάτω Πίνακας 3 περιγράφονται τα αρχεία εισόδου όσον αφορά αριθμό στοιχείων του LINEITEM πίνακα και το μέγεθος του αρχείου εισόδου σε Megabytes.

A/A	Scale Factor	Number of Elements	Memory Residence (Mbytes)
1	0.1	600572	70.8
2	0.2	1201144	141.6
3	0.3	1801716	212.4
4	0.4	2402288	283.2
5	0.5	3002860	354
6	1	6005760	708.1

Πίνακας 3: Πληροφορίες αρχείων εισόδου

Εκτελέσαμε λοιπόν την σειριακή εφαρμογή για τα έξι αυτά αρχεία από είκοσι φορές για το καθένα και εξάγαμε την πιο κάτω γραφική παράσταση Σχήμα 5.3



Σχήμα 5.3: Χρόνοι εκτέλεσης σειριακού προγράμματος

Παρατηρούνται αρκετά χαμηλοί εώς και μηδαμινοί χρόνοι της σειριακής εκτέλεσης για όλα τα αρχεία εισόδου.

5.5 Μετατροπή αλγορίθμου σε MaxCompiler υλοποίηση και πειραματικές μετρήσεις

5.5.1 Φάση 1

Αφού μελετήθηκε ο αλγόριθμος της εφαρμογής μας θα πρέπει τώρα να ξεκινήσουμε την διαδικασία μετατροπής της C μορφής του επερωτήματος σε υλοποίηση για τον MaxCompiler εφαρμόζοντας τις απαραίτητες αλλαγές. Για την μετατροπή ενός προγράμματος θα πρέπει να ανευρεθεί το κομμάτι του κώδικα το οποίο εκτελείται πιο εντατικά το οποίο είναι ο αλγόριθμος που περιγράφεται στο υποκεφάλαιο Περιγραφή Αλγορίθμου5.2. Το κομμάτι αυτό λοιπόν θα πρέπει να αφαιρεθεί από τον κώδικα μας και να αντικατασταθεί αργότερα από την υλοποίηση μας στον MaxCompiler. Δημιουργήσαμε ένα νέο MaxCompiler project με την βοήθεια του MaxIDE και αντιγράψαμε τον κώδικα του επερωτήματος στο αρχείο του νέου μας project με όνομα query6CpuCode.c. Στο αρχείο αυτό εισάγουμε και συμπεριλαμβάνουμε τις απαραίτητες βιβλιοθήκες του MaxCompiler:

```
#include "Maxfiles.h"

#include "MaxSLiCInterface.h"
```

Στη υλοποίηση μου επέλεξα να μετατρέψω τον πίνακα line_item στον ανάστροφο του και ο πίνακας στην δική μου υλοποίηση έχει 4 γραμμές και LINEITEMX στήλες.

```
float line_item[4][LINEITEMX];
```

Ο λόγος που επέλεξα να μετατρέψω τον πίνακα είναι γιατί έτσι θα γινόταν ευκολότερη η μεταφορά των streams δεδομένων στο FPGA αφού θα έστελλα την διεύθυνση του αρχικού στοιχείου κάθε γραμμής και λόγο της stream σχεδίασης και υποδοχής δεδομένων αυτόματα ο MaxCompiler θα έπαιρνε σαν επόμενο στοιχείο το ανάλογο. Θα σταλούν δηλαδή στο υλικό τέσσερα streams εισόδου που το κάθε ένα θα αντιπροσωπεύει μία γραμμή του πίνακα και τα ανάλογα δεδομένα της κάθε γραμμής (π.χ. τιμή, έκπτωση, ημερομηνία, ποσότητα).

Αφού αφαιρέσουμε το κομμάτι του κώδικα που προαναφέραμε θα αντικατασταθεί στον κώδικα με μια κλήση SLiC (Basic Static) με το όνομα του πακέτου που θα ετοιμάσουμε για την υλοποίηση μας και τις παραμέτρους που επιθυμούμαι. Οι παράμετροι αυτοί είναι η μεταβλητή `end` που αντιπροσωπεύει των αριθμό επαναλήψεων του κώδικα του πυρήνα (Kernel) που θα υλοποιήσουμε καθώς επίσης και το μέγεθος σε στοιχεία των εισαγόμενων και εξαγόμενων streams. Τα streams αυτά θα είναι οι 4 γραμμές του πίνακα με end στοιχεία η καθεμιά και τα εξαγόμενα streams στα οποία θα εισαχθούν τα αποτελέσματα από την υλοποίηση μας και θα επιστραφούν στο κυρίως πρόγραμμα ένα για όλα τα αθροίσματα (`sum_table`) και ένα για τον μετρητή(`res_table`) του πόσες φορές ίσχυσαν οι συνθήκες στα δεδομένα που αποστείλαμε. Έτσι οι κλήση θα έχει ως ακολούθως:

```
query6(end,line_item[0],line_item[1],line_item[2],line_item[3]
,res_table,sum_table);
```

Μετά την εκτέλεση της πιο πάνω γραμμής κώδικα το μόνο που χρειάζεται να πράξουμε επιπλέον είναι να αθροίσουμε όλα τα στοιχειά των δύο πινάκων για να εξάγουμε τα τελικά μας αποτελέσματα όσον αφορά το τελικό άθροισμα και την τελική τιμή του μετρητή.

Αφού ολοκληρώσαμε τις αλλαγές που απαιτούνταν στον κώδικα του κυρίως προγράμματος θα πρέπει να προγραμματίσουμε και να διαμορφώσουμε τον διαχειριστή και τον πυρήνα αναλόγως των απαιτήσεων της εφαρμογής μας.

Θα ξεκινήσω από το αρχείο EngineParameters του οποίου ο κώδικας είναι πολύ απλός και 4 γραμμές μόνο.

```
package query;
import com.maxeler.maxcompiler.v2.build.EngineParameters;

public class query6EngineParameters extends EngineParameters {

      public query6EngineParameters(String[] args) {
            super(args);
      }}
```

Το μόνο που χρειάζεται απ' ότι βλέπετε είναι η συνάρτηση `super(args)` Που η χρήση της είναι η ίδια όπως και στην Java δηλαδή η κλήση της του supe-class κατασκευαστή του αντικειμένου engineParameters που καθορίζει σιωπηρά το DFE μοντέλο καθώς επίσης και το ποιο .max αρχείο θα καλεστεί αναλόγως της εκτέλεσης (RunRule) που έχει επιλεγεί.

Ακολούθως θα υλοποιήσουμε τον Kernel ο οποίος πρέπει να προγραμματιστεί ώστε να εκτελεί τις λειτουργίες μιας επανάληψης του βρόγχου του αλγόριθμου μας. Πρώτο βήμα η δήλωση της κλάσης του πυρήνα μας κι ότι κληρονομεί από την κλάση Kernel. Ακολούθως επέλεξα να δηλώσω ένα DFEType για να μην χρειάζεται συνέχεια να γράφω τον τύπο στην κανονική του μορφή `dfeFloat(8,24)` που όπως έχουμε πει σημαίνει κινητής διαστολής αριθμός 32 bits 8 για το ακέραιο μέρος και 24 bits για το δεκαδικό και η δήλωση του kernel με τις παραμέτρους του.

```
class query6Kernel extends Kernel {
private static final DFEType type = dfeFloat(8, 24);
protected query6Kernel(KernelParameters parameters, int nx,
double discount){ super(parameters)
```

Στα ουσιώδη τώρα, όπως προαναφέραμε έχουμε αποστείλει τις 4 γραμμές του πίνακα line_item σαν τα εισαγόμενα μας stream. Υπάρχουν δύο τρόποι να τα διαχειριστούμε και να τα αναθέσουμε σε DFEVar αντικείμενα. Ο πρώτος είναι ο απεικονιζόμενος στο Σχήμα 5.4 με τον οποίο παίρνουμε τις εισόδους σαν stream και τις αναθέτουμε σε αντικείμενα DFEVar και ακολούθως με την μέθοδο stream.offset(str,off) μπορούμε να μετακινούμαστε σε αυτά μπρος και πίσω για να μεταβούμε στη θέση του στοιχείου που επιθυμούμε και να το αναθέσουμε σε ένα DFEVar αντικείμενο.

```
DFEVar inStream = io.input("inStream", type);
DFEVar inStream1 = io.input("inStream1", type);
DFEVar inStream2 = io.input("inStream2", type);
DFEVar inStream3 = io.input("inStream3", type);


DFEVar s0=stream.offset(inStream, 0);
DFEVar s1=stream.offset(inStream1, 0);
DFEVar s2=stream.offset(inStream2, 0);
DFEVar s3=stream.offset(inStream3, 0);
```

Σχήμα 5.4: Stream offset

Αφού όμως η θέση του stream που θα επιθυμούμε θα είναι πάντα η παρούσα δεν προτιμάται ο πιο πάνω τρόπος αλλά η απευθείας ανάθεση στις DFEVar.

```
DFEVar s0=io.input("s0", type);
DFEVar s1=io.input("s1", type);
DFEVar s2=io.input("s2", type);
DFEVar s3=io.input("s3", type);
```

Σχήμα 5.5: Ανάθεση τιμών στα DFEVar αντικείμενα

Δηλώνουμε λοιπόν 4 DFEVar αντικείμενα (s0,s1,s2,s3) που το καθένα παίρνει το στοιχείο από την γραμμή με τον αντίστοιχο αριθμό.

Ακολούθως θα πρέπει να γίνουν οι έλεγχοι που γίνονται και στον αλγόριθμο για να ελεγχθεί εάν ισχύουν οι απαραίτητε συνθήκες και να δηλωθούν ακόμη δύο DFEVar αντικείμενα τα οποία θα πάρουν και την ανάλογη τιμή. Ο MaxCompiler επιτρέπει τις λογικές εκφράσεις και τις συγκρίσεις DFEVar με float τιμές αφού στην ουσία έχουν τον ίδιο τύπο άρα θα τις χρησιμοποιήσουμε. Επίσης θα γίνει χρήση του multiplexer if-statement. Δηλώσαμε λοιπόν δύο DFEVars (sum και res) και θέσαμε ότι εάν ισχύουν οι απαραίτητες συνθήκες μας το sum θα ισούται με s0*s2 και το res με ένα αλλιώς και το δύο θα ισούνται με μηδέν και χρησιμοποιήσαμε 2 multiplexer if-statement ένα για το καθένα και ακολούθως στέλνουμε τα αποτελέσματα τους σαν εξόδους του πυρήνα μας.

```
DFEVar sum=((s2>=1994.0)&(s2<1995.0)
&(s1>(discount2-0.01))&(s1<=(discount2+0.01))
&(s3<24.0))? s0*s2 : 0;


DFEVar res=((s2>=1994.0)&(s2<1995.0)
&(s1>(discount2-0.01))&(s1<=(discount2+0.01))
&(s3<24.0))?1:0;

io.output("s", sum, type);
```

```
io.output("r", res,type);
```

Μετά την ολοκλήρωση της διαμόρφωσης του πυρήνα μας θα πρέπει τώρα να ετοιμάσουμε και την διαμόρφωση του διαχειριστή της υλοποίησης μας. Αρχικά γίνεται η δήλωση της κλάσης του διαχειριστή μας σαν public `class query6Manager` και συνεχίζουμε με την δήλωση οποιονδήποτε σταθερών επιθυμούμε. Στην περίπτωση μας επιθυμούμε την δήλωση τριών μία για την ονομασία του πυρήνα μας, μια για το μέγιστο αριθμό στοιχείων που μπορεί να περιέχει ένα stream που είναι ίσο με τη σταθερά του κυρίως προγράμματος μας LINEITEMX και επίσης δηλώσαμε την έκπτωση (0,01).

```
private static final String s_kernelName = "afKernel";
private static int nxMax=10000000;
private static final double DISCOUNT= 0.01;
```

Κάθε διαχειριστής έχει την δική του συνάρτηση main που εκτελούνται η απαραίτητες λειτουργίες έτσι και ο δικός μας και θα ακολουθηθούν τα βήματα όπως τα περιγράψαμε πιο πάνω (4.1.3MaxCompiler Managers (Διαχειριστές)). Ξεκινούμε από την δημιουργία των παραμέτρων της μηχανής, ακολούθως δηλώνετε ο διαχειριστής με τις παραμέτρους του, μετά δηλώνετε και θέτετε ο πυρήνας με τις παραμέτρους του, καθορίζονται οι είσοδοι/εξόδοι της υλοποίησης μας (όλοι από το CPU), καλούμαι την δημιουργία της SLiC διεπαφής (με παράμετρο τη συνάρτηση `interfaceDefault()`) και τέλος την μέθοδο κτισίματος.

```
     public static void main(String[] args) {
query6EngineParameters parms=new query6EngineParameters(args);
     Manager manager = new Manager(parms);
     Kernel kernel   = new query6Kernel(
manager.makeKernelParameters(s_kernelName),nxMax,DISCOUNT2);

manager.setKernel(kernel);
manager.setIO(        link("s0",   IODestination.CPU),
                      link("s1",   IODestination.CPU),
                      link("s2",IODestination.CPU),
                      link("s3",IODestination.CPU),
                 link("s", IODestination.CPU),
                 link("r", IODestination.CPU));

           manager.createSLiCinterface(interfaceDefault());
           manager.build();}
```

Η `interfaceDefault()` συνάρτηση ο κώδικας της οποίας φαίνεται πιο κάτω είναι απαραίτητη και αρκετά χρήσιμη για να καθορίσουμε κάποιες παραμέτρους όπως των αριθμό των ticks ή αν προτιμάτε πόσες φορές θα επαναλάβει την διαδικασία εκτέλεσης του κώδικα του ο πυρήνας μας (`setTicks(s_kernelName, N);`), καθώς επίσης να θέσουμε το μέγεθος σε bytes και τον τύπο το streams που θα εισαχθούν και θα εξάχθουν.

```
private static EngineInterface interfaceDefault() {
      EngineInterface engine_interface = new EngineInterface();
      CPUTypes   type = CPUTypes.FLOAT;
      int        size = type.sizeInBytes();

InterfaceParam N=engine_interface.addParam("N", CPUTypes.INT);

      engine_interface.setTicks(s_kernelName, N);
      engine_interface.setStream("s0",   type, N * size);
      engine_interface.setStream("s1",   type, N * size);
      engine_interface.setStream("s2",   type, N * size);
      engine_interface.setStream("s3",   type, N * size);
      engine_interface.setStream("s", type, N * size);
      engine_interface.setStream("r",type,N*size);
      return engine_interface;}
```

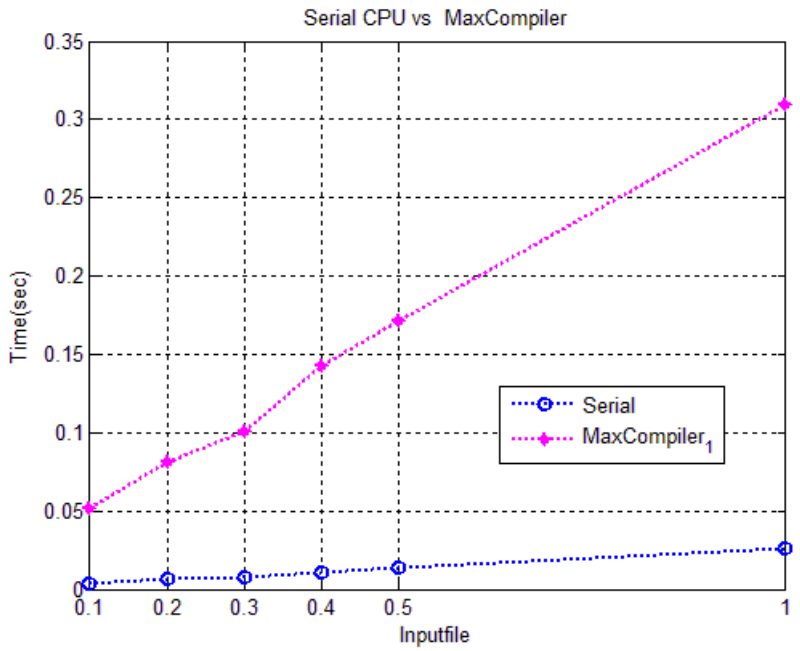
Η συνάρτηση επιστρέφει ένα τύπου `EngineInterface` αντικείμενο με όλα όσα προαναφέραμε και θα είναι παράμετρος της συνάρτησης `createSLiCinterface`.

Αφού συγγράψαμε και προγραμματίσαμε όλα τα απαραίτητα κομμάτια της MaxCompiler υλοποίησης μας ελέγξαμε την ορθή λειτουργία του προγράμματος μας τρέχοντας την προσομοίωση του και αφού βεβαιωθήκαμε για την ορθότητα του το μεταγλωττίσαμε και για το πραγματική εκτέλεση του στο Maxeler DFE.

```
FINAL RESOURCE USAGE
LUTs: 14077 / 207360 (6.79%)
FFs: 18255 / 207360 (8.80%)
BRAMs: 49 / 324 (15.12%)
DSPs: 2 / 192 (1.04%)
```

Σχήμα 5.6: Resource Usage Report Φάσης 1

Το επόμενο βήμα ήταν να εκτελέσουμε την υλοποίηση μας για τα έξι αρχεία εισόδου που επιλέξαμε και να μετρήσουμε την διάρκεια εκτέλεσης του κομματιού που υλοποιήθηκε στον MaxCompiler ούτος ώστε να κάνουμε σύγκριση με το σειριακό. Εκτελέσαμε λοιπόν την εφαρμογή μας είκοσι φορές για κάθε αρχείο εισόδου, υπολογίσαμε τον μέσο όρο της κάθε εκτέλεσης για κάθε αρχείο εισόδου και παράξαμε την πιο κάτω γραφική παράσταση (Σχήμα 5.7) στην οποία συγκρίνουμε τους χρόνους εκτέλεσης της σειριακής εφαρμογής και της δικής μας υλοποίησης.



Σχήμα 5.7: Σύγκριση σειριακού με MaxCompiler

Τα αποτελέσματα δεν μας αφήνουν ικανοποιημένους καθώς το σειριακό πρόγραμμα είναι κατά πολύ γρηγορότερο από την MaxCompiler υλοποίηση. Δοκίμασα επίσης να εκτελέσω την υλοποίηση αλλάζοντας την SLiC διεπαφή από Basic Static σε Advanced Static (Σχήμα 5.8) και Advanced Dynamic (Σχήμα 5.9) όμως και στις δύο περιπτώσεις δεν είχαμε καμιά βελτίωση στην απόδοση. Μάλιστα στην Advanced Dynamic είχα μετρήσει το χρόνο των διάφορων κομματιών της κλήσης εκτέλεσης και διαπίστωσα στις μετρήσεις που έκανα ότι όσο χρόνο χρειάζεται η εκτέλεση του σειριακού κώδικα με αρχείο εισόδου inp_1 (~0,027) ο MaxCompiler χρειάζεται περίπου για την φόρτωση του .max αρχείου και τον καθορισμό των παραμέτρων στην μηχανή και συγκεκριμένα μεταξύ 0,017 – 0,020 σε δευτερόλεπτα για όλα τα αρχεία εισόδου. Βάση αυτού του γεγονότος φαντάζει αδύνατη η βελτίωση της σειριακής εκτέλεσης της συγκεκριμένης εφαρμογής(επερωτήματος) με την χρήση του MaxCompiler. Παρόλα αυτά θα συνεχίσουμε την προσπάθεια για βελτίωση της πρώτης μας αυτής υλοποίησης χρησιμοποιώντας διαφορετικές υλοποιήσεις και μεθόδους του MaxCompiler.

```
max_file_t *maxfile = query6_init();
max_engine_t *engine = max_load(maxfile, "*");
query6_actions_t run_q6;
run_q6.param_N = end;
run_q6.instream_s0 = line_item[0];
run_q6.instream_s1 = line_item[1];
run_q6.instream_s2 = line_item[2];
run_q6.instream_s3 = line_item[3];
run_q6.outstream_r = res_table;
run_q6.outstream_s = sum_table;

query6_run(engine, &run_q6);
max_unload(engine);
```

Σχήμα 5.8: Advanced Static SLiC interface

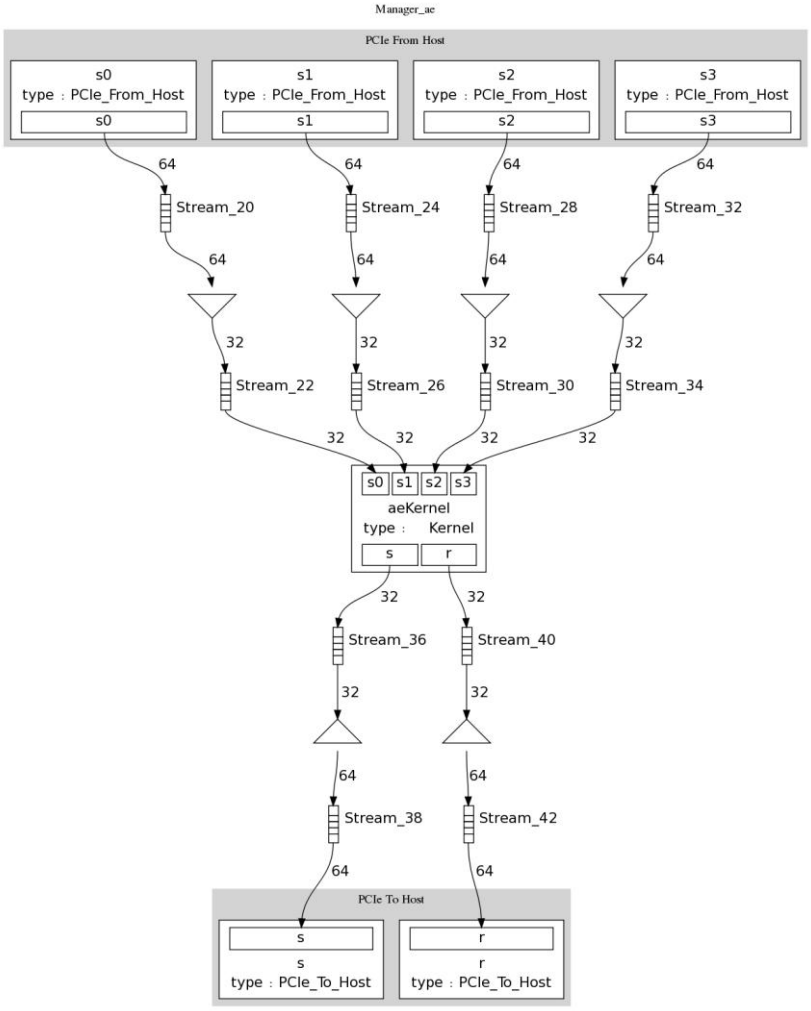
```
max_file_t *maxfile = query6_init();
      max_engine_t *engine = max_load(maxfile, "*");
max_actions_t* act = max_actions_init(maxfile, "default");
      max_set_param_uint64t(act, "N", end);
max_queue_input(act, "s0", line_item[0], end*sizeof(float));
max_queue_input(act, "s1", line_item[1], end*sizeof(float));
max_queue_input(act, "s2", line_item[2], end*sizeof(float));
max_queue_input(act, "s3", line_item[3], end*sizeof(float));
max_queue_output(act, "s", sum_table, end*sizeof(float));
      max_run(engine, act);
            max_unload(engine);
```

Σχήμα 5.9: Advanced Dynamic SLiC interface

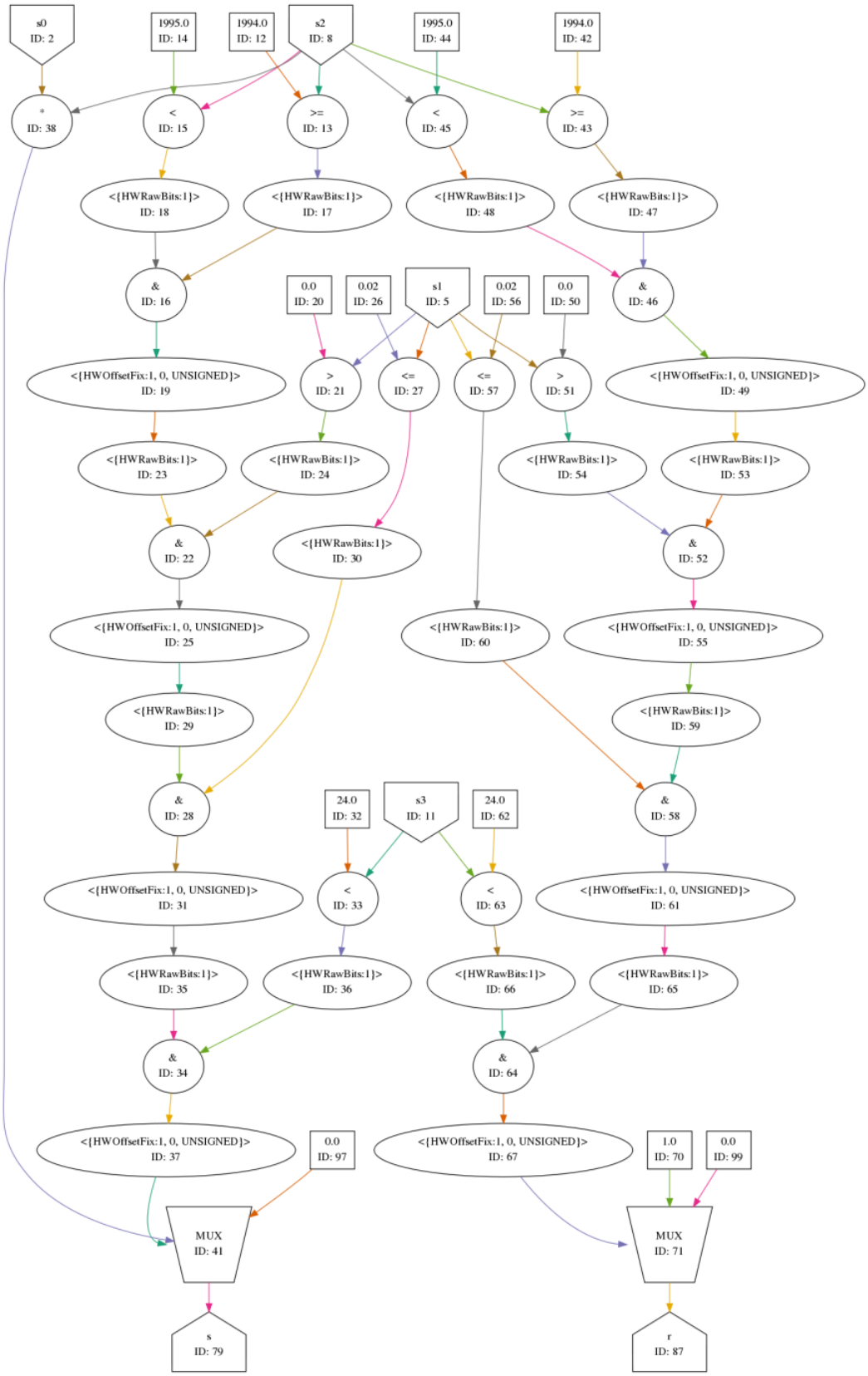
Στην παρούσα υλοποίηση σκέφτηκα επίσης να αλλάξω την συχνότητα ρολογιού του FPGA που χρησιμοποιούμε από 75MHz που είναι προεπιλεγμένο στην μέγιστη δυνατή του τιμή που είναι τα 243MHz και δοκίμασα την συγκεκριμένη αλλαγή σε εκτελέσεις με το αρχείο εισόδου inp_1 (μεγαλύτερο) ώστε να ελέγξω τυχόν βελτίωση. Η βελτίωση επετεύχθητε όμως δεν μπορεί να χαρακτηριστεί σε καμιά περίπτωση ικανοποιητική ιδιαίτερα αν ληφθεί υπόψη σαν παράμετρος η κατανάλωση ενέργειας. Οι χρόνοι που μετρήθηκαν σε 20 εκτελέσεις με μέγιστη συχνότητα ρολογιού ήταν όλοι κάτω από 0,30 δευτερόλεπτα με γρηγορότερο το 0,294063 και υψηλότερο το

0,299594 ενώ ο μέσος όρος με την προεπιλεγμένη συχνότητα των 75MHz ήταν 0,310056 δευτερόλεπτα.

Επίσης έχω χρησιμοποιήσει την εντολή maxRenderGraph για να παράξω τα γραφήματα απεικόνισης του Kernel και του Manager της εφαρμογής μου τα οποία παρουσιάζονται στα πιο κάτω σχήματα (Σχήμα 5.10 , Σχήμα 5.11).



Σχήμα 5.10: Γράφημα Manager Φάσης 1



Σχήμα 5.11: Γράφημα Kernel Φάσης 1

5.5.2 Φάση 2 Δύο Πυρήνες

Συνεχίσαμε λοιπόν την προσπάθεια μας για βελτίωση της απόδοσης της εφαρμογής και στη φάση αυτή θα χρησιμοποιήσουμε δύο αντί για ένα πυρήνα στην υλοποίηση μας μοιράζοντας τους τα δεδομένα εισόδου έτσι ώστε να ελπίσουμε σε καλύτερη επίδοση της προηγούμενης.

Για την μετατροπή αυτή δεν θα γίνει καμιά αλλαγή στον κώδικα του πυρήνα μας αφού επιθυμούμαι να εκτελεί ακριβώς τις ίδιες λειτουργίες με προηγουμένως όμως θα πρέπει να γίνουν αλλαγές στους κώδικες του διαχειριστή και του κυρίως προγράμματος.

Θα ξεκινήσουμε από τις αλλαγές στον κώδικα του διαχειριστή. Εδώ αντί για την δήλωση ενός kernel θα γίνει η δήλωση δύο kernels οι οποίοι θα δηλωθούν σαν KernelBlock αντικείμενα και δοθεί διαφορετικό όνομα στον καθένα. Σημαντικό είναι επίσης να αναφέρουμε ότι στην υλοποίηση αυτή δεν χρησιμοποιούμε τον Standard Manager αλλά δημιουργούμε τον δικό μας Custom Manager.

```
public class q6Manager extends CustomManager{

KernelBlock k1 = addKernel(new q6Kernel
(makeKernelParameters("q6Kernel1"),nxMax,DISCOUNT));
KernelBlock k2 = addKernel(new q6Kernel
(makeKernelParameters("q6Kernel2"),nxMax,DISCOUNT));
```

Ακολούθως θα δηλωθούν οκτώ DFELink όσα και τα streams εισόδου στην υλοποίηση αυτή και ακολούθως να διαμοιραστούν στους δύο kernels καθορίζοντας στο καθένα σε ποιο DFEVar θα ανατεθεί. Τα οκτώ streams εισόδου που θα αποσταλούν από το CPU οι 4 γραμμές του πίνακα μας μέχρι την μέση και ξανά οι 4 γραμμές του πίνακα line_item από τη μέση και μετά.

```
DFELink s10 = addStreamFromCPU("s10");
DFELink s11 = addStreamFromCPU("s11");
DFELink s12 = addStreamFromCPU("s12");
DFELink s13 = addStreamFromCPU("s13");

DFELink s20 = addStreamFromCPU("s20");
DFELink s21 = addStreamFromCPU("s21");
DFELink s22 = addStreamFromCPU("s22");
DFELink s23 = addStreamFromCPU("s23");


k1.getInput("s0")<==s10;
k1.getInput("s1")<==s11;
k1.getInput("s2")<==s12;
k1.getInput("s3")<==s13;
```

```
k2.getInput("s0")<==s20;
k2.getInput("s1")<==s21;
k2.getInput("s2")<==s22;
k2.getInput("s3")<==s23;
```

Επίσης θα δηλωθούν ακόμη τέσσερα DFELink για τα εξαγόμενα streams που μέσω αυτών θα αποσταλούν τα αποτελέσματα στο κυρίως πρόγραμμα, και για αυτά ισχύει η διαμοίραση και ανάθεση τους στους 2 μας kernels καθορίζοντας ότι οι εξόδοι των kernels θα ανατεθούν σ' αυτά και ακολούθως θα επιστραφούν στο κυρίως πρόγραμμα.

```
DFELink sum1=addStreamToCPU("sum1");
DFELink r1 =addStreamToCPU("r1");

DFELink sum2=addStreamToCPU("sum2");
DFELink r2 =addStreamToCPU("r2");

sum1<== k1.getOutput("sum");
r1<== k1.getOutput("r");

sum2<== k2.getOutput("sum");
r2<== k2.getOutput("r");
```

Στην συνάρτηση interfaceDefault() έχουμε ακριβώς τις ίδιες λειτουργίες με αυτές της Φάσης 1 άλλα αυτή τη φορά εις διπλούν αφού έχουμε διπλασιάσει τους πυρήνες μας και τα streams εισόδου / εξόδου.

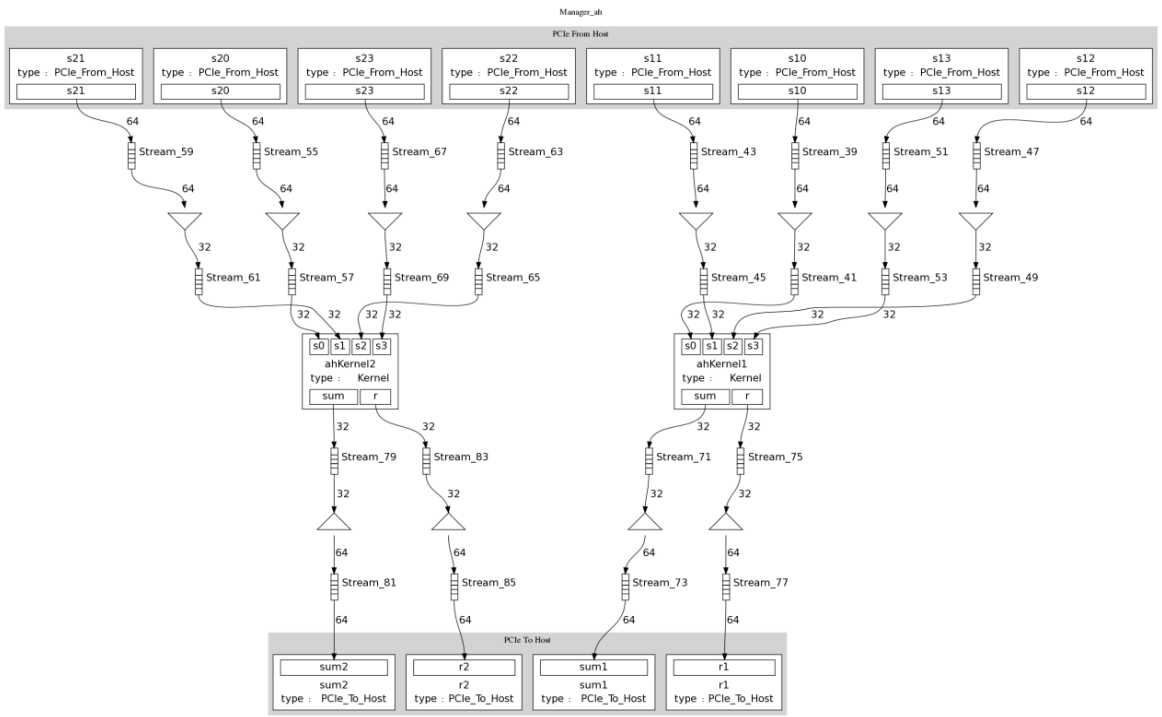
```
private static EngineInterface interfaceDefault() {
     EngineInterface engine_interface = new EngineInterface();
          CPUTypes type = CPUTypes.FLOAT;
          int size = type.sizeInBytes();

InterfaceParam N=engine_interface.addParam("N", CPUTypes.INT);

          engine_interface.setTicks("ahKernel1", N);
          engine_interface.setTicks("ahKernel2", N);


          engine_interface.setStream("s10", type, N * size);
          engine_interface.setStream("s11", type, N * size);
          engine_interface.setStream("s12", type, N * size);
          engine_interface.setStream("s13", type, N * size);
          engine_interface.setStream("sum1", type, N * size);
          engine_interface.setStream("r1", type, N * size);

          engine_interface.setStream("s20", type, N * size);
          engine_interface.setStream("s21", type, N * size);
          engine_interface.setStream("s22", type, N * size);
          engine_interface.setStream("s23", type, N * size);
          engine_interface.setStream("sum2", type, N * size);
          engine_interface.setStream("r2", type, N * size);
```

```
            return engine_interface;
      }
```

Και τέλος για τον κώδικα του διαχειριστή μας παρουσιάζουμε το περιεχόμενο της main του.

```
public static void main(String[] args) {
      q6EngineParameters params = new q6EngineParameters(args);
      q6Manager manager = new q6Manager(params);
      manager.createSLiCinterface(interfaceDefault());
      manager.build();
      }
```

Εκτελέσαμε και πάλι την εντολή maxRenderGraph για να παράξουμε τα γραφήματα των Kernels και του Manager μας. Τα γραφήματα των δύο kernels είναι ακριβώς τα ίδια με το γράφημα του Kernel στη Φάση 1 όπως παρουσιάζεται στο Σχήμα 5.11 .όμως το γράφημα του διαχειριστή μας (Σχήμα 5.12) διαφέρει αφού σε αυτή την υλοποίηση τα εισαγόμενα (8) streams μοιράζονται σε δύο πυρήνες και στην ουσία ο κάθε πυρήνας αναλαμβάνει από μισό υπολογισμό (πίνακα). Επίσης με την ολοκλήρωση της εκτέλεσης θα πρέπει και πάλι να αθροιστούν τα στοιχεία των εξαγόμενων streams για να έχουμε τα τελικά αποτελέσματα (sum, result_count) της εφαρμογής.



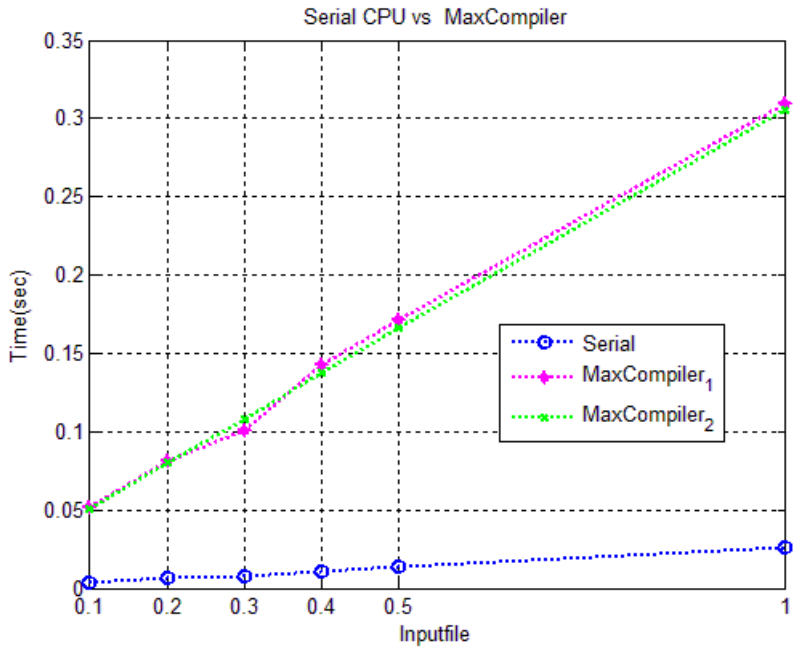
Σχήμα 5.12: Γράφημα Manager Φάσης 2

Για τις αλλαγές στο κυρίως πρόγραμμα σε σχέση με την Φάση 1 είναι στην κλήση εκτέλεσης που θα μετατραπεί ως:

```
q6(end/2,line_item[0],line_item[1],line_item[2],line_item[3],
line_item[0]+(end/2),line_item[1]+(end/2),line_item[2]+(end/2)
,line_item[3]+(end/2),res_table1,res_table2,sum_table1,
sum_table2);
```

άρα θα χρειαστεί η επιπρόσθετη δήλωση 4 νέων πινάκων μεγέθους end/2 και η μη δήλωση των 2 πινάκων της προηγούμενης φάσης μεγέθους end στοιχείων.

Την υλοποίηση μας αυτή έχουμε επίσης εκτελέσει και με τα έξι αρχεία εισόδου (20 φορές για το καθένα) ώστε να ελέγξουμε το χρόνο εκτέλεσης του αλγόριθμου μας και να τον συγκρίνουμε με τους χρόνους της σειριακής εκτέλεσης και της πρώτης μας υλοποίησης στη Φάση 1. Υπολογίζοντας τον μέσο όρο του χρόνου εκτέλεσης παράξαμε μια νέα γραφική παράσταση στην οποία παρουσιάζεται οι σύγκριση των τριών υλοποιήσεων στο Σχήμα 5.13



Σχήμα 5.13: Γραφική παράσταση Σειριακού vs Φάσης 1 vs Φάσης 2

Παρατηρώντας την γραφική παράσταση στο Σχήμα 5.13 και του μέσου όρου των χρόνων εκτέλεσης η υλοποίηση της Φάσης 2 έχει ελαφρώς καλύτερους χρόνους από την πρώτη υλοποίηση για όλα τα αρχεία εισόδου πλην των χρόνων για τις εκτελέσεις με το αρχείο εισόδου inp_0,3. Αναλυτικότερα οι χρόνοι εκτέλεσης παρουσιάζονται στον Πίνακας 4.

	0.1	0.2	0.3	0.4	0.5	1
Σειριακό	**0.003181**	**0.006326**	**0.008055**	**0.010733**	**0.013342**	**0.026516**
MaxCompiler$_1$	**0.051355**	**0.081112**	**0.100378**	**0.143062**	**0.171150**	**0.310056**
MaxCompiler$_2$	**0.050705**	**0.080724**	**0.108381**	**0.137385**	**0.166592**	**0.305905**

Πίνακας 4: Μέσος Όρος Χρόνου Εκτέλεσης Σειριακού, Φάσης1 και Φάσης2

Η ελαφρά αυτή βελτίωση δεν μπορεί να χαρακτηριστεί ικανοποιητική αφού καταρχάς δεν είναι εμφανής και κατά δεύτερον κυμαίνεται στα ίδια επίπεδα απόστασης από την σειριακή. Η μη εμφάνιση βελτίωσης των επιδόσεων στην υλοποίηση αυτή οφείλεται κυρίως στο μεγάλο για την συγκεκριμένη εφαρμογή (σχετικά με τους χρόνους σειριακής εκτέλεσης) communication και loading .max αρχείο overhead.

```
FINAL RESOURCE USAGE
LUTs: 20677 / 207360 (9.97%)
FFs: 26674 / 207360 (12.86%)
BRAMs: 71 / 324 (21.91%)
DSPs: 4 / 192 (2.08%)
```

Σχήμα 5.14: Resource Usage Report Φάσης 2

Στην αναφορά χρήσης πόρων παρατηρούμαι δέσμευση και χρήση περισσότερων πόρων κάθε είδους και αυτό είναι λογικό λόγω της χρήσης δύο πυρήνων και των περισσοτέρων εισαγόμενων και εξαγόμενων streams.

5.5.3 Φάση 3 (4 πυρήνες με 2 κλήσεις)

Αναλογιζόμενος κάποιος θα έλεγε αφού έχουμε έστω αυτή την ελάχιστη βελτίωση με το να μοιράσουμε τον πίνακα δεδομένων και τους υπολογισμούς στην Φάση 2 ας συνεχίσουμε αυτή την διαδικασία τεμαχισμού του πίνακα line_item και σπάσιμο του υπολογισμού σε πολλαπλούς πυρήνες μέχρι να επιτύχουμε ικανοποιητική βελτίωση σε σχέση με την πρώτη υλοποίηση. Αυτό όμως δεν μπορεί να επιτευχθεί καθώς οι Max2 κάρτες της Maxeler (όπως την δική μας) δέχονται ως μέγιστο αριθμό εισαγόμενων streams από το CPU οκτώ όσες δηλαδή έχουμε στην υλοποίηση της Φάσης 2. Έτσι σκέφτηκα να πράξω κάτι διαφορετικό το οποίο μου κινείσαι το ενδιαφέρον κατά πόσο θα μας προσέφερε καλύτερη ή χειρότερη επίδοση. Παίρνω λοιπόν ακριβώς την ίδια υλοποίηση με αυτήν της Φάσης 2 αλλά θα κάνω δύο φορές την κλήση της εκτέλεσης, την πρώτη για το πρώτο μισό του πίνακα και για την ακρίβεια για τα πρώτα δύο τέταρτα και την δεύτερη για τα υπόλοιπα δύο τέταρτα. Άρα λοιπόν θα πράξουμε την εξής αλλαγή στον κώδικα του κυρίως προγράμματος.

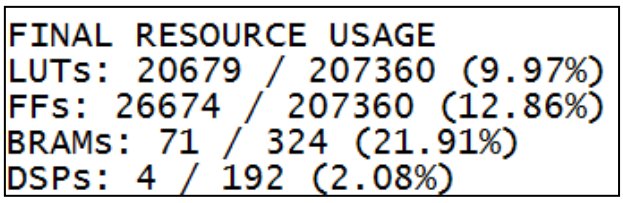
Πρώτη κλήση:

```
q6(end/4,line_item[0],line_item[1],line_item[2],line_item[3],
line_item[0]+(end/4),line_item[1]+(end/4),line_item[2]+(end/4)
,line_item[3]+(end/4),res_table1,res_table2,sum_table1,
sum_table2);
```

Δεύτερη κλήση:

```
q6(end/4,line_item[0]+(end/2),line_item[1]+(end/2),
line_item[2]+(end/2),line_item[3]+(end/2),
line_item[0]+(end*3/4),line_item[1]+(end*3/4),
line_item[2]+(end*3/4),line_item[3]+(end*3/4),
res_table3,res_table4,sum_table3, sum_table4);
```

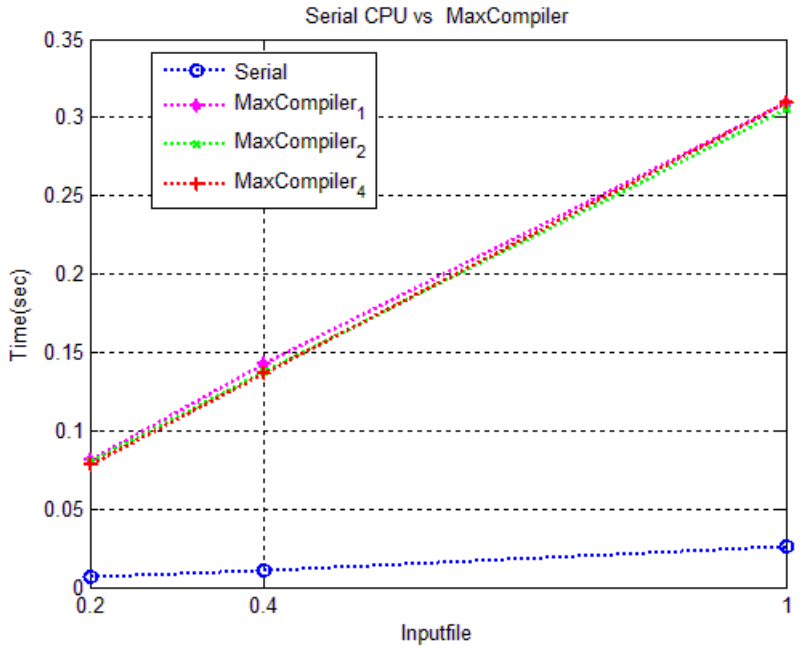
Οι δύο κλήσεις είναι γραμμένες στον κώδικα η μία κάτω από την άλλη. Τα γραφήματα πυρήνα και διαχειριστή δεν διαφέρουν με αυτά της Φάσης 2. Η συγκεκριμένη υλοποίηση εκτελέστηκε για 20 φορές για τρία αρχεία εισόδου (inp_0.2, 0.4 και 1) και παρουσιάστηκαν πανομοιότυποι χρόνοι με τις δύο προηγούμενες υλοποιήσεις. Στην μέτρηση του χρόνου με το μεγαλύτερο αρχείο εισόδου (inp_1) που η διάρκεια εκτέλεσης του είναι κοντά στα 0.30 δευτερόλεπτα η πρώτη κλήση εκτελείται κατά μέσο όρο 0,16 και η δεύτερη κατά 0,14. Η αναφορά χρήσης πόρων (Resource Usage Report) είναι σχεδόν η ίδια με αυτή της Φάσης 2 με μόνη διαφορά ότι χρησιμοποιούνται 2 περισσότερα LUTs (lookup tables).



Σχήμα 5.15: Resource Usage Report Φάσης 3

5.6 Τελικά αποτελέσματα

Τέλος θα συγκρίνουμε και τις τέσσερις υλοποιήσεις για τα τρία αρχεία εισόδου που εκτελέσαμε την υλοποίηση της Φάσης 3. Οι χρόνοι εκτέλεσης και των τριών υλοποιήσεων μας στον MaxCompiler έχουν πανομοιότυπους χρόνους εκτέλεσης όμως καμία δεν πλησιάζει τον χρόνο εκτέλεσης του σειριακού. Καταφέραμε επιτυχώς όμως να δείξουμε πως μια εφαρμογή μπορεί να μετατραπεί ,σε υλοποίηση MaxCompiler, και να εκτελεστεί με διαφορετικούς τρόπους. Στην πιο κάτω γραφική παράσταση παρουσιάζεται η σύγκριση των τεσσάρων υλοποιήσεων.



Σχήμα 5.16: Γραφική παράσταση σύγκρισης των 4 εφαρμογών μας

Λόγω του ότι δεν διακρίνονται με ευκολία οι γραμμές για τις τρεις εκτελέσεις του MaxCompiler στη γραφική παράσταση ένα πίνακα με τους μέσους όρους των χρόνων εκτέλεσης τους.

	0.2	**0.4**	**1**
Σειριακό	0.006326	0.010733	0.026516
MaxCompiler$_1$	0.081112	0.143062	0.310056
MaxCompiler$_2$	0.080724	0.137385	0.305905
MaxCompiler$_4$	0.078005	0.136683	0.309802

Πίνακας 5: Μέσος Όρος χρόνου εκτέλεσης όλων των υλοποιήσεων

Μέσω της γραφικής παράστασης και του πίνακα των τελικών μας αποτελεσμάτων παρατηρείται ότι η υλοποίηση της Φάσης 3 έχει τους καλύτερους χρόνους για τις εκτελέσεις με τα αρχεία εισόδου inp_0.2 και inp_0.4 ενώ για το μεγαλύτερο αρχείο εισόδου του καλύτερους χρόνους έχει η υλοποίηση της Φάσης 2. Αρνητικό μπορεί να χαρακτηριστεί το γεγονός ότι, εκτός του ότι οι χρόνοι των υλοποιήσεων μας είναι πολύ κοντινοί μεταξύ τους, απέχουν πάρα πολύ από τους χρόνους της σειριακής εφαρμογής και ίσως εν τέλει αποδεικνύεται όχι και η καταλληλότερη επιλογή εφαρμογής για να υλοποιηθεί μέσω του εργαλείου του MaxCompiler. Ο χρόνος επικοινωνίας και η καθυστέρηση φόρτωσης του .max αρχείου έχουν μεγάλο αρνητικό αντίκτυπο στο συνολικό χρόνο εκτέλεσης στις υλοποιήσεις του MaxCompiler.

Κεφάλαιο 6 Συμπεράσματα

6.1 Συμπεράσματα

Με την ολοκλήρωση της εργασίας μας είμαστε σε θέση να εξάγουμε τα γενικά μας συμπεράσματα όσον αφορά την γενική χρήση των FPGAs σαν επιταχυντές υλικού καθώς και τα συμπεράσματα μας για την νέα αυτή τεχνολογίας μεταγλώττισης που μας προσφέρεται μέσω του εργαλείου MaxCompiler για αξιοποίηση τους.

Μέσα από την μελέτη και ανάλυση σχετικών εργασιών και δημοσιεύσεων γύρω από το θέμα μας προκύπτει η τάση για περαιτέρω χρησιμοποίηση των FPGAs σε συστήματα βάσεων δεδομένων για την βελτιστοποίηση διαφόρων ειδών επερωτημάτων η οποία δεν μπορεί να μπορεί να επιτευχθεί από καμιά υπολογιστική τους μονάδα. Μπορούν να χρησιμοποιηθούν είτε για ταχύτερη προσκόμιση των δεδομένων από την πηγή τους είτε και σαν συνεπεξεργαστές. Η streaming σχεδίαση και η διαφορετικότητα λειτουργίας τους μαζί με την βαθιά διασωλήνωση που προσφέρουν, με σωστή τους αξιοποίηση μπορεί να επιτευχθεί τεράστια επιτάχυνση σε σχέση με την παραδοσιακή εκτέλεση σε επεξεργαστή/ες ιδιαίτερα στην επεξεργασίας μεγάλων όγκων δεδομένων. Οι μικρές υπολογιστικές μονάδες των FPGAs δεν πρέπει να υποτιμόνται σε καμία περίπτωση αφού παρόλο που μπορούν να εκτελέσουν μόνο απλές λογικές συνήθως εκφράσεις δίνουν την δυνατότητα να εκτελούνται εκατομμύρια τέτοιες ανά κύκλο ρολογιού στο FPGA και γενικώς την μέγιστη αξιοποίηση των διαθέσιμων πόρων τέτοιων καρτών εάν παραστεί χρήσιμο και αναγκαίο. Ιδιαίτερες και σημαντικές επιταχύνσεις παρατηρούνται σε επερωτήματα ταιριάσματος εικόνας, ανίχνευσης κειμένου, αναλυτικών λειτουργιών, της JOIN διαδικασίας καθώς επίσης και της προεπεξεργασίας τους.

Ο MaxCompiler είναι ένα νέο εργαλείο το οποίο ακόμη βρίσκεται στην διαδικασία αναγνώρισης και προσπάθειας συνεχής βελτίωσης όμως μέσα από την εργασία μας μπορούμε να εξάγουμε συμπεράσματα γύρω από την χρήση του. Στα θετικά του μπορούμε να συμπεριλάβουμε την δυνατότητα που δίνει στο χρήστη να διαχειριστεί και να προγραμματίσει ένα FPGA μέσω μιας αφαιρετικής υψηλού επιπέδου γλώσσας προγραμματισμού αποφεύγοντας έτσι την χρονοβόρα διαδικασία συγγραφής κώδικα σε γλώσσες περιγραφείς υλικού (HDL). Επίσης μια υλοποίηση στον MaxCompiler (Manager & Kernels) μπορεί να χρησιμοποιηθεί για κάποια διαφορετική (αλλά με

παρόμοιες λειτουργίες) εφαρμογή από την αρχική για την οποία σχεδιάστηκε χωρίς καμιά αλλαγή και χωρίς να χρειάζεται η επαναμεταγλώττιση του κώδικα της μηχανής αρκεί μόνο η προσαρμογή της νέας εφαρμογής (που μπορεί να είναι γραμμένη ακόμη και σε άλλη γλώσσα προγραμματισμού) στην ήδη υπάρχουσα υλοποίηση του υλικού. Ακόμη στα θετικά προσθέτω την δυνατότητα εξαγωγής γραφημάτων των πυρήνων και του διαχειριστή με την βοήθεια των οποίων μπορούμε να αντιληφθούμε σε καλύτερο βαθμό την λειτουργία της υλοποίησης μας, καθώς επίσης γίνετε και ευκολότερη απασφαλμάτωση.

Στα αρνητικά του MaxCompiler συμπεριλαμβάνονται ο περιορισμένος αριθμό πηγών στις οποίες μπορείς να ανατρέξεις για βοήθεια για το λόγο ότι είναι ακόμη μια καινούργια καινοτομία στο χώρο καθώς επίσης και ο περιορισμός χρήσης του μόνο για υλικό κατασκευασμένο και ειδικά σχεδιασμένο από την εταιρεία παράγωγης του Maxeler Technologies.

6.2 Μελλοντική Εργασία

‘Ως μελλοντική εργασία της δουλειάς μου θα ήθελα να προτείνω την υλοποίηση μεγαλύτερου εύρους εφαρμογών και άλλων αλγορίθμων βάσεων δεδομένων από το TPC-H benchmark με την χρήση του MaxCompiler ώστε να υπάρχει πληθώρα αποτελεσμάτων και συγκρίσεων και να εξαχθούν περισσότερα και καλύτερα συμπεράσματα.

Επίσης θα ήταν καλό να γίνουν μετρήσεις όσον αφορά κατανάλωση ενέργειας τέτοιων υλοποιήσεων σε σχέση με τις παραδοσιακές εκτελέσεις στους επεξεργαστές αφού στις μέρες μας συχνά γίνονται μετρήσεις για τέτοιες συγκρίσεις. Η χαμηλότερη κατανάλωση ενέργειας είναι ένα αρκετά σημαντικό όφελος και λαμβάνεται σοβαρά υπόψη σαν μια ισχυρή παράμετρος στις συγκρίσεις διαφορετικών εκτελέσεων της ίδιας εφαρμογής.

Βιβλιογραφία

[1] MOORE, B. C. On the flexibility offered by state feedback in multivariable systems beyond closed loop eigenvalue assignment. Automatic Control, IEEE Transactions on, 1976, 21.5: 689-692.

[2] ΘΕΟΔΩΡΌΠΟΥΛΟΣ, Παναγιώτης. Συστήματα παράλληλης επεξεργασίας. 2008.

[3] BARNEY, Blaise, et al. Introduction to parallel computing. Lawrence Livermore National Laboratory, 2010, 6.13: 10.

[4] TANENBAUM, Andrew. Structured computer organization. 2005.

[5] MUELLER, Rene; TEUBNER, Jens. FPGA: what's in it for a database?. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. ACM, 2009. p. 999-1004.

[6] JEAN, Jack SN, et al. Query processing with an fpga coprocessor board. In: Proc. 1st Int. Conf. Engineering of Reconfigurable Systems and Algorithms. 2001.

[7] SUKHWANI, Bharat, et al. Database analytics acceleration using FPGAs. In: Proceedings of the 21st international conference on Parallel architectures and compilation techniques. ACM, 2012. p. 411-420.

[8] Maxeler MaxCompiler http://www.maxeler.com/products/software/maxcompiler/

[9] TECH, Maxeler. Maxcompiler White Paper. 2011. http://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf.

[10] TECH, Maxeler. Programming MPC Systems White Paper. 2013.

http://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf

[11] *Multiscale Dataflow Programming,* Maxeler Technologies, December 11, 2012.

http://asearch.ac.uk/sites/default/files/maxelerdataflowprogramming.pdf

[12] *MaxCompiler Manager Compiler Tutorial,* Maxeler Technologies, 2012.

[13] *Acceleration Tutorial, Loops and Pipelining,* Maxeler Technologies, 2012.

[14] RESNIK, Jack. The Source-Sink Model. Embryo Project Encyclopedia, 2012.

[15] KOROLIJA, Nenad, et al. Accelerating Lattice-Boltzman Method Using Maxeler DataFlow Approach. The IPSI BgD Transactions on Internet Research, 34.

[16] JENSEN, Jonas Julian. Reconfigurable FPGA Accelerator for Databases. 2012.

[17] MANDLE, Aaron Richard. FPGA Based Hardware Acceleration: A Case Study in Protein Identification. 2008.

[18] ERNST, Rolf; HENKEL, Jörg; BENNER, Thomas. Hardware-software cosynthesis for microcontrollers. Design & Test of Computers, IEEE, 1993, 10.4: 64-75.

[19] ELES, Petru, et al. System level hardware/software partitioning based on simulated annealing and tabu search. Design automation for embedded systems, 1997, 2.1: 5-32.

[20] GUPTA, Rajesh K.; DE MICHELI, Giovanni. Hardware-software cosynthesis for digital systems. Design & Test of Computers, IEEE, 1993, 10.3: 29-41.

[21] BHATT, Ajay V. Creating a PCI Express interconnect. URL www. pcisig. com/specifications/pciexpress/technical library/pciexpress whitepaper. pdf, 2002.

[22] WYLIE, Andrew. The first integrated circuits. 2009

[23] Learn FPGA design to develop your own customized embeddd system.URL http://www.eeherald.com/section/design-guide/fpga_design.html

[24] PELL, Oliver. Multiscale Dataflow Computing (Building vertically in a horizontal world). MuCoCoS 2013

[25] COUNCIL, Transaction Processing Performance. TPC Benchmark H (TPC-H). pp. 16-17, 38.

[26] PRATAS, Frederico, et al. Open the Gates: Using High-level Synthesis Towards Programmable LDPC Decoders on FPGAs.