Σχολή
Ηλεκτρολόγων
Μηχανικών &
Μηχανικών
Υπολογιστών

Mapping (Structured Unstructured Grid Algorithms) on FPGA Using High Level Synthesis Tools

---

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΜΕΛΛΟΣ ΑΛΕΞΑΝΔΡΟΣ

Μέλη Επιτροπής:

Καθηγητής Πνευματικάτος Διονύσιος (Επιβλέπων)

Καθηγητής Ζερβάκης Μιχαήλ

Καθηγητής Καλαιτζάκης Κωνσταντίνος

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ.Πνευματικάτο που μου έδωσε την ευκαιρία να συνεργαστώ μαζί του και να ασχοληθώ με νέα και ενδιαφέροντα πράγματα που συνάντησα στην παρούσα διπλωματική. Επίσης, θα ήθελα να ευχαριστήσω τα μέλη της επιτροπής κ.Ζερβάκη και κ.Καλαιτζάκη.

Στη συνέχεια, θα ήθελα να ευχαριστήσω ιδιαίτερα τον κ.Χρυσό όπου με βοήθησε με τις κατευθύνσεις που μου έδινε καθόλη τη διάρκεια της διπλωματικής και ήταν παρών σε κάθε απορία και πρόβλημα που αντιμετώπισα.

Ακόμα, δεν γίνεται να παραλείψω την οικογένεια μου, Κατερίνα, Γίωργο και Παναγιώτη όπου έδειξαν ιδιαίτερη υπομονή και με βοήθησαν ψυχολογικά σε κάθε δυσκολία που αντιμετώπισα όλα αυτά τα χρόνια και τους ευχαριστώ πραγματικά για όλα.

Τέλος, θέλω να ευχαριστήσω τους φίλους μου για όλες τις όμορφες στιγμές που έχουμε ζήσει και για την στήριξη τους όλα αυτά τα χρόνια σε κάθε μου προσπάθεια.

Περίληψη

Τα τελευταία χρόνια η ανάγκη επεξεργασίας μεγάλου όγκου δεδομένων σε μικρό χρονικό διάστημα, έστρεψε το ενδιαφέρον στη δημιουργία προγραμματων όπου συνδυάζουν το software και το hardware με σκοπό την εκμετάλλευση των πλεονεκτημάτων που παρέχει το καθένα. Η ανάγκη αυτή οδήγησε το Phil Colela στην έμπνευση επτά αλγοριθμικών μεθόδων με μεγάλη φορητότητα, οι οποίες χρησιμοποιήθηκαν ως benchmarks σε διάφορες πλατφόρμες εκμεταλλευόμενες τα πλεονεκτήματα του παράλληλου προγραμματισμού. Στη συνέχεια οι μέθοδοι αυτοί επεκτάθηκαν σε δεκατρείς από ομάδα ερευνητών του Berkeley.

Παράλληλα, τα τελευταία χρόνια απλουστεύτηκε η απεικόνιση ενός αλγορίθμου στο hardware με τη βοήθεια του εργαλείου Vivado High Level Synthesis. Οι διαδικασίες έγιναν πιο αυτοματοποίημενες και η δημιουργία του RTL αρχείου αρκετά πιο εύκολη για το προγραμματιστή.

Ο στόχος λοιπόν αυτής της διπλωματικής, είναι η απεικόνιση δύο αλγορίθμων στο hardware με βάση την αρχιτεκτονική Decoupled Access/Execute ως απώτερο σκοπό τη βελτιστοποίηση της απόδοσης τους. Πιο συγκεκριμένα γίνεται απεικόνιση των μεθόδων Jacobi και Serial Based. Οι αλγόρθιμοι αυτοί υπάγονται στους δεκατρείς νάνους και ανήκουν στην κατηγορία των δομημένων και μη δομημένων πλεγμάτων αντίστοιχα. Οι δύο αυτοί διαφορετικοί τύποι αλγορίθμων υλοποίηθηκαν με κοινό framework, το Decoupled Access/Execute (DAE). Τα στάδια μετατροπής ενός αλγορίθμου με βάση το παραπάνω framework είναι απλά και συγκεκριμένα. Για καθένα από τους παραπάνω αλγορίθμους πραγματοποιήθηκαν τρεις διαφορετικές υλοποιήσεις στο εργαλείο της Vivado HLS. Τέλος, πραγματοποιήθηκε σύγκριση στην απόδοση κάθε υλοποίησης με την αρχική βελτιστοποιημένη υλοποίηση σε software.

Λέξεις Κλειδιά

Παράλληλος προγραμματισμός, Δεκατρείς Νάνοι, Δομημένα Πλέγματα, Μη Δομημένα Πλέγματα, Jacobi Method, Serial Based Method, Αρχιτεκτονική DAE, Vivado High Level Synthesis

Abstract

In the latest years, the need to process large volumes of data in a short time period has shifted the interest in creating programs that combine software and hardware. This need led Phil Colela to the inspiration of seven algorithmic methods with great portability on various platforms that were used as benchmarks, exploiting the advantages of parallel programming. These methods were extended to thirteen by a Berkeley group of researchers.

Simultaneously, in the past few years, the visualization of an algorithm in hardware has been simplified with the help of the Vivado High Level Synthesis tool. As a result, procedures have become more automated and the creation of the RTL file has become easier for the developer,as well.

The aim of this Diploma Thesis is implement two algorithms in hardware according to the DAE architecture and for the optimization system performance. Specifically, Jacobi and Serial Based methods are mapped. These algorithms fall into the 13 dwarfs and belong to the category of structural and non-welded mesh respectively. These two different types of algorithms have been implemented in a common framework, Decoupled Access / Execute (DAE). The stages of converting an algorithm based on the above framework are simple and specific. For each of the above algorithms, three different implementations were carried out on the Vivado HLS tool. The first and main implementation is performed with knowledge of DAE architecture. The second and the third ones are implemented based on the DAE architecture but differ in the way of data storage, where the second is internally made by the FPGA and the third is externally made. Finally, both the times of the second and the third implementation are stated and compared with the initial optimized implementation in software.

Keywords

Parallel Computing, Thirteen Dwarfs, Structured Grid, Unstructured Grid, Jacobi Methood, Serial Based Method, DAE Architecture

Περιεχόμενα

Κεφάλαιο 1 ..... **10**

1.1 Εισαγωγή στους δεκατρείς νάνους **11**

1.2 Εισαγωγή στην αρχιτεκτονική DAE **12**

1.3 Συνδυασμός DAE με τους δεκατρείς νάνους **13**

1.4 Συνεισφορά **13**

Κεφάλαιο 2 ..... **15**

2.1 Υλοποιήσεις της χρήση DAE για βελτίωση της απόδοσης **15**

2.2 Υλοποιήσεις της χρήσης των Νάνων για βελτίωση της απόδοσης **16**

Κεφάλαιο 3 ..... **18**

3.1 Δεκατρεις Νάνοι **18**

3.2 Περιγραφή των νάνων **19**

3.2.1 Dense Linear Algebra 19
3.2.2 SparceLinearAlgebra 19
3.2.3 Spectral Methods 20
3.2.4 N-Body Methods 20
3.2.5 Map Reduce & Monte Carlo 21
3.2.6 Combinational Logic 21
3.2.7 Dynamic Programming 21
3.2.8 Backtrack and Branch and Bound 22
3.2.9 Finite State Machine 22
3.2.10 GraphTraversal 22
3.2.11 Construct Graphical Models 23
3.2.12 Αποτελεσματικότητα νάνων και κλάδων εφαρμογής 23

3.3 Δομημένα Πλέγματα **24**

3.4 Μέθοδος Jacobi **29**

3.5 Μη Δομημένα Πλέγματα **31**

3.6 Μέθοδος Serial Based **32**

Κεφάλαιο 4 ..... **34**

4.1 Περιγραφή Αρχιτεκτονικής DAE **34**

4.2 Αρχιτεκτονική DAER **37**

4.3 Ενεργειακή Αποδοτικότητα **40**

Κεφάλαιο 5 ..... **42**

5.1 Υλοποιήσεις δομημένων πλεγμάτων 42
5.1.1 1η Υλοποίηση δομημένων πλεγμάτων 42
5.1.2 2η Υλοποίηση δομημένων πλεγμάτων 47
5.1.3 3η Υλοποίηση δομημένων πλεγμάτων 48

5.2 Υλοποιήσεις μη δομημένων πλεγμάτων 49
5.2.1 1η Υλοποίηση μη δομημένων πλεγμάτων 49
5.2.2 2η Υλοποίηση Μη Δομημένων Πλεγμάτων 53
5.2.3 3η Υλοποίηση Μη Δομημένων Πλεγμάτων 53

5.3 Directives 54

Κεφάλαιο 6 57

6.1 Περιγραφή πλατφόρμων και εργαλειών 57

6.2 Πόροι Συστήματος 59

6.3 Απόδοση συστημάτων 60
6.3.1 Περιγραφή των datasets 60
6.3.2 Χρόνοι εκτέλεσης των αλγορίθμων για τα δομημένα πλέγματα από το HLS 61
6.3.3 Χρόνοι εκτέλεσης των αλγορίθμων για τα μη δομημένα πλέγματα από το HLS 62
6.3.4 Χρόνοι εκτέλεσης των αλγορίθμων για τα δομημένα και μη δομημένα πλέγματα στη Convey 64

Κεφάλαιο 7 69

7.1 Συμπεράσματα 69

7.2 Μελλοντική εργασία 70

Βιβλιογραφία 71

Παράρτημα 74

Α Κώδικας Δομημένων Πλεγμάτων 74
Α.1 Υλοποίηση 1η 74
Α.2 Υλοποίηση 2η 77
Α.3 Υλοποίηση 3η 80

Β Κώδικας Μη Δομημένων Πλεγμάτων 82
Β.1 Υλοποίηση 1η 82
Β.2Υλοποίηση 2η 87
Β.3Υλοποίηση 3η 89

Λίστα εικόνων

Εικόνα 1: Τομείς εφαρμογής των νάνων 24
Εικόνα 2: Μορφή regular static grid 26
Εικόνα 3: Μορφή transformed structured grids 26
Εικόνα 4: Μορφή composite structured grids 26
Εικόνα 5: Μορφή block structured grids 27
Εικόνα 6: Μορφή adaptive grids 27
Εικόνα 7: Μορφή block adaptive grids 28
Εικόνα 8: Μορφή recursively structured adaptive grids 28
Εικόνα 9: Περιοχές εφαρμογής μεθόδου Jacobi 30
Εικόνα 10: Εφαρμογή μεθόδου Jacobi σε ένα σημείο 30
Εικόνα 11: Παράδειγμα εφαρμογής της μεθόδου serial based 33
Εικόνα 12: Αρχιτεκτονική DAE 35
Εικόνα 13: Αρχιτεκτονική DAER 39
Εικόνα 14: Υλοποίηση της μεθόδου Jacobi 43
Εικόνα 15: Εφαρμογή της αρχιτεκτονικής DAER στα δομημένα πλέγματα 46
Εικόνα 16: Υλοποίηση της μεθόδου Serial based 50
Εικόνα 17 :Εφαρμογή της αρχιτεκτονικής DAER στα μη δομημένα πλέγματα 52
Εικόνα 18: Εφαρμογή dataflow directive 55
Εικόνα 19: Χαρακτηριστικά της πλατφόρμας Zynq zc706 58
Εικόνα 20: Χαρακτηριστικά της πλατφόρμας Convey 58
Εικόνα 21: Πόροι συστήματος που χρησιμοποιήθηκαν για τα δομημένα πλέγματα 59
Εικόνα 22: Πόροι συστήματος που χρησιμοποιήθηκαν για τα μη δομημένα πλέγματα 60
Εικόνα 23: Πίνακας χρόνων δομημένων πλεγμάτων 61
Εικόνα 24: Γράφημα απόδοσης για τα δομημένα πλέγματα 62
Εικόνα 25: Πίνακας χρόνων για τα μη δομημένα πλέγματα στην FPGA 63
Εικόνα 26: Γράφημα απόδοσης για τα μη δομημένα πλέγματα στην FPGA 64
Εικόνα 27: Πίνακας χρόνων για τα δομημένα πλέγματα στην Convey 65
Εικόνα 28: Γράφημα απόδοσης για τα δομημένα πλέγματα στην Convey 66
Εικόνα 29: Πίνακας χρόνων για τα μη δομημένα πλέγματα στην Convey 67
Εικόνα 30: Γράφημα απόδοσης για τα μη δομημένα πλέγματα στην Convey 67
Εικόνα 31: Λειτουργική μονάδα fetch για τα δομημένα πλέγματα 75
Εικόνα 32: Φόρτωση δεδομένων από την κύρια μνήμη 76
Εικόνα 33: Λειτουργική μονάδα process για τα δομημένα πλέγματα 77
Εικόνα 34: Συνάρτηση test function 79
Εικόνα 35: Συνάρτηση read memory 80
Εικόνα 36: Συνάρητη test function 81
Εικόνα 37: Λειτουργική υπομονάδα fetch0 για τα μη δομημένα πλέγματα 82
Εικόνα 38: Λειτουργική υπομονάδα fetch1 για τα μη δομημένα πλέγματα 83
Εικόνα 39: Λειτουργική υπομονάδα fetch2 για τα μη δομημένα πλέγματα 83

Εικόνα 40: Ανάκτηση δεδομένων υπομονάδας fetch0 84
Εικόνα 41: Ανάκτηση δεδομένων υπομονάδας fetch1 84
Εικόνα 42: Ανάκτηση δεδομένων υπομονάδας fetch2 84
Εικόνα 43: Κλήσεις λειτουργικών μονάδων fetch και process στην κύρια μνήμη 86
Εικόνα 44: Λειτουργική μονάδα process για τα μη δομημένα πλέγματα 87
Εικόνα 45: Συνάρτηση test funtion 89
Εικόνα 46: Συνάρτηση test funtion 91

Κεφάλαιο 1

Τα τελευταία χρόνια πραγματοποιήθηκε μια σημαντική αλλαγή στο υλικό του υπολογιστή. Συγκεκριμένα, το 2005 οι κύριες εταιρίες μικροεπεξεργαστών ανακοίνωσαν ότι στα μελλοντικά προιόντα τους θα τοποθετηθούν πολλαπλοί επεξεργαστές σε ένα μόνο chip κάτι το οποίο θα συντελέσει στην εκτόξευση της απόδοσης των εφαρμογών στο εγγύς μέλλον. Εν συνεχεία δημιουργήθηκε και η ιδέα του παράλληλου προγραμματισμού για την πλήρη εκμετάλευση των πολλαπλών πόρων. Ωστόσο, η δημιουργία ενός προγράμματος με τις επιθυμητές προοπτικές ενός κατασκευαστή και με συνδυασμό την εφαρμογή παράλληλου προγραμματισμού καθιστούσε αρκετά δύσκολη τη ζωή του προγραμματιστή. Αρκετές ήταν οι φορές που με μία λάθος εκτίμηση η ορθότητα και η απόδοση ενός αλγορίθμου καταλήγε να είναι χειρότερη.

Αναλογιζόμαστε ότι μας παρουσιάζεται μία διαφορετική προοπτική στο πως θα προσεγγίζουμε τη δημιουργία των προγραμμάτων και θα εκμεταλλευόμαστε πλήρως τις δυνατότητες της παραλληλίας που μας παρέχει η αρχιτεκτονική πολλών πυρήνων. Για την κατανόηση της σημασίας του παράλληλου προγραμματισμού και την μη αποστροφή των προγραμματιστών από τέτοιες υλοποιήσεις εμπνεύστηκαν και δημιουργήθηκαν από το Phil Collela το 2004 επτά αλγοριθμικοί μέθοδοι τους οποίους ονόμασε νάνους. Συγχρόνως, δημιουργήθηκαν νέες αρχιτεκτονικές με σκοπό την απλούστευση του παράλληλου προγραμματισμού όπως είναι η αρχιτεκτονική decoupled access/execute. Στο κεφάλαιο αυτό θα παρουσιάσουμε τις αλγοριθμικές μεθόδους που υπάγονται στους δεκατρείς νάνους και την αρχιτεκτονική Decoupled Access/Execute (DAE), καθώς και στην υλοποίηση των νάνων με βάση την αρχιτεκτονική DAE. Τέλος, θα γίνει μια μικρή εισαγωγή των επόμενων κεφαλαίων και αναφέρεται η συνεισφορά της παρούσας διπλωματικής.

1.1 Εισαγωγή στους δεκατρείς νάνους

Αρχικά ο Phil Colela προσδιόρισε επτά αλγοριθμικές μεθόδους τις οποίες και ονόμασε νάνους[1]. Πρωταρχικός του στόχος ήταν η ένταξη όλων των κοινών αλγορίθμων σε ένα μικρό σύνολο μεθόδων. Ισχυρίστηκε, ότι καθε κατηγορία εμφανίζει κοινά πρότυπα υπολογισμού και επικοινωνίας. Στην ουσία κάθε κατηγορία νάνων παρουσιάζει οικογένειες αλγορίθμων, όπου εμφανίζουν όμοιες υπολογιστικές ιδιότητες. Οι παραπάνω αλγοριθμικές μεθόδοι δέχθηκαν και θα δεχθούν αρκετές αλλαγές με το πέρασμα των χρόνων, ωστόσο τα πρότυπα υπολογισμού παρέμειναν και θα παραμείνουν τα ίδια παίζοντας σπουδαίο ρόλο στο εγγύς μέλλον για την βελτίωση εφαρμογών σε διάφορους τομείς της επιστήμης και της μηχανικής. Ένα άλλο χαρακτηριστικό τους είναι ότι διακρίνονται για τα υψηλά επίπεδα αφαίρεσης, ώστε να βρίσκουν εφαρμογή σε ένα ευρύ φάσμα εφαρμογών. Εξαιτίας της μεγάλης αποδοτικότητας που τους διέπουν οι μέθοδοι αυτοί είναι ικανοί να εκτελεστούν οπουδήποτε, π.χ CPU, GPU ακόμα και στις FPGAs κάτι το οποίο είναι αρκετά σημαντικό διότι γνωρίζουμε ότι είναι αρκετά επεκτάσιμες και είναι ιδιαίτερα χαμηλού κόστους.

Στη συνέχεια, μια ομάδα ερευνητών από το Berkeley επέκτεινε τους νάνους σε δεκατρείς εξετάζοντας και εφαρμόζοντας τους σε διάφορους τομείς[1]. Απέδειξαν ότι οι νάνοι έχουν αρκετά καλή απόδοση, μειωμένη κατανάλωση ενέργειας ιδιαίτερα σε πολυπύρηνους επεξεργαστές με αποτέλεσμα να δημιουργηθούν νέα κριτήρια για το σχεδιασμό και την αξιολόγηση παράλληλων μοντέλων προγραμματισμού. Οι ερευνητές ισχυρίζονται ότι θα διατηρήσουν τις καλές αποδόσεις τους και σε μελλοντικότερες πλατφόρμες.

Τέλος οι νάνοι συνεισφέρουν στην διευκόλυνση της ζωής του προγραμματιστή καθιστώντας την υλοποίηση παράλληλων προγραμμάτων αρκετά πιο εύκολη. Εξαιτίας τις κατηγοριοποίησης των αλγορίθμων, ο προγραμματιστής μπορεί να επιλέξει εκ των προτέρων την κατάλληλη αλγοριθμική μέθοδο για την υλοποίηση του προγράμματος που επιθυμεί. Έτσι και η σχεδίαση σε παράλληλου προγραμματισμού γίνεται πιο εύκολη και πιο αποδοτική εφόσον μπορούμε να αποφασίσουμε εκ των προτέρων πότε θα χρησιμοποιήσουμε GPU και άλλους επιταγχυντές και πότε CPU.

1.2 Εισαγωγή στην αρχιτεκτονική DAE

Ο παράλληλος προγραμματισμός έχει γίνει το κυρίαρχο πρότυπο στην αρχιτεκτονική των υπολογιστών και κυρίως με τη μορφή των πολλαπλών πυρήνων. Η εφεύρεση του έχει γίνει πολλά χρόνια τώρα, αλλά τα τελευταία χρόνια το ενδιαφέρον για δημιουργία παράλληλων αρχιτεκτονικών έχει μεγαλώσει εξαιτίας διαφόρων φυσικών περιορισμών που εμποδίζουν τη βελτίωση της συχνότητας.

Στους περισσότερους μικροεπεξεργαστές υψηλής απόδοσης εφαρμόζεται δυναμικός προγραμματισμός ή αλλιώς out-of-order. Ο επεξεργαστής εκτελεί τις εντολές ανάλογα με την διαθεσιμότητα και όχι ανάλογα με την σειρά του προγράμματος. Έτσι, ο επεξεργαστής δεν περιμένει την ολοκλήρωση μιας εντολής για την ανάκτηση της επόμενης με αποτέλεσμα να εξαλείπονται τυχών καθυστερήσεις όταν τα δεδομένα που περιμένει δεν είναι έτοιμα. Η σπουδαιότητα του δυναμικού προγραμματιισμού αυξάνεται καθώς μεγαλώνει ο αριθμός των εντολών και η διαφορά της ταχύτητας μεταξύ της κυριας μνήμης και του επεξεργαστή διευρύνεται.

Μία εκ των αρχιτεκτονικών που επιτυγχάνει την απόκτηση των οφελών του δυναμικού προγραμματισμού είναι και η DAE με την διαφορά ότι δημιουργεί ουρές που επιτρέπουν την αποσύνδεση της ανάκτηση δεδομένων από την εκτέλεση των αριθμητικών πράξεων. Η ιδέα της αρχιτεκτονικής αυτής προήλθε από τον James E. Smith το 1982[2]. Το αρχικό instruction stream χωρίζεται σε δύο λειτουργικές μονάδες, fetch και process χρησιμοποιώντας ένα buffer. Στόχος του buffer είναι η πλήρης αποσύνδεση των προσβάσεων στη μνήμη με την επεξεργασία των δεδομένων για να επιτευχθεί υψηλή απόδοση εκμεταλλευόμενοι τον παραλληλισμό. Έτσι, οι καθυστερήσεις λόγω μνήμης μειώνονται και συγχρόνως με τη μείωση των αστοχιών στη μνήμη cache επιτυγχάνεται υψηλή απόδοση.

Ακόμα συμβάλει στην βελτίωση της ενεργειακής απόδοσης του συστήματος. Ανάλογα με το στάδιο που βρισκόμαστε μπορούμε να χρησιμοποιήσουμε και την κατάλληλη συχνότητα. Έτσι, μεταβάλοντας τη συχνότητα σε μικρότερες τιμές όσο αναφορά τις ανακτήσεις μνήμης και σε μεγαλύτερες στην επεξεργασία των δεδομένων πετυχαίνουμε την βέλτιστη κατανάλωση ενέργειας.

Συνοψίζοντας, η αρχιτεκτονική DAE παρέχει βέλτιστη απόδοση και κατανάλωση ενέργειας σε παράλληλα μοντέλα προγραμματισμού εξαιτίας τη μεγάλης αποσύνδεσης των σταδίων ανάκτησης και επεξεργασίας δεδομένων.

1.3 Συνδυασμός DAE με τους δεκατρείς νάνους

Παραπάνω αναφέρθηκαν ο σκοπός της δημιουργίας της αρχιτεκτονικής DAE και η δημιουργία των δεκατριών αλγοριθμικών μέθοδων που ονομάστηκαν νάνοι. Αυτά τα δύο παρέχουν τη δυνατότητα της δημιουργίας ενός προγράμματος που διακρίνεται για την ευελιξία του και για την αποδοτικότητα του. Ακόμα, καθιστούν εύκολη τη ζωή του προγραμματιστή εφόσον πραγματοποιεί τετριμένα βήματα για την δημιουργία της εφαρμογής που επιθυμεί με πρότυπα παράλληλου προγραμματισμού. Μπορούμε να αναλογιστούμε ότι η υλοποίηση μιας εκ των δεκατριών αλγοριθμικών μεθόδων στα πρότυπα της αρχιτεκτονική DAE θα εκτοξεύσει την απόδοση του συστήματος.

Εάν εξετάσουμε τους νάνους θα δούμε ότι οι αλγόριθμοι τους περιέχουν προσπελάσεις στη μνήμη και ανάκτηση μεγάλου όγκου δεδομένων. Με το διαχωρισμό του instruction streamσε δύο λειτουργικές μονάδες όπου η πρώτη είναι αρμόδια για την ανάκτηση των διευθύνσεων και η δεύτερη για την επεξεργασία των αποτελεσμάτων καταλαβαίνουμε ότι η αρχιτεκτονική DAE βελτιώνει την απόδοση των παραπάνω αλγοριθμικών μεθόδων εφόσον δεν χρειάζεται να έχει εκτελεστεί μία εντολή για να εκτελεστεί η επόμενη.

1.4 Συνεισφορά

Συνοπτικά στη διπλωματική αυτή υλοποιήσαμε δύο αλγόριθμους που υπάγονται στους δεκατρείς νάνους και συγκεκριμένα στα δομημένα και μη δομημένα πλέγματα με τη χρήση του εργαλείου της Vivado High Level Synthesis(HLS)και ως γνώμονα την αρχιτεκτονική DAE. Το εργαλείο της vivadomας παρέχει αυτοματοποιημένους επιταχυντές και dataflow directives όπου βελτίωσαν την απόδοση του προγράμματος. Πραγματοποιήθηκαν τρεις υλοποιήσεις για κάθε αλγόριθμο και συγκρίναμε την απόδοση και την κατανάλωση της ενέργειας με τον αρχικό σχεδιασμό.

Η συνεισφορά της παραπάνω διπλωματικής συνοψίζεται στα παρακάτω:

- Απεικόνιση δύο αλγορίθμων με διαφορετικά χαρακτηριστικά με τη χρήση το εργαλείου Vivado σε αναδιατασσόμενη λογική. Το εργαλείο αυτό παρέχει αυτοματοποιημένους τρόπους απεικόνισης εφαρμογών σε αναδιατασσόμενη λογική και μας βοηθάει να μετατρέψουμε πολύ εύκολα εφαρμογές από γλώσσα υψηλού προγραμματισμού όπως η C/C++ .
- Βελτιστοποίηση των αρχιτεκτονικών των παραπάνω εφαρμογών που πραγματοποιήθηκαν στο Vivado HLS με τη χρήση του framework DAE(DAER) που αναλύεται στο κεφάλαιο 5. Το συγκεκριμένο framework βελτιστοποιεί την απόδοση διαφόρων εφαρμογών προσφέροντας streaming επεξεργασία και καλύπτοντας διάφορες αλγοριθμικές δυσκολίες όπως οι εξαρτήσεις δεδομένων κατά την επεξεργασία.
- Χρήση συγκεκριμένων εφαρμογών που υπάγονται στους δεκατρείς νάνους ως benchmarks, και την απεικόνιση τους με βάση το DAE framework σε αναδιατασσόμενη λογική.
- Μέτρηση απόδοσης της απεικόνισης των παραπάνω εφαρμογών σε πλατφόρμα αναδιατασσόμενης λογικής. Εξαγωγή συμπερασμάτων και μελλοντικές προεκτάσεις.

Κεφάλαιο 2

Στο συγκεκριμένο κεφάλαιο αναφέρονται προηγούμενες εργασίες που χρησιμοποιούν το framework DAE για την απεικόνιση σε αναδιατασσόμενη λογική διαφόρων εφαρμογών. Επίσης επισημαίνονται προηγούμενες εργασίες που χρησιμοποιούν τους νάνους σε διαφορετικές πλατφμόρμες.

2.1 Υλοποιήσεις της χρήση DAE για βελτίωση της απόδοσης

Η αρχιτεκτονική DAE δημιουργήθηκε με σκοπό την βελτίωση της απόδοσης των προγραμμάτων. Το μεγαλύτερο της πλεονέκτημα είναι ότι παρέχει τη δυνατότητα απόκρυψης καθυστέρησης πρόσβασης στη μνήμη εκμεταλλευόμενη την ιδιότητα του prefetching. Το κέρδος φυσικά είναι ότι τα δεδομένα έρχονται την κατάλληλη στιγμή για επεξεργασία και αποφεύγονται τυχών αστοχίες στη μνήμη. Ωστόσο, τα τελευταία χρόνια το ενδιαφέρον των αρχιτεκτονικών DAE έχει συρρικνωθεί και αυτό γιατί ορισμένες φορές δεν είναι κατάλληλες σε γενικές εφαρμογές διότι δεν είναι δομημένες.

Στο παρελθόν υπήρξαν πολλές προηγούμενες εργασίες με βάση την αρχιτεκτονική DAE για την αποδοτική απεικόνιση των αλγορίθμων στο hardware. Αρχικά τα τελευταία χρόνια παρατηρούμε ότι οι εφαρμογές των πολυμέσων έχουν κυρίαρχο φόρτο εργασίας. Οι εφαρμογές αυτές είναι δομημένες κάτι το οποίο βοηθάει στην ανάπτυξη της αρχιτεκτονικής DAE. Με βάση λοιπόν την αρχιτεκτονική αυτή δημιουργήθηκε μία παρόμοια αρχιτεκτονική που ονομάζεται MediaBreeze[3] όπου όμοια και αυτή αποσυνδέει την κύρια εκτέλεση του προγράμματος σε δύο μέρη. Το πρώτο μέρος και το κύριο όπου, περιέχει χρήσιμους υπολογισμούς που απαιτούνται από το κύριο φόρτο του αλγορίθμου και το δεύτερο μέρος αποτελεί υποστηριχτική μονάδα και είναι αρμόδιο για λειτουργίες όπως η δημιουργία και μετασχηματισμός των διευθύνσεων, αποθήκευση και φόρτωση και βρόχους επανάληψης. Ο κύριος σκοπός της συγκεκριμένης

αρχιτεκτονικής είναι να βοηθήσει στη εκτέλεση των χρήσιμων εντολών με την διαδικασία της αποσύζεξης. Έτσι, εκμεταλλεύεται τις βασικές έννοιες της αρχιτεκτονικής DAE για την βελτίωση των προγραμμάτων. Αποτελέσματα έχουν δείξει ότι η βελτίωση της απόδοσης κυμαίνεται από 1,05x εώς 16,7x.

Εν συνεχέια παρουσιάστηκε η αρχιτεκτονική MTDAE (MultithreadDecoupledAccess/Execute)[4] με σκοπό την εκμετάλευση της παραλληλίας. Υλοποιήθηκε αρχικά στο MARS-Mcomputer το 1980. Στοχεύει στην συνολική ενίσχυση της αποδοτικότητας της μηχανής και ενός νήματος. Ο συνδυασμός των πολλαπλών νημάτων και της αρχιτεκτονικής DAE επιτρέπουν στον επεξεργαστή να εκμεταλλεύονται τον παραλληλισμού με τις πολλαπλές μονάδες εκτέλεσης που ανατίθονται σε ένα νήμα. Ο επεξεργαστής μπορεί να μεταβεί σε άλλο νήμα ενώ παράλληλα οι διευθύνσεις και τα δεδομένα επεξεργάζονται από το προηγούμενο νήμα. Πολλές αρχιτεκτονικές χρησιμοποιούν MTDAE όπως HEP, MASA, Horizon για εναλλαγή μεταξύ των νημάτων σε κάθε κύκλο. Οι αποδόσεις αυτών των νημάτων βελτιώθηκαν από 30x μέχρι και 90x

2.2 Υλοποιήσεις της χρήσης των Νάνων για βελτίωση της απόδοσης

Με το πέρασμα των χρόνων αντιλαμβανόμενοι τη σπουδαιότητα των νάνων στο παράλληλο προγραμματισμό έγιναν προσπάθειες για την εξάπλωση τους σε ευρύτερες υπολογιστικές μεθόδους. Η αρχιτετονική με μορφή streaming που παρέχει η FPGA αποτέλεσε κύρια αρχιτεκτονική για βελτιστοποιήσεις των αποδόσεων σε τομείς όπως η οικονομία και η δυναμική των υγρών, τομείς που όπως θα δούμε και παρακάτω βρίσκουν εφαρμογή και οι δεκατρείς νάνοι. Ωστόσο, ο προγραμματισμός αυτών των πλακετών για την εκμετάλλευση του παράλληλου προγραμματισμού απαιτεί υψηλές γνώσεις γλώσσας προγραμματισμού χαμηλού επιπέδου. Για την αντιμετώπιση αυτού του προβλήματος μία καλή λύση είναι η χρησιμοποίηση της OpenCL η οποία παρέχει εύχρηστο και φορητό μοντέλο προγραμματισμού για CPUs,GPUs και τώρα και FPGAs. Για την βελτίωση της απόδοσης των OpenCL kernels στις FPGA δημιουργήθηκαν νέες μέθοδοι κάτω από συγκεκριμένους περιορισμούς του υλικού. Στη συνέχεια, αυτές οι τεχνικές εφαρμόστηκαν στο Open Dwarfs benchmark suite όπου

συμπεριλαμβάνει πολλές και διάφορες μεθόδους για την αξιολόγηση της βελτιστοποίησης ανάλογα με την απόδοση και τους πόρους που χρησιμοποιήθηκαν.

Ένα χαρακτηριστικό των δεκατριών νάνων είναι ότι μπορούν να βρούν εφαρμογή σε αρκετές πλατφόρμες. Για την ανάδειξη της εφαρμογής αυτής χρησιμοποιήθηκαν σαν benchmarks σε αρκετές πλατφόρμες πολυπήρηνων επεξεργαστών της εταιρίας AMD, στη πλατφόρμα Intel Xeon Phi P1750 και τέλος χρησιμοποιήθηκε η FPGA, Xilinx Virtex-6 LX760[5]. Οι αλγοριθμικοί μέθοδοι που χρησιμοποιήθηκαν ως benchmarks ήταν N-Body Methods, Dynamic Programming, Structured Grid, Graph Traversal, Combinational Logic, Sparce Linear Algebra. Ειδικότερα η αλγοριθμική μέθοδος των δομημένων πλεγμάτων ήταν ιδιαίτερα αποδοτική στον επεξεργαστή της πλατφόρμας Intel Xeon Phi P1750. Στην FPGA αναλόγως τη χρήση των επιταγχυντών άλλαζε και η απόδοση του συστήματος. Με τη χρήση πέντε επιταγχυντών πραγματοποιήθηκε πολύ καλή απόδοση πλησιάζοντας την απόδοση της πλατφόρμας της Intel Xeon Phi.

Κεφάλαιο 3

Στο συγκεκριμένο κεφάλαιο γίνεται μια εισαγωγή στις αλγοριθμικές μεθόδους των δεκατριών νάνων. Γίνεται μία μικρή επισκόπη όλων των νάνων και εκτενέστερη περιγραφή των αλγορίθμων με επεξεργασία πάνω σε δομημένα και μη δομημένα πλέγματα.. Επίσης ,περιγράφονται οι αλγόριθμοι που θα υλοποιηθουν στο εργαλείο της Vivado για την βελτίωση της απόδοσης τους.

3.1 Δεκατρεις Νάνοι

Οι νάνοι εμπνεύστηκαν από το Phil Collela το 2005 με τη τότε μορφή του να έιναι επτά αλγοριθμικοί μέθοδοι. Εν συνεχεία με τη συμβολή μιας ομάδας ερευνητών από το Berkeley οι αλγόριθμοι επεκτάθηκαν σε δεκατρείς. Διακρίνονται για τα μεγάλα επίπεδα αφαίρεσης και όλα τα μέλη έχουν κοινά πρότυπα επικοινωνίας και υπολογισμού καθιστώντας τους εύχρηστους σε ευρύ φάσμα εφαρμογών. Κάτι το οποίο είναι ξεχωριστό με τους νάνους είναι ότι οι ερευνητές δεν εστίασαν στο υπάρχον υλικό και τα υπάρχοντα προγραμματιστικά μοντέλα και προσπάθησαν να εστιάσουν σε μελλοντικές απαιτήσεις και εφαρμογές. Για το λόγο αυτό, αποτελούν και θα αποτελέσουν και στο μέλλον κριτήρια απόδοσης για παράλληλα μοντέλα προγραμματισμού.

Η κατηγοριοποίηση των μεθόδων είναι αρκετά σημαντική και δημιουργήθηκε για να καλύπτει τις ανάγκες όλων των κοινών αλγορίθμων.

Η τελική μορφή των δεκατριών νάνων συντελείται από τις παρακάτω αλγοριθμικούς μεθόδους:

- Dense Linear Algebra
- Sparce Linear Algebra
- Spectral Methods
- N-Body Methods
- Structured Grids

- Unstructured Grids
- Map Reduce &Monte Carlo
- Combinational Logic
- Graph Traversal
- Dynamic Programming
- Backtrack and Branch+Bound
- Construct Graphical Models
- Finite State Machine

3.2 Περιγραφή των νάνων

3.2.1 Dense Linear Algebra

Οι εφαρμογές αυτές ,περιλαμβάνουν γραμμικές πράξεις διανυσμάτων και πινάκων. Υπάρχουν τρία διαφορετικά επιπέδα που περιλαμβάνουν πολλαπλασιασμούς μεταξύ διανυσμάτων, πινάκων και πινάκων με διανυσμάτων.Στην ουσία, αυτές οι μέθοδοι είναι αρμόδιες για εφαρμογές που χρειάζονται προσβάσεις στην μνήμη και έχουν υψηλό βαθμό εξαρτήσεων δεδομένων. Χρησιμοποιούνται κυρίως σε πλατφόρμες της γραμμικής άλγεβρας όπως το LAPACK, όπου είναι μια βιβλιόθηκη λογισμικού για την αριθμητική γραμμική άλγεβρα και περιέχει ρουτίνες για την επίλυση γραμμικών εξισώσεων ,συστημάτων. Ένα άλλο πεδίο εφαρμογής είναι το datamining και κυρίως το streamcluster όπου βοηθά στην ομαδοποίηση μεγάλου όγκου δεδομένων από διάφορους τομείς όπως η οικονομία και κυρίως οι οικονομικές συναλλαγές.

3.2.2 SparceLinearAlgebra

Οι αλγόριθμοι sparce matrix επίλυουν τα ίδια προβλήματα με τη dense linear algebra με μία διαφορά.Συνήθως, προτιμούνται κυρίως όταν οι πίνακες που εισέρχονται ως είσοδοι έχουν μεγάλο αριθμό μηδενικών καταχωρήσεων. Αυτό παίζει σημαντικό ρόλο στη αποθήκευση των

δεδομένων και στην αποδοτικότητα του αλγορίθμου. Η μέθοδος επίλυσης ενός προβλήματος με αυτούς τους αλγόριθμους πραγματοποιείται με την επαναληπτική μέθοδο. Ξεκινάμε με μία υποθετική τιμή και επαναλαμβάνουμε τον αλγόριθμο μέχρι η τιμή του λάθους να έιναι αμελητέα ή στις τιμές που έχει ορίσει ο χρήστης ως συνθήκη τερματισμού. Όπως και στη Dense Linear Algebra βρίσκει χρήση σε συστήματα όπου απαιτούνται προσβάσεις στη μνήμη όπου τα δεδομένα εμφανίζουν υψηλό βαθμό εξαρτήσεων με τη διαφορά ότι η εφαρμογή της είναι αρκετά πιο πολύπλοκη και σύνθετη από τη Dense Linear Algebra. Εφαρμογές αυτών των μεθόδων βρίσκουμε κυρίως στην ανάλυση πεπερασμένων στοιχείων όπου είναι μια μέθοδος που προβλέπει πως θα αντιδράσει ένα προιόν στις δυνάμεις του πραγματικού κόσμου όπως δονήσεις, θερμότητα και άλλα φυσικά φαινόμενα.

3.2.3 Spectral Methods

Είναι μέθοδοι, οπου χρησιμοποιούνται στα εφαρμοσμένα μαθηματικά για την επίλυση διαφορικών εξισώσεων και περιλαμβάνουν τον μετασχηματισμό Fourrier. Οι μέθοδοι αυτοί συνδέονται με την ανάλυση πεπερασμένων στοιχείων διατυπόνοντας μια πιο σφαιρική προσεγγιση. Αυτό, τις καθιστά αρκετά σημαντικές για τον εντοπισμό σφάλματος και μάλιστα ορισμένες φορές είναι η μοναδική μέθοδος για τον εντοπισμό των σφαλμάτων. Εφαρμόζεται σε τομείς όπως η δυναμική των ρευστών όπου περιγράφει την ροή των υγρών ακόμα και στις προβλέψεις του καιρού και την κβαντομηχανική.

3.2.4 N-Body Methods

Οι αλγόριθμοι αυτοί επικεντρώνονται στους υπολογισμούς που εξαρτώνται απο τις αλληλεπιδράσεις μεταξύ δύο η περισσότερων διακριτών σημείων. Βρίσκουν εφαρμογή σε διάφορους τομείς όπως η μοριακή μοντελοποίηση, δηλαδή στον υπολογισμό της συμπεριφοράς των μορίων. Ακόμα, λαμβάνει χώρα και σε υπολογισμούς όπου βοηθουν να κατανοήσουμε την αλληλεπίδραση των σωματιδίων στο σύμπαν.

3.2.5 Map Reduce & Monte Carlo

Ο νάνος αυτός ονομαζόταν αρχικά "Monte Carlo", εξαιτίας της τεχνική της χρήσης στατιστικών μεθόδων βασισμένων σε επαναλαμβανόμενες τυχαίες δοκιμές. Τα μοτίβα που ορίζονται από το μοντέλο προγραμματισμού Map Reduce είναι μια πιο γενική έκδοση της ίδιας ιδέας δηλαδή η επαναλαμβανόμενη ανεξάρτητη εκτέλεση μιας λειτουργίας, όπου μας βοηθά να καταλήξουμε σε κάποια συγκεντρωτικά αποτελέσματα στο τέλος. Σχεδόν δεν απαιτείται επικοινωνία μεταξύ των διαδικασιών. Εφαρμόζονται σε κατανεμημένες αναζητήσεις, τρόπος που χρησιμοποιείται και στις αναζητήσεις στο google και στη βιοπληροφορική όπου χρησιμοποιείται για την ευθυγράμμιση του DNA ή των πρωτεινών για τον προσδιορισμό ομοιότητας των στοιχείων

3.2.6 Combinational Logic

Εκτελούνται απλές εργασίες σε μεγάλους όγκους δεδομένων συγκεντρώνοντας το αποτέλεσμα σε μία τελική λύση. Χρησιμοποιείται κυρίως για τεχνικές αποκρυπτογράφησης, κρυπτογράφησης και γενικά ότι έχει να κάνει με εκτέλεση απλών διεργασιών σε πολύ μεγάλες ποσότητες δεδομένων. Είναι πολύ απλή σε σχέση με άλλους νάνους.

3.2.7 Dynamic Programming

Είναι μια αλγοριθμική τεχνική όπου υπολογίζει λύσεις με την επίλυση απλούστερων υπερκαλυπτόμενων υποπροβλημάτων. Είναι ιδιαίτερα εφαρμόσιμο σε προβλήματα βελτιστοποίησης. Η τεχνική αυτή χρησιμοποιείται σε προβλήματα γράφων, όπως για την επίλυση του αλγορίθμου Floyd-Warshall που βρίσκει τον μικροτερο και πιο αποτελεσματικό μονοπάτι σε ένα γράφο. Επίσης χρησιμοποιούνται και στην βιοπληροφορική για την ευθυγράμμιση αλληλουχιών όπως και η εύρεση ομοιοτήτων για το DNA και το RNA

3.2.8 Backtrack and Branch and Bound

Οι αλγόριθμοι αυτοί είναι αποτελεσματικοί για την επίλυση διαφόρων προβλημάτων αναζήτησης και βελτιστοποίησης. Η αλγοριθμική μέθοδος backtrack and branch and bound λειτουργεί με την αρχή του διαίρει και κυρίευε. Ο χώρος αναζήτησης υποδιαιρείται σε μικρότερος χώρους (branching) και τα όρια που βρέθηκαν κρατούνται σε κάθε υποπεριοχή για να ελεγχθουν στη συνέχεια. Όταν τα όρια μιας μεγάλης υποπεριοχής περιέχει τιμές οι οποίες δεν αντιστοιχούν σε λύσεις του προβλήματος τότε καταρρίπτουμε ολόκληρη τη περιοχή. Πολλά προβλήματα μπορούν να επιλυθούν με αυτόν τον αλγόριθμο όπως integer linear programming όπου είναι μια μαθηματική βελτιστοποίηση όπου οι τιμές του προγράμματος πρέπει να είναι αυστηρά ακέραιοι. Ακόμα βρίσκει εφαρμογή και στο πρόβλημα Boolean satisfiability όπου ελέγχει εάν υπαρχει ερμηνεία που να ικανοποιεί μια δεδομένη Boolean σχέση.

3.2.9 Finite State Machine

Συνδέει τα σημεία μετάβασης από μία κατάσταση σε μία άλλη.Εφαρμόζεται σε τομείς όπως αποκωδικοποιήσεις video και datamining, δηλαδή μια μορφή υπολογισμού προτύπων με μεγάλο όγκο δεδομένων. Χρησιμοποιούνται κυρίως στους κλάδους της στατιστικής και των βάσεων δεδομένων.

3.2.10 GraphTraversal

Διασχίζουν γραφήματα με μεγάλο αριθμό αντικειμένων εξετάζοντας τα χαρακτηριστικά του κάθε αντικειμένου. Οι εφαρμογές αυτές συνήθως περιλαμβάνουν έμμεσες αναζητήσεις και ελάχιστους υπολογισμούς. Αυτές οι εφαρμογές παίζουν σημαντικό ρόλο σε περιπτώσεις αναζητήσεων, ταξινομήσεων και ανίχνευση συγκρούσεων.

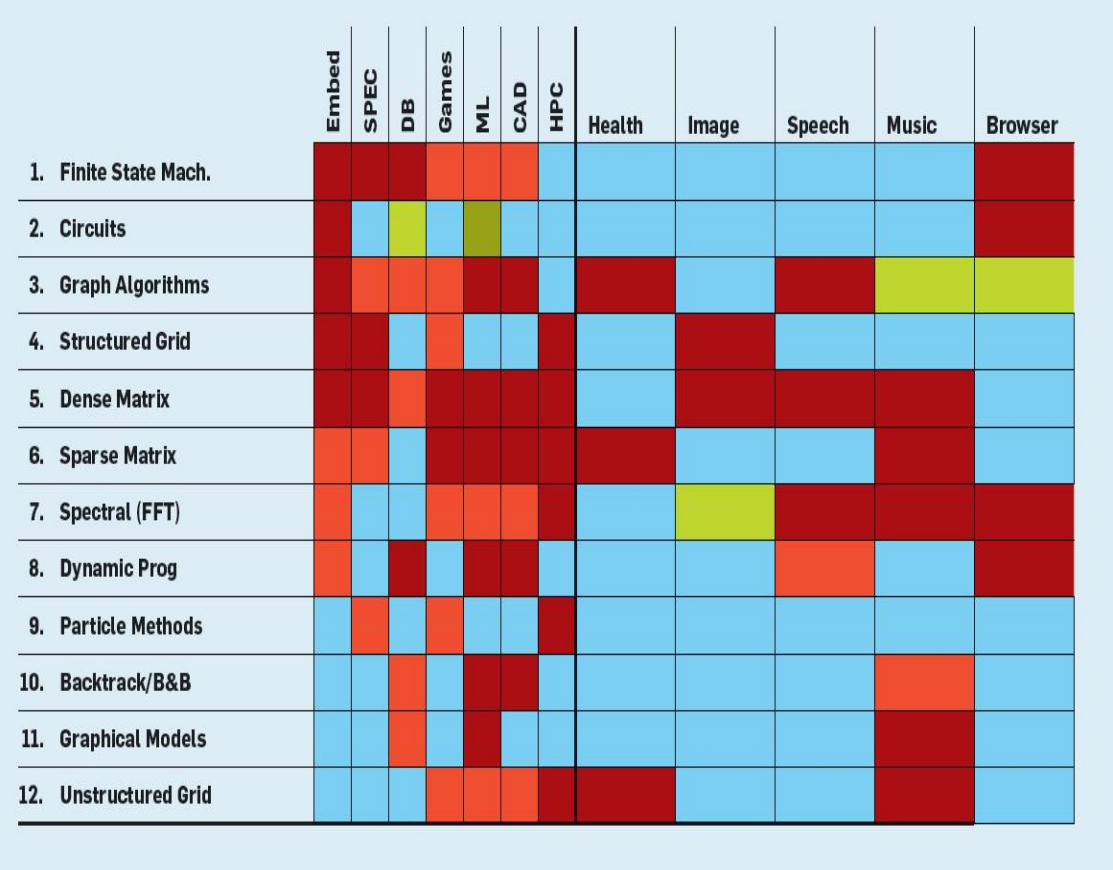
3.2.11 Construct Graphical Models

Είναι ένα γραφικό μοντέλο όπου οι κόμβοι αντιπροσωπεύουν μεταβλητές και οι άκρες αντιπροσωπεύουν υπό συνθηκη πιθανότητες. Χρησιμοποιούνται κυρίως σε τομείς όπως η βιολογία όπου με στατιστικές εκτιμήσεις μπορούν να μεταφέρουν χρήσιμες πληροφορίες σχετικά με το γενετικό υλικό. Ακόμα επιλύουν μοντέλα Markov όπου χρησιμοποιούνται για την χρονική αναγνώριση προτύπων όπως χειρογραφία και ομιλίες.

Οι μέθοδοι structured και unstructured grid θα αναλυθούν αμέσως μετά εκτενέστερα δίοτι έιναι οι δύο μέθοδοι που μας απασχόλησαν στη συγκεκριμένη διπλωματική και αυτοί όπου χαρτογραφήθηκαν στο εργαλείο της Vivado hls.

3.2.12 Αποτελεσματικότητα νάνων και κλάδων εφαρμογής

Οι παραπάνω μέθοδοι εξετάστηκαν για την αποτελεσματικότητα τους στα ενσωματωμένα συστήματα,σε βάσεις δεδομένων, textmining, παιχνίδια, γραφικά και μηχανική μάθηση. Ακόμα έπαιξαν σημαντικό ρόλο όσο αναφορά την βελτίωση της απόδοσης σε τομείς όπως η ιατρική αλλά και σε διάφορους άλλους τομείς όπως η επεξεργασία εικόνας, βίντεο, φωνής και μουσικής. Παρακάτω, ο πίνακας αναδεικνύει το χώρο εφαρμογής μερικών απο των παραπάνω νάνων. Ανάλογα με την ένταση του χρώματος των κελιών καταλαβαίνουμε την αυξημένη δράση τον μεθόδων στο συγκεκριμένο πεδίο.



Εικόνα 1: Τομείς εφαρμογής των νάνων

3.3 Δομημένα Πλέγματα

Ένα πρόβλημα δύο, τριών και περισσότερων διαστάσεων μπορεί να περιγραφεί με ένα δομημένο πλέγμα το οποιο υλοποιείται σαν ένα ν-διάστατο πίνακα. Το σχήμα των δομημένων πλεγμάτων είναι στατικό και προσδιορίσιμο. Στην επιστήμη της μηχανικής είναι συχνά απαραίτητη η παρουσίαση σημείων ενός πλέγματος και των τιμών τους τα οποία ενημερώνονται από τα γειτονικά τους σημεία. Η τιμή αυτή μπορεί να προσωμοιάζει τη θερμοκρασία, το χρώμα ενός pixel μιας εικόνας κ.α. Κάθε σημείο εξόδου υπολογίζεται από την τιμή που περιέχει το κάθε στοιχείο και την τιμή των πλησιέστερων γειτονικών σημείων. Τα δεδομένα παρουσιάζονται σαν έναν κανονικό πολυδιάστατο πλέγμα όπου πραγματοποιείται συνεχής ενημέρωση του πλέγματος και μάλιστα η ενημέρωση αυτή γίνεται ταυτόχρονα σε όλα τα στοιχεία χρησιμοποιώντας τις τιμές των γειτονικών τους στοιχείων. Αυτή η ενημέρωση μπορεί να γίνεται παράλληλα σε όλα τα σημεία του πλέγματος αρκεί βέβαια να υπάρχουν δύο

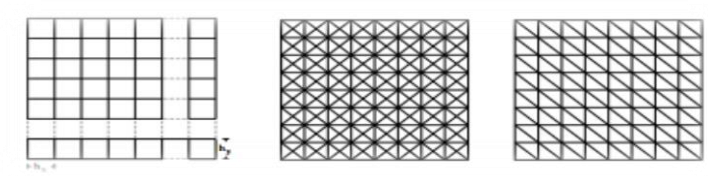
αντίγραφα του πλέγματος για να μην δημιουργηθεί πρόβλημα με τις γειτονικές τιμές και σταματάει μέχρι να φτάσει σε ένα συγκεκριμένο όριο σφάλματος. Η αφορμή της δημιουργίας των δομημένων πλεγμάτων υπήρξε ο υπολογισμός της διανομής θερμότητας με τον υπολογισμό μερικών διαφορικών εξισώσεων σε πολυδιάστατες συστοιχίες. Η παράλληλη χαρτογράφηση πραγματοποιείται με την αντιστοίχηση των υποπλεγμάτων σε ενα επεξεργαστή. Ένα σημαντικό πλεόνεκτημα των πλεγμάτων είναι ότι οι διευθύνσεις της μνήμης είναι έυκολα υπολογίσιμες, κάτι το οποίο μας βοηθά να κάνουμε ευκολότερα την χαρτογράφηση μας στο hardware. Ακόμα οι μέθοδοι αυτόι έχουν και μεγάλη χρονική τοπικότητα δίοτι επιτρέπουν την επαναχρησιμοποίηση δεδομένων σε σχετικά μικρό χρονικό διάστημα.

Η πρώτη διεργασία για την επίλυση ενός προβλήματος με δομημένα πλέγματα είναι με την τεχνική της αποσύνθεσης. Το πλέγμα είναι χωρισμένο σε γειτονικά κομμάτια και σε κάθε γειτονικό κομμάτι έχει ανατεθεί μια εργασία. Η πιο απλή περίπτωση ενός δομημένου πλέγματος είναι η λύση προβληματων που σχεδιάζονται σε 1, 2, 3 διαστάσεις πλεγμάτων. Κάθε επεξεργαστής έχει εκχωρηθεί σε ένα κομμάτι του πλέγματος που καθορίζεται στατικά η δυναμικά από ένα αλγόριθμο διαμέρισης. Κάθε επεξεργαστής υπολογίζει τότε τις ενημερώσεις των σημείων του πλέγματος σε κάθε επανάληψη.

Μία σημαντική μέθοδος που παίζει ρόλο στην ενημέρωση των σημείων ενός δομημένου πλέγματος είναι η μέθοδος double buffering. Χρησιμοποιούμε δύο αντίγραφα δεδομένων, ένα για τα δεδομένα που πρόκειτε να υπολογίσουμε και ένα για τα δεδομένα της προήγουμενης επανάληψης. Αυτό επιτρέπει την πρόσβαση στις τιμές που απαιτούνται για την ενημέρωση του συστήματος. Αυτή τη μέθοδο θα την συναντήσουμε και στη μέθοδο Jacobi όπου θα δούμε παρακάτω.

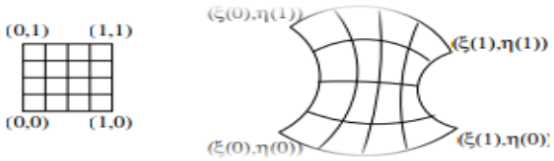
Υπάρχουν διάφορες μορφές δομημένων πλεγμάτων όπως για παράδειγμα:

- **Regular Static Grid**: Ορθογώνια ή καρτεσιανά πλέγματα κυρίως δύο ή τριών διαστάσεων. Οι διαστάσεις των πλεγμάτων παραμένουν όπως ήταν στην αρχή με αποτέλεσμα να παρέχεται υψηλή χωρική τοπικότητα. Ωστόσο,αν και βοηθάει στην χρήση δεδομένων γειτονικών θέσεων, έχει περιορισμένη χρονική τοπικότητα.



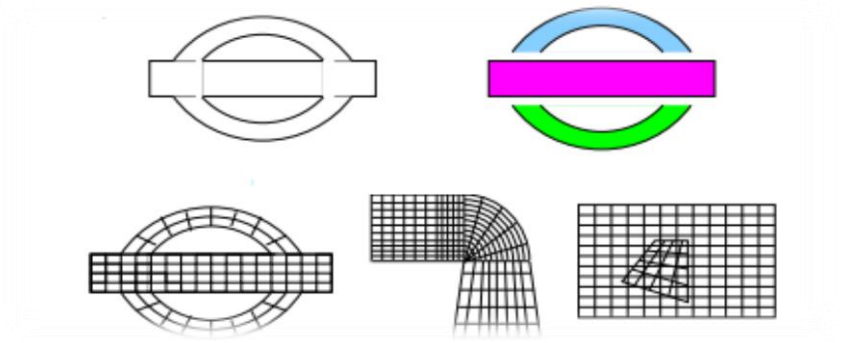
Εικόνα 2: Μορφή regular static grid

- **Transformed Structured Grids:** Τα regular static grids μετασχηματίζονται σε transformed structured grids Το μοναδιαίο τετράγωνο μετασχηματίζεται σε υπολογιστικό τομέα, όπου στην ουσία είναι μια πιο απλοποιημένη μορφή του φυσικού τομέα ως προς τη γεωμετρική αναπαράσταση. Αυτή η μορφή διατηρεί τα φυσικά σημαντικά χαρακτηριστικά, αλλά συγχρόνως αγνοεί μικρές λεπτομέρειες απλοποιώντας το σύστημα.



Εικόνα 3: Μορφή transformed structured grids

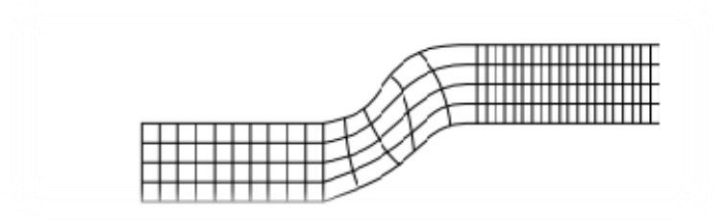
- **Composite Structured Grids:** Υποδιαίρεση πολύπλοκων πλεγμάτων σε απλούστερες μορφές και στη συνέχεια χρησιμοποιούνται regular meshes για κάθε τομέα που διαμερίσαμε.



Εικόνα 4: Μορφή composite structured grids

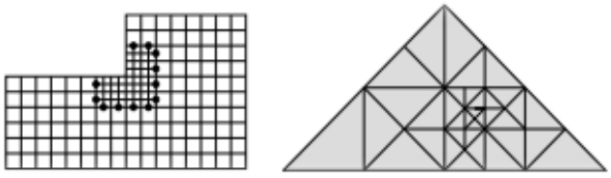
- **Block Structured Grids:** Είναι ειδική περίπτωση των σύνθετων πλεγμάτων, όπου γίνεται υποδιαίρεση σε λογικά ορθογώνια υποπλέγματα. Τα υποπλέγματα αυτά

δημιουργούνται με αδόμητο τρόπο, ωστόσο εξασφαλίζεται η σύνεχεια τους. Αρκετά δημοφιλής είναι στην υπολογιστική ροή ο κλάδος της μηχανικής των ρευστών. Βοηθούν και βελτιώνουν τους υπολογισμούς που απαιτούνται για την προσομοίωση της αλληλεπίδρασης των υγρών με των αερίων.



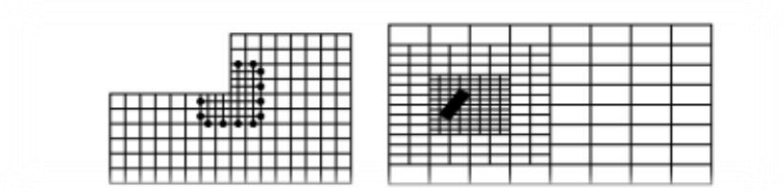
Εικόνα 5: Μορφή block structured grids

- **Adaptive Grids:** Βελιώνει τοπικά την απόδοση των στοιχείων όπου θέλουμε να εστιάσουμε και επιθυμούμε υψηλότερη ανάλυση. Παρέχει υψηλή χρονική και χωρική τοπικότητα. Οι εφαρμογές των adaptive πλεγμάτων βοήθησαν αρκετά τους τομείς της αστροφυσικής, τη κλιματική μοντελοποιήση, βιοφυσική και αρκετούς άλλους τομείς.



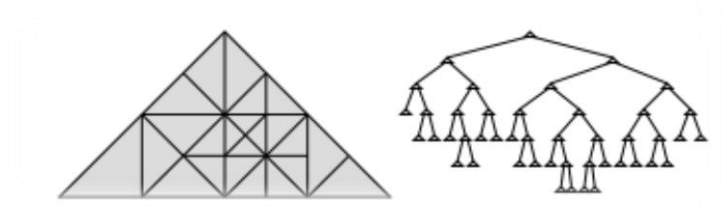
Εικόνα 6: Μορφή adaptive grids

- **Block Adaptive Grids:** Έιναι παρόμοια με τα block structured grids. Διατηρούν την κανονική τους δομή με αποτέλεσμα να έιναι αποδοτικοί στην αποθήκευση και την επεξεργασία ,ωστόσο έχουν μεγάλο πρόβλημα στην προσαρμοστικότητα.



Εικόνα 7: Μορφή block adaptive grids

- **Recursively Structured Adaptive Grids:** Βασίζονται στην παραλληλοποίση των πατεράδων κελιών το οποίο οδηγεί στην μετατροπή τους σε δομές δέντρων. Είναι αρκετά ευέλικτα ωστόσο έχουν μειώνεκτημα την χρήση αναδρομικών συναρτήσεων, οι οποίες είναι αρκετά πολύπλοκες.



Εικόνα 8: Μορφή recursively structured adaptive grids

Με τη βοήθεια των πλεγμάτων, βελτιστοποιήθηκαν αρκετές εφαρμογές κυρίως στο τομέα της επεξεργασίας εικόνας, εξελίσσοντας και τον τομέα της ιατρικής παράλληλα. Τα πλέγματα, εστιάζουν ακόμα σε εφαρμογές ενσωματωμένων συστημάτων και λιγότερο σε παιχνίδια.

Παρέχουν ακόμα την δυνατότητα της χαρτογράφησης ενός uniprocessor, δηλαδή ενός συστήματος με μια κεντρική μονάδα επεξεργασίας όπου εκτελένουνται διάφορα tasks. Τα σημεία του πλέγματος μπορούν να χρησιμοποιήθουν με οποιαδήποτε σειρά παρέχοντας έτσι υψηλη χωρική και χρονική τοπικότητα. Επίσης,υποστηρίζουν την παραλληλοποίηση ενός συστήματος κάτι το όποιο ανεβάζει την απόδοση του αλγορίθμου κατακόρυφα. Η δυνατότητα της παράλληλης χαρτογράφησης, όπου κάθε υποπλέγμα αντιστοιχίζεται σε ένα επεξεργαστή επιτυγχάνει την μείωση της καθυστέρησης στέλνωντας μεγάλυτερο όγκο δεδομένων σε λιγότερους κύκλους ρολογιού.

Ο αλγόριθμος που επιλέξαμε να χαρτογραφήσουμε στο hardware και να βελτιστοποιήσουμε με τη βοήθεια των directives που μας παρέχει το Vivado HLS είναι η μέθοδος Jacobi. Ο αλγόριθμος αυτός είναι μια αναπαράσταση του πλεγματος, όπου γίνεται ταυτόχρονη ενημέρωση των στοιχείων.

3.4 Μέθοδος Jacobi

Ο αλγόριθμος αυτός χρησιμοποιείται κυρίως για την έυρεση ριζών γραμμικών εξισώσεων και συστημάτων. Το γραμμικό σύστημα **A*x=b** μπορεί να γραφεί και στην μορφή αθροίσματος, δηλαδή :

$$\sum_{k=1}^{n} a_{i,k}x_k = b_k \,, \mathrm{i} = 1,2,3 \ldots \mathrm{n}$$

Αναλύουμε το άθροισμα σε 3 όρους ως εξής:

$$\sum_{k=1}^{i-1} a_{i,k}x_k + a_{i,i}x_i + \sum_{k=\mathrm{i}+1}^{n} a_{i,k}x_k \,, \mathrm{i} = 1,2,3 \ldots \mathrm{n}$$

και επιλύουμε την σχέση αυτή ως προς x_i δηλαδή:

$$x_i = \frac{1}{a_{i,i}}\left(b_i - \left(\sum_{k=1}^{i-1} a_{i,k}x_k + \sum_{k=\mathrm{i}+1}^{n} a_{i,k}x_k \right) \right) , \mathrm{i} = 1,2,3 \ldots \mathrm{n}$$

Η τελευτάια αυτή μας οδηγεί στην λεγόμενη μέθοδος Jacobi :

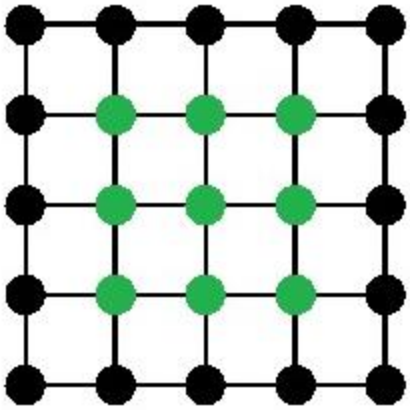
$$x_i^{(m+1)} = \frac{1}{a_{i,i}}\left(b_i - \left(\sum_{k=1}^{i-1} a_{i,k}x_k^{(m)} + \sum_{k=\mathrm{i}+1}^{n} a_{i,k}x_k^{(m)} \right) \right) , \mathrm{i} = 1,2,3.., \mathrm{n}\kappa\alpha\iota\mathrm{m} = 0,1,2, \ldots$$

όπου m δηλώνει τον αριθμό των επαναλήψεων.Κρίτηριο τερματισμού της συνθήκης αυτής συνήθως είναι η σχέση:

$$||x_i^{(m+1)} - x_i^{(m)}||\infty < \varepsilon$$

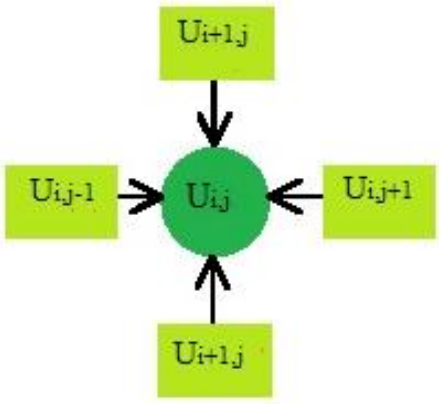
όπου ε είναι η ζητούμενη ακρίβεια, η οποία εκφάζει την μέγιστη διαφορά των στοιχείων μεταξύ δύο διαδοχικών προσεγγίσεων του διανύσματος της λύσης.

Σε ένα δομημένο πλέγμα ο υπολογισμός της μεθόδου Jacobi πραγματοποιείται ως εξής. Έστω ότι έχουμε ένα πίνακα 5x5 η μέθοδος Jacobi εφαρμόζεται στις πράσινες περιοχές του πίνακα όπως φαίνεται στην παρακάτω εικόνα.



Εικόνα 9: Περιοχές εφαρμογής μεθόδου Jacobi

Κάθε τμήμα της περιοχής αυτής υπολογίζει τη νέα του τιμή με βάση τις γειτονικές του τιμές. Δηλαδή ο υπολογισμός της τιμής ενός κελιού υπολογίζεται διαιρώντας το άθροισμα των γειτονικών του τεσσάρων σημείων με τον αριθμό των γειτόνων δηλαδή με το 4, όπως φαίνεται στο σχήμα.



Εικόνα 10: Εφαρμογή μεθόδου Jacobi σε ένα σημείο

Το γεγονός, ότι για να προχωρήσουμε στην επόμενη επανάληψη πρέπει να έχουμε τα αποτελέσματα της προηγούμενης, η μέθοδος Jacobi καθίσταται αρκετά αργή σε σχέση με άλλους αλγορίθμους. Ωστόσο,είναι αρκετά χρήσιμη διότι αποτελεί βάση για άλλες μεθόδους. Επομένως, υλοποιώντας την μέθοδο αυτή στο εργαλείο Vivado HLS θα κάνουμε χρήση της παραλληλοποιήσης βελτιώνοντας τον χρόνο εκτέλεσης της.

3.5 Μη Δομημένα Πλέγματα

Το σύνολο των δεδομένων τυπικά ορίζεται σαν ένα πλέγμα που καλύπτει τις επιφάνειες ή τους όγκους αυτών των αντικειμένων. Τα στοιχεία ενός μη δομημένου πλέγματος εμφανίζουν αρκετές ομοιότητες με τα στοιχεία των δομημένων πλεγμάτων εφόσον οι τιμές τους υπολογίζονται ομοίως με τις κατάλληλες αριθμητικές πράξεις των τιμών που βρίσκονται στα γειτονικά στοιχεία. Η ουσιαστική τους διαφορά είναι ότι ανήκουν σε ένα ακαθόριστο πλέγμα. Επομένως το κάθε στοιχείο δεν έχει τον ίδιο αριθμό γειτόνων. Το πλέγμα συνήθως αντιπροσωπεύει κάποιον όγκο ή επιφάνεια και οι αριθμητικές πράξεις που γίνονται συνήθως είναι λύσεις διαφορικών εξισώσεων.

Τα μη δομημένα πλέγματα διαθέτουν δομές δεδομένων συνδεδεμένες με μια λίστα με δείκτες όπου παρακολουθούν την τοποθεσία και την “γειτονιά” των στοιχείων που χρησιμοποιούνται για τον υπολογισμό των αριθμητικών πράξεων που απαιτούνται. Όπως και στη sparse γραμμική άλγεβρα,οι ενημερώσεις περιλαμβάνουν πολλαπλές αναφορές στην μνήμη. Για την ενημέρωση ενος σημείου πρώτα καθορίζεται μια λίστα με τα γειτονικά σημεία και μετά φορτώνει τις τιμές τους.

Οι εφαρμογές τους συνήθως απαιτούν την μοντελοποίηση φυσικών ποσοτήτων όπως θερμότητα και πίεση όπου συσχετίζονται με το χρόνο. Δηλαδή κατά πόσο αυτές οι ποσότητες άλλαξαν με την παροδο του χρόνου. Τα σύνολα των μη δομημένων πλεγμάτων μπορεί να αποτελούνται από μεγάλο όγκο δεδομένων. Λαμβάνουν χώρα κυρίως σε εφαρμογές που δραστηριοποιούνται στο κλάδο της δυναμικής των ρευστών όπου στην ουσία είναι ένας κλάδος που χρησιμοποιεί αριθμητικές αναλύσεις και δομές δεδομένων για την επίλυση και την ανάλυση προβλημάτων σχετικά με τα ρευστά. Για παράδειγμα ο σχεδιασμός των φτερών των αεροπλάνων αποτελεί μοντελοποίηση της δυναμικής των ρευστών. Στην περίπτωση αυτή η επιφάνεια των

μορφών αεροτομής ορίζεται σαν ένα πλέγμα όπου οι γεωμετρικές ιδιότητες καθορίζουν στην ουσία τη λύση του προβλήματος.

Αυτοί οι κώδικες έχουν υψηλό βαθμό παραλληλοποίησης,ωστόσο επειδή η λήψη των τιμών γίνονται μέσω της προσπέλασης της μνήμης και της ανάκτησης των διευθύνσεων οδηγεί σε κακή χωρική τοπικότητα.

Η συλλογή των δεδομένων μπορούν να αναπαρασταθουν σαν ένα γράφημα όπου τα άκρα αντιπροσωπεύουν την γεωμετρική σχέση με τον πλησιέστερο γείτονα. Τα πρότυπα επικοινωνίας σε τέτοιου είδους προβληματα ακολουθούν τη σχέση του πλησιέστερου γείτονα. Δηλαδή η τιμή ενός σημείου θα εξαρτηθεί στο μέλλον από τις τιμές της γεωμετρικής γειτονιάς του.

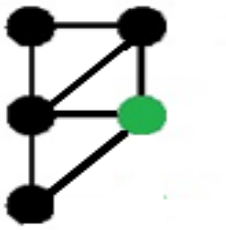
Παρέχουν την δυνατότητα παράλληλης χαρτογράφησης εφόσον η δομή του πλέγματος μπορεί να διαιρεθεί σε υποπεριοχές. Ωστόσο η χαρτογράφηση τους στο hardware ορισμένες φορές έιναι αρκετά δύσκολη όταν συγκεκριμένα αφορούν ακάνονιστα πλέγματα διότι το memory latency έιναι περιορισμένο.

3.6 Μέθοδος Serial Based

Ο αλγόριθμος που υλοποιήθηκε για τη συγκεκριμένη αλγοριθμική μέθοδο των μη δομημένων πλεγμάτων έχει αρκετές ομοιότητες με την μέθοδο Jacobi. Όπως, γνωρίζουμε υπάρχει μια σημαντική διαφορά μεταξύ των δομημένων και των μη δομημένων πλεγμάτων και αυτό αφορά το σχήματους. Όπως γνωρίζουμε στη δεύτερη περίπτωση το σχήμα τους είναι ακαθόριστο. Γι αυτό, το λόγο κάθε σημείο του πλέγματος δεν έχει συγκεκριμένο αριθμό γειτόνων. Επομένως, στο συγκεκριμένο αλγόριθμο υπολογίζουμε την τιμή μιας κορυφής του κελιού με το άθροισμα των γειτονικών τιμών που εμπεριέχονται στα γειτονικά κελιά. Το άθροισμα που προκύπτει το διαιρούμε με τον αριθμό των γειτόνων.

Αξίζει να σημειωθεί ότι το πλέγμα αυτό είναι χωρισμένο σε τρίγωνα και οι κορυφές είναι κατα αύξουσα σειρά επομένως τα γειτονικά κελιά απέχουν δύο η τρεις θέσεις από τη τιμή του

κελιού που επιθυμούμε να υπολογίσουμε. Ωστόσο, εφόσον δεν έχει συγκεκριμένο αριθμό γειτόνων το κάθε κελί ,πρώτα λαμβάνει τις τιμές από ενα πίνακα που εμπεριέχει τις θέσεις που βρίσκονταιτα γειτονικά κελιά και έπειτα γίνεται η πράξη της πρόσθεσης.



Εικόνα 11: Παράδειγμα εφαρμογής της μεθόδου serial based

Όπως, βλέπουμε και στο παραπάνω παράδειγμα το σημείο που μας ενδιαφέρει έχει τρία γειτονίκα σημεία. Στο συγκεκριμένο πλέγμα άλλο ένα έχει τρία γειτονικά σημεία και όλα τα υπόλειπα έχουν δύο.

Επίσης στον αλγόριθμο αυτό όπως γίνεται και στην αλγοριθμική μέθοδο των μη δομημένων πλεγμάτων η ενημέρωση και ο υπολογισμός των στοιχείων γίνεται ταυτόχρονα. Εν τέλει ο αλγόριθμος των μη δομημένων πλεγμάτων έχει την παρακάτω μορφή :

```
for(j = 0; j < numberOfNeighbors; j ++)
      {

tempArray[i] += ArrayUnstructureGrid[neighbors[j+i]];

      }
TempArray[i] /= numberOfNeighbors;
```

Συνοψίζοντας, στο κεφάλαιο αυτό παρουσιάσαμε τα δομημένα και μη δομημένα πλέγματα, αλγοριθμικές μεθόδους που ανήκουν στους δεκατρείς νάνους. Επίσης,αναφέρονται οι αλγόριθμοι που θα χαρτογραφηθούν στο hardware με σκόπο τη βελτίωση της απόδοσης τους. Οι αλγόριθμοι αυτοί υπάγονται στα δομημένα και μη δομημένα πλέγματα και είναι οι μέθοδοι Jacobi,Serial Based.

Κεφάλαιο 4

Στο συγκεκριμένο κεφάλαιο περιγράφεται η αρχιτεκτονική decoupledaccess/executeκαι τα βήματα που απαιτούνται για την υλοποίηση της. Αναλύεται λεπτομερώς η δομή της και πόσο συμβάλειστην βελτίωση της απόδοσης και της κατανάλωσης ενέργειας μιας εφαρμογής.

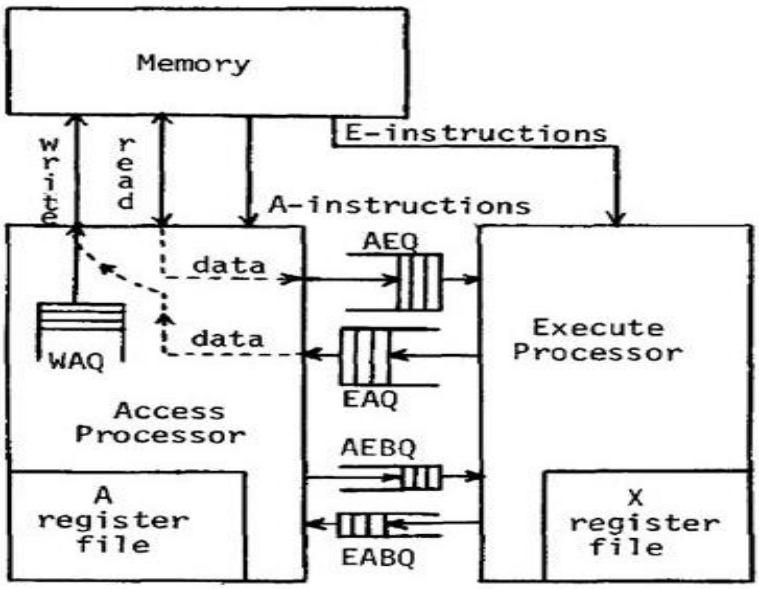
4.1 Περιγραφή Αρχιτεκτονικής DAE

Τα τελευταία χρόνια οι απαιτήσεις για την επεξεργασία των δεδομένων αυξάνονται και μαζί τους και ο όγκος των δεδομένων. Γι αυτό το λόγο, οι περισσότερες προσπάθειες επικεντρώνονται σε υπολογιστικά συστήματα όπου προσπαθούν να συνδυάσουν την ευελιξία του software με την παροχή της υψηλής απόδοσης απο το hardware. Ωστόσο, ορισμένες φορές ο μεγάλος όγκος δεδομένων μπορεί να προκαλέσει σοβαρές καθυστερήσεις και μειώνει την προσπάθεια για επιτάγχυνση του προγράμματος από τους επιταγχυντές του υλικού. Για την επιλύση αυτού του προβλήματος ο χρήστης θα πρέπει να βελτιστοποιήσει το σύστημα της ανάκτησεις των δεδομένων.

Έτσι γεννήθηκε η ιδέα της αρχιτεκτονικής decouple access/execute που προήλθε από το James E. Smith το 1982. Στην προσέγγιση του οι μονάδες εκτέλεσης δεν περιέιχαν υπολογισμούς διευθύνσεων και ήταν αρμόδιες μόνο για τις αριθμητικές και λογικές πράξεις. Στόχος της αρχιτεκτόνικής αυτής ήταν η μείωση της εκτέλεσης του προγράμματος με ελάχιστη δυνατή επέμβαση από το χρήστη.

Είναι αρχιτεκτονική υψηλών αποδόσεων και βασίζεται στην μεγάλη αποσύζευξη της πρόσβασης και της επεξεργασίας δεδομένων. Το αρχικό instruction stream χωρίζεται σε δύο λειτουργικές μονάδες, στη μονάδα ανάκτησης δεδομένων και στη μονάδα επεξεργασίας δεδομένων όπου επικοινωνούν μεταξύ τους με FIFO ουρές. Συγκρίνοντας την αρχιτεκτονική DAE με μία άλλη συμβατή αρχιτεκτονική αντιλαμβανόμαστε ότι υπάρχει μεγάλη διαφορά στο κέρδος απόδοσης, διότι μειώνει την πολυπλοκότητα στη σχεδίαση και την επικοινωνία του

επεξεργαστή με την κύρια μνήμη. Πειραματικά αποτελέσματα εχουν δείξει οτι οι βελτιστοποιήσεις μπορουν να κάνουν πιο αποδοτικό το πρόγραμμα 2,28 πιο γρήγορο και να μειώσουν την κατανάλωση ενέργειας κατά 15%.



Εικόνα 12: Αρχιτεκτονική DAE

Κάθε μονάδα έχει το δικό της instruction stream, τον Access processor και τον Execute processor. Οι δύο αυτοί επεξεργαστές, εκτελούν ξεχωριστά προγράμματα με διαφορετικές λειτουργίες αλλά έχουν παρόμοιο διάγραμμα ροής. Έπειτα, το κάθε ένα instruction stream έχει το δικο τους αρχείοκαταχωρητών που αποτελείται από 32 καταχωρητές ακεραίων. Στον επεξεργαστή A οι καταχωρητές δηλώνονται ως καταχωρητές A0,A1.. ενώ στον επεξεργαστή E ,ως X0,X1... Τα δύο stream είναι ανεξάρτητες λειτουργικές μονάδες όπου επικοινωνούν με την μνήμη μέσω ουρών.

Ο Access processor έιναι υπεύθυνος, για την προσπέλαση της μνήμης εκτελώντας όλες τις απαραίτητες λειτουργίες για την μεταφορά δεδομένων από και προς την μνήμη. Δηλαδή υπολογίζει τις διευθύνσεις και εκτελεί όλες τις αιτήσεις αναγνώσης και εγγραφής που είναι απαραίτητες. Γι αυτό το λόγο αφαιρούμε στο κομμάτι αυτό ότι έχει να κάνει με αριθμητικές και λογίκες πράξεις, οι οποίεις βρίσκονται μόνο στον execute processor. Έπειτα τα δεδομένα είτε χρησιμοποιούνται στον access είτε τοποθετούνται στην AEQ(AccesstoExecute)FIFO ουρά και αποστέλλονται στο execute processor. Είναι αναγκαίο ο access processor να εκτελεστεί πρώτα

απο τον execute processor ώστε να έχει γίνει η ανάκτηση των δεδομένων που θα χρειαστούν για την επεξεργασία. Επίσης, με την προεπεξεργασία των δεδομένων στον access processor, εξαλείφονται οι περισσότερες αστοχίες της μνήμης cache και μετατρέπονται σε ευστοχίες βοηθώντας αρκετά τον execute processor. Έτσι, στον execute processor όπου έχουν εξαλειφθεί οι περισσότερες αστοχίες μειώνονται αισθητά οι καθυστερήσεις και συγχρόνως η απόδοση του προγράμματος βελτιώνεται αισθητά.

Εκεί λοιπόν, εφόσον λάβει ο execute processor όλα τα απαραίτητα δεδομένα που έρχονται απο την AEQ, τα επεξεργάζεται τα και τα χρησιμοποιέι για να εκτελέσει τις πράξεις που απαιτούνται για την επίλυση του αλγορίθμου. Στη συνέχεια εισέρχονται στην EAQ(ExecutetoAccess) FIFO ουρά όπου αποστέλλονται πίσω στον access processor. Τέλος, αφού εκδοθεί η εντολή αποθήκευσης αποστέλλονται στην μνήμη για την αποθηκευση των αποτελεσμάτων.

Ο υπολογισμός των διευθύνσεων απο τον access processor και τα αποτελέσματα που παράγει ο execute processor πραγματοποιούνται παράλληλα. Ο access processor παράγει τις διευθύνσεις όπου πρέπει να αποθηκευτούν τα αποτελέσματα χωρίς απαραίτητα να έχουν έρθει τα αποτελέσματα απο τον execute processor. Οι διευθύνσεις αυτές κρατούνται παράλληλα στην WAQ(writeaddress) και όταν φθάσουν τα δεδομένα σε συνδυασμό με την πρώτη διεύθυνση της WAQ αποστέλλονται στη μνήμη. Αυτό ο συνδυασμός γίνεται αυτόματα με το που φθάσουν τα δεδομένα.

Αξίζει να σημειωθεί ότι υπάρχει μια τρίτη λειτουργική μονάδα ξεχωριστη από τον A και E επεξεργαστή όπου χειρίζεται τα δεδομένα και τις διευθύνσεις όπου χρειάζονται να γραφούν. Ο EAQ επίσης μπορεί να χρησιμοποιηθεί για την μεταφορά δεδομένων στον A processor τα όποια δεν πρόκειτε να γραφούν στην μνήμη αλλά χρησιμοποιούνται για τον υπολογισμο διευθύνσεων. Σε κάποιες περιπτώσεις μπορεί να χρειαστεί διπλός υπολογισμός και στους δυο επεξεργαστές ώστε να αποφευχθεί το γεγονός να περιμένει ο A τα δεδομένα απο τον E επεξεργαστή. Όταν πάμε να χρησιμοποιήσουμε την αρχιτεκτονική DAE σε συνδυασμό με το software πρέπει να έιμαστε αρκετά προσεχτικοί και να συντονίζουμε κατάλληλα τα δεδομένα που επέρχονται από τους δύο επεξεργαστές έτσι ώστε να μεταδίδονται από τις ουρές με τη σωστή σειρά. Σε αρκετές περιπτώσεις το access stream προηγείται του execute με αποτέλεσμα να έχουμε σημαντικές μειώσεις όσο αναφορά την καθυστέρηση της ανάκτησης των δεδομένων από την μνήμης.

Όπως αναφέραμε, ο υπολογισμός των διευθύνσεων μπορεί να υπολογισθεί πριν ακόμα τα δεδομένα είναι έτοιμα. Οι διευθύνσεις αυτές παραμένουν στο WAQ. Τα δεδομένα διαβιβάζονται από το EAQ στο WAQ πριν πάνε στην μνήμη. Η ανάκτηση των διευθύνσεων πριν ακόμα τα δεδομένα να είναι έτοιμα, είναι σημαντικός παράγοντας για την βελτίωση της επίδοσης του προγράμματος διότι φορτώνει νέες εντολές χωρίς να έχει γίνει η αποθήκευση της προηγούμενης .Ωστόσο, δημιουργείται ένα σημαντικό πρόβλημα ορισμένες φορές, όπου η νέα εντολή που φορτώνεταιχρησιμοποιεί την ίδια θέση μνήμης με την προηγούμενη. Γι αυτό το λόγο, ορισμένες φορές χρησιμοποιούμε interlocks όπου στην ουσία για να φορτώσουμε μια νέα εντολή περιμένουμε την αποθήκευση των δεδομένων της προηγούμενης. Μία άλλη επίσης λύση, έιναι να κάνουμε έναν έλεγχο όποιας καινούργιας διεύθυνσης φορτώνεται με τις διευθύνσεις που έιναι αποθηκευμένες στο WAQ. Εάν υπάρχει ίδια, τότε η φόρτωση νέας εντολής πρέπει να περιμένει μέχρι να σταματήσει να ισχύει ο παραπάνω έλεγχος. Ωστόσο, έιναι μια πιο ακριβής λύση συγκριτικά με την προηγούμενη.

4.2 Αρχιτεκτονική DAER

Λαμβάνοντας υπόψιν την αρχιτεκτονική decouple access/execute γεννήθηκε η ιδέα της αποτύπωσης μιας αρχιτεκτονικής με υψηλές αποδόσεις που εκμεταλλεύεται το software και το hardware. Η προτεινόμενη αρχιτεκτονική απεικονίζεται στην εικόνα 13 και μπορεί να εφαρμοστεί σε ένα ευρύ φάσμα εφαρμογών. Η χαρτογράφηση του αλγορίθμου με βάση αυτή τη δομή απαιτεί κάποια συγκεκριμένα βήματα. Το framework της παραπάνω αρχιτεκτονικής ακολουθεί τρία βασικά βήματα μετασχηματισμού. Αρχικά ο κώδικας χωρίζεται σε δύο λειτουργικές μονάδες. Η πρώτη μονάδα είναι αρμόδια για τις προσβάσεις στη μνήμη και η δεύτερη για την εκτέλεση του κύριου αλγορίθμου. Έπειτα, επιλύει όλες τις εξαρτήσεις μνήμης που μπορούν να προκύψουν και τέλος μετατρέπει τις δύο αυτές λειτουργικές μονάδες σε μηχανές ροής δεδομένων.

Η ερώτηση λοιπόν που προκύπτει είναι γιατι να προτιμήσουμε την χαρτογράφηση εφαρμογών με την αρχιτεκτονική DAER?

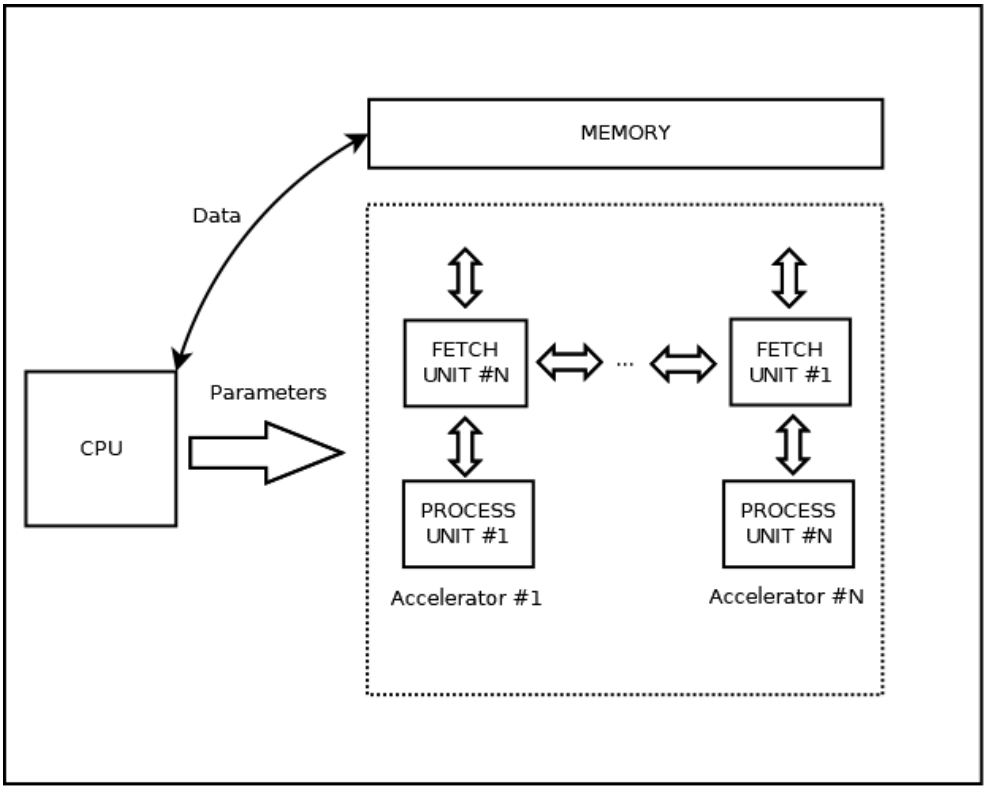
Αρχικά, όπως και η αρχιτεκτονική DAE μας παρέχει μεγάλη ευελιξία στη χαρτογράφηση ποικίλων εφαρμογών. Εάν αναλογιστούμε δε και ότι οι νάνοι που επιθυμούμε να χαρτογραφήσουμε διακρίνονται για τα υψηλά επίπεδα αφαίρεσης τότε ο συνδυασμός αυτών των δύο παρέχουν ακόμα μεγαλύτερη ευελιξία στη χαρτογράφηση ποικίλων εφαρμογών. Έπειτα, η αρχιτεκτονική αυτή έιναι αρκετά αποδοτική σε εφαρμογές που απαιτούν streams για την μεταφορά δεδομένων και είναι ικανή να επιλύσει τυχών αλληλεξαρτήσεις που μπορούν να προκύψουν κατά την διάρκεια της προσπέλασης στην μνήμη.

Πρώτα, χωρίζουμε το κώδικα μας σε δύο νέες λειτουργικές μονάδες όπου τις ονομάζουμε fetch και process αντίστοιχα. Η πρώτη μονάδα είναι αρμόδια για την προσπέλαση της κύριας μνήμης. Επίσης, μπορεί να λαμβάνει δεδομένα από την κύρια μνήμη και τέλος είναι αρμόδια για την αποθήκευση των αποτελεσμάτων σε αυτή. Η δεύτερη είναι αρμόδια για την επίλυση όλων των αριθμητικών πράξεων και για την κύρια υλοποίηση του αλγορίθμου. Χρησιμοποιούνται FIFO interfaces για την μεταφορά των δεδομένων από την μονάδα fetch με την κύρια μνήμη και από την κύρια μνήμη στην process μονάδα. Επομένως ,το πρόγραμμα γίνεται αρκετά ευέλικτο και καθιστά εύκολο των χειρισμό όλων των εφαρμογών που απαιτούν μεταφορές δεδομένων σε μορφή streaming.

Ακόμα η αρχιτεκτονική αυτή παρέχει την δυνατότητα χαρτογράφησης εφαρμογών που έιναι βασισμένες σε διεργασίες. Κάθε διεργασία μπορεί να χαρτογραφηθεί σε ένα επιταγχυντή όπου ο κάθε επιταγχυντής μπορεί να επικοινωνεί με τους υπόλοιπους μέσω της fetch unit. Επομένως, ορισμένες φορές η μονάδα fetch μπορεί να παρακάμψει να λάβει δεδομένα απο την κύρια μνήμη και να λάβει άμεσα από τις υπόλοιπες μονάδες fetch.

Παρουσιάζοντας την παραπάνω αρχιτεκτονική συμπαιρένουμε ότι έχει πολλά να προσφέρει στην απόδοση μιας εφαρμογής. Ωστόσο, σε περιπτώσεις όπου ολόκληρη η εφαρμογή μπορεί να χαρτογραφηθεί σε ένα απλό επιταγχυντή πρέπει να είμαστε προσεχτικοί και να επιλύσουμε τυχών προβλήματα που μπορούν να προκέιψουν με τις εξαρτήσεις μνήμης κατά τηνς διάρκεις της προσπέλασης της. Ένας τρόπος επίλυσης είναι η διαίρεση της μονάδας fetch σε υπομονάδες τα οποία εκτελόυνται παράλληλα σε μορφή pipiline. Επομένων, είναι αρκετά

κατανοητό χωρίς να έχουμε παρουσιάσει ακόμα τα αποτελέσματα των χρόνων και της βελτιώσης που επέφερε αυτή η αρχιτεκτονική ότι μπορεί να παρέχει αρκετά υψηλά επίπεδα αποδοτικότητας.



Εικόνα 13: Αρχιτεκτονική DAER

Όπως γνωρίζουμε οι επιταγχυντές στο υλικό γίνονται ολοένα και πιο δημοφιλείς διότι παρέχουν την δυνατότητα της αύξησης απόδοσης του προγράμματος και συγχρώνως την μείωση της κατανάλωσης της ενέργειας. Ωστόσο, ο σχεδιασμός είναι αρκετά απαιτητικός, διότι η μεταφορά δεδομένων από την εξωτερική μνήμη στον επιταγχυντή πρέπει να γίνει αρκετά προσεχτικά. Όμως με την δημιουργία αυτοματοποιημένων εργαλίων όπως το Vivado HLS μπορούμε να υλοποιήσουμε τέτοιες εφαρμογές ευκολότερα.

Προβλέπεται ότι τα μελλοντικά υπολογιστικά συστήματα θα πρέπει να βασίζονται σε επιταγχυντές του υλικού για την γενική βελτίωση των εφαρμογών. Ήδη, πολλα κινητά χρησιμοποιούν σε μεγάλο βαθμό επιταγχυντές για την εκτέλεση εργασιών όπως επεξεργασία μουσικής και βίντεο.

Το HLS ειναι μια αρκετα ελπιδοφορα εφαρμογή για να δρομολογησει τον σχεδιασμο ενός πολύπλοκου προβλήματος,καθως επιτρέπει την αυτόματη δημιουργία επιταγχυντών υλικού. Δυστυχώς, ακόμα και στο HLS η παροχή των δεδομένων από την μνήμη πρέπει να γίνεται με προσεχτικό τρόπο και να συντονιζονται σωστα με χειροκινητες βελτιστοποιησεις προκειμενου να επιτευχθει σωστη αποδοση στους επιταγχυντες υλικου. Ακόμα παρέχει την δυνατότητα χρησιμοποιήσης dataflow directives για την βελτιστοποιήση της μονάδας επεξεργασίας οι οποίες θα αναλυθουν αργότερα.

Στην συγκεκριμένη διπλωματική θα χαρτογραφήσουμε τις αλγοριθμικές μεθόδους για τα δομημένα και μη πλεγματα. Η χαρτογράφηση θα γίνει με την βοήθεια του εργαλείου vivadohls. Ο αλγόριθμος που χρησιμοποιείται για την υλοποιήση των δομημένων πλεγμάτων είναι η μέθοδος Jacobi.

4.3 Ενεργειακή Αποδοτικότητα

Η ενεργειακή αποδοτικότητα έχει καταστεί ένας από τα πιο σημαντικούς παράγοντες όσο αναφορά τη σχεδίαση στο hardware λόγω του περιορισμένου χρόνου ζωής της μπαταρίας και του ενεργειακού κόστους. Το γεγονός ότι δεν μπορεί να παρέχεται σταθερή πυκνότητα ισχύος έχει ώς αποτέλεμα την αδυναμία εξοικονόμηση τάσης και συχνότητα. Γι αυτό τον τρόπο πρέπει να βρούμε προγράμματα τα οποία ξοδεύουν το χρόνο τους περιμένοντας δεδομένα από την μνήμη και συνεπώς δεν έιναι εθάισθητα στις αλλαγές.

Όπως αναφέρεμε παραπάνω η αρχιτεκτονική DAE χωρίζεται σε δύο φάσεις την access και την execute. Και όπως θα δούμε είναι μια αρχιτεκτονική που βοηθάει αρκετά την ενεργειακή απόδοση του προγράμματος και μειώνει την κατανάλωση ισχύος. Πρώτα απ'όλα χωρίζεται σε δύο τμήματα όπου χρησιμοποιούν την ίδια μνήμη και μας παρέχει την δυνατότητα να χρησιμοποιήσουμε την βέλτιστη συχνότητα ξεχωριστά. Η λειτουργική μονάδα access μεταβάλει τις περισσότερες αποτυχίες της cache σε ευστοχίες πριν εισέλθει στην λειτουργική μονάδα execute. Επίσης, δαπανά ένα μικρό μέρος του χρόνου στον υπολογισμό των διευθύνσεων και τον περισσότερο μέρος περιμένει να έρθουν τα δεδομένα από την μνήμη. Έτσι,έχει ως αποτέλεσμα να μην επηρεάζεται από την συχνότητα του συστήματος. Άρα,στη συγκεκριμένη μονάδα δεν

απαιτείται και ούτε χρειάζεται μεγάλη τιμή η συχνότητα παρά μόνο η ελάχιστη δυνατή τιμή για την εκτέλεση αυτής της φάσης. Όλο αυτό συνδράμει στην εξοικονόμηση ενέργειας και στην αποφυγή της επιβάρυνσης της απόδοσης του προγράμματος.

Όσο αναφορά την execute μονάδα γνωρίζουμε ότι οι περισσότερες αστοχίες της cashe έχουν εξαλειφθεί από την προεπεξεργασία των δεδομένων στην access λειτουργική μονάδα. Επομένως μειώνοντας τις αστοχίες της μνήμης μειώνονται και οι καθυστερήσεις με αποτέλεσμα στη συγκεκριμένη μονάδα από άποψη μεταφορά δεδομένων η υψηλότερη συχνότητα συνίσταται ως καταλλήλότερη.

Κεφάλαιο 5

Στην ενότητα αυτή θα παρουσιάσουμε τις τρεις διαφορετικές υλοποιήσεις που έγιναν στο Vivado HLS για την βελτίωση της απόδοσης και της κατανάλωσης ενέργειας ακολουθώντας πιστά το framework που παρουσιάσαμε στην προηγούμενη ενότητα.

5.1 Υλοποιήσεις δομημένων πλεγμάτων

5.1.1 1η Υλοποίηση δομημένων πλεγμάτων

Αρχικά, έχουμε τον αλγόριθμο που παρουσιάζεται παρακάτω και είναι η μέθοδος Jaccobi. Ο αλγόριθμος αυτός, υλοποιείται στο test bench το οποίο είναι αρμόδιο για την επαλήθευση και τη σωστή λειτουργία του υλικού.

```
Void Solver_solve(struct Solver* solve)
{
double t[ROWS][COLS]
int i,j;
bool converged = false;
solver->iterations = 0;

        while(!converged)
        {
                converged = true;
                solver->iterations++;

        for(i = 1; i < solver->ymax - 1; i ++)
        {
                for(j = 1; j < solver->xmax - 1; j ++)
                {
                        t[i][j]=0;
                        solver->tempGrid[i][j] = 0.25 * (solver->grid[i-1][j]+ solver->grid[i+1][j]+
solver->grid[i][j-1] + solver->grid[i][j+1]);

                if(converged && abs(solver->tempGrid[i][j] - solver->grid[i][j]) >= solver->epsilon)
converged = false;
    }
}

for(i=0;i<ROWS;i++)
  {
```

```
for(j=0;j<COLS;j++)
     {
t[i][j]=solver->grid[i][j];
solver->grid[i][j]=solver->tempGrid[i][j];
solver->tempGrid[i][j]=t[i][j];
     }
   }

}
for(i=0;i<ROWS;i++)
{
for(j=0;j<COLS;j++)
solver->result[i][j]=solver->grid[i][j];
}
}
```

Εικόνα 14: Υλοποίηση της μεθόδου Jacobi

Ο παραπάνω αλγόριθμος είναι η μέθοδος Jacobi υλοποιημένη σε κώδικα C. Έχουμε δύο πίνακες το gridκαι τον tempGrid. Στον πίνακα grid αρχικά αποθηκεύονται οι τιμές που διαβάζονται από το αρχείο. Ο πίνακας αυτός είναι ένας πίνακας **NxN**. Ομοίως και ο tempGrid έχει το ίδιο μέγεθος με το grid. Ο tempGrid στην ουσία είναι ένας βοηθητικός πίνακας όπου κρατάει τις νέες ενημερωμένες τιμές του πλέγματος. Δεν είναι δυνατόν να αποθηκεύονται οι καινούργιες τιμές που δημιουργούνται στο πίνακα grid δίοτι υπάρχει περίπτωση να επηρεάσει τους επόμενους υπολογισμούς. Όταν τελειώσει ο υπολογισμός του κάθε επιτρεπτού σημειού του πίνακα grid εξετάζεται η διαφορά, της τιμής που προκύπτει με την αρχική τιμή του. Εάν, είναι μεγαλύτερη του σφάλματος που έχει οριστεί εξαρχής, τότε ο παραπάνω υπολογισμός ξαναεκτελείται μέχρι η διαφορα να είναι μικρότερη από το σφάλμα. Σε περίπτωση που η διαφορά είναι μεγαλύτερη από το σφάλμα, γίνεται ανταλλαγή τιμών του πίνακα grid με τις τιμές του tempGrid και εκτελείται εκ νέου ο υπολογισμός των τιμών.

Η χαρτογράφηση του παραπάνου αλγορίθμου στο εργαλείο της vivado με βάση της αρχιτεκτονικής DAE θα γίνει με τη δημιουργία δύο λειτουργικών μονάδων όπως αναφέραμε και στο 4ο κεφάλαιο. Η πρώτη λειτουργική μονάδα θα ονομάζεται fetch και θα είναι αρμόδια για την προσπέλαση στη μνήμη και την ανάκτηση των απαιτούμενων διευθύνσεων. Αξίζει να σημειωθεί ότι θα απαιτηθεί μία λειτουργική μονάδα fetch εφόσον δεν προκύπτει από τον κώδικα κάποιο είδος εξάρτησης στη μνήμη. Η δεύτερη λειτουργική μονάδα ονομάζεται process και είναι αρμόδια για κάθε λογική και αριθμητική πράξη που εμπεριέχει ο αλγορίθμος. Μελετώντας τον αλγόριθμο καταλαβαίνουμε ότι τα δεδομένα τα οποία πρέπει να φορτωθούν και να εισέλθουν

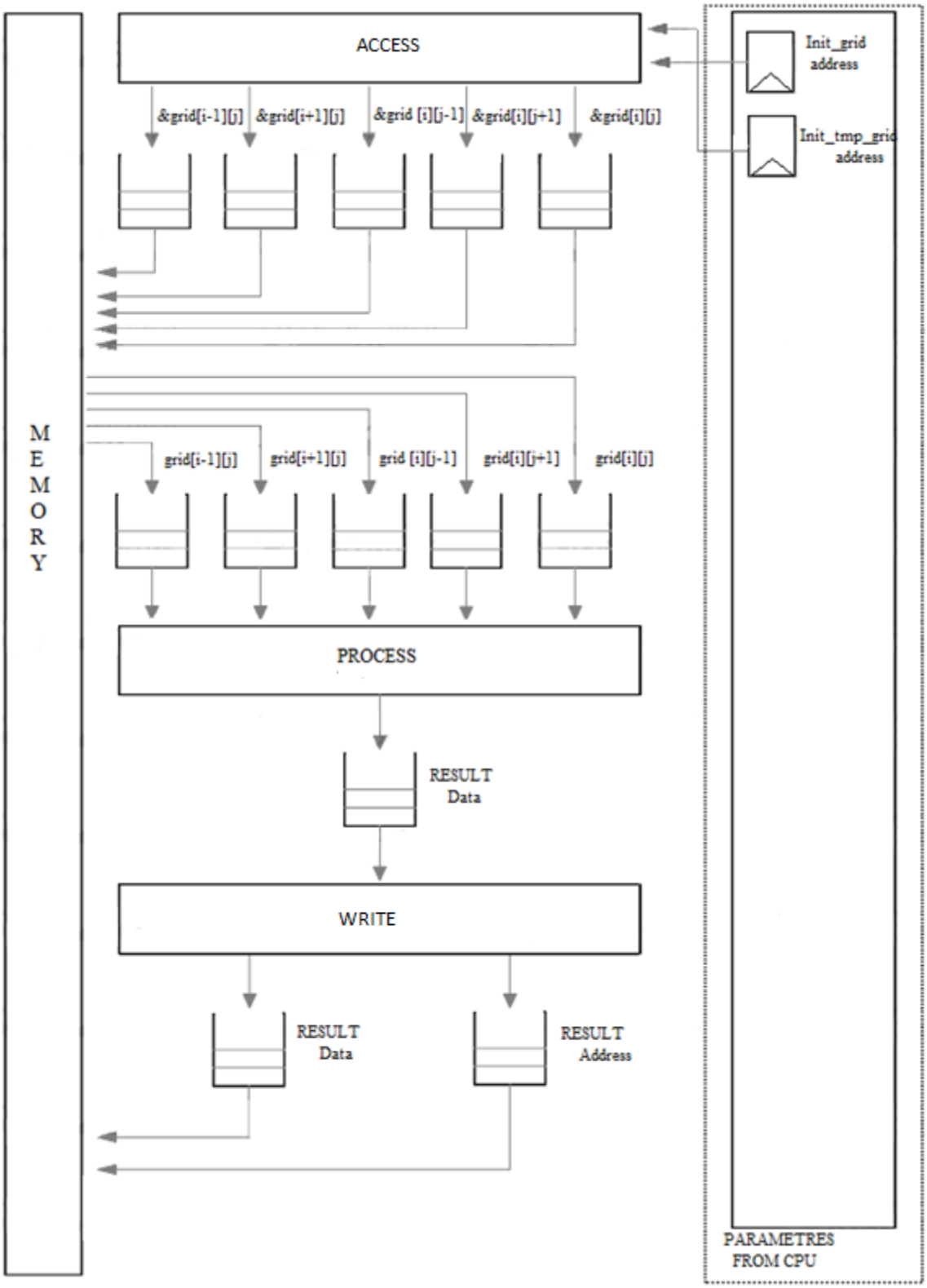
στη μονάδα process είναι οι εξής τιμές: grid[i-1][j], grid[i+1][j], grid[i] [j-1], grid[i] [j+1] για τον απαιτούμενο υπολογισμό και τις τιμές του grid[i] [j] για τη σύγκριση του με το σφάλμα.Όπου i και j ο αριθμός θέσης στον πίνακα grid.

Κατά την απεικόνιση στο harware, θα πρέπει να μετατραπούν αυτοί οι πίνακες σε μονοδιάστατους εφόσον επιθυμούμε την μεταφορά δεδομένων με τη μορφή streams και τη χρήση FIFOs ουρών. Έτσι, θα κάνουμε χρήση του τύπου i*γραμμές+j για την ακριβή μετατροπή ενός δισδιάστατου σε μονοδιάστατο πίνακα.

Η ροή δεδομένων ακολουθεί τη παρακάτω διαδρομή όπου εκτός απο την περιγραφή θα παρατεθεί και το διάγραμμα. Αρχικά, η λειτουργική μονάδα fetch λαμβάνει σαν είσοδο από την CPU την αρχική τιμή της διεύθυνσης του πίνακα grid και tempGrid. Η μονάδα αυτή βγάζει σαν έξοδο 5 δεδομένα που αποθηκεύονται σε αντίστοιχες FIFO ουρές. Αυτές οι ουρές αποθηκεύουν προσωρινά όλες τις διευθύνσεις των τιμών που αναφέρθηκαν παραπάνω για τον υπολογισμό του κύριου φόρτου εργασίας του αλγορίθμου. Αξίζει να σημειωθεί ότι η τοποθέτηση των τιμών στις ουρές γίνεται αντίστοιχα, έτσι ώστε οι τιμές που θα χρειαστούν για τον υπολογισμό μιας θέσης του νέου πίνακα grid να βρίσκονται στις ίδιες θέσεις στις ουρές. Στη συνέχεια ,τα δεδομένα των ουρών αυτών κατευθύνονται στη κύρια μνήμη από όπου θα φορτώθουν τα δεδομένα που βρίσκονται σε αυτές τις διευθύνσεις και θα τοποθετηθούν σε FIFOs ουρές και εισέρχονται στη μονάδα process. Εφόσον, γίνουν οι απαραίτητοι υπολογισμόι και με συνδυασμό τον έλεγχο με την τιμή του σφάλματος η μονάδα process βγάζει σαν έξοδο δύο FIFO ουρές. Η πρώτη εμπεριέχει όλες τις τιμές των αποτελεσμάτων του κύριου αλγορίθμου και η άλλη εμπεριέχει τις διευθύνσεις τους. Τέλος γίνεται το αίτημα εγγραφής αυτών των αποτελεσμάτων στην κύρια μνήμη.

Αξίζει να σημειωθεί ότι στη μονάδα process συναντήσαμε την εξής δυσκολία. Εφόσον, υπολογίσουμε την νέα τιμή το πίνακα grid την τοποθετούμε σε ένα προσωρινό πίνακα tempGrid και συγκρίνουμε τη διαφορά της νέα τιμής και της παλίας με το σφαλμα. Σε περίπτωση που είναι μεγαλύτερη από το σφάλμα η διαφορά πραγματοποιείται εναλλαγή των πινάκων grid και tempGrid και γίνονται εκ νέου οι υπολογισμοί. Αυτη η εναλλαγή δημιουργούσε πρόβλημα διότι στη process φορτώναμε τα δεδομένα της grid. Για την επίλυση αυτού του προβήματος ανάλογα με την επανάληψη στην οποία βρισκόμασταν στη συνάρτηση fetch δινόταν σαν είσοδο είτε η διεύθυνση του grid έιτε του tempGrid και η process έπαιρνε τα ανάλογα δεδομένα.

Τα παραπάνω βήματα ακολουθήθηκαν για τη χαρτογράφηση της μεθόδου Jacobi στο hardware με τη χρήση του εργαλείου της Vivado και με γνώμωνα την αρχιτεκτονική DAE. Παρακάτω παρατίθεται το διάγραμμα όπου φαίνεται αναλυτικά η ροή του προγράμματος.



Εικόνα 15: Εφαρμογή της αρχιτεκτονικής DAER στα δομημένα πλέγματα

5.1.2 2η Υλοποίηση δομημένων πλεγμάτων

Έχοντας σαν γνώμονα την πρώτη υλοποίηση, η οποία είναι και η βασική αρχιτεκτονική, δημιουργήσαμε δύο εκ νέου υλοποιήσεις με την μόνη διαφορά το σημείο αποθήκευσης των δεδομένων. Βασικό χαρακτηριστικό της συγκεκριμένης υλοποίησης είναι η προφόρτωση των δεδομένων στη BRAM. Η BRAM διαθέτει τη δυνατότητα αποθήκευσης μικρού όγκου δεδομένων αλλά προσφέρει την παράλληλη πρόσβαση σε διαφορετικές θέσεις μνήμης και την γρήγορη πρόσβαση στα δεδομένα. Ομοίως με την πρώτη υλοποίηση χωρίσαμε την αρχική συνάρτηση σε δύο λειτουργικές μονάδες την fetch και τη process.

Στη συνέχεια, δημιουργούμε τη συνάρτηση Test Function όπου εκεί μέσα πραγματοποιούνται όλες οι κλήσεις των συναρτήσεων. Εσωτερικά της πραγματοποιούνται στατικές δηλώσεις πινάκων για την αποθήκευση των διεθύνσεων και των δεδομένων. Όπως είπαμε και στην πρώτη υλοποίηση η fetch θα βγάλει σαν εξόδους 5 FIFOs ουρές. Ομοίως,θα γίνει και σε αυτή την υλοποίηση, ωστόσο τα δεδομένα θα φορτωθούν από την BRAM και όχι από την κύρια μνήμη. Οι έξοδοι της συνάρτησης fetch,οι οποίεις περιέχουν τις διευθύνσεις των δεδομένων που είναι απαραίτητα για τον υπολογισμό του αλγορίθμου θα εισέλθουν στη συνάρτηση read memory όπου αυτή με τη σειρά της θα φορτώσει τα δεδομένα από την εσωτερική BRAM της FPGA σε 5 νέες FIFOsουρές. Τέλος, οι ουρές αυτές από έξοδοι της read memory θα γίνουν είσοδοι της process.

Η συνάρτηση testFunction εκτελεί με μορφή pipeline τις συναρτήσεις fetch ,readmemory και process. Πρώτα εκτελείται η συνάρτηση fetchUnit όπου δεν έχει καμμία διαφορά με την πρώτη υλοποίηση. Έπειτα, εκτελείται η συνάρτηση read_memory. Ανάλογαμε τον αριθμό της επανάληψης λαμβάνει και διαφορετική αρχική τιμή διεύθυνσης. Στην πρώτη επανάληψη έχουμε τον αριθμό της δεύθυνσης της grid ενώ στην δεύτερη την διεύθυνση της tempGrid. Και τέλος εκτελείται η process όπου δεν διαφέρει καθόλου με την process της πρώτης υλοποίησης.

5.1.3 3η Υλοποίηση δομημένων πλεγμάτων

Η Τρίτη υλοποίηση διαφέρει ελάχιστα με την δεύτερη υλοποίηση όσο αφορά τον τρόπο αποθήκευσης δεδομένων. Οι αποθηκεύσεις των δεδομένων θα πραγματοποιούνται στην εξωτερική μνήμη. Αρχικά έχει και αυτή τρείς συναρτήσεις υλοποιημένες στο hardware. Συγχρόνως θα πραγματοποιήθει η εξής αλλαγή σε σχέση με τη δεύτερη υλοποίηση, θα εισάγουμε το while loop όπου υπήρχε στο software μέσα στην testFunction. Οι συναρτήσεις fetch, read_memory και process παραμένουν ίδιες. Μπορεί, αρχικά να μην φαίνεται μεγάλη η αλλαγή, αλλα υπάρχει διαφορά στους χρόνους. Το Vivado παρέχει εντολές όπου μπορούμε να βελτιστοποιήσουμε τη ροή των δεδομένων στα loops επομένως η Τρίτη υλοποίηση γίνεται ακόμα πιο γρήγορη. Τα loops έχουν επιρροή στο latency το οποίο καθορίζει το χρόνο εκτέλεσης μιας διεργασίας.

Αυτή είναι και η τελική μας υλοποίηση όπου θα τη συγκρίνουμε άλλωστε με την αρχική μας υλοποίηση όσο αναφορά την απόδοση και την κατανάλωση ενέργειας.

Για να γίνει η εισαγωγή του loop εσωτερικά της testfunction πρέπει να δηλώσουμε ένα πίνακα ακεραίων conVal με μέγεθος ίδιο με το grid. Ο πίνακας αυτός θα δωθεί σαν όρισμα και στη συνάρτηση process όπου μετά τις απαραίτητες πράξεις που θα γίνονται θα βγάζει σαν έξοδο τον πίνακα αυτό. Θέλουμε να δημιουργήσουμε ένα πίνακα όπου για κάθε έναν έλεγχο που θα πραγματοποιείται με τον αριθμό του σφάλματος να εισάγει εάν είναι μικρότερο του σφάλματος την τιμή 0 και εάν είναι μεγαλύτερο την τιμή 1. Αυτό θα γίνει τόσες φορές όσες και το μέγεθος του πίνακα. Ο επαναληπτικός βρόγχος εκτελείται συνεχόμενα εκτός και αν του δωθεί εντολή να σταματήσει. Μετά και το πέρας της συνάρτησης process θα γίνει ο έλεγχος έτσι ώστε να παρθεί η απόφαση εάν θα σταματήσει η όχι η εκτέλεση του βρόγχου. Εάν λοιπον βρεθεί στο πίνακα έστω και ένα στοιχείο το οποίο να περιέχει την τιμή 0 δηλαδή ο έλεγχος με τον αριθμό του σφάλματος να ήταν μεγαλύτερος τότε εκτελείται εκ νέου ο επαναληπτικός βρόγχος αλλιως έχουμε το τέλος του προγράμματος.

5.2 Υλοποιήσεις μη δομημένων πλεγμάτων

Στο συγκεκριμένο κεφάλαιο θα παρουσιαστούν οι τρεις διαφορετικές υλοποιήσεις που πραγματοποιήθηκαν για την αλγοριθμική μέθοδο των μη δομημένων πλεγμάτων για την βελτιστοποίηση της απόδοσης. Ο αλγόριθμος που υπάγεται στη συγκεκριμένη μέθοδο είναι η Serial Based μέθοδος. Οι υλοποιήσεις και ο αλγόριθμος θα αναλυθούν εκτενώς παρακάτω. Η πρώτη και κύρια υλοποίηση η οποία βασίζεται στην αρχιτεκτονική DAE και οι άλλες δύο οι οποίες διαφέρουν στο τρόπο αποθήκευσης των δεδομένων. Στη δεύτερη γίνεται αποθήκευση των δεδομένων στην εσωτερική μνήμη της FPGA και η τρίτη στην εξωτερική.

Ο αλγόριθμος Serial Based, ο οποίος παρουσιάζεται παρακάτω, αρχικά υλοποιείται στο test bench και ελέγχεται η σωστή λειτουργία του.

5.2.1 1η Υλοποίηση μη δομημένων πλεγμάτων

Η αρχιτεκτονική της πρώτης και κύριας υλοποίησης γίνεται με βάση της αρχιτεκτονικής DAE. Αρχικά, εκτελούμε τον αλγόριθμο Serial Based στο test bench και ελέγχουμε την ορθότητα του. Ο αλγόριθμος αυτός παρουσιάζεται παρακάτω και είναι υλοποιημένος στη γλώσσα C++.

```
void Solver::solve()
{
bool converged = false;
int counter = 0;

while(!converged)
   {
converged = true;

for (unsigned int i = 0; i < numVertex; i ++)
      {
compute(neighborhood[i], numNeighbors[i], value ,  tempValue + i);

if (converged && (abs(tempValue[i] - value[i]) >= epsilon)) converged = false;
      }

double * t = value;
value = tempValue;
tempValue = t;
```

```
counter++;
   }
resultValue = value;

printf("SW Iterations: %d\n", counter);
}
```

```
static inline void compute(unsigned int *vertices, unsigned int num, double *input, double *output)
{
      *output = 0;
for(unsigned int i = 0; i < num; i ++)
      {
         *output += input[vertices[i]];
               }

      *output /= num;

}
```

Εικόνα 16: Υλοποίηση της μεθόδου Serial based

Αρχικά έχουμε τον πίνακα Value όπου περιέχει τις τιμές του κάθε κελιού του μη δομημένου πλέγματος. Στη συνέχεια δηλώνονται οι δυο πίνακες neighborhood και numNeighbors. Το πρώτο υποδηλώνει τα γειτονικά σημεία ενός κελιού του μη δομημένου πλέγματος και το δεύτερο τον αριθμό των γειτονικών κελιών. Τέλος, χρησιμοποιείται ένας βοηθητικός πίνακας tempValue για την αποθήκευση των ενημερωμένων τιμών του πλέγματος. Εσωτερικά της solve καλείται η compute με ορίσματα τους πίνακες :neighborhood, numNeighbors, value και tempValue. Εσωτερικά της compute υπολογίζεται ο μέσος όρος ενός κελιού υπολογίζοντας το άθροισμα των κελιών που υπάγονται στην ίδια γειτονιά και διαιρώντας το με τον αριθμό των γειτόνων. Το αποτέλεσμα αυτό αποθηκεύεται στο πίνακα tempValue όπου είναι η έξοδος της συνάρτησης compute. Στη συνέχεια η διαφορά της εξόδου που παράχθηκε συγκρίθηκε με την προηγούμενη τιμή και σε περίπτωση που το σφάλμα είναι μεγαλύτερο από ένα όριο η διαδικασία επαναλαμβάνεται. Ο κώδικας αυτός θα προστεθεί στο testbench όπου θα γίνεται και η επαλήθευση με το κώδικα που θα γράψουμε στο hardware.

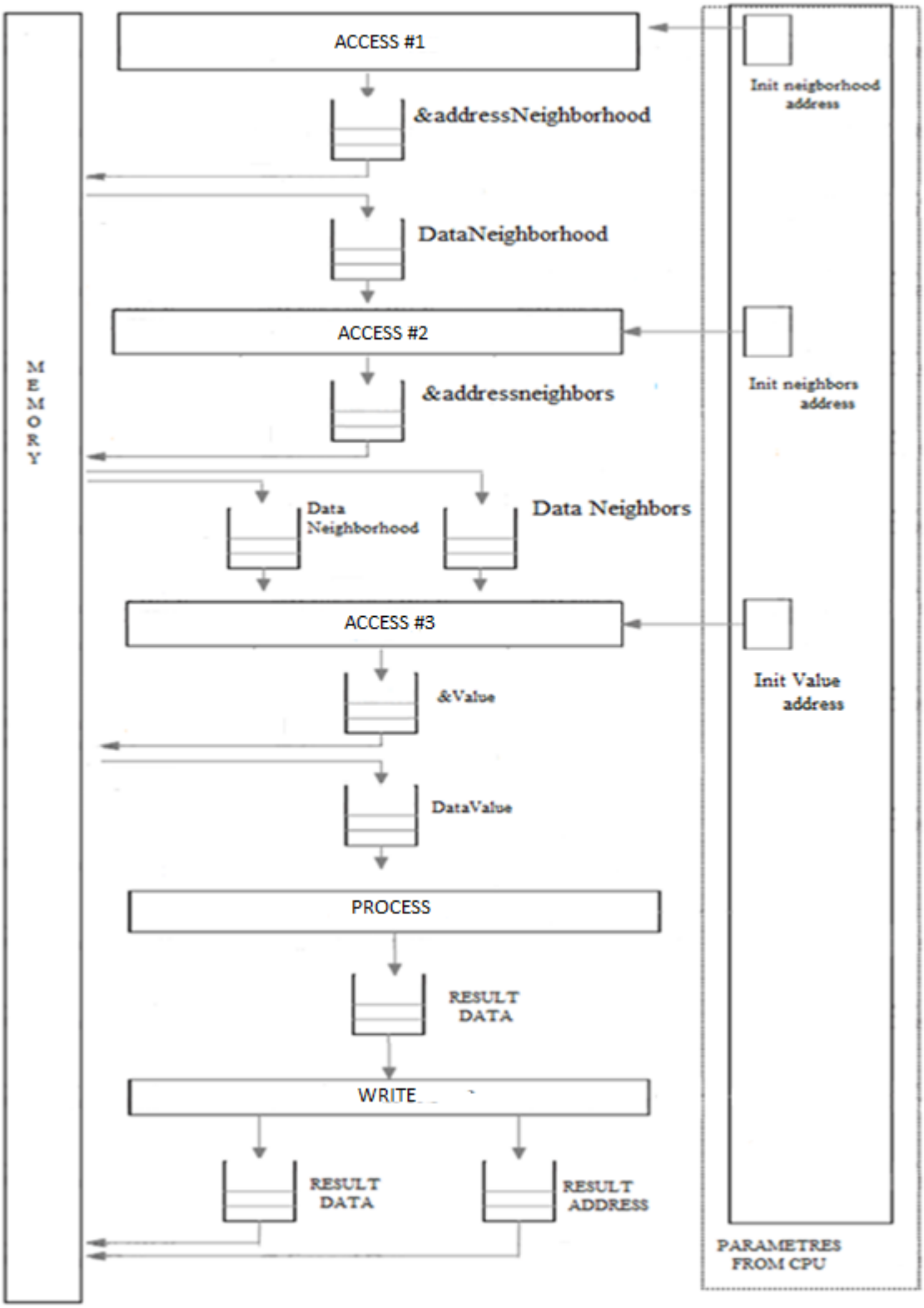
Λαμβάνοντας υπόψιν την αρχιτεκτονική DAE θα πρέπει ο κώδικας ομόιως όπως και των δομημένων πλεγμάτων να διασπαστεί σε δύο λειτουργικές μονάδες, τη fetch και την process. Με

την πρώτη λειτουργική μονάδα θα κάνουμε τις προσβάσεις στη μνήμη. Στη συγκεκριμένη αλγοριθμική μέθοδο συναντήσαμε εξαρτήσεις μνήμης οπότε χρειάστηκε να την χωρίσουμε σε τρεις υπομονάδες. Θα γίνει προσπέλαση σε τρεις διαφορετικού πίνακες για την εκτύπωση των απαραίτητων διευθύνσεων. Με την fetch0 θα λαμβάνονται οι διευθύνσεις για το πίνακα neighborhood. Έπειτα,οι διευθύνσεις αυτές θα εισέρχονται στη κύρια μνήμη όπου θα λάβουμε τα δεδομένα που θα είναι οι είσοδοι για τη δεύτερη υπομονάδα fetch1. Αυτή υπομονάδα θα βγάζει σαν εξόδους τις διευθύνσεις του neighbors πίνακα όπου και αυτές με τη σειρά ακολουθούν την ίδια σειρά με τις διευθύνσεις του neighborhood. Έτσι, η μνήμη θα μας δώσει τα νέα δεδομένα που στην ουσία είναι οι τιμές του πίνακα neighbors οι οποίες βρίσκονται στις θέσεις neighborhood[i]. Τέλος,τα δεδομένα αυτά εισέρχονται στη τρίτη και τελευταία υπομονάδα fetch3. Η υπομονάδα αυτή θα υπολογίσει τις διευθύνσεις του value πίνακα που βρίσκονται στις θέσεις neighbors[neighborhood[i]]. Οι διευθύνσεις αυτές κατευθύνονται στη μνήμη από όπου θα διαβαστούν τα δεδομένα αυτών των διευθύνσεων.

Τα δεδομένα αυτά τέλος εισέρχονται στη δεύτερη μονάδα, process όπου γίνεται ο φόρτος του κύριου αλγορίθμου. Δηλαδή, γίνονται όλες οι αριθμητικές και λογικές πράξεις και εξέρχονται τα τελικά αποτελέσματα. Τα αποτελέσματα αυτά γίνονται είσοδοι στην τελική μονάδα fetch. Η μονάδα αυτή έχει δύο FIFOs ουρές. Η πρώτη εμπεριέχει όλα τα δεδομένα των αποτελεσμάτων και η δεύτερη όλες τις διευθύνσεις των αποτελεσμάτων. Το τελικό στάδιο είναι το αίτημα εγγραφής αυτών των αποτελεσμάτων στην κύρια μνήμη.

Αξίζει να σημειωθεί ότι τα αποτελέσματα των διευθύνσεων και των δεδομένων που εισέρχονται και εξέρχονται απο τις λειτουργικές μονάδες και την κύρια μνήμη γίνεται με μορφή streams και υλοποιούνται με FIFOs ουρές.

Στη συνέχεια όπως είπαμε παραπάνω η κάθε μία υπομονάδα βγάζει σε μορφή streams τις απαραίτητες διευθύνσεις σε FIFOs ουρών, που απαιτούνται για να γίνει εφικτή η πρόσβαση στη κύρια μνήμη. Έπειτα, διαβάζονται τα δεδομένα απο την κύρια μνήμη όπου και αυτά τοποθετούνται σε πίνακες όπου εν συνεχεία τοποθετούνται σε FIFOs ουρές και εισέρχονται στην επόμενη λειτουργική υπομονάδα. Παρακάτω παρατίθεται το διάγραμμα ροής όπου φαίνονται ξεκάθαρα τα βήματα που αναλύσαμε.



Εικόνα 17 :Εφαρμογή της αρχιτεκτονικής DAER στα μη δομημένα πλέγματα

5.2.2 2η Υλοποίηση Μη Δομημένων Πλεγμάτων

Βασισμένοι στην πρώτη υλοποίηση δημιουργήσαμε τη δεύτερη υλοποίηση όπου γίνεται προφόρτωση των δεδομένων στη BRAM. Για το λόγο αυτό δημιουργούμε μία νέα συνάρτηση TestFunction όπου εκεί πέρα πραγματοποιούνται όλες οι κλήσεις των συναρτήσεων fetch και process. Ακόμα, δηλώνονται οι πίνακες που θα αποθηκευσούν τις διευθύνσεις και τα δεδομένων αυτών όπου εισέρχονται και εξέρχονται απο τίς λειτουργικές μονάδες με την μορφή FIFO ουρών. Οι έξοδοι των λειτουργικών μονάδων fetch γίνονται είσοδοι της συνάρτησης readmemory όπου είναι αρμόδια για τη φόρτωση των δεδομένων από την εσωτερική μνήμη της FPGA. Εξαιτίας του διαχωρισμού της λειτουργικής μονάδας fetch σε τρεις υπομονάδες, θα διαχωρίσουμε και την readmemory σε τρεις διαφορετικές όπου η κάθε μία θα είναι αρμόδια για τη φόρτωση των δεδομένων που βρίσκονται στις αντίστοιχες διευθύνσεις.

.Η readmemory0 θα δεχθεί σαν είσοδο την ουρά που θα περιέχει τις διευθύνσεις numNeighbors και θα βγάλει σαν έξοδο τα δεδομένα που εμπεριέχονται σε αυτές τις διευθύσεις. Η read_memory1 θα πάρει σαν είσοδο τα δεδομένα του numneighbors και τις διευθύνσεις του neighborhood και θα βγάλει σαν έξοδο τα δεδομένα του neighborhood[numneighbors] και η read_memory2 θα πάρει σαν είσοδο τα δεδομένα του neighborhood[numNeighbors] και τις διευθύνσεις του value και θα βγάλει σαν τελική έξοδο τα δεδομένα του values[neighborhood[numNeighbors]]. Τα δεδομένα αυτά θα εισαχθούν σαν είσοδο στη λειτουργική μονάδα process η οποία θα βγάλει το τελικό αποτέλεσμα. Έπειτα θα γίνει ο απαραίτητος έλεγχος με το σφάλμα και θα συνεχιστεί εκ νέου ο υπολογισμός ή θα τερματίσει το πρόγραμμα.

5.2.3 3η Υλοποίηση Μη Δομημένων Πλεγμάτων

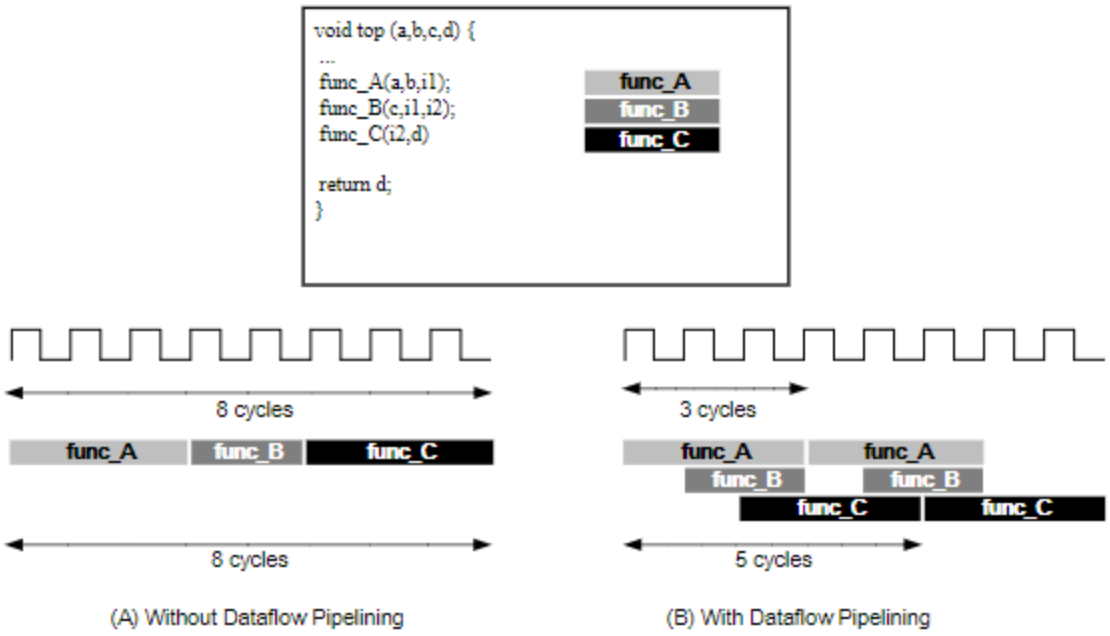
Η συγκεκριμένη υλοποίηση διαφέρει από τη δεύτερη μόνο στο τρόπο αποθήκευσης των δεδομένων. Οι αποθηκεύσεις των δεδομένων γίνονται στην εξωτερική μνήμη. Επίσης, πραγματοποιείται και η εξής αλλαγή, γίνεται η εισαγωγή του while loop από το software στο hardware. Το εργαλείο της Vivado παρέχει τη δυνατότητα εισαγωγής directives για τους βρόγχους επανάληψης όπου βελτιώνει ακόμα πιο πολύ την απόδοση του συστήματος. Όλες οι

συναρτήσεις fetch, read_memory και process παραμένουν ίδιες. Για να εισάγουμε το whileloop μέσα στη testFunction πρέπει να εισάγουμε και ένα νέο πίνακα, τον conVal, με μέγεθος ίσο με το value όπου θα αποθηκεύει 0 ή 1 ανάλογα με το αποτέλεσμα του ελέγχου που πραγματοποίησε. Θα αποθηκεύει τη τιμή 0 όταν απαιτείται να ξαναγίνει νεός υπολογισμός τιμών και νέος έλεγχος με το σφάλμα και την τιμή 1 όταν ο έλεγχος είναι σωστός. Στο τέλος της κλήσης της συνάρτησης process θα προστεθεί μια forloop όπου θα γίνεται πρόσβαση και έλεγχος του πίνακα conVal. Εάν δεν υπάρχει αποθηκευμένη τιμή 0 τότε θα γίνεται break και τα δεδομένα που έβγαλε η συνάρτηση processθα οδηγούνται στη κύρια μνήμη και θα γίνονται αιτήματα εγγραφής.

5.3 Directives

Στην ενότητα αυτή θα αναλυθούν οι directives που χρησιμοποιήθηκαν για την βελτιώση της απόδοσης του αλγορίθμου. Αρχικά, όλοι οι πίνακες στη vivado μετατρέπονται σε FIFOs ουρές. Έτσι, χρησιμοποιήσαμε την εντολή, **#pragmaHLSinterface =apfifo_port.** Με την συγκεκριμένη εντολή μπορούμε και να ελέγξουμε το μέγεθος της ουράς. Ακόμα χρησιμοποιήσουμε dataflow directives για την βελτιώση της απόδοσης όπως για παράδειγμα τη **#pragmaHLSdataflow.** Η ακόλουθη εντολή επιτρέπει την εφαρμογή διεργασίων να τρέχουν με τη μορφή pipeline. Παρατηρούμε από την εικόνα, ότι στην πρώτη υλοποίηση όπου δεν χρησιμοποιούμε τη συγκεκριμένη directive, οι τρεις συναρτήσεις υλοποιούνται σε 8 κύκλους. Ενώ, στη δεύτερη υλοποιήση έπειτα από την χρησιμοποίηση της dataflow directive υλοποιείται σε 5 κύκλους. Αυτό γίνεται διότι η δεύτερη συνάρτηση δεν περιμένει να τελείωσει η πρώτη και η τρίτη ομοίως για η δεύτερη. Άρα, μειώνοντας το latency της συνάρτησης αυξάνεται η απόδοση του αλγόριθμου και η μειώνεται η κατανάλωση ενέργειας.

Αξίζει να σημειωθεί ότι για τη συγκεκριμένη υλοποίηση η συγκεκριμένη dataflow directive είναι πολύ σημαντική διότι όπως είχαμε εξηγήσει και στο τέταρτο κεφάλαιο ο υπολογισμός των διευθύνσεων από τη fetch με τα αποτελέσματα που παράγει η process γίνονται παράλληλα.



Εικόνα 18: Εφαρμογή dataflow directive

Τέλος, χρησιμοποιήθηκε και η συνάρτηση **#pragmaHLSLOOP_MERGE**. Αρκετά συχνά γίνεται χρήση πολλών διαδοχικών βρόχων κάτι το οποίο μπορεί να προκαλέσει αύξηση του latency δημιουργώντας περιττούς κύκλους ρολογιού. Το γεγονός αυτό αποτρέπει την προσπάθεια για βελτιστοποίηση. Ένας τρόπος επίλυσης είναι η συγκεκριμένη συνάρτηση όπου επιτυγχάνεται η συγχώνευση διαδοχικών βρόχων μειώνοντας την καθυστέρηση όπου μπορεί να προκύψει επιτυγχάνοντας τη βελτίωση της απόδοσης του αλγορίθμου.

Πραγματοποιήθηκαν λοιπόν, τρεις διαφορετικές υλοποιήσεις για κάθε αλγόριθμο στο εργαλείο Vivado HLS. Η πρώτη και κύρια υλοποίηση με γνώμωνα την αρχιτεκτονική DAE και οι άλλες δύο υλοποιήσεις όπου διέφεραν από τη πρώτη στο τρόπο αποθήκευσης των δεδομένων. Στη δεύτερη η αποθήκευση πραγματοποιήθηκε στην εσωτερική μνήμη BRAM και η τριτη στην εξωτερική μνήμη. Δεν συναντήσαμε ιδιαίτερες δυσκολίες στη σχεδίαση του RTL εξαιτίας των αυτοματοποιήμενων διαδικασιών που παρέχει το εργαλείο Vivado HLS. Επίσης, το εργαλείο αυτό παρέχει τη δυνατότητα στη χρήση directives για την βελτίωση της απόδοσης του συστήματος. Τέλος, η αρχιτεκτονική DAE αποδείχθηκε κατάλληλη για υλοποιήσεις όπου

συναντάμε εξαρτήσεις στη μνήμη εκμεταλλευόμενοι όσο μπορούμε τα πλεονεκτήματα του παράλληλου προγραμματισμού.

Κεφάλαιο 6

Στο κεφάλαιο αυτό περιγράφονται οι πλατφόρμες και τα εργαλεία που χρησιμοποιήσαμε για την απεικόνιση των αγορίθμων. Στη συνέχεια, παρουσιάζονται οι χρόνοι εκτέλεσης τόσο στο hardware όσο και στο software. Πραγματοποιείται σύγκριση των χρόνων, και παρουσιάζεται η μεταβολή της απόδοσης. Τέλος, παρατίθονται οι πόροι του συστήματος που χρησιμοποιήθηκαν για κάθε αλγόριθμο.

6.1 Περιγραφή πλατφόρμων και εργαλειών

Το εργαλείο που χρησιμοποιήσαμε για την απεικόνιση των αλγορίθμων που υπάγονται στα δομημένα και μη πλέγματα είναι το Vivado HLS. Το εργαλείο αυτό αποτελεί αυτοποιημένη μετατροπή μιας σχεδίασης από C, C++, SystemC σε RTL υλοποίηση η οποία με τη σειρά της γίνεται σύνθεση σε μια fpga. Αυτό, καθιστά τη ζωή του προγραμματιστή ευκολότερη, εφόσον η σχεδίαση του RTL αρχείου δεν γίνεται χειροκίνητα, κάτι το οποίο μάλιστα, ελαχιστοποιεί και την πιθανότητα των λαθών. Η σύνθεση υψηλού επιπέδου δέχεται σαν είσοδο μια συνάρτηση υλοποιήμενη σε C/C++ καθώς και ένα αρχείο test bench το οποίο εχει δημιουργηθεί για την επαλήθευση της ορθής λειτουργίας του συστήματος. Το αρχείο αυτό αποθηκεύει όλα τα αποτελέσματα τα οποία συγκρίνονται με τα αποτελέσματα εξόδου της συνάρτησης. Μετά την επαλήθευση της σωστής λειτουργίας του προγράμματος ακολουθεί η διαδικασία της σύνθεσης σε μία FPGA. Στο στάδιο αυτό παρέχεται η δυνατότητα στο χρήστη να πειραματιστεί ανάλογα με τις προδιαγραφές που επιθυμεί να έχει το πρόγραμμα και να χρησιμοποιήσει και τις κατάλληλες directives. Τέλος, ελέγχεται η ορθότητα του αρχείου RTL.

Το εργαλείο της Vivado εμπεριέχει δίαφορες FPGAs με διαφορετικά χαρακτηριστικά. Το rtl αρχείο σχετίζεται με την επιλογή της πλατφόρμας. Στη συγκεκριμένη διπλωματική, η FPGA που επιλέξαμε είναι η ZYNQ ZC706. Η πλατφόρμα εμπεριέχει 350 Logic Cells, 900 DSP Slices, 218600 LUTs, 437200 Flip Flops, 1090 BRAMs. Η επιλογή αυτή έγινε με γνώμονα την εσωτερική μνήμη της συγκεκριμενης πλατφόρμας διότι, επιθυμούμε όσο το δυνατόν

μεγαλύτερη. Όσο το δυνατόν μεγαλύτερη εσωτερική μνήμη τόσο μεγαλύτερο και το dataset το οποίο θα χρησιμοποιηθεί.

Τα χαρακτηριστικά της παραπάνω πλατφόρμας είναι τα παρακάτω:

Zynq – 7000 XC7Z045-2FFG900C AP SoC
Dual ARM-Cortex-A9 core processors
1 GB DDR3 memory SODIMM on PL side
1GB DDR3 component memory on PS side

Εικόνα 19: Χαρακτηριστικά της πλατφόρμας Zynq zc706

Στη συνέχεια, η δεύτερη πλατφόρμα που χρησιμοποιούμε για να τρέξουμε την τρίτη υλοποίηση όπου γίνεται αποθήκευση των δεδομένων στην εξωτερική μνήμη είναι η Convey.

Τα χαρακτηριστικά της Convey είναι τα εξής:

Standard Intel ® x86-64 Server
Convey Coprocessor FPGA-based
Hybrid-Core Shared Memory(HCGSM)

Εικόνα 20: Χαρακτηριστικά της πλατφόρμας Convey

Έχει παρατηρηθεί, ότι η απόδοση ενός προγράμματος βελτιώνεται από 5x μέχρι και 25x. Συγχρόνως, παρατηρείται μείωση της κατανάλωσης ενέργειας εώς και 90%. Ο προγραμματισμός γίνεται σε C, C++, Fortan. Η συγκεκριμένη πλατφόρμα, περιέχει τον κύριο επεξεργαστή,ο οποίος είναι Intel x86. Ο επεξεργαστής αυτός συνδέεται με τον συνεπεξεργαστή της Convey όπου έχει σχεδιαστεί για να συμπληρώνει τον κύριο επεξεργαστή. Ο συνεπεξεργαστής περιέχει 14 FPGAs οι οποίες εκτελούν συγκεκριμένες λειτουργίες, οι οποίες ονομάζονται personalities και αντιπροσωπεύουν το χρόνο εκτέλεσης μιας εφαρμογής. Οι τέσσερις FPGAs χρησιμεύουν για την εκτέλεση των εντολών. Οι δύο χειρίζονται την επικοινωνία του συνεπεξεργαστή της Convey

με τον επεξεργαστή Intel x86. Τέλος, οι υπόλοιπες 8 FPGAs παρέχουν την δυνατότητα της επιτάχυνσης της μνήμης. Με τη συγκεκριμένη πλατφόρμα δεν αντιμετωπίζουμε προβλήματα με το μέγεθος των dataset όπου δημιουργείται με την εσωτερική μνήμη της FPGA.

Αφού τρέξαμε και στις δύο πλατφόρμες τις αντίστοιχες υλοποιήσεις συγκρίναμε τους χρόνους με την αρχική υλοποίηση των αλγορίθμων η οποία ήταν υλοποιημένη στο software. Δηλαδή, εκτελέσαμε την αρχική υλοποίηση των δύο αλγορίθμων και βγάλαμε τους χρόνους εκτέλεσης. Εν συνεχεία, εκτελέσαμε την δεύτερη υλοποίηση των δύο αλγορίθμων στην οποία πραγματοποιείται αποθήκευση των δεδομένων στη εσωτερική μνήμη της FPGA και συγκρίναμε τους χρόνους με την αρχική υλοποίηση. Τέλος,πραγματοποιήθηκε το ίδιο πράγμα και για την τρίτη υλοποίηση όπου γίνεται αποθήκευση των δεδομένων στην εξωτερική μνήμη και εκτελέστηκε στη Convey. Οι χρόνοι παρατίθονται στο κεφάλαιο 6.3 με αναλυτικά γραφήματα όπου φαίνεται η διαφορά στην απόδοση των υλοποιήσεων στο hardware με αυτή στο software.

6.2 Πόροι Συστήματος

Κάθε λειτουργική μονάδα απαιτεί διαφορετικό αριθμό πόρων. Για παράδειγμα η λειτουργική μονάδα process εμπεριέχει αριθμιτικές και λογικές πράξεις. Για την υλοποίηση των πράξεων αυτών απαιτείται από το σύστημα τη χρήση DSP. Οι πράξεις του πολλαπλασιασμού και της πρόσθεση χρειάζονται περισσότερες από μία DSP για τον υπολογισμό τους. Το LUT αντιπροσωπεύει το LookUp Table και στην ουσία είναι ένας πίνακας ο οποίος καθορίζει ποια είναι η έξοδος για τις συγκεκριμένες εισόδους. Δηλαδή η διασύνδεση λογικών πυλών μπορεί να γίνει από ένα LUT. Παρακάτω παρουσιάζονται οι πόροι που χρησιμοποιήθηκαν για την υλοποίηση του κάθε αλγορίθμου.

Structured Grid				
	BRAM	**DSP**	**FF**	**LUT**
Total	0	14	306	678
Available	1090	900	437200	218600
Utilization	0.00%	1.55%	0.70%	3.1%

Εικόνα 21: Πόροι συστήματος που χρησιμοποιήθηκαν για τα δομημένα πλέγματα

Unstructured Grid				
	BRAM	**DSP**	**FF**	**LUT**
Total	0	14	240	365
Available	1090	900	437200	218600
Utilization	0.00%	1.55%	0.55%	1.67%

Εικόνα 22: Πόροι συστήματος που χρησιμοποιήθηκαν για τα μη δομημένα πλέγματα

6.3 Απόδοση συστημάτων

6.3.1 Περιγραφή των datasets

Για την εκτέλεση των δύο υλοποιήσεων των αλγορίθμων δομημένων και μη πλεγμάτων έγινε χρήση δύο διαφορετικών dataset. Δεν ήταν εφικτό να γίνει το ίδιο αρχείο γιατί, η εσωτερική μνήμη της FPGA δεν επέτρεπε το ίδιο αριθμό δεδομένων όπως επιτρέπει η εξωτερική μνήμη.

Η δημιουργία του dataset γίνεται με τον εξής τρόπο. Αρχικά, δίνουμε έναν αριθμό γραμμών και στηλών που επιθυμούμε να έχει ο πίνακας που εισάγεται και εν συνεχεία δημιουργείται το αρχείο dataset. Κάθε αλγόριθμος απαιτεί ένα πίνακα για τον υπολογισμό του κύριου φόρτου του αλγορίθμου. Γι αυτό το λόγο, δημιουργούμε δύο πίνακες, έναν για την κύρια επεξεργασία στο software και έναν για την κύρια επεξεργασία στο hardware.Οι πίνακες αυτοί θα περιέχουν και τα τελικά αποτελέσματα. Έτσι, ο πίνακας που απαιτείται στο software γεμίζει με τυχαίο τρόπο με τη συνάρτηση rand(). Τέλος, αντιγράφουμε τις τιμές στον αντίστοιχο πίνακα του hardware.

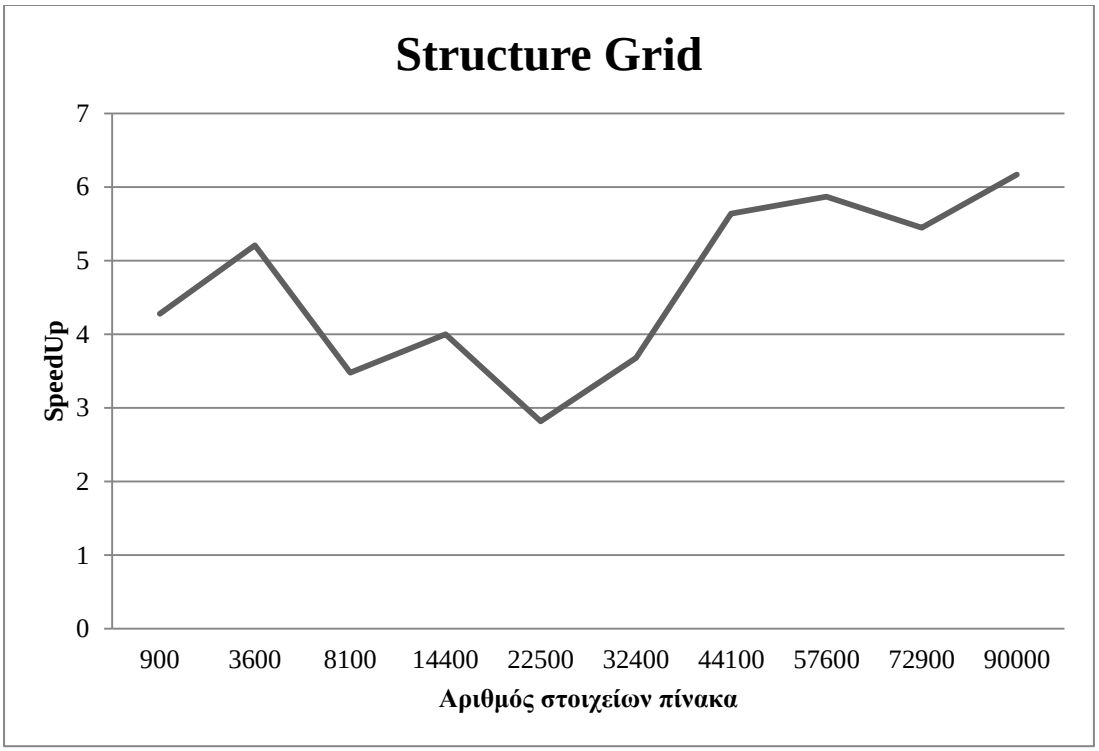
Στο κεφάλαιο αυτό θα παρατεθούν οι χρόνοι εκτέλεσης των δομημένων πλεγμάτων με τα διαφορετικά datasets όπου επιλέχθηκαν.

6.3.2 Χρόνοι εκτέλεσης των αλγορίθμων για τα δομημένα πλέγματα από το HLS

Στο κεφάλαιο αυτό παρατίθονται οι χρόνοι εκτέλεσης των δομημένων πλεγμάτων. Δόθηκαν διάφορα μεγέθη dataset όπου όπως καταλαβαίνουμε όσο μεγάλωνε το μέγεθος τόσο αυξανόταν και ο χρόνος εκτέλεσης. Οι χρόνοι παρατίθενται παρακάτω μαζί με ένα γράφημα στο οποίο φαίνεται η διαφορά στην απόδοση του αλγορίθμου. Αξίζει να σημειωθεί, ότι η χαρτογράφηση του αλγορίθμου στο hardware σύμφωνα με την αρχιτεκτονική DAE βελτίωσε την απόδοση του συστήματος εώς και 2 φορές παραπάνω. Στη συνέχεια εκμεταλευόμενοι την παραλληλία που παρέχεται από την αρχιτεκτονική αυτή και εκμεταλευόμενοι και την δυνατότητα του εργαλείου της Vivado να επιτρέπει στο πρόγραμμα να εκτελείται σε μορφή pipeline εκτόξευσε την απόδοση του συστήματος εώς και 4,5 φορές παραπάνω. Τα παράλληλα modules ήταν δύο, το ένα της fetch και το άλλο της process. Συνολικά, κατά μέσο όρο είχε μία σταθερή βελτίωση της τάξεως του 4x. Σαφώς, η πάραλληλη υλοποίηση δεν θα ήταν εφικτή εάν δεν επέτρεπε τέτοιου είδους υλοποιήσεις η FPGA.

Structured Grid		
Table Size	**Software Time(sec)**	**Hardware Parallel Time (sec)** **(2 parallel modules)**
30x30	0,012	0,002
60x60	0.048	0,009
90x90	0.660	0.190
120x120	1.320	0.330
150x150	1.410	0.500
180x180	2.580	0.710
210x210	5.420	0.960
240x240	7.230	1.230
270x270	8.340	1.540
300x300	11.920	1.930

Εικόνα 23: Πίνακας χρόνων δομημένων πλεγμάτων



Εικόνα 24: Γράφημα απόδοσης για τα δομημένα πλέγματα

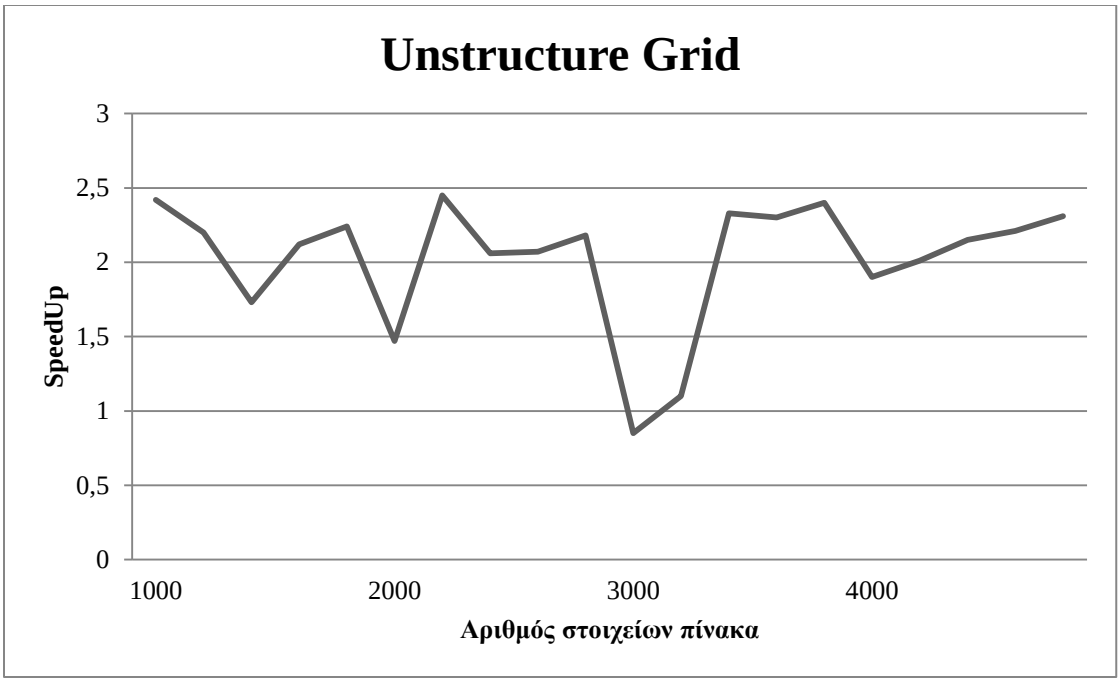
6.3.3 Χρόνοι εκτέλεσης των αλγορίθμων για τα μη δομημένα πλέγματα από το HLS

Όπως και στην προηγούμενη ενότητα θα παραθέσουμε τους χρόνους υλοποίησης των μη δομημένων πλεγμάτων. Στο συγκεκριμένο αλγόριθμο αντιμετωπίσαμε το πρόβλημα της χωρητικότητας στη μνήμη της FPGA. Για το λόγο αυτό χρησιμοποιήσαμε μικρότερο μέγεθος dataset. Ωστόσο, παρόλο την διεύρυνση του dataset οι χρόνοι παρέμειναν αρκετά μικροί κάτι το οποίο δεν έκανε εύκολη την σύγκριση των δύο υλοποιήσεων, του software και του hardware. Στον συγκεκριμένο αλγόριθμο,η χαρτογράφηση στο hardware δεν επέφερε βελτίωση της απόδοσης και καθιστούσε απαραίτητη την παραλληλοποίηση του προγράμματος. Η κύρια αιτία είναι πως στο συγκεκριμένο αλγόριθμο συναντάμε πολλές αλληλεξαρτήσεις στη μνήμη. Μετά την παραλληλοποίηση οι χρόνοι μειώθηκαν και το πρόγραμμα βελτιώθηκε και έγινε κατά μέσο όρο 2 φορές πιο αποδοτικό. Στη συνέχεια,παραθέτουμε τον πίνακα με τους χρόνους και το σχετικό γράφημα. Στο πίνακα φαίνονται το μέγεθος του πίνακα vertex σε συνδυασμό με

διαφορετικό μέγεθος του πίνακα triangle. Ανάλογα με τους συνδυασμούς που προκύπτουν αλλάζει και ο αριθμός των επαναλήψεων. Όσες περισσότερες επαναλήψεις έχουμε τόσο μεγαλώνει και ο χρόνος εκτέλεσης του προγράμματος. Ωστόσο, στις περισσότερες επαναλήψεις και στους μεγαλύτερους χρόνους παρατηρήθηκε και η μεγαλύτερη βελτίωση στην απόδοση όπως φαίνεται και παρακάτω:

Unstructured Grid				
Vertex Number	**Triangle Number**	**Iterations**	**Software Time(sec)**	**Hardware Parallel Time** **(4 parallel modules)**
1000	200	570	0,00800	0,00330
	400	1026	0,17000	0,07700
	600	413	0,00970	0,00560
	800	111	0,00340	0,00160
	1000	73	0,00220	0,00098
2000	400	452	0,02300	0,15600
	800	2342	0,13000	0,05600
	1200	267	0,01200	0,05800
	1600	198	0,01100	0,00530
	2000	75	0,00480	0,00220
3000	600	659	0,07400	0,05700
	1200	1233	0,16700	0,15100
	1800	314	0,02800	0,01200
	2400	165	0,01500	0,00650
	3000	80	0,01200	0,00500
4000	800	1153	0,07400	0,03900
	1600	2166	0,16700	0,08300
	2400	282	0,02800	0,01300
	3200	134	0,01500	0,00680
	4000	97	0,01200	0,00520

Εικόνα 25: Πίνακας χρόνων για τα μη δομημένα πλέγματα στην FPGA



Εικόνα 26: Γράφημα απόδοσης για τα μη δομημένα πλέγματα στην FPGA

6.3.4 Χρόνοι εκτέλεσης των αλγορίθμων για τα δομημένα και μη δομημένα πλέγματα στη Convey

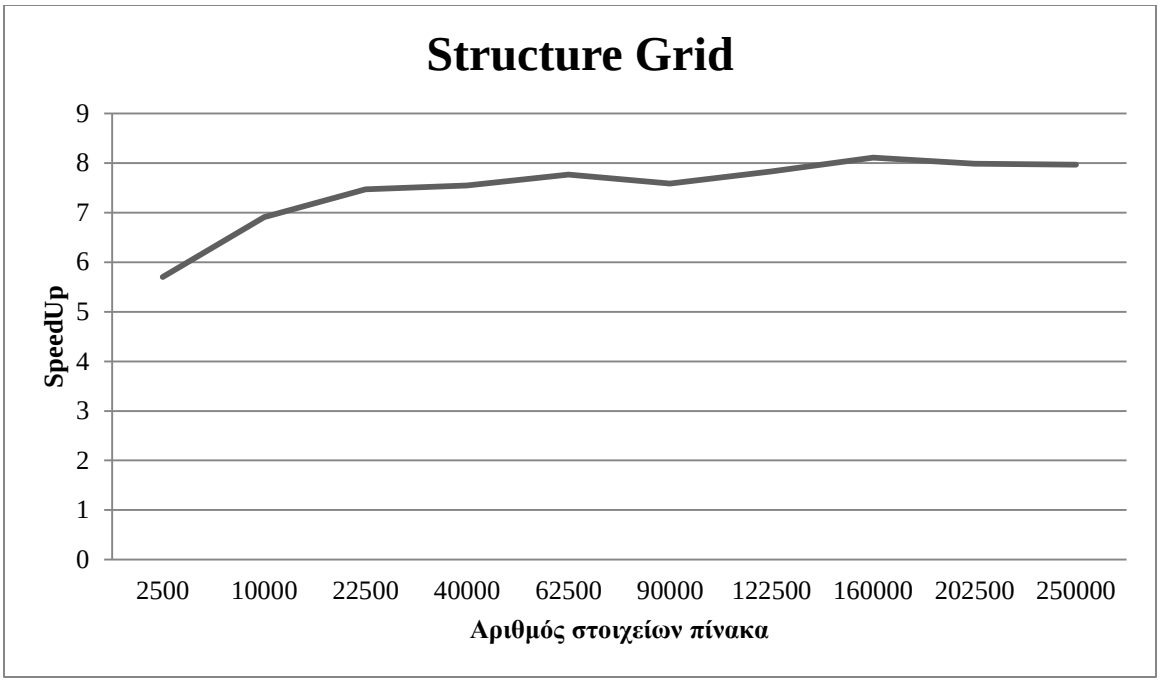
Στην ενότητα αυτή εκτελέσαμε την τελευταία υλοποίηση στη Convey με διαφορετικού μεγέθους dataset και συγκρίναμε τους χρόνους του υλοποιημένου αλγορίθμου στο software με αυτούς του hardware. Οι χρόνοι και στη Convey έδειξαν το αναμενόμενο αποτέλεσμα. Δηλαδή, η υλοποίηση του αλγορίθμου με την αρχιτεκτονική DAE επέφερε βελτίωση στην απόδοση καθιστώντας την εως και δυο φορές πιο γρήγορη. Ορισμένες φορές ήταν 2,5 φορές πιο αποδοτική από την αρχική και άλλες φορές είχαν περίπου ίσους χρόνους ανάλογα με τον αριθμό των δεδομένων. Ελάχιστες ήταν οι φορές που η αρχιτεκτονική DAE κατέστησε πιο αργό το σύστημα από ότι ήταν το αρχικό. Στις δύο αυτές περιπτώσεις ο αριθμός των επαναλήψεων ήταν μεγάλος ωστόσο δεν μπορούμε να θεωρήσουμε ως δεδομένο ότι αυτό επηρέασε τη διαφορά των

δύο υλοποιήσεων στους χρόνους ,εφόσον σε ανάλογο μεγάλο αριθμό επναλήψεων ο χρόνος στο hardware ήταν μικρότερος απο αυτόν στο software. Τα αποτελέσματα επιβεβαιώνουν τις πειραματικές ενδείξεις όπου έδειξαν ότι η αρχιτεκτονική DAE μεταβάλει την απόδοση του συστήματος κατά 2,28 πιο γρήγορη όσο αφορά τα μη δομημένα πλέγματα. Μεγάλη αύξηση στην απόδοση υπήρξε στα δομημένα πλέγματα όπου το πρόγραμμα έγινε και 7 x πιο αποδοτικό. Συνεπώς,καταφέραμε να απεικονίσουμε τους αλγόριθμους των δομημένων και μη δομημένων πλεγμάτων στο εργαλείο της vivado και συγχρόνως να γίνει και πιο αποδοτικό.

Πιο αναλυτικά παρατίθεται οι πίνακες και τα γραφήματα όπου δέιχνουν τους χρόνους που πραγματοποίησε η κάθε εκτέλεση στο software και στο hardwareγια διαφορετικό αριθμό δεδομένων,και επαναλήψεων που πραγματοποίησε ο αλγόριθμος.

Structured Grid		
Table Size	**Software Time(sec)**	**Hardware Parallel Time**
50x50	0,19	0,03
100x100	4,31	0,062
150x150	19,18	2,57
200x200	54,07	7,16
250x250	142,97	18,39
300x300	259,85	34,24
350x350	494,52	63,15
400x400	988,81	121,96
450x450	1382,16	172,92
500x500	2281,76	286,28

Εικόνα 27: Πίνακας χρόνων για τα δομημένα πλέγματα στην Convey



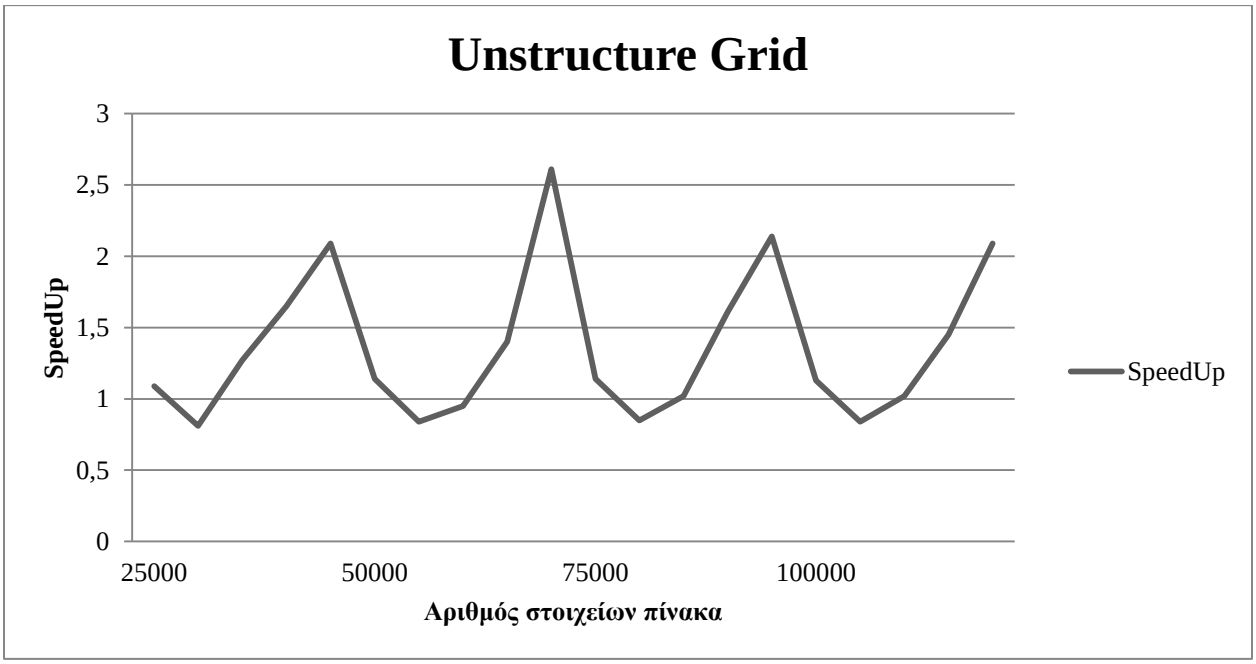
Εικόνα 28: Γράφημα απόδοσης για τα δομημένα πλέγματα στην Convey

Στο συγκεκριμένο αλγόριθμο εκτοξεύσαμε την απόδοση και ξεπεράσαμε την αρχική εκτίμηση της βελτίωσης κατά 2,5 φορές. Ο συγκεκριμένος αλγόριθμος είχε την μεγαλύτερη βελτίωση στην απόδοση,εφόσον στην fpga άγγιξε το 4,5 % και στη convey μέχρι και το 8%. Στη συνέχεια θα παρουσιάσουμε τους χρόνους του αλγορίθμου των μη δομημένων πλεγμάτων όπου δεν περιμένουμε μεγάλη διαφορά με τους χρόνους της εκτέλεσης στην fpga εφόσον στον αλγόριθμο αυτό υπάρχουν αλληλοεξαρτήσεις στα δεδομένα. Παρακάτω παρουσιάζονται οι χρόνοι και το γράφημα των μη δομημένων πλεγμάτων.

Unstructured Grid				
Vertex Number	**Triangle Number**	**Iterations**	**Software Time(sec)**	**Hardware Parallel Time**
25000	5000	1523	1,05	0,97
	10000	7269	6,01	7,46
	15000	677	0,83	0,65
	20000	229	0,45	0,27
	25000	278	0,59	0,28
	10000	1885	2,66	2,33

50000	20000	7460	12,89	15,36
	30000	873	2,17	2,28
	40000	360	1,31	0,93
	50000	281	1,29	0,49
75000	15000	6498	13,28	11,69
	30000	5935	15,76	18,53
	45000	946	3,59	3,52
	60000	426	2,26	1,41
	75000	379	2,44	1,14
100000	20000	5093	14,10	12,51
	40000	5869	21,00	25,06
	60000	950	4,87	4,79
	80000	846	5,21	3,58
	100000	210	2,35	1,12

Εικόνα 29: Πίνακας χρόνων για τα μη δομημένα πλέγματα στην Convey



Εικόνα 30: Γράφημα απόδοσης για τα μη δομημένα πλέγματα στην Convey

Στο παραπάνω αλγόριθμο οι χρόνοι εκτέλεσης στη Convey δεν είχαν μεγάλες διαφορές σε σχέση με την fpga όπου και στις δύο περιπτώσεις επιτύγχαμε την βελτίωση του προγράμματος κατά 2,5 φορές μέσο όρο.

Ανακεφαλαιώνοντας, παρατηρήσαμε ότι και στις δύο υλοποιήσεις των αλγοριθμικών μεθόδων δομημένων και μη δομημένων πλεγμάτων πραγματοποιήθηκε βελτίωση στην απόδοση. Στα δομημένα πλέγματα η απόδοση έγινε 5x φορές πιο γρήγορη κατά μέσο όρο στην περίπτωση όπου έχουμε αποθήκευση των δεδομένων εσωτερικά της FPGA και 7x φορές πιο γρήγορο στη περίπτωση οπου η αποθήκευση των δεδομένων γίνεται στην εξωτερική μνήμη. Αντιθέτως, στα μη δομημένα πλέγματα και στις δύο υλοποιήσεις είχαμε παρόμοια ποσοστά βελτίωσης της απόδοσης με κατά μέσο όρο 2,5x φορές πιο γρήγορο.

Τέλος, αξίζει να σημειωθεί ότι, αντιμετωπίσαμε πρόβλημα σε μεγάλο αριθμό δεδομένων όσο αναφορά τη δεύτερη υλοποίηση όπου γινόταν η αποθήκευση των δεδομένων εσωτερικά της FPGA κάτι το οποίο δεν αντιμετωπίστηκε στην τρίτη υλοποίηση εφόσον η αποθήκευση γινόταν στην εξωτερική μνήμη.

Κεφάλαιο 7

7.1 Συμπεράσματα

Στη παρούσα διπλωματική παρουσιάστηκε η χρήση δύο εκ των δεκατριών νάνων, ως benchmarks στη πλατφόρμα ZYNQ ZC706 και στη Convey. Οι αλγόριθμοι που υλοποιήθηκαν ήταν οι μέθοδοι Jacobi και Serial Based, οι οποίοι υπάγονται στις αλγοριθμικές μεθόδους των δομημένων και μη δομημένων πλεγμάτων. Στην απεικόνιση και των δύο αυτών αλγορίθμων χρησιμοποιήθηκε το κοινό framework DAE. Το παραπάνω framework λειτούργησε εξαιρετικά με τις συγκεκριμένες αλγοριθμικές μεθόδους εφόσον παρουσιάζονται προβλήματα εξαρτήσης των δεδομένων στη μνήμη και είναι ιδανικό για τέτοιου είδος προβλήματα όπως αναφέραμε και παραπάνω. Το πρόγραμμα χωρίζεται σε δύο λειτουργικές μονάδες τη fetch και τη process. Η πρώτη είναι αρμόδια για την ανάκτηση των διευθύνσεων και η δεύτερη για το υπολογισμό του κύριου φόρτου εργασίας του αλγορίθμου. Οι διαδικασίες αυτές γίνονται παράλληλα. Η λειτουργική μονάδα process δεν περιμένει την ανάκτηση όλων των διευθύνσεων παραμόνο αυτών που απαιτούνται για τον υπολογισμό των λογικών πράξεων. Επομένως,παρατηρείται μεγάλη βετίωση στην απόδοση εκμεταλλευόμενοι τα πλεονεκτήματα του παράλληλου προγραμματισμού.

Οι υλοποίησεις πραγματοποιήθηκαν στο εργαλείο της Vivado HLS. Παλαιότερα, η σχεδίαση RTL ήταν αρκετά δύσκολη και χρειαζόταν μεγάλη εξειδίκευση από τους προγραμματιστές. Ωστόσο, το εργαλείο Vivado HLS παρέχει μεγάλη ευελιξία στη χαρτογράφηση των αλγορίθμων με τη προσφορά της αυτοματοποιημένης διαδικασίας της σύνθεσης του αρχείου rtl και του verification με αποτέλεσμα η υλοποίηση στο hardware να καθίσταται αρκετά πιο εύκολη.Επίσης,παρέχει directives που συμβάλλουν στην βελτίωση της απόδοσης.

Στο εγγύς μέλλον,οι υλοποιήσεις που συνδυάζουν το software και το hardware θα γίνουν ολοένα και περισσότερες και το εργαλείο Vivado HLS θα παίξει σημαντικό ρόλο στην βελτίωση της απόδοσης των συστημάτων.

7.2 Μελλοντική εργασία

Όπως είδαμε παραπάνω η απεικόνιση των δύο αλγορίθμων στο hardware και η εκτέλεση των αλγορίθμων σε πλατφόρμες όπως FPGAs και Convey συντέλεσαν στη βελτίωση της απόδοσης τους. Αναλογιζόμαστε, ότι οι μελλοντικές εφαρμογές, οι οποίες θα αφορούν την επεξεργασία και τη μεταφορά μεγάλου όγκου δεδομένων θα στραφούν σε ανάλογες υλοποιήσεις με αυτές που εφαρμόσαμε στη παρούσα διπλωματική. Για το λόγο αυτό πρέπει να πειραματιστούμε και με άλλες πλατφόρμες βγάζοντας πιο ολοκληρωμένα συμπεράσματα. Αρχικά, με τη χρήση νεότερων FPGA όπως είναι η Ultrascale. Οι συγκεκριμένες πλατφόρμες έχουν μειωμένη κατανάλωση και συγχρόνως διακρίνονται για τις υψηλότερες ταχύτητες σε σχέση με FPGAs προηγούμενης γενιάς. Βρίσκουν εφαρμογή κυρίως σε προγράμματα με υψηλές απαιτήσεις όπου χρειάζεται αποθήκεση, επεξεργασία και μεταφορά μεγάλου όγκου δεδομένων. Μία ακόμα εναλλακτική χρήση πλατφόρμας είναι αυτή της Maxeler. Η πλατφόρμα αυτή δίνει τη δυνατότητα για βελτίωση στην απόδοση του συστήματος μέχρι και 30% σε σχέση με μία συμβατική CPU. Η υλοποίηση πραγματοποιείται από το συνδυασμό του hardware και του software, χρησιμοποιώντας γλώσσες όπως η C,C++,Java για την αύξηση της απόδοσης σε προγράμματα που απαιτούν τη γρήγορη ροή των δεδομένων.

Επίσης, η διαμέριση ενός δομημένου πλέγματος σε υποπλέγματα και συγχρόνως με τη χρήση κατάλληλων δομών μας παρέχεται η δυνατότητα αύξησης του παραλληλισμού εσωτερικά της αρχιτεκτονικής εκμεταλλευόμενοι στο έπακρο τα πλεονεκτήματα του παράλληλου προγραμματισμού. Με τη σωστή χρήση και των αυτοματοποιημένων directives που μας παρέχει το εργαλείο Vivado HLS αυξάνουμε ακόμα περισσότερο την παραλληλία και την απόδοση του συστήματος.

Βιβλιογραφία

[1]K. Asanovic, P. Husbands, Plishker, J. Shalf, S. W. Williams and K. A. Yellick, The Landscape of Parallel Computing Reasearch: A view from Berkeley, Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley (Dec 2006)

[2]James E. Smith "Decoupled Access/Execute Computer Architectures", Department of Electrical and Computer Engineering, University of Wisconsin-Madison,Madison,Wisconsin 53706

[3]Deependra Talla and Lizy K.John ''A Decoupled Architecture for Accelerating Multimedia Applications''Laboratory for Computer Architecture Department of Electrical and Computer Engineering The University of Texas, Austin, TX 78712

[4]Michael Sung, Ronny Krashinsky, and Krste Asanovic ''Multithreading Decoupled Architectures for Complexity-Effective General Purpose Computing ''

[5]Konstantinos Krommydas,Wu-chun Feng ,Christos D Antonopoulos,Nikolaos Bellas ''Open Dwarfs Characterization of Dwarf-Based Benchmarks on Fixed Reconfigurable Architectures''

[6]W. Feng, H. Lin, T. Scongland and J. Zhang "OpenCl and 13 Dwarfs: A Work in Progress"Department of Computer Science Virginia Tech Blacksburg, VA 24060, USA

[7]Mariza Ferro, Antonio R. Mury, Laion F. Manfroi, Bruno Schlze "High Performance Computing Evaluation A methodology based on Scientific Application Requirements"National Laboratory of Scientific Computing, Getulio Vargas 333, Petropolis, Rio de Janeiro

[8]E. M. Daoudi, A. Lakhouaja, and H. Outada "Study of the Parallel Block One-Sided Jacobi Method"University of Mohamed First, Faculty Sciences Department of Mathematics and Computer Science LaRI Laboratory, 60 000 Ouijda, Morocco

[9]ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC

[10]Xilinx Inc. ¨Vivado Design Suite User Guide Getting Started UG910´´

[11]Xilinx Inc. ¨Vivado Design Design Suite Tutorial: High Level Synthesis UG871´´

[12]Xilinx Inc. ¨Vivado Design Suite User Guide: Synthesis UG901´´

[13]Xilinx. Inc ´´Improving Performance Vivado HLS 2013.3 Version´´

[14]George Charitopoulos, Charalampos Vatsolakis, Stefanos Sidiropoulos, Grigorios Chrysos and Dionisios N. Pnevmatikatos "A Decoupled Access-Execute Architecture for Reconfigurable Accelerators" School of Electrical and Computer Engineering Technical University of Crete, Chania, Greece 73100

[15]Prof. Gerhard Wellein, Dr Georg Hager "The seven dwarfs of HPC Possible projects" MuCoSim 27.04.2010 HPC Services, Regionales Rechenzentrum Erlangen (RRZE) Department fur Informatik

[16]Michael Bader "HPC-Algorithms and Applications Dwarf #5 Structured Grids"Winter 2012/2013 Technische Universitat Munchen

[17]Michael Bader "HPC-Algorithms and Applications Dwarf #6 Unstructured Grids"Winter 2012/2013 Technische Universitat Munchen

[18]The Convey HC-2 Computer Architectural Overview

[19]Tao Chen and G. Edward Suh "Efficient Data Supply for Hardware Accelerators with Prefetching and Access/Execute Decoupling"Cornell University, Ithaca, NY 14850, USA

[20]Shaoyi Cheng and John Wawrynek "Architectural Synthesis of Computational Pipelines with Decoupled Memory Access" Department of EECS, UC Berkeley, California, USA 94720

[21]Chen-Han-Ho Sung Jin Kim Karthikeyan Sankaralingam Vertical Research Group "Memory Access Dataflow" Department of Computer Sciences, University of Winscosin-Madison

[22]Krommydas, K. Owaida, M. Antonopoulos, C. Bellas, N.& Feng, W.C.(2013) "On the portability of the OpenCL dwarfs on fixed and reconfigurable parallel platforms. In Proceeding of the Iternational Conference on Parallel and Distributed Systems"(ICPADS '13) (pp.432-433). Korea: Seoul.

[23]K. Koukos, P.Ekemark, G. Zacharopoulos, V. Spiliopoulos, S. Kaxiras, and A.Jimborean, "Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-

purpose programs", in Proceedings of the 25th International Conference on Compiler Construction Pages 121-131

[24]A. Berrached, L. D. Coraor, P. T. Hulina "A decoupled access/execute architecture for efficient access of structured data" in Systems Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference

Παράρτημα

Α Κώδικας Δομημένων Πλεγμάτων

Α.1 Υλοποίηση 1η

Στηνενότητααυτή παρατίθεται ο κώδικας των μη

Στην ενότητα αυτή παρατίθεται η πρώτη υλοποίηση των δομημένων πλεγμάτων.Αποτελείται από δύο συνάρτησεις οι οποίες αναλύθηκαν ,τη fetchκαι τη process.

```
voidfetchUnit(intinit_address,int *dt1,int *dt2,int *dt3,int *dt4,int *dt5)
{

#pragma HLS INTERFACE ap_stable port=init_address
#pragma HLS INTERFACE ap_fifo port=dt5
#pragma HLS INTERFACE ap_fifo port=dt4
#pragma HLS INTERFACE ap_fifo port=dt3
#pragma HLS INTERFACE ap_fifo port=dt2
#pragma HLS INTERFACE ap_fifo port=dt1

#pragma HLS DATAFLOW
#pragma HLS CLOCK domain=default

int i,j,index_grid1,index_grid2,index_grid3,index_grid4,index_grid5;
for(i = 0; i < ROWS ; i ++)
{
for(j = 0; j < COLS; j ++)
{
int l=i*COLS+j;
if((i!=0 && j!=0) && (i!=ROWS-1 && j!=COLS-1) )
{
index_grid1=init_address+((i-1)*COLS+j)*8;
*dt1=index_grid1;

index_grid2=init_address+((i+1)*COLS+j)*8;
*dt2=index_grid2;

index_grid3=init_address+(i*COLS+j-1)*8;
```

```
*dt3=index_grid3;

index_grid4=init_address+(i*COLS+j+1)*8;
*dt4=index_grid4;



}
else
*dt1= *dt2= *dt3= *dt4=0;

index_grid5=init_address+(i*COLS+j)*8;
*dt5=index_grid5;

dt1++;
dt2++;
dt3++;
dt4++;
dt5++;
}
}
}
```

Εικόνα 31: Λειτουργική μονάδα fetch για τα δομημένα πλέγματα

```
while(!converged)
     {
converged =true;
iter++;

                  //Klhsh fetchUnit//
if(iter==1){
fetchUnit((int)&grid2[0],matrix_addr1,matrix_addr2,matrix_addr3,matrix_addr4,matrix_addr5);
        }
else{
fetchUnit((int)&tempGrid2[0],matrix_addr1,matrix_addr2,matrix_addr3,matrix_addr4,matrix_addr5);
        }
for(i = 0; i < ROWS ; i ++)
        {
for(j = 0; j < COLS; j ++)
           {
int l=i*COLS+j;

if((i!=0 && j!=0) && (i!=ROWS-1 && j!=COLS-1) )
               {
                  ptr1=(double*)matrix_addr1[l];
values1[l]=*ptr1;

                  ptr2=(double*)matrix_addr2[l];
values2[l]=*ptr2;

                  ptr3=(double*)matrix_addr3[l];
values3[l]=*ptr3;

                  ptr4=(double*)matrix_addr4[l];
values4[l]=*ptr4;
```

```
            }

else
values1[l]=values2[l]=values3[l]=values4[l]=0;
                    ptr5=(double*)matrix_addr5[l];
values5[l]=*ptr5;
          }
        }
```

```
processUnit(tempGrid2,(int)&tempGrid2[0],matrix_addrTempGrid,values1,values2,values3,values4,values5,
matrix_convAddr,matrix_conVal,(int)&converged,solver->epsilon);

for(i=0;i<ROWS*COLS;i++)
          {

if(matrix_conVal[i]==false)
converged=false;
          }
    }
for(i=0;i<ROWS*COLS;i++)
result2[i]=tempGrid2[i];
```

Εικόνα 32: Φόρτωση δεδομένων από την κύρια μνήμη

```
void processUnit(double *tmpG2,int init_addr,int *tmpGridAddr,double *val1,double *val2,double
*val3,double *val4,double *val5,int *conAdd,int *conVal,int initConverged_Add,double eps){

int i,j,index_grid;
#pragma HLS INTERFACE ap_stable port=eps
#pragma HLS INTERFACE ap_stable port=initConverged_Add
#pragma HLS INTERFACE ap_fifo port=conVal
#pragma HLS INTERFACE ap_fifo port=conAdd
#pragma HLS INTERFACE ap_fifo port=val5
#pragma HLS INTERFACE ap_fifo port=val4
#pragma HLS INTERFACE ap_fifo port=val3
#pragma HLS INTERFACE ap_fifo port=val2
#pragma HLS INTERFACE ap_fifo port=val1
#pragma HLS INTERFACE ap_fifo port=tmpGridAddr
#pragma HLS INTERFACE ap_stable port=init_addr
#pragma HLS INTERFACE ap_fifo port=tmpG2
#pragma HLS DATAFLOW
#pragma HLS CLOCK domain=default
for(i = 0; i < ROWS ; i ++)
 {
for(j = 0; j < COLS; j ++)
   {
int l=i*COLS+j;
      index_grid=init_addr+l*8;
      *tmpGridAddr=index_grid;
      *conAdd = initConverged_Add;
if((i!=0 && j!=0) && (i!=ROWS-1 && j!=COLS-1) )
        {
```

```
            *tmpG2=0.25*(*val1+*val2+*val3+*val4);

if(*tmpG2>*val5)
                {
if(*tmpG2 -*val5 >= eps)
                        *conVal = false;
else
                        *conVal = true;
                }
else
                {
if(*tmpG2-*val5 >= eps)
                        *conVal = false;
else
                        *conVal= true;
                }
          }
else
            *(conVal+l) = true;
tmpGridAddr++;
conAdd++;
tmpG2++;
val1++;val2++;val3++;val4++;val5++;
conVal++;
    }
 }
```

Εικόνα 33: Λειτουργική μονάδα process για τα δομημένα πλέγματα

Α.2 Υλοποίηση 2η

```
while(!converged)
{
converged =true;
iter++;

if(iter==1)
            TestFunction((int)&grid2[0],(int)&tempGrid2[0],tempGrid2,matrix_conVal,solver->epsilon,iter);

else
            TestFunction((int)&grid2[0],(int)&tempGrid2[0],resultHw,matrix_conVal,solver->epsilon,iter);


for(i=0;i<ROWS*COLS;i++)
            {
if(matrix_conVal[i]==false)
converged=false;
            }
```

```
    }

for(i=0;i<ROWS*COLS;i++)
result2[i]=resultHw[i];
```

```
void TestFunction(int init_address,int init_addrTmpGrid,double *resultH,int *conVal,double eps,int
iterations)
{
int sgrid_address1[ROWS*COLS];
int sgrid_address2[ROWS*COLS];
int sgrid_address3[ROWS*COLS];
int sgrid_address4[ROWS*COLS];
int sgrid_address5[ROWS*COLS];

double sgrid_data1[ROWS*COLS];
double sgrid_data2[ROWS*COLS];
double sgrid_data3[ROWS*COLS];
double sgrid_data4[ROWS*COLS];
double sgrid_data5[ROWS*COLS];

#pragma HLS INTERFACE ap_stable port=init_address
#pragma HLS INTERFACE ap_stable port=init_addrTmpGrid
#pragma HLS INTERFACE ap_fifo port=resultH
#pragma HLS INTERFACE ap_fifo port=conVal
#pragma HLS INTERFACE ap_stable port=eps
#pragma HLS INTERFACE ap_stable port=iterations
#pragma HLS INTERFACE ap_fifo port=sgrid_address1
#pragma HLS INTERFACE ap_fifo port=sgrid_address2
#pragma HLS INTERFACE ap_fifo port=sgrid_address3
#pragma HLS INTERFACE ap_fifo port=sgrid_address4
#pragma HLS INTERFACE ap_fifo port=sgrid_address5
#pragma HLS INTERFACE ap_fifo port=sgrid_data1
#pragma HLS INTERFACE ap_fifo port=sgrid_data2
#pragma HLS INTERFACE ap_fifo port=sgrid_data3
#pragma HLS INTERFACE ap_fifo port=sgrid_data4
#pragma HLS INTERFACE ap_fifo port=sgrid_data5

#pragma HLS DATAFLOW
#pragma HLS CLOCK domain=default

if(iterations==1)
{
  fetchUnit(init_address,sgrid_address1,sgrid_address2,sgrid_address3,sgrid_address4,sgrid_address5);
read_memory(init_address,sgrid_address1,sgrid_address2,sgrid_address3,sgrid_address4,sgrid_address5,sgri
d_data1,sgrid_data2,sgrid_data3,sgrid_data4,sgrid_data5);
processUnit(resultH,sgrid_data1,sgrid_data2,sgrid_data3,sgrid_data4,sgrid_data5,conVal,eps);
}
else
{
fetchUnit(init_addrTmpGrid,sgrid_address1,sgrid_address2,sgrid_address3,sgrid_address4,sgrid_address5);
```

```
read_memory(init_addrTmpGrid,sgrid_address1,sgrid_address2,sgrid_address3,sgrid_address4,sgrid_address
5,sgrid_data1,sgrid_data2,sgrid_data3,sgrid_data4,sgrid_data5);
processUnit(resultH,sgrid_data1,sgrid_data2,sgrid_data3,sgrid_data4,sgrid_data5,conVal,eps);

}
}
```

Εικόνα 34: Συνάρτηση test function

```
void read_memory(int init_address,int *sgrid_address1,int
*sgrid_address2,int *sgrid_address3,int *sgrid_address4,int *sgrid_address5,double *sgrid_data1,double
*sgrid_data2,double *sgrid_data3,double *sgrid_data4,double *sgrid_data5)
{
#pragma HLS INTERFACE ap_stable port=init_address
#pragma HLS INTERFACE ap_fifo port=sgrid_address5
#pragma HLS INTERFACE ap_fifo port=sgrid_address4
#pragma HLS INTERFACE ap_fifo port=sgrid_address3
#pragma HLS INTERFACE ap_fifo port=sgrid_address2
#pragma HLS INTERFACE ap_fifo port=sgrid_address1
#pragma HLS INTERFACE ap_fifo port=sgrid_data5
#pragma HLS INTERFACE ap_fifo port=sgrid_data4
#pragma HLS INTERFACE ap_fifo port=sgrid_data3
#pragma HLS INTERFACE ap_fifo port=sgrid_data2
#pragma HLS INTERFACE ap_fifo port=sgrid_data1

#pragma HLS DATAFLOW
#pragma HLS CLOCK domain=default

double grid_array[ROWS*COLS] = {4276,32,202,2690,3903,8400,865,6219,6609};
int i, j, k;
int ind1,ind2,ind3,ind4,ind5,*ptr1,*ptr2,*ptr3,*ptr4,*ptr5;

for(i = 0; i < ROWS ; i ++)
 {
for(j = 0; j < COLS; j ++)
   {
int l=i*COLS+j;

        ind5=(*(sgrid_address5+l)-init_address)/8;
        *(sgrid_data5+l)=*(grid_array+ind5);

if((i!=0 && j!=0) && (i!=ROWS-1 && j!=COLS-1) )
           {
              ind1=(*(sgrid_address1+l)-init_address)/8;
              *(sgrid_data1+l)=*(grid_array+ind1);

              ind2=(*(sgrid_address2+l)-init_address)/8;
              *(sgrid_data2+l)=*(grid_array+ind2);

              ind3=(*(sgrid_address3+l)-init_address)/8;
              *(sgrid_data3+l)=*(grid_array+ind3);
```

```
            ind4=(*(sgrid_address4+l)-init_address)/8;
            *(sgrid_data4+l)=*(grid_array+ind4);
          }
else
            *(sgrid_data1+l)=*(sgrid_data2+l)=*(sgrid_data3+l)=*(sgrid_data4+l)=0;
}
   }
 }
```

Εικόνα 35: Συνάρτηση read memory

Α.3 Υλοποίηση 3η

```
void TestFunction(int init_address,int init_tmpAddress,double *resultH,double eps,int iterations,double
*gridArray,double *tmpGridArray){

int sgrid_address1[ROWS*COLS];
int sgrid_address2[ROWS*COLS];
int sgrid_address3[ROWS*COLS];
int sgrid_address4[ROWS*COLS];
int sgrid_address5[ROWS*COLS];

double sgrid_data1[ROWS*COLS];
double sgrid_data2[ROWS*COLS];
double sgrid_data3[ROWS*COLS];
double sgrid_data4[ROWS*COLS];
double sgrid_data5[ROWS*COLS];

int i,flag=1;
int conVal[ROWS*COLS];

#pragma HLS INTERFACE ap_stable port=init_address
#pragma HLS INTERFACE ap_stable port=init_tmpAddress
#pragma HLS INTERFACE ap_fifo port=resultH
#pragma HLS INTERFACE ap_stable port=eps
#pragma HLS INTERFACE ap_stable port=iterations
#pragma HLS INTERFACE ap_fifo port=gridArray
#pragma HLS INTERFACE ap_fifo port=tmpGridArray

#pragma HLS INTERFACE ap_fifo port=sgrid_address5
#pragma HLS INTERFACE ap_fifo port=sgrid_address4
#pragma HLS INTERFACE ap_fifo port=sgrid_address3
#pragma HLS INTERFACE ap_fifo port=sgrid_address2
#pragma HLS INTERFACE ap_fifo port=sgrid_address1

#pragma HLS INTERFACE ap_fifo port=sgrid_data5
#pragma HLS INTERFACE ap_fifo port=sgrid_data4
#pragma HLS INTERFACE ap_fifo port=sgrid_data3
#pragma HLS INTERFACE ap_fifo port=sgrid_data2
#pragma HLS INTERFACE ap_fifo port=sgrid_data1
#pragma HLS INTERFACE ap_fifo port=conVal
```

```
#pragma HLS DATAFLOW
#pragma HLS CLOCK domain=default
while (1)
{
iterations++;
if(iterations==1)
  {
fetchUnit(init_address,sgrid_address1,sgrid_address2,sgrid_address3,sgrid_address4,sgrid_address5);

read_memory(init_address,sgrid_address1,sgrid_address2,sgrid_address3,sgrid_address4,sgrid_address5,sgri
d_data1,sgrid_data2,sgrid_data3,sgrid_data4,sgrid_data5,gridArray);
processUnit(resultH,sgrid_data1,sgrid_data2,sgrid_data3,sgrid_data4,sgrid_data5,conVal,eps);

for(i=0;i<ROWS*COLS;i++)
        {
if(conVal[i]==false)
flag=0;
        }

if(flag==0)
break;

  }
else
  {
fetchUnit(init_address,sgrid_address1,sgrid_address2,sgrid_address3,sgrid_address4,sgrid_address5);

read_memory(init_tmpAddress,sgrid_address1,sgrid_address2,sgrid_address3,sgrid_address4,sgrid_address5
,sgrid_data1,sgrid_data2,sgrid_data3,sgrid_data4,sgrid_data5,tmpGridArray);
processUnit(resultH,sgrid_data1,sgrid_data2,sgrid_data3,sgrid_data4,sgrid_data5,conVal,eps);

for(i=0;i<ROWS*COLS;i++)
                {
if(conVal[i]==false)
flag=0;

                }
if(flag==0)
break;
  }
}

}
break;
  }
}

}
```

Εικόνα 36: Συνάρητη test function

Β Κώδικας Μη Δομημένων Πλεγμάτων

Β.1 Υλοποίηση 1η

```
void fetchUnit0(uint64_t numVertex, uint64_t init_addressNumNeighbors, uint64_t *addrNumNeighbors)
{
#pragma HLS CLOCK domain=default
#pragma HLS DATAFLOW

#pragma HLS INTERFACE ap_stable port=numVertex
#pragma HLS INTERFACE ap_stable port=init_addressNumNeighbors
#pragma HLS INTERFACE ap_fifo depth=4096 port=addrNumNeighbors

		uint64_t i;
		for(i = 0; i < numVertex; i++)
		{
#pragma HLS PIPELINE
			*addrNumNeighbors = init_addressNumNeighbors + i*sizeof(uint64_t);
			addrNumNeighbors++;
		}
}dt3++;
dt4++;
dt5++;
}
}
}
```

Εικόνα 37: Λειτουργική υπομονάδα fetch0 για τα μη δομημένα πλέγματα

```
void fetchUnit1(uint64_t numVertex, uint64_t init_addressNeighborhood, uint64_t *NumNeighbors, uint64_t
*addrNeighborhood)
{
#pragma HLS CLOCK domain=default
#pragma HLS DATAFLOW
#pragma HLS INTERFACE ap_stable port=numVertex
#pragma HLS INTERFACE ap_stable port=init_addressNeighborhood
#pragma HLS INTERFACE ap_fifo depth=4096 port=NumNeighbors
#pragma HLS INTERFACE ap_fifo depth=4096 port=addrNeighborhood
		uint64_t i, j, temp;
		for(i = 0; i < numVertex; i++)
		{
#pragma HLS LOOP_MERGE
			temp = *NumNeighbors;
			NumNeighbors++;

			for(j = 0; j < temp; j ++)
			{
#pragma HLS PIPELINE
```

```
                *addrNeighborhood = init_addressNeighborhood + (i*numVertex + j)*sizeof(uint64_t);
                addrNeighborhood++;
            }
        }
}
```

Εικόνα 38: Λειτουργική υπομονάδα fetch1 για τα μη δομημένα πλέγματα

```
void fetchUnit2(uint64_t numVertex, uint64_t init_addressValue, uint64_t *NumNeighbors, uint64_t *Neighborhood, uint64_t *addrValue, uint64_t *addrValue_prev)
{
#pragma HLS CLOCK domain=default
#pragma HLS DATAFLOW

#pragma HLS INTERFACE ap_stable port=numVertex
#pragma HLS INTERFACE ap_stable port=init_addressValue
#pragma HLS INTERFACE ap_fifo depth=4096 port=NumNeighbors
#pragma HLS INTERFACE ap_fifo depth=4096 port=Neighborhood
#pragma HLS INTERFACE ap_fifo depth=4096 port=addrValue
#pragma HLS INTERFACE ap_fifo depth=4096 port=addrValue_prev

        uint64_t i, j, temp, temp1;

        for(i = 0; i < numVertex; i++)
        {
#pragma HLS LOOP_MERGE
            temp = *NumNeighbors;
            NumNeighbors++;

            for(j = 0; j < temp; j ++)
            {
#pragma HLS PIPELINE
                temp1 = *Neighborhood;
                *addrValue = init_addressValue + temp1*sizeof(uint64_t);

                Neighborhood++;
                addrValue++;
            }

            *addrValue_prev = init_addressValue + i*sizeof(uint64_t);
            addrValue_prev++;
        }
}
```

Εικόνα 39: Λειτουργική υπομονάδα fetch2 για τα μη δομημένα πλέγματα

```
for(i=0;i<numvertex;i++)
{
	ptr1=(uint64_t*)NumNeighbors_address[i];
	NumNeighbors_data[i]=*ptr1;

}
```

Εικόνα 40: Ανάκτηση δεδομένων υπομονάδας fetch0

```
for(i=0;i<numvertex;i++)
{
	ptr2=(uint64_t*)Neighborhood_address[i];
	Neighborhood_data[i]=*ptr2;

}
```

Εικόνα 41: Ανάκτηση δεδομένων υπομονάδας fetch1

```
for(i=0;i<numvertex;i++)
{
	ptr3=(int64_t*)Values_address1[i];
	Values_data1[i]=*ptr3;

	ptr4=(int64_t*)Values_address2[i];
	Values_data2[i]=*ptr4;

}
```

Εικόνα 42: Ανάκτηση δεδομένων υπομονάδας fetch2

```
while(!conVal)
{
			conVal=true;

			if(iterations % 2 == 0)
			{
				fetchUnit0(numvertex, (uint64_t)NumNeighbors_HW,
NumNeighbors_address);


				for(i=0;i<numvertex;i++)
				{
					ptr1=(uint64_t*)NumNeighbors_address[i];
					NumNeighbors_data[i]=*ptr1;

				}

				fetchUnit1(numvertex,  (uint64_t)Neighborhood_HW , NumNeighbors_data,
Neighborhood_address);

				for(i=0;i<numvertex;i++)
				{
					ptr2=(uint64_t*)Neighborhood_address[i];
					Neighborhood_data[i]=*ptr2;
```

```
                                        }

                                        fetchUnit2(numvertex, (uint64_t)&Values_HW_1[0], NumNeighbors_data,
Neighborhood_data, Values_address1, Values_address2);

                                        for(i=0;i<numvertex;i++)
                                        {
                                                ptr3=(int64_t*)Values_address1[i];
                                                Values_data1[i]=*ptr3;

                                                ptr4=(int64_t*)Values_address2[i];
                                                Values_data2[i]=*ptr4;

                                        }

                                        processUnit(numvertex, epsilon, (uint64_t)&Results[0], NumNeighbors_data,
Values_data1, Values_data2, &conVal, Results, Results_address);

                                                        for(int i =0; i < numvertex; i++)
                                                        {
                                                                Values_HW_2[i] =
Results[(Results_address[i]-(uint64_t)&Results[0])/sizeof(int64_t)];

                                                        }

                                                        result = 0;

                                                }

                                else
                                {
                                        fetchUnit0(numvertex, (uint64_t)NumNeighbors_HW,
NumNeighbors_address);

                                        for(i=0;i<numvertex;i++)
                                        {
                                                ptr1=(uint64_t*)NumNeighbors_address[i];
                                                NumNeighbors_data[i]=*ptr1;

                                        }

                                        fetchUnit1(numvertex,  (uint64_t)Neighborhood_HW , NumNeighbors_data,
Neighborhood_address);

                                        for(i=0;i<numvertex;i++)
                                        {
                                                ptr2=(uint64_t*)Neighborhood_address[i];
                                                Neighborhood_data[i]=*ptr2;

                                        }


                                        fetchUnit2(numvertex, (uint64_t)&Values_HW_2[0], NumNeighbors_data,
Neighborhood_data, Values_address1, Values_address2);
```

```
                                for(i=0;i<numvertex;i++)
                                {
                                        ptr3=(int64_t*)Values_address1[i];
                                        Values_data1[i]=*ptr3;

                                        ptr4=(int64_t*)Values_address2[i];
                                        Values_data2[i]=*ptr4;

                                }

                                processUnit(numvertex, epsilon, (uint64_t)&Results[0], NumNeighbors_data,
Values_data1, Values_data2, &conVal, Results, Results_address);

                                                for(int i =0; i < numvertex; i++)
                                                {
                                                        Values_HW_1[i] =
Results[(Results_address[i]-(uint64_t)&Results[0])/sizeof(int64_t)];
                                                }

                                                result = 1;

                                        }

                                        iterations++;
        }
```

Εικόνα 43: Κλήσεις λειτουργικών μονάδων fetch και process στην κύρια μνήμη

```
void processUnit(uint64_t numVertex, int64_t eps, uint64_t init_addressResult, uint64_t *NumNeighbors,
int64_t *value, int64_t *prev_value, uint64_t *conVal, int64_t *result_data, uint64_t *result_address)
{
#pragma HLS CLOCK domain=default
#pragma HLS DATAFLOW

#pragma HLS INTERFACE ap_fifo depth=4096 port=result_address
#pragma HLS INTERFACE ap_fifo depth=4096 port=result_data
#pragma HLS INTERFACE ap_fifo depth=2 port=conVal
#pragma HLS INTERFACE ap_fifo depth=4096 port=prev_value
#pragma HLS INTERFACE ap_fifo depth=4096 port=value
#pragma HLS INTERFACE ap_fifo depth=4096 port=NumNeighbors
#pragma HLS INTERFACE ap_stable port=init_addressResult
#pragma HLS INTERFACE ap_stable port=eps
#pragma HLS INTERFACE ap_stable port=numVertex

        uint64_t i, j, temp;
        int64_t sum;
        uint64_t converged = true;
        int64_t final_result = 0, previous_result, result1, result2, eps_local;

        for (i = 0; i <numVertex; i ++)
        {
#pragma HLS LOOP_MERGE
                temp = *NumNeighbors;
```

```
                        NumNeighbors++;

                        sum = 0;

                        previous_result = *prev_value;
                        prev_value++;

                        final_result = previous_result;

                        if(temp > 0)
                        {
                                sum = 0;
                                for(j = 0; j < temp; j++)
                                {
#pragma HLS PIPELINE
                                        sum = sum + *value;
                                        value++;
                                }

                                final_result = sum/temp;

                                result1 = final_result - previous_result - eps;
                                result2 = previous_result - final_result - eps;

                                if(result1 >= 0 || result2 >= 0)
                                        converged = false;
                        }
                        *result_address = init_addressResult + i*sizeof(uint64_t);
                        *result_data = final_result;

                        result_address++;
                        result_data++;

                }
                *conVal = converged;
}
```

Εικόνα 44: Λειτουργική μονάδα process για τα μη δομημένα πλέγματα

Β.2Υλοποίηση 2η

```
void TestFunction(uint64_t numvertex, int64_t eps, uint64_t *solution, uint64_t *NumNeighbors_HW,
uint64_t *Neighborhood_HW, int64_t *Values_HW_1, int64_t *Values_HW_2, int64_t *Results, uint64_t
*Results_address)
{
        uint64_t NumNeighbors_address[4096];
        uint64_t NumNeighbors_data[4096];

        uint64_t Neighborhood_address[4096];
        uint64_t Neighborhood_data[4096];

        uint64_t Values_address1[4096];
        uint64_t Values_address2[4096];
```

```
	int64_t Values_data1[4096];
	int64_t Values_data2[4096];

	int iterations = 0;
	uint64_t conVal;

	while((!conVal))
	{
		conVal=true;

		if(iterations % 2 == 0)
		{
			fetchUnit0(numvertex, (uint64_t)NumNeighbors_HW,
NumNeighbors_address);

			read_memory_0(numvertex, (uint64_t)NumNeighbors_HW,
NumNeighbors_address, NumNeighbors_data);

			fetchUnit1(numvertex,  (uint64_t)Neighborhood_HW , NumNeighbors_data,
Neighborhood_address);

			read_memory_1(numvertex,  (uint64_t)Neighborhood_HW ,
NumNeighbors_data, Neighborhood_address, Neighborhood_data);

			fetchUnit2(numvertex, (uint64_t)&Values_HW_1[0], NumNeighbors_data,
Neighborhood_data, Values_address1, Values_address2);

			read_memory_2(numvertex, (uint64_t)&Values_HW_1[0],
NumNeighbors_data, Values_address1, Values_address2, Values_data1, Values_data2, Values_HW_1);

			processUnit(numvertex, eps, (uint64_t)&Results[0], NumNeighbors_data,
Values_data1, Values_data2, &conVal, Results, Results_address);

			for(int i =0; i < numvertex; i++)
			{
				Values_HW_2[i] = Results[(Results_address[i]-
(uint64_t)&Results[0])/sizeof(int64_t)];

			}

			*solution = 0;

			if(conVal == true)
			{
				break;
			}
		}
		else
		{
			fetchUnit0(numvertex, (uint64_t)NumNeighbors_HW,
NumNeighbors_address);

			read_memory_0(numvertex, (uint64_t)NumNeighbors_HW,
NumNeighbors_address, NumNeighbors_data);
```

```
                                        fetchUnit1(numvertex,  (uint64_t)Neighborhood_HW , NumNeighbors_data,
Neighborhood_address);

                                        read_memory_1(numvertex,  (uint64_t)Neighborhood_HW ,
NumNeighbors_data, Neighborhood_address, Neighborhood_data);

                                        fetchUnit2(numvertex, (uint64_t)&Values_HW_2[0], NumNeighbors_data,
Neighborhood_data, Values_address1, Values_address2);

                                        read_memory_2(numvertex, (uint64_t)&Values_HW_2[0],
NumNeighbors_data, Values_address1, Values_address2, Values_data1, Values_data2, Values_HW_2);

                                        processUnit(numvertex, eps, (uint64_t)&Results[0], NumNeighbors_data,
Values_data1, Values_data2, &conVal, Results, Results_address);

                                        for(int i =0; i < numvertex; i++)
                                        {
                                                Values_HW_1[i] = Results[(Results_address[i]-
(uint64_t)&Results[0])/sizeof(int64_t)];
                                        }
                                        *solution = 1;
                                        if(conVal == true)
                                        {
                                                break;
                                        }
                        }
                        iterations++;
        }
        printf("HW Iterations: %d\n", iterations+1);
}
```

Εικόνα 45: Συνάρτηση test funtion

Β.3Υλοποίηση 3η

```
void TestFunction(uint64_t numvertex, int64_t eps, uint64_t *solution, uint64_t *NumNeighbors_HW,
uint64_t *Neighborhood_HW, int64_t *Values_HW_1, int64_t *Values_HW_2, int64_t *Results, uint64_t
*Results_address)

uint64_t NumNeighbors_address[4096];
uint64_t NumNeighbors_data[4096];

uint64_t Neighborhood_address[4096];
uint64_t Neighborhood_data[4096];

uint64_t Values_address1[4096];
uint64_t Values_address2[4096];
int64_t Values_data1[4096];
int64_t Values_data2[4096];

int iterations = 0;
uint64_t conVal;

while(1)
```

```
{
if(iterations % 2 == 0)
  {
fetchUnit0(numvertex, (uint64_t)NumNeighbors_HW, NumNeighbors_address);

     read_memory_0(numvertex, (uint64_t)NumNeighbors_HW, NumNeighbors_address,
NumNeighbors_data);

fetchUnit1(numvertex,  (uint64_t)Neighborhood_HW , NumNeighbors_data, Neighborhood_address);

     read_memory_1(numvertex,  (uint64_t)Neighborhood_HW , NumNeighbors_data, Neighborhood_address,
Neighborhood_data);

fetchUnit2(numvertex, (uint64_t)&Values_HW_1[0], NumNeighbors_data, Neighborhood_data,
Values_address1, Values_address2);

     read_memory_2(numvertex, (uint64_t)&Values_HW_1[0], NumNeighbors_data, Values_address1,
Values_address2, Values_data1, Values_data2, Values_HW_1);

processUnit(numvertex, eps, (uint64_t)&Results[0], NumNeighbors_data, Values_data1, Values_data2,
&conVal, Results, Results_address);

for(int i =0; i < numvertex; i++)
     {
        Values_HW_2[i] = Results[(Results_address[i]-(uint64_t)&Results[0])/sizeof(int64_t)];

     }

     *solution = 0;

if(conVal == true)
     {
break;
     }
  }
else
  {
fetchUnit0(numvertex, (uint64_t)NumNeighbors_HW, NumNeighbors_address);

     read_memory_0(numvertex, (uint64_t)NumNeighbors_HW, NumNeighbors_address,
NumNeighbors_data);

fetchUnit1(numvertex,  (uint64_t)Neighborhood_HW , NumNeighbors_data, Neighborhood_address);

     read_memory_1(numvertex,  (uint64_t)Neighborhood_HW , NumNeighbors_data, Neighborhood_address,
Neighborhood_data);

fetchUnit2(numvertex, (uint64_t)&Values_HW_2[0], NumNeighbors_data, Neighborhood_data,
Values_address1, Values_address2);

     read_memory_2(numvertex, (uint64_t)&Values_HW_2[0], NumNeighbors_data, Values_address1,
Values_address2, Values_data1, Values_data2, Values_HW_2);

processUnit(numvertex, eps, (uint64_t)&Results[0], NumNeighbors_data, Values_data1, Values_data2,
&conVal, Results, Results_address);
```

```
for(int i =0; i < numvertex; i++)
      {
         Values_HW_1[i] = Results[(Results_address[i]-(uint64_t)&Results[0])/sizeof(int64_t)];
      }

      *solution = 1;

if(conVal == true)
      {
break;
      }
   }
iterations++;
}
printf("HW Iterations: %d\n", iterations+1);
```

Εικόνα 46: Συνάρτηση test funtion