ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Neuron Simulation Acceleration in FPGAs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Ιωάννη Μαγκανάρη

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής

Ιωάννης Μαγκανάρης

Αθήνα, Νοέμβριος 2018

(this page is left intentionally blank)

Εθνικο Μετσοβιο Πολυτεχνειο

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ
ΣΥΣΤΗΜΑΤΩΝ

Neuron Simulation Acceleration in FPGAs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Ιωάννη Μαγκανάρη

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την Τρίτη 4 Δεκεμβρίου 2018.

.....	.....	.....
Δημήτριος Ι. Σούντρης	Κιαμάλ Ζ. Πεκμεστζή	Γιώργος Ματσόπουλος
Καθηγητής	Καθηγητής	Καθηγητής

Αθήνα, Νοέμβριος 2018

.....
Ιωάννης Μαγκανάρης
Διπλωματούχος Φοιτητής του Εθνικού Μετσόβιου Πολυτεχνείου

Περίληψη

Η Νευροεπιστήμη μελετά τον ανθρώπινο εγκέφαλο, το νευρικό σύστημα και πως αυτό λειτουργεί και οργανώνεται. Αυτή η μελέτη εστιάζεται κυρίως στον ανθρώπινο εγκέφαλο και πως αυτός καθορίζει την συνείδηση και την συμπεριφορά του. Ενώ στο παρελθόν τα σχετικά πειράματα εκτελούνταν σε εργαστήρια με μικρό αριθμό νευρώνων ή τον ίδιο τον εγκέφαλο, σήμερα οι νευροεπιστήμονες χρησιμοποιούν υπολογιστές για να προσομοιώσουν δίκτυα νευρώνων με μεγάλη ακρίβεια, πολυπλοκότητα και μέγεθος. Επιπλέον, αυτά τα πειράματα τους βοηθάνε να παρακολουθούν περισσότερες μεταβλητές και τους παρέχουν την δυνατότητα να οπτικοποιούν μεγάλα δίκτυα, βοηθώντας τους περαιτέρω στην έρευνα τους.

Τα όλο και πιο σύνθετα και μεγάλα δίκτυα νευρώνων που θέλουν να προσομοιώσουν οι νευροεπιστήμονες δημιούργησαν την ανάγκη για επιτάχυνσή τους σε διάφορες πλατφόρμες και αρχιτεκτονικές. Ενώ υπάρχουν διάφοροι γενικοί προσομοιωτές που καλύπτουν τις βασικές ανάγκες των επιστημόνων, οι περισσότεροι από αυτούς δεν είναι βελτιστοποιημένοι για τα σύγχρονα υπολογιστικά συστήματα και επομένως δεν πετυχαίνουν την καλύτερη δυνατή απόδοση με αποτέλεσμα οι προσομοιώσεις να χρειάζονται ώρες ή ακόμα και μέρες για να ολοκληρωθούν, καθυστερώντας την έρευνα.

Η διπλωματική αυτή προσπαθεί να καλύψει την ανάγκη για επιταχυνόμενες προσομοιώσεις χρησιμοποιώντας την πλατφόρμα Μηχανών Ροής Δεδομένων της Maxeler για να επιταχύνει μια προσομοίωση ενός Προσαρμοστικού Εκθετικού μοντέλου Νευρώνα Συσόρευσης-και-Πυροδότησης σε ένα δίκτυο με Πλαστικότητα εξαρτώμενη από τις χρονικές στιγμές Πυροδότησης των Νευρώνων. Η προσομοίωση αρχικά μεταφράστηκε από τον προσομοιωτή Brian σε ένα πρόγραμμα γραμμένο στη γλώσσα προγραμματισμού C και στη συνέχεια εξελίχθηκε για τις Μηχανές Ροής Δεδομένων. Η πλατφόρμα της Maxeler βασίζεται σε FPGA και χρησιμοποιεί έναν γράφο ροής δεδομένων για να επεξεργαστεί τα δεδομένα, αποσυνδέοντας την λογική από την μνήμη που βρίσκονται τα δεδομένα. Στις Μηχανές Ροής Δεδομένων ο χρόνος των υπολογισμών μετατρέπεται σε υπολογισμούς στον χώρο.

Αυτή η υλοποίηση κατάφερε να επιταχύνει την προσομοίωση μέχρι και 8 φορές σε σχέση με τον Προσομοιωτή Brian και είναι ικανή να προσομοιώσει δίκτυα με πάνω από 20000 νευρώνες, κρατώντας την ίδια λειτουργικότητα των Συνάψεων με τον Brian. Στις μετρήσεις των προσομοιώσεων παρατηρήθηκαν μεγάλες μεταβολές στους λόγους της επιτάχυνσης των Μηχανών Ροών Δεδομένων σε σχέση με την προσομοίωση σε C εξαιτίας της εξαρτώμενης από γεγονότα φύσης της προσομοίωσης και του ντετερμινιστικού χρόνου εκτέλεσης που απαιτούν τα FPGA. Αυτό το γεγονός αποτελεί σημείο ιδιαίτερου ενδιαφέροντος και αναλύεται περισσότερο στην διπλωματική.

Λέξεις Κλειδιά

Maxeler, DFE, Dataflow Programming, Προσομοίωση Νευρώνων, Brian, FPGA, παραλληλοποίηση, Adaptive Exponential Integrate-and-Fire model, STDP, in-silico experiment

Abstract

Neuroscience studies the human brain, the nervous system and how it functions and organizes itself. The main focus of these studies are the human brain and how it defines each person's consciousness and behavior. While in the past most of the experiments were done in labs studying small amounts of neurons or the brain itself, nowadays neuroscientists use computers to simulate neuronal networks in great detail, complexity and size. Those simulations also help them keep track of more variables and grant them the ability to visualize large networks, aiding them further in their research.

The ever more complex and large neuron networks that neuroscientists want to simulate has generated a need to accelerate neuron simulations in different platforms and architectures. While there are a lot of universal simulators that cover neuroscientists' basic needs, most of them are not optimized for modern computer systems and consequently don't achieve the best performance possible, causing the simulations to demand hours or days in execution time, delaying research.

This diploma thesis attempts to appease the need for accelerated simulation by utilizing the Maxeler Data Flow Engine platform to accelerate an Adaptive Exponential Integrate-and-Fire neuron model with Spike-timing Dependent Plasticity which is widely used by neuroscientists. The simulation was firstly imported from Brian Simulator to C programing language and then developed for the DFEs. Maxeler DFE platform is built with FPGAs and uses a dataflow graph to process data, decoupling logic from memory. In the DFEs the computation in time is transformed into a computation in space.

This implementation was able to accelerate neuron simulation up to x8 times in comparison to the Brian Simulator and is able to simulate networks of more than 20000 neurons, while keeping the same functionality of synapses with the Brian Simulator. However, there was observed a variation in the acceleration rates of the DFEs in comparison to the C and Brian Simulator due to the event-driven architecture of the simulation and the deterministic runtime of the FPGAs. This fact constitutes a point of interest and is investigated further in this diploma thesis.

Keywords

Maxeler, DFE, Dataflow Programming, Neuron Simulation, Brian, FPGA, parallelization, Adaptive Exponential Integrate-and-Fire model, STDP, in-silico experiment

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Σούντρη για όλες τις γνώσεις και εμπειρίες που μου μετέφερε κατά τη διάρκεια των σπουδών μου και της εκπόνησης της διπλωματικής μου εργασίας.

Επίσης ευχαριστώ ιδιαίτερα τον μεταδιδακτορικό φοιτητή κ. Σιδηρόπουλο και τον διδακτορικό φοιτητή κ. Χατζηκωνσταντή για την καθοδήγηση και την στενή συνεργασία μας καθ' όλη τη διάρκεια της εργασίας μου στην διπλωματική καθώς και τον κ. Σμάραγδο, την κα. Κολιογεώργη και τον κ. Ζερβάκη για την πολύτιμη βοήθεια τους.

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

Εισαγωγή

Η διπλωματική εργασία αυτή προέκυψε από το ενδιαφέρον μου για την Νευροεπιστήμη και την Επιστήμη των Υπολογιστών. Οι δύο αυτές επιστήμες καταφέρνουν να συνδυαστούν στον τομέα της Υπολογιστικής Νευροεπιστήμης, ο οποίος τα τελευταία χρόνια συγχέεται ίσως πολλές φορές με αυτόν την Νευροεπιστήμης. Πολλοί νευροεπιστήμονες, πέρα από τα πειράματα με φυσικούς νευρώνες in-vitro ή in-vivo χρησιμοποιούν πειράματα in-silico. Αυτό σημαίνει ότι χρησιμοποιούν τους υπολογιστές για να αναπαραστήσουν πειράματα τα οποία λόγω πολυπλοκότητας και της ιδιαίτερης φύσης τους δεν μπορούν να γίνουν στην πραγματικότητα. Τα συνεχόμενα και πιο πολύπλοκα αυτά πειράματα σε συνδυασμό με την πολύπλοκη δομή του νευρικού συστήματος συνιστούν ιδιαίτερα απαιτητικά πειράματα από άποψη υπολογιστικών πόρων. Σε αυτό τον τομέα προσπαθεί να βοηθήσει την έρευνα των νευροεπιστημόνων αυτή η διπλωματική.

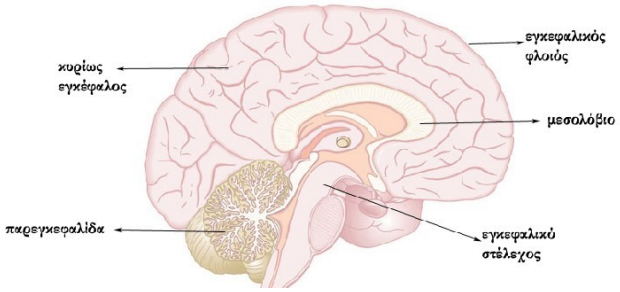
Μέσω της χρήσης της πλατφόρμας της Maxeler γίνεται προσπάθεια επιτάχυνσης μιας προσομοίωσης νευρώνων ενός Προσαρμοστικού Εκθετικού Συσσώρευσης-και-Πυροδότησης (Adaptive Exponential Integrate-and-fire, εν συντομία AdEx) μοντέλου σε ένα δίκτυο με Πλαστικότητα εξαρτώμενη από τις στιγμές πυροδότησης των Νευρώνων (Spike-timing Dependent Plasticity). Το πρώτο κομμάτι της διπλωματικής αφιερώνεται στην παραγωγή μιας προσομοίωσης στη γλώσσα προγραμματισμού C με βάση μιας προσομοιωσης αυτού του μοντέλου που έχει γίνει στον προσομοιωτή Brian. Στη συνέχεια, αφού έχει παραχθεί η προσομοίωση σε C, γίνεται επιτάχυνση της μέσω των Μηχανών Ροής Δεδομένων (DataFlow Engines) της Maxeler, οι οποίες είναι υλοποιημένες με FPGAs.

Νευροεπιστήμη

Ένας μέσος εγκέφαλος ενηλίκου ζυγίζει περίπου 1.4 κιλά, που ισοδυναμεί με το 2% του βάρους του ατόμου. Ωστόσο, καταναλώνει το 20% της συνολικής ενέργειας που καταναλώνει το σώμα του σε μια μέρα χαλάρωσης. Επιπλέον είναι το πιο πολύπλοκο όργανο του νευρικού συστήματος του ανθρώπου με 100 δισεκατομμύρια νευρικά κύτταρα και 100 τρισεκατομμύρια νευρικές συνάψεις, τα οποία έχουν εξελιχθεί στην σημερινή τους κατάσταση από την αρχή της ζωής. Η Νευροεπιστήμη είναι μια επιστήμη η οποία συνδυάζει πολλές άλλες επιστήμες όπως τομείς της Βιολογίας όπως η φυσιολογία, η ανατομία, η μοριακή και εξελικτική βιολογία και η κυτταρολογία, την Στατιστική, τα μαθηματικά και τομείς της Ιατρικής όπως η ψυχολογία. Η Νευροεπιστήμη μελετά τον ανθρώπινο νευρικό σύστημα, πως δουλεύει, πως είναι οργανωμένο και πως αναπτύσσεται. Οι βασικοί στόχοι της Νευροεπιστήμης είναι:

- Η κατανόηση του ανθρώπινου εγκεφάλου και του πως δουλεύει
- Η κατανόηση και η περιγραφή του πως το κεντρικό νευρικό σύστημα αναπτύσσεται, ωριμάζει και διατηρείται
- Η ανάλυση και η κατανόηση των νευρικών και ψυχολογικών διαταραχών καθώς και η εύρεση τρόπων αποτροπής και θεραπείας

Όλα τα χρόνια ερευνών για τον ανθρώπινο εγκέφαλο έχουν καταλήξει στο ότι ο εγκέφαλος είναι το κεντρικό όργανο του ανθρώπινου νευρικού συστήματος και μαζί με την σπονδυλική στήλη αποτελούν το κεντρικό νευρικό σύστημα. Ο εγκέφαλος θα μπορούσε να παραλληλιστεί με μια Κεντρική Μονάδα Επεξεργασίας σε έναν υπολογιστή. Μαζεύει όλες τις πληροφορίες από τις αισθήσεις και τα νεύρα από όλο το σώμα, τις επεξεργάζεται και παίρνει αποφάσεις οι οποίες στέλνονται στο απαιτούμενο μέρος του σώματος. Οι αποφάσεις αυτές ανάλογα με το τι έχουν να κάνουν παίρνονται και σε ξεχωριστό μέρος του εγκεφάλου. Έτσι, ο εγκέφαλος είναι χωρισμένος σε διαφορετικές περιοχές, τον κύριο εγκέφαλο, το εγκεφαλικό στέλεχος και την παρεγκεφαλίδα.



Σχήμα 1: Οργάνωση Ανρθώπινου Εγκεφάλου

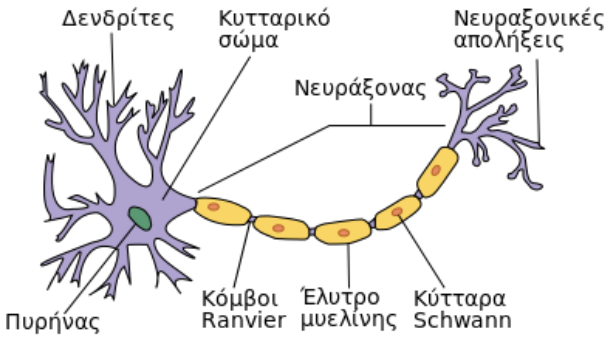
Ο κύριος εγκέφαλος είναι το μεγαλύτερο κομμάτι του ανθρώπινου εγκεφάλου. Ο εγκεφαλικός φλοιός καλύπτει το μεγαλύτερο κομμάτι του κυρίως εγκεφάλου, είναι το πιο εξωτερικό κομμάτι του εγκεφάλου και έχει μια εξωτερική κάλυψη από γκρίζα ύλη που αποτελείται από νευρωνικό ιστό και περιέχει τα σώματα νευρωνικών κυττάρων. Επίσης χωρίζεται σε αριστερό και δεξί εγκεφαλικά ημισφαίρια που ενώνονται με το μεσολόβιο, το οποίο είναι μια πυκνή δέσμη νευραξόνων που επιτρέπει την ανταλλαγή πληροφοριών μεταξύ των δύο ημισφαιρίων. Τα δύο ημισφαίρια διαιρούνται περαιτέρω σε μετωπιαίο, βρεγματικό, κροταφικό και ινιακό λοβό. Κάθε μια από αυτές τις περιοχές του εγκεφάλου επικεντρώνεται σε μια συγκεκριμένη λειτουργία.

Όλα τα μέρη του εγκεφάλου αποτελούνται από δύο βασικά κύτταρα. Τους νευρώνες και τα νευρογλοιακά κύτταρα. Τα δεύτερα είναι μη-νευρικά κύτταρα με ποικίλα σχήματα τα οποία συμβάλλουν στην διατήρηση της ομοιόστασης και παρέχουν στήριξη και προστασία στους νευρώνες του εγκεφάλου.

Νευρώνας

Ο νευρώνας αποτελεί το δομικό κύτταρο και τη λειτουργική μονάδα του νευρικού συστήματος. Οι νευρώνες χαρακτηρίζονται από το κυτταρικό σώμα που περιέχει τον πυρήνα και ένα

μεγάλο αριθμό οργανιδίων. Πέρα από αυτόν έχουν διάφορους τύπους κλαδιών ή και καθόλου, τα οποία χρησιμοποιούνται σαν είσοδος ή έξοδος στο κύτταρο. Τα κλαδιά τα οποία χρησιμοποιούνται για είσοδο ονομάζονται δενδρίτες. Στους περισσότερους νευρώνες ολόκληρο το σώμα τους είναι μέρος υποδοχής. Τα κλαδιά τα οποία χρησιμοποιούνται σαν έξοδος ονομάζονται άξονες και προεξέχουν από το σώμα ή κάποιον δενδρίτη. Τα σήματα που στέλνουν συνήθως εξέρχονται από την άκρη του άξονα.

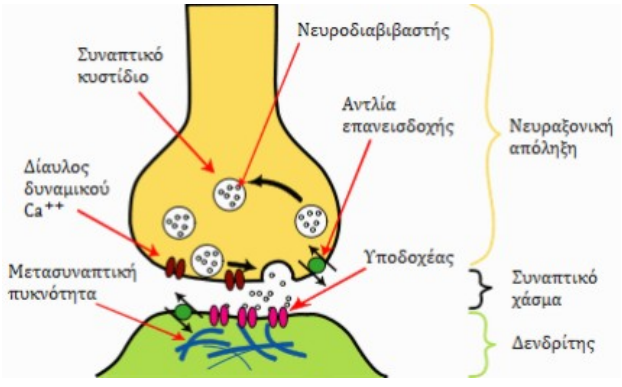


Σχήμα 2: Νευρώνας

Οι νευρώνες έχουν μια μεγάλη ποικιλία ηλεκτροφυσιολογικών ιδιοτήτων που προστίθενται στο μεγάλο σετ από ηλεκτρικές ιδιότητες και τα είδη λειτουργίας τους. Η βασική λειτουργία των νευρώνων είναι η συγκέντρωση νευροδιαβιβαστών και η παραγωγή ηλεκτρικών παλμών. Πέρα από αυτό ελάχιστες ιδιότητες έχουν προστεθεί, όπως η Πλαστικότητα που εκφράζεται σαν Μακροπρόθεσμη Παράταση νευρικών παλμών και την Μακροπρόθεσμη Κατάπτωση νευρικών παλμών.

Συνάψεις

Οι νευρώνες επικοινωνούν μεταξύ τους μέσω των Συνάψεων που τους συνδέουν και μεταφέρουν ηλεκτρικά ή χημικά σήματα στο νευρώνα στόχο ή σε κάποιο άλλο κύτταρο-στόχο. Σε μια σύναψη, η πλασματική μεμβράνη του πηγαίου νευρώνα (προσυναπτικός) έρχεται σε κοντινή παράθεση με την μεμβράνη του στοχευμένου κυττάρου (μετασυναπτικού).



Σχήμα 3: Σύναψη

Οι συνάψεις χωρίζονται σε δύο κατηγορίες. Στις χημικές, όπου η ηλεκτρική δραστηριότητα

στον προσυναπτικό νευρώνα μετατρέπεται μέσω των ελεγχόμενων από δυναμικό πυλών καλίου σε απελευθέρωση νευροδιαβιβαστών που συνδέονται με τους δέκτες που βρίσκονται στην μεμβράνη του μετασυναπτικού κυττάρου και στις ηλεκτρικές, όπου οι προ- και μετα συναπτικές μεμβράνες συνδέονται με ειδικά κανάλια που ονομάζονται χασματοσυνδέσεις και είναι ικανές να μεταδώσουν ένα ηλεκτρικό ρεύμα από το προ- στο μετά- συναπτικό κύτταρο.

Βαθμίδες των Λεπτομερειών Μοντελοποίησης Νευρώνων

Τα 3 βασικά μοντέλα για την αναπαράσταση νευρώνων είναι:

- Τα μοντέλα με βάση την διαγωγιμότητα: Περιέχουν μεγάλο βαθμό λεπτομερειών καθώς προσομοιώνουν την δομή των νευρώνων μέσω πολλαπλών, διασυνδεδεμένων τμημάτων που το καθένα συμπεριφέρεται σαν να είναι ηλεκτρικά συμπαγές
- Τα μοντέλα Συσσώρευσης-και-Πυροδότησης: Πρόκειται για πιο αφηρημένο μοντέλο νευρώνων από το πρώτο. Για να πυροδοτηθούν πρέπει το δυναμικό το οποίο υπολογίζεται μέσω ενός συμβολικού μοντέλου να είναι μεγαλύτερο από το κατώφλι τους. Απλοποιούν ριζικά την γεωμετρία των κυττάρων
- Τα μοντέλα Συχνότητας Εκκένωσης: Πρόκειται για το πιο αφηρημένο μοντέλο, έχοντας μια συνεχής τιμής, μεταβαλλόμενης με τον χρόνο τιμής για την Συχνότητα Εκκένωσης της εξόδου των νευρώνων

Προγραμματιζόμενες Συστοιχίες Πυλών σε Πεδίο (FPGA)

Τα FPGA είναι ολοκληρωμένα κυκλώματα στα οποία οι λογικές πύλες και οι συνδέσεις μεταξύ τους προγραμματίζονται από τον χρήστη. Για να γίνει αυτό ο προγραμματιστής χρησιμοποιεί μια Γλώσσα Περιγραφής Υλικού (Hardware Description Language, HDL) η οποία μεταφράζεται στο υλικό μέσω των μεταγλωττιστών που παρέχουν οι εταιρίας παραγωγής των FPGA. Μέσω της περιγραφής που έχει δώσει ο χρήστης με μια γλώσσα HDL, είναι δυνατόν να ορίσει ακριβώς τον αλγόριθμο που θέλει να υλοποιήσει, να δημιουργήσει ένα σχέδιο σωλήνωσης και να ελέγξει με ακρίβεια την χρονοδρομολόγηση του συστήματος μιας και πρόκειται για σύγχρονα κυκλώματα. Τα FPGA αποτελούνται από πολλά Λογικά Τμήματα υλικού, τα οποία αποτελούνται από Look-Up Tables, λογικές πύλες και Καταχωρητές, Σκληρά Τμήματα υλικού που είναι σχεδιασμένα για να επιτελούν συγκεκριμένες πράξεις και ένα βασικό ρολόι ή και παραπάνω περιφερειακά ρολόγια. Το περιορισμένο υλικό που υπάρχει σε ένα ολοκληρωμένο τσιπ περιορίζει τα σχέδια που μπορούν να υλοποιηθούν σε αυτό και για αυτό προτείνονται συνδεσμολογίες που ενώνουν πολλά τσιπ μαζί για την δημιουργία ενός εικονικά ολοκληρωμένου κυκλώματος. Πέρα από αυτό τον περιορισμό, χρειάζεται γνώση της ειδικής γλώσσας προγραμματισμού για να γραφτεί κάποιο πρόγραμμα σε αυτή την πλατφόρμα αλλά και αρκετή ώρα για να γίνει η μεταγλώττιση ενός προγράμματος για FPGA. Σήμερα υπάρχουν αρκετές προτάσεις για την βελτίωση αυτών των θεμάτων καθώς τα FPGA λόγω της μεγάλης απόδοσης τους, της ασφάλειάς τους και της χαμηλής ενέργειας που καταναλώνουν έχουν πληθώρα εφαρμογών στο διάστημα, σε αυτοκίνητα, σαν επιταχυντές αλλά και πλατφόρμες πρωτοτυποποίησης.

Maxeler

Η διπλωματική αυτή έχει βασιστεί στην πλατφόρμα των Μηχανών Γράφων Δεδομένων της Maxeler. Μια Μηχανή Ροής Δεδομένων υλοποιείται με την χρήση FPGA. Το μοντέλο

προγραμματισμού που υλοποιεί αυτή η πλατφόρμα απαιτεί τον διαχωρισμό της λογικής από τα δεδομένα. Μια Μηχανή Ροής Δεδομένων ουσιαστικά δημιουργεί έναν Γράφο Δεδομένων, ο οποίος τροφοδοτείται με δεδομένα από την μνήμη στην οποία είναι αποθηκευμένα τα δεδομένα σε κάθε 'τικ' του ρολογιού της πλατφόρμας και παράγει τα αντίστοιχα δεδομένα στην έξοδο σε κάθε 'τικ'. Μέσα σε αυτόν τον γράφω επίσης δημιουργείται αυτόματα από τον μεταφραστή της πλατφόρμας μια σωλήνωση ώστε να αυξάνεται ο ρυθμός παραγωγής χρήσιμης εξόδου από τον γράφο. Οι μνήμες τις οποίες χρησιμοποιεί μια Μηχανή Ροής Δεδομένων είναι μια Μεγάλη Μνήμη (LMEM) η οποία αποτελείται από DDR3 Ram Dimms τα οποία είναι συνδεδεμένα πολύ κοντά με το ολοκληρωμένο FPGA και μια μικρότερη (FMEM) η οποία ουσιαστικά είναι μέσα στο FPGA και είναι τα BRAMs τα οποία υπάρχουν μέσα σε αυτό. Η πλατφόρμα αυτή βοηθάει πολύ τον προγραμματισμό λόγω των γλωσσών υψηλού επιπέδου που χρησιμοποιούνται, αποκρύπτοντας έτσι τις δυσκολίες και λεπτομέρειες προγραμματισμού των FPGA, ενώ μπορούν ακόμα να αξιοποιηθούν τα πλεονεκτήματα των FPGA όπως ο μεγάλος ρυθμός εξόδου χρήσιμων δεδομένων αλλά και η μεγάλη παραλληλοποίηση. Το προγραμματιστικό μοντέλο της πλατφόρμας αποτελείται από ένα κομμάτι κώδικα που τρέχει στον κλασικό επεξεργαστή της πλατφόρμας και είναι υπεύθυνο να καλεί τον Πυρήνα και να αρχικοποιεί την Μηχανή Ροής Δεδομένων, από τον Πυρήνα που αποτελεί το τμήμα του κώδικα που περιγράφει τον σχεδιασμό της Μηχανής Ροής Δεδομένων και έναν Μαναγερ που είναι υπεύθυνος για την μεταγλώττιση και το σετάρισμα της Μηχανής Ροής Δεδομένων και των περιφερειακών της όπως διάφορες ρυθμίσεις της και τον καθορισμό των ροών δεδομένων της από και τις μνήμες ή τον επεξεργαστή.

Σχετικό Έργο στην Επιταχυνόμενη Υπολογιστική Νευροεπιστήμη

Η Υπολογιστική Νευροεπιστήμη είναι ένας κλάδος της Νευροεπιστήμης που χρησιμοποιεί μαθηματικά μοντέλα, ανάλυση θεωρημάτων και αφαιρέσεις του εγκεφάλου για να κατανοήσει τις αρχές που διέπουν την εξέλιξη, την δομή, την φυσιολογία και τις διαισθητικές ικανότητες του νευρικού συστήματος. Όπως αναφέρθηκε και προηγουμένως, οι Νευροεπιστήμονες έχουν αντικαταστήσει πολλά in-vivo και in-vitro πειράματα με in-silico πειράματα, δηλαδή με προσομοιώσεις κομματιών του εγκεφάλου σε υπολογιστές. Τα κομμάτια αυτά μπορεί να είναι από μικρά δίκτυα νευρώνων μέχρι ολόκληρες περιοχές του εγκεφάλου με τρισδιάστατη μορφολογία. Για αυτές τις προσομοιώσεις χρησιμοποιούνται κυρίως κάποια ευρέως δεδομένα πλαίσια προσομοιώσεων όπως το Neuron, το Nest, το Brian, το Moose και το Genesis. Για την προσομοίωση μικρών δικτύων και την δοκιμή διαφόρων μοντέλων χρησιμοποιείται επίσης κατά κόρον και το Matlab. Τα πλαίσια αυτά έχουν το καλό ότι είναι εύκολα στη χρήση και υπάρχει μεγάλη βιβλιογραφία βασισμένη σε αυτά, ωστόσο δεν εκμεταλλεύονται τις δυνατότητες των σύγχρονων υπολογιστικών συστημάτων όπως οι πολλαπλοί επεξεργαστές, τα πολλαπλά Νήματα, οι Κάρτες Επεξεργασίας Γραφικών Γενικού Σκοπού, τα FPGA, υπερυπολογιστές και δομές στο Σύννεφο.

Ειδική αναφορά αξίζει στο Brainframe, από το οποίο προέκυψε αυτή η διπλωματική. Το

Brainframe είναι μια ονλινε πλατφόρμα επιτάχυνσης η οποία τρέχει τα πειράματα σε ετερογενή συστήματα που αποτελούνται από Intel Xeon-Phi CPUs, NVidia GP-GPUs και Maxeler Μηχανές Ροής Δεδομένων προσφέροντας όχι μόνο επιτάχυνση αλλά και μικρότερη κατανάλωση ενέργειας. Το μεγάλο πλεονέκτημα της είναι η ευκολία χρήσης καθώς είναι προσαρμοσμένη στο πλαίσιο της PyNN και χρησιμοποιεί την γλώσσα προγραμματισμού Python, τα οποία είναι ευρέως διαδεδομένα και γνωστά στους νευροεπιστήμονες. [GS17; APDY09]

Περιγραφή προβλήματος

Η προσομοίωση της διπλωματικής εργασίας βασίζεται σε ένα Προσαρμοστικό Εκθετικό Μοντέλο Συσσώρευσης-και-Πυροδότησης (Adaptive Exponential Integrate-and-Fire, AdEx) σε ένα δίκτυο με Πλαστικότητα εξαρτώμενη από τις στιγμές πυροδότησης των Νευρώνων (Spike-timing Dependent Plasticity). Για να υλοποιηθεί αυτή η προσομοίωση στην πλατφόρμα της Maxeler χρειάστηκε πρώτα η μετάφραση της από Python που χρησιμοποιεί ο Προσομοιωτής Brian σε ένα πρόγραμμα C. Στη συνέχεια, αυτό το πρόγραμμα χρειάστηκε να αναπτυχθεί χρησιμοποιώντας την αρχιτεκτονική και την Διεπαφή Προγραμματισμού Εφαρμογών της Maxeler.

Το Προσαρμοστικό Εκθετικό Μοντέλο Συσσώρευσης-και-Πυροδότησης μοντέλο νευρώνα είναι ένα μοντέλο νευρώνα Πυροδότησης με δύο μεταβλητές. Η πρώτη εξίσωση περιγράφει την δυναμική της πιθανότητας της μεμβράνης του νευρώνα και περιέχει έναν όρο ενεργοποίησης του νευρώνα. Το δυναμικό εισέρχεται σε μια δεύτερη εξίσωση που περιγράφει την προσαρμοστικότητα. Οι ακριβείς διαφορικές εξισώσεις μπορούν να βρεθούν στο αγγλικό κομμάτι της διπλωματικής.

Η Πλαστικότητα εξαρτώμενη από τις στιγμές πυροδότησης των Νευρώνων είναι μια μη συμμετρική μορφή της μάθησης του Hebbian που εξαρτάται από τον χρόνο και και ιδιαίτερα με τις χρονικές στιγμές μεταξύ της πυροδότησης των προ- και μετα-συναπτικών νευρώνων. Θεωρείται πως μαζί με άλλες μορφές Πλαστικότητας, ευθύνεται για την μάθηση και την αποθήκευση πληροφοριών στον εγκέφαλο καθώς και την ανάπτυξη και την βελτίωση των νευρικών κυκλωμάτων κατά την ανάπτυξη του εγκεφάλου. Στη συγκεκριμένη προσομοίωση ασχολούμαστε και με τις δύο μορφές έκφρασης της, δηλαδή σε σχέση με τον προ- και τον μετα-συναπτικό νευρώνα. Οι ακριβείς διαφορικές της εξισώσεις φαίνονται στο αγγλικό κομμάτι του κειμένου.

Προσομοιωτής Brian και μετατροπή της προσομοίωσης σε C

Ο Brian είναι ένας προσομοιωτής νευρώνων παλμών. Στόχος του προσομοιωτή αυτού είναι να αποκρύψει τις λεπτομέρειες της υλοποίησης της λύσης της προσομοίωσης από τους νευροεπιστήμονες, ώστε να μπορούν να αφοσιωθούν μόνο στα μοντέλα των νευρώνων που χρησιμοποιούν. Για αυτό τον λόγο χρησιμοποιεί την γλώσσα προγραμματισμού Πψτηον και υποστηρίζει την διεπαφή της PyNN που χρησιμοποιείται ευρέως για τον καθορισμό των μοντέλων.

Η μετατροπή της δοσμένης προσομοίωσης από Brian σε C έγινε σε συνεργασία με έναν συμφοιτητή στα πλαίσια της διπλωματικής του εργασίας επίσης. Για την αναπαράσταση του νευρικού δικτύου σύμφωνα με τον Brian χρειάστηκε ο ορισμός τριών δομών δεδομένων οι

οποίοι αναπαριστώνται στο Brian. Αυτές είναι οι Νευρώνες Εισόδου, οι οποίοι δεν διέπονται από κάποιο μοντέλο παρά μόνο δημιουργούν παλμούς σύμφωνα με μια γεννήτρια συνάρτηση, οι πραγματικοί AdEx νευρώνες και οι Συνάψεις. Οι Νευρώνες Εισόδου στην C αναπαριστώνται με έναν πίνακα από ακέραιους αριθμούς. Για κάθε Νευρώνα Εισόδου υπάρχει μια μεταβλητή στον πίνακα αυτόν που αν είναι 1 τότε σημαίνει ότι στο συγκεκριμένο βήμα της προσομοίωσης παράχθηκε από αυτόν τον νευρώνα ένας παλμός και αν είναι 0 δεν παράχθηκε. Οι AdEx νευρώνες αναπαριστώνται από έναν πίνακα με μιας ειδικής δομής δεδομένων που ονομάσαμε Νευρον και περιέχει όλες τις μεταβλητές κάθε νευρώνα. Οι Συνάψεις αναπαριστώνται από έναν δισδιάστατο πίνακα γειτνίασης, όπου κάθε στοιχείο του αναφέρεται στη σύναξη μεταξύ δύο νευρώνων. Οι νευρώνες που θεωρούνται σαν παραγωγοί αντιστοιχούν στις γραμμές του πίνακα ενώ οι νευρώνες που θεωρούνται στόχοι αντιστοιχούν στις στήλες του πίνακα. Η προσομοίωση σε C είναι ικανή να προσομοιώσει δύο τύπους προσομοιώσεων ανάλογα με την συνδεσμολογία των AdEx νευρώνων. Ο πρώτος τύπος είναι η NxM προσομοίωση όπου Νευρώνες Εισόδου απλά συνδέονται με Νευρώνες AdEx σε ένα νευρικό δίκτυο δύο επιπέδων. Ο δεύτερος τύπος είναι η MxM προσομοίωση, όπου ένα σύνολο από AdEx νευρώνες συνδέεται με διάφορους τρόπους με άλλους AdEx νευρώνες του ίδιου συνόλου, σχηματίζοντας ένα δίκτυο ενός επιπέδου με διασυνδέσεις.

Σε κάθε βήμα της προσομοίωσης χρειάζεται να εκτελεστούν 4 βασικές λειτουργίες με την σειρά με την οποία θα αναφερθούν, λόγω μεταξύ τους εξαρτήσεις. Η πρώτη είναι η συνάρτηση SolveNeurons, η οποία ενημερώνει τις μεταβλητές των AdEx Νευρώνων. Η δεύτερη είναι η συνάρτηση InitializeSpikeArray που υπολογίζει ποιοί από τους Νευρώνες Εισόδου χρειάζεται να παράγουν έναν παλμό στο συγκεκριμένο βήμα της προσομοίωσης. Η τρίτη είναι η συνάρτηση UpdateSynapses_pre η οποία ενημερώνει τις τιμές των Συνάψεων ανάλογα με την δραστηριότητα των νευρώνων που είναι προσυναπτικοί (Εισόδου ή AdEx). Η τελευταία συνάρτηση είναι η UpdateSynapses_post η οποία ενημερώνει τις τιμές των Συνάψεων ανάλογα με την δραστηριότητα των μετασυναπτικών νευρώνων. Αυτές οι συναρτήσεις, στην πραγματικότητα αυτό που κάνουν είναι να επιλύουν τις διαφορικές εξισώσεις των μοντέλων των Νευρώνων και των Συνάψεων.

Υλοποίηση σε DFE

Αφού έγινε η υλοποίηση σε C της προσομοίωσης ακολουθεί η προσπάθεια επιτάχυνσης της σε Maxeler DFE. Για να γίνει αυτό, το πρώτο πράγμα που πρέπει να αλλάξει είναι η αναπαράσταση των δομών δεδομένων, ώστε το διάβασμα τους να γίνεται με αποδοτικό τρόπο από την DFE. Αρχικά ο πίνακας των AdEx νευρώνων μετατρέπεται σε έναν μονοδιάστατο πίνακα από μεταβλητές κινητής υποδιαστολής όπου κάθε 6 στοιχεία του αντιπροσωπεύουν έναν νευρώνα (5 πραγματικές μεταβλητές, 1 για παδδινγ). Με αυτόν τον τρόπο σε κάθε 'τικ' της DFE όταν χρειάζεται να διαβαστεί έναν νευρώνας, ουσιαστικά διαβάζονται 6 στοιχεία του πίνακα των AdEx νευρώνων. Κάτι αντίστοιχο γίνεται και με τον πίνακα γειτνίασης των συνάψεων. Κάθε σύναψη αναπαριστάνεται από 12 μεταβλητές κινητής υποδιαστολής (11 πραγματικές, 1 για padding). Επίσης γίνεται αντιστροφή των στηλών με τις γραμμές του και έτσι οι γραμμές

αντιστοιχούν στους νευρώνες-στόχους ενώ οι στήλες στους νευρώνες-πηγές. Αυτός ο πίνακας είναι επίσης μονοδιάστατος αλλά διευθυτοδοτείται σαν δισδιάστατος ώστε να κρατήσει την φυσική του υπόσταση. Ο πίνακας των Παλμών Εισόδου μετατρέπεται σε έναν πίνακα από μεταβλητές 8 bit(1 byte) για εξοικονόμηση χώρου στην μνήμη. Όλοι αυτοί οι πίνακες λόγω του μεγάλου τους μεγέθους σώζονται στην Μεγάλη Μνήμη των DFE που έχει μέγεθος 48 ΓΒ. Μετά από υπολογισμούς σχετικούς με αυτή, για την προσομοίωση όπου χρησιμοποιούνται μεταβλητές κινητής διαστολής μονής ακριβείας ο μέγιστος αριθμός νευρώνων είναι 20352.

Αρχιτεκτονική Πυρήνα

Για την εύρεση της πιο αποδοτικής αρχιτεκτονικής για την επίλυση της προσομοίωσης δοκιμάστηκαν πολλές εκδόσεις και διαφορετικές λύσεις ώστε να υπάρξει και εξοικείωση με την πλατφόρμα της Maxeler. Οι διαφορετικές εκδόσεις περιλάμβαναν την φόρτωση διαφορετικών πυρήνων για την λύση της προσομοίωσης αντί για ενός, χρήση διαφορετικών τύπων δεδομένων και παραλληλοποίηση της προσομοίωσης. Αυτό που γίνεται σε όλους τους πυρήνες είναι ότι τα δεδομένα των Νευρώνων και των Συνάψεων διαβάζονται από την Μεγάλη Μνήμη της DFE και οι ανανεωμένες τιμές τους γράφονται στις ίδιες θέσεις.

Πολλαπλοί Πυρήνες

Η υλοποίηση με τη χρήση πολλαπλών Πυρήνων έγινε σε αντιστοιχία με την C. Κάθε διακριτή συνάρτηση του κώδικα σε C μετατράπηκε σε πυρήνα με στόχο να φορτώνεται σε κάθε βήμα της προσομοίωσης η DFE με όλους τους πυρήνες με την ίδια σειρά που καλούνται στην C. Για όλους τους πυρήνες χρησιμοποιήθηκαν αριθμοί κινητής υποδιαστολής διπλής ακρίβειας.

Ο πυρήνας της SolveNeurons είναι υπεύθυνος για την ενημέρωση των AdEx νευρώνων. Συγκεκριμένα σε κάθε 'τικ' του DFE διαβάζεται ένας νευρώνας από την Μεγάλη Μνήμη (LMem) και ενημερώνονται οι τιμές του παράγοντας έξοδο σε κάθε 'τικ'. Η βελτιστοποίηση αυτού του πυρήνα έχει να κάνει με την παραλληλοποίηση σε επίπεδο νευρώνων. Αυτό σημαίνει ότι σε κάθε 'τικ' επιλύονται 2 νευρώνες ταυτόχρονα, μειώνοντας τα απαιτούμενα 'τικς' στο μισό. Φυσικά και χωρίς την παραλληλοποίηση σε επίπεδο νευρώνων υπάρχει παραλληλοποίηση των πράξεων στο υλικό αφού διαφορετικές πράξεις γίνονται ταυτόχρονα. Για τον πυρήνα αυτό απαιτούνταν από τον επεξεργαστή να γράψει στη μνήμη τον πίνακα των νευρώνων και να περάσει στον πυρήνα τις παραμέτρους των διαφορικών εξισώσεων των νευρώνων.

Η συνάρτηση UpdateSynapses_pre είναι υπεύθυνη για την επίλυση των διαφορικών εξισώσεων των συνάψεων με βάση την δραστηριότητα των προσυναπτικών νευρώνων. Σε αυτόν τον πυρήνα διαβάζεται αρχικά ένας νευρώνας-στόχος και στη συνέχεια διαβάζονται οι συνάψεις που αντιστοιχούν σε αυτόν και για κάθε μια από αυτές διαβάζεται και ο προσυναπτικός νευρώνας. Αν κάποιος από τους προσυναπτικούς νευρώνες έχει παράξει παλμό, τότε αυτή η σύναψη ενημερώνεται βάση των διαφορικών εξισώσεων που διέπουν την συμπεριφορά της. Η βελτιστοποίηση σε αυτόν τον πυρήνα έχει να κάνει με την αποθήκευση των νευρώνων που διαβάζονται πολλές φορές στην γρήγορη μικρή μνήμη του FPGA. Η παραλληλοποίηση δεν ήταν δυνατή αρχικά λόγω μη επιλυμένων εξαρτήσεων μεταξύ δεδομένων. Από τον επεξεργαστή απαιτείται να γράψει στην LMem τους πίνακες των AdEx νευρώνων, τον πίνακα των Συνάψεων και τον

πίνακα των Νευρώνων Εισόδου.

Η συνάρτηση UpdateSynapses_post χρειάστηκε περισσότερες δοκιμές και εκδόσεις για να βρεθεί η πιο αποδοτική εξαιτίας του ότι έπρεπε να περαστεί δύο φορές ο πίνακας των συνάψεων. Στο πρώτο πέρασμα γινόταν η επίλυση κάποιων διαφορικών και ο υπολογισμός ενός μέσου όρου βάσει των τιμών των συνάψεων που ενημερωνόντουσαν και στο δεύτερο πέρασμα επιλυόντουσαν διαφορικές εξισώσεις βάσει αυτού του μέσου όρου. Για να γίνουν αυτά τα δύο περάσματα δοκιμάστηκαν τρεις εκδόσεις. Η πρώτη έκδοση ήταν η φόρτωση από τον επεξεργαστή δύο διαφορετικών πυρήνων μέσω του κώδικα του επεξεργαστή και η δεύτερη μέσω του Manager της ΔΦΕ. Η τελευταία και πιο αποδοτική έκδοση της συνάρτησης UpdateSypapses_post βασίζεται σε έναν μοναδικό πυρήνα ο οποίος μέσω ρολογιών δημιουργεί μια εσωτερική επανάληψη 2 βημάτων. Στο πρώτο βήμα γίνεται το πρώτο πέρασμα των συνάψεων, υπολογίζονται όσες μεταβλητές χρειάζονται και στο δεύτερο επιλύονται οι διαφορικές βάσει αυτών των μεταβλητών. Ωστόσο για να γίνει μέσα στον ίδιο πυρήνα το διάβασμα και το γράψιμο στην μνήμη δύο φορές και να γίνουν ορατές οι αλλαγές και τις δύο φορές απαιτήθηκε η δημιουργία Ειδικών Ροών Εντολών για την Μνήμη. Αυτές οι εντολές προερχόντουσαν από τον ίδιο τον πυρήνα και αφορούσαν τις διευθύνσεις που διαβάζονταν και γραφόντουσαν στην μνήμη καθώς και σε ποιές χρονικές στιγμές. Αυτές οι Ειδικές Ροές όμως επέφεραν καθυστερήσεις στο διάβασμα και το γράψιμο στη μνήμη με αποτέλεσμα να χρειαστεί να υπάρξουν περαιτέρω επαναλήψεις μέσα στον πυρήνα αυτόν. Ο τρόπος με τον οποίο ανανεώνονται οι Συνάψεις σε αυτόν τον πυρήνα και στα δύο περάσματα είναι ο εξής. Κάθε κάποια τακτά 'τικς' διαβάζεται ένας νευρώνας-στόχος και ανάλογα με το αν έχει παραχθεί από αυτόν ένας παλμός τότε ενημερώνονται οι τιμές όλων των Συνάψεων με τις οποίες συνδέεται με την σειρά.

Μοναδικός Πυρήνας

Ο μοναδικός πυρήνας ουσιαστικά εκτελεί ότι ακριβώς και οι πολλαπλοί πυρήνες, οι οποίοι συνδυάζονται σε αυτόν με την χρήση μετρητών. Με αυτόν τον τρόπο δημιουργούνται τρία βήματα σε αυτόν τον πυρήνα. Στο πρώτο γίνεται η επίλυση των διαφορικών των Νευρώνων. Το διάβασμα και το γράψιμο στη μνήμη γίνεται πάντα με τη χρήση Ειδικών Ροών Εντολών για την Μνήμη. Λόγω αυτού υπάρχουν καθυστερήσεις και έτσι ενημερώνεται η τιμή ενός νευρώνα κάθε 2 'τικ'. Στο δεύτερο βήμα εκτελείται η συνάρτηση UpdateSynapses_pre μαζί με το πρώτο πέρασμα στον πίνακα των συνάψεων της UpdateSynapses_post. Έτσι για αυτό το βήμα χρειάζεται ένα πέρασμα στον πίνακα των συνάψεων αλλά κάθε σύναψη χρειάζεται 16 'τικς' για να υπολογιστεί, λόγω του υπολογισμού ενός αθροίσματος κινητής υποδιαστολής που γίνεται για τον υπολογισμό του μέσου όρου. Το τρίτο βήμα περιλαμβάνει την εκτέλεση των διαφορικών που απομένουν από την συνάρτηση UpdateSynapses_post και χρειάζονται τον μέσο όρο που υπολογίζεται στο προηγούμενο βήμα. Σε αυτό το βήμα υπολογίζεται μια σύναψη ανά 4 'τικς'. Ο συνολικός αριθμός 'τικς' που χρειάζεται είναι: Number_of_STDP_variables + timesteps*(2*N_Group_T + 16*(N_S+N_Group_S)*N_Group_T + 4*(N_S+N_Group_S)*N_Group_T). Αυτός ο πυρήνας καταλαμβάνει μετά από την μεταλώττιση το 96.14% της λογικής του FPGA και έτσι δεν είναι δυνατή κάποια παραλληλοποίηση σε

επίπεδο νευρώνων ή συνάψεων γιατί αυτό θα σήμαινε την χρήση περισσότερης λογικής για την ταυτόχρονη επίλυση παραπάνω πράξεων.

Μετά από σύγκριση με την έκδοση με τους πολλαπλούς πυρήνες, διαπιστώθηκε ότι για την ίδια προσομοίωση η έκδοση με τους πολλούς πυρήνες χρειάζεται σχεδόν 9.5 δευτερόλεπτα για την φόρτωση και την εκφόρτωση των πυρήνων στην DFE, που χρειάζεται να γίνουν σε κάθε βήμα της προσομοίωσης και έτσι δεν είναι αποδοτική. Επιπλέον ο μοναδικός πυρήνας χρειάστηκε λιγότερο χρόνο για την επίλυση της ίδιας προσομοίωσης, άρα και ο σχεδιασμός του είναι πιο αποδοτικός.

Η βελτιστοποίηση αυτού του πυρήνα βασίζεται στην χρήση αριθμών κινητής υποδιαστολής μονής ακρίβειας, πράγμα το οποίο οδηγεί στην χρήση λιγότερης λογικής για τον ίδιο αλγόριθμο και κατ' επέκταση την παραλληλοποίηση σε επίπεδο Συνάψεων. Αυτή η αλλαγή ωστόσο επιφέρει την ύπαρξη αποκλίσεων στις τιμές που υπολογίζονται, οι οποίες ωστόσο είναι αρκετά μικρές ώστε να θεωρηθούν ασήμαντες. Αρχικά για την χρήση αριθμών κινητής υποδιαστολής μονής ακρίβειας χρειάστηκε απλά να τεθεί ο κατάλληλος τύπος στην αρχή του κώδικα του πυρήνα, ώστε όλες οι μεταβλητές που τίθενται με βάσει αυτό να γίνουν μονής ακρίβειας. Στη συνέχεια για να παραλληλοποιηθούν οι υπολογισμοί των συνάψεων χρειάζεται να διαβάζονται δύο ή παραπάνω συνάψεις μαζί στο δεύτερο και τρίτο βήμα του πυρήνα. Στη συνέχεια πρέπει να λυθεί η εξάρτηση που προκύπτει εξαιτίας του αθροίσματος για τον μέσο όρο. Το πρόβλημα αυτό λύθηκε με την δημιουργία ενός δέντρου αθροιστών (κάνοντας το λεγόμενο ρεδυσινγ) που καταλήγουν στο τελικό άθροισμα. Με αυτόν τον τρόπο στους ίδιους χρόνους για κάθε βήμα με τον πυρήνα με αριθμούς διπλής ακριβείας, γίνονται οι διπλάσιες πράξεις για βαθμό παραλληλοποίησης δύο. Η χρήση της λογικής με χρήση βαθμού παραλληλοποίησης δύο έγινε:
Τελική Χρήση Υλικού
Κατανάλωση Λογικής: 252278 / 262400 (96.14%)
Πρωτεύοντα FFs: 364674 / 524800 (69.49%)
Δευτερεύοντα FFs: 67155 / 524800 (12.80%)
Πολλαπλασιαστές (18ξ18): 776 / 3926 (19.77%)
Κουτιά DSP: 388 / 1963 (19.77%)
Κομμάτια Μνήμης (M20K): 1440 / 2567 (56.10%)

Παρατηρούμε πως υπάρχει διαθέσιμος χώρος και έτσι δοκιμάστηκε και παραλληλοποίηση βαθμού τέσσερα. Ωστόσο αυτή οδήγησε σε χρήση της λογικής στο 114% περίπου και έτσι δεν ήταν εφικτή η σχεδίαση του πυρήνα στο FPGA. Επιπλέον οι συχνές προσβάσεις στη μνήμη είχαν ως αποτέλεσμα την χρήση μεγαλύτερων καθυστερήσεων για κάθε βήμα του πυρήνα και συνεπώς δεν προέκυπτε μείωση στον αριθμό των 'τικς' για τα οποία τρέχει ο πυρήνας.

Τα αποτελέσματα της επιτάχυνσης σε σχέση με τον πυρήνα που χρησιμοποιεί αριθμούς κινητής υποδιαστολής διπλής ακρίβειας φαίνονται στον παρακάτω πίνακα:

Τέλος, για να μπορεί η προσομοίωση να τρέξει για πάνω από μια ώρα σε DFE, χρειάζεται να σπάσει η προσομοίωση σε κομμάτια της μίας ώρας. Έτσι ο κώδικας του επεξεργαστή διαιρεί την προσομοίωση σε βήματα ώστε κάθε παράθυρο βημάτων να τρέχει για λιγότερο από μια ώρα και τελικά να είναι εφικτή η προσομοίωση μεγάλων δικτύων που διαρκεί πάνω από μια ώρα.

Πίνακας 1: Επιτάγχυνση με αριθμούς κινητής υποδιαστολής με μόνη ακρίβεια σε σχέση με διπλή

Πείραμα	1.1	1.2	1.3	2.1	2.2	2.3	3	4
Νευρώνες Εισόδου	384	384	384	384	384	384	0	0
ΑδΕξ Νευρώνες Πηγές	0	0	0	0	0	0	384	1152
ΑδΕξ Νευρώνες Στόχοι	384	384	384	1152	1152	1152	384	1152
Διαστήματα μεταξύ Παλμών Εισόδου (βήματα)	1	2	5	1	2	5	-	-
Χρόνος Επεξεργαστή (s)	2.38	1.24	0.56	7.83	4.56	2.24	4.10	52.21
Χρόνος DFE Μονής Ακρίβειας (s)	2.34	2.34	2.34	6.98	6.98	6.98	2.54	22.70
Επιτάχυνση Μονής Ακρίβειας vs CPU	1.02	0.53	0.24	1.12	0.65	0.32	1.61	2.30
Χρόνος DFE Διπλής Ακρίβειας (s)	3.02	3.02	3.02	9.03	9.02	9.02	3.47	31.07
Επιτάχυνση Διπλής Ακρίβειας vs CPU	0.79	0.41	0.19	0.87	0.50	0.25	1.18	1.68
Επιτάχυνση Μονής vs Διπλής Ακρίβειας	1.29	1.29	1.29	1.29	1.29	1.29	1.37	1.37

Αποτελέσματα

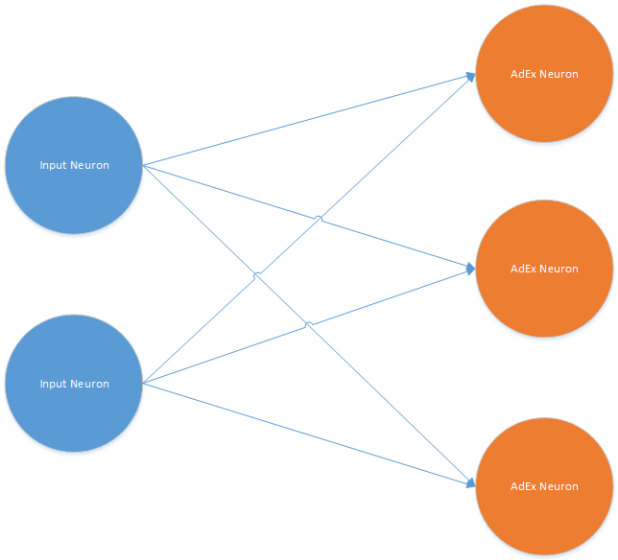
Για την συλλογή των μετρήσεων και την παρατήρηση της επιτάχυνσης της προσομοίωσης χρησιμοποιήθηκε η DFE με όνομα MAIA, η οποία αποτελείται από ένα FPGA τσιπ της Altera, τύπου Stratix V, το οποίο περιέχει 262400 προσαρμοστικά λογικά κομμάτια υψηλής απόδοσης, 3926 DSP κινητής υποδιαστολής και 2567 M20K μπλοκ μνήμης. Επίσης έχει 48 GB DDR3 μνήμη μέχρι 933 MHz. Για όλα τα πειράματα σε DFE χρησιμοποιήθηκε ο μονός πυρήνας με παραλληλοποίηση και αριθμούς κινητής υποδιαστολής μονής ακρίβειας. Ο πυρήνας χρησιμοποίησε το αρχικό ρολόι των 150 MHz, με την μνήμη να λειτουργεί στα 400 MHz. Ο επεξεργαστής που χρησιμοποιήθηκε είναι ένας Intel Xeon E5-2658A v3 με 12 πυρήνες και 24 νήματα, συχνότητα λειτουργίας 2.2 GHz με Turbo Συχνότητα 2.9 GHz και 30 MB Smart-Cache. Ο υπολογιστής είχε επίσης 128 GB DDR4 Ram. Όλα τα πειράματα στον επεξεργαστή τρέχουν σε έναν πυρήνα.
Ο κώδικας για την λήψη των μετρήσεων βρίσκεται στο εξής GitHub repository:
`https://github.com/iomaganaris/AdexSimMaxeler`

Οι παράμετροι της προσομοίωσης που επιλέχθηκαν να μεταβάλλονται είναι:

- Ο αριθμός των Νευρώνων Εισόδου και των AdEx Νευρώνων ώστε να διαπιστωθεί αν υπάρχει μεταβολή στην επιτάχυνση σε σχέση με το μέγεθος του δικτύου
- Η συχνότητα των παλμών που παράγουν οι Νευρώνες Εισόδου ώστε να παρατηρηθούν οι διαφορές στην επιτάχυνση ανάλογα με την δραστηριότητα του δικτύου
- Η σύνδεση των Νευρώνων για να παρατηρηθούν οι διαφορές στην επιτάχυνση ανάλογα με το ποσοστό των συνδέσεων των Νευρώνων
- Ο αριθμός των βημάτων της προσομοίωσης ώστε να διαπιστωθεί αν μεταβάλλεται η επιτάχυνση με το μέγεθος της προσομοίωσης

NxM Προσομοίωση

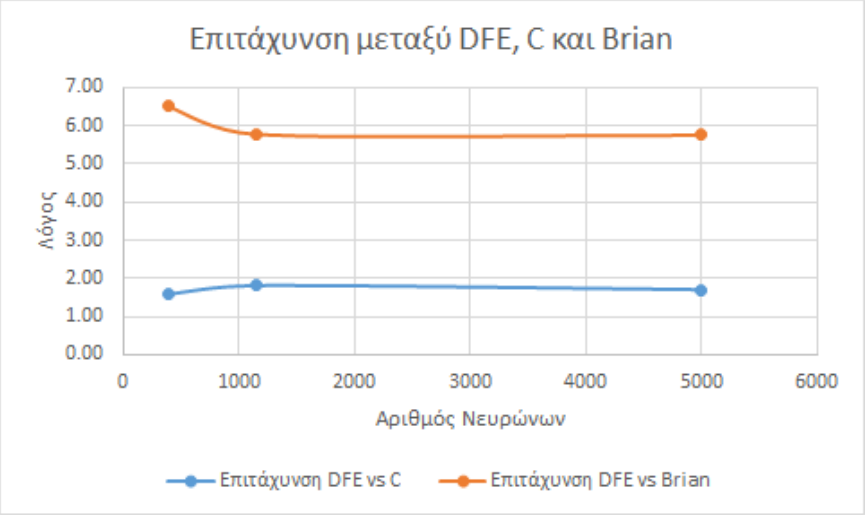
Σε αυτόν τον τύπο προσομοίωσης όπως αναφέρθηκε και νωρίτερα, συνδέονται Νευρώνες εισόδου, που παράγουν μόνο παλμούς δεδομένης μιας συνάρτησης παραγωγής τους, με Νευρώνες AdEx σε ένα δίκτυο δύο επιπέδων. Μια εικόνα του δικτύου αυτού φαίνεται από κάτω:
Αριθμός Νευρώνων Εισόδου: 2
Αριθμός AdEx Νευρώνων σαν Πηγές: 0
Αριθμός AdEx Νευρώνων σαν Στόχοι: 3
Συνδεσιμότητα: 100%
Αριθμός Συνάψεων: 6



Σχήμα 4: NxM Δίκτυο Νευρώνων

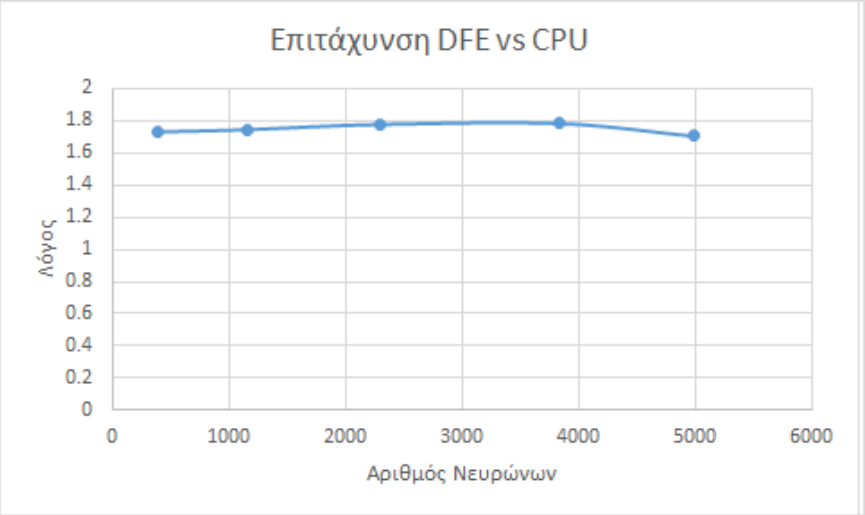
Οι μπλε κύκλοι αναπαριστάνουν τους Νευρώνες εισόδου, οι πορτοκαλοί τους AdEx και οι ευθείες γραμμές τις Συνάψεις.

Για κάθε πείραμα αυτού του τύπου οι νευρώνες αρχικοποιήθηκαν βάσει του άρθρου στο οποίο βασίστηκε η διπλωματική. Για την αρχικοποίηση των Συνάψεων μεταβλήθηκαν οι ίδιες τιμές οι οποίες είχαν αρχικοποιηθεί και στο άρθρο αλλά με έναν αύξοντα αριθμό που επαναλαμβάνεται μετά από το 16777216 που είναι ο μεγαλύτερος αριθμός που μπορούν να αναπαραστήσουν οι αριθμοί κινητής υποδιαστολής μονής ακρίβειας χωρίς να έχουν παραλείψει κάποιον προηγούμενο.
Αρχικά γίνεται μια σύγκριση ανάμεσα σε όλες τις υλοποιήσεις. Οι μεταβλητές που κρατούνται σταθερές είναι:
Βήματα προσομοίωσης: 1000
Βήμα προσομοίωσης: 1ms
Διάστημα μεταξύ παλμών Νευρώνων Εισόδου: 1 βήμα
Συνδεσιμότητα: 100%
Τα αποτελέσματα παριστάνονται στο παρακάτω διάγραμμα. Σε αυτό ο αριθμός των Νευρώνων Εισόδου είναι ίδιος με τον αριθμό Νευρώνων ΑδΕξ και παριστάνεται από τον οριζόντιο άξονα.



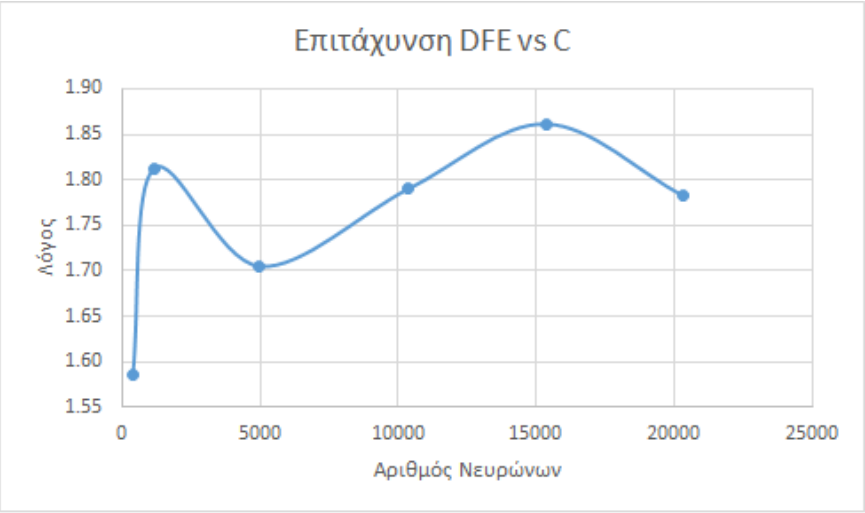
Σχήμα 5: Επιτάγχυνση μεταξύ DFE, C και Brian

Για να συγκρίνουμε καλύτερα τα αποτελέσματα μεταξύ C και DFE, γίνεται άλλο ένα πείραμα με τις ίδιες μεταβλητές με το προηγούμενο, όπου ωστόσο ο αριθμός των Νευρώνων AdEx παραμένει σταθερός στο 4992.



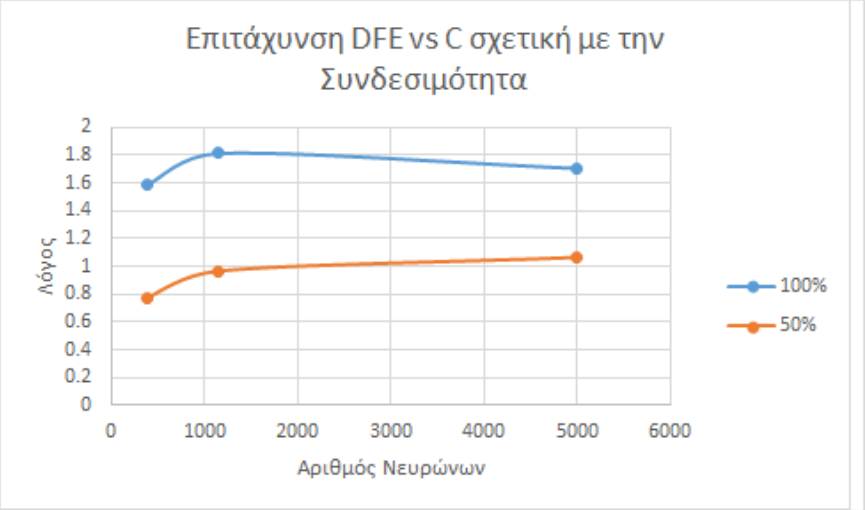
Σχήμα 6: Επιτάχυνση DFE vs CPU με σταθερό αριθμό Νευρώνων AdEx

Προσθέτοντας στις παραπάνω μετρήσεις, μετρήσεις για μεγαλύτερα μεγέθη δικτύων τα οποία όμως έχουν τρέξει για 100 βήματα προσομοίωσης λόγω του ότι διαφορετικά χρειαζόντουσαν υπερβολικά πολύ χρόνο για να τρέξουν (μέρες).



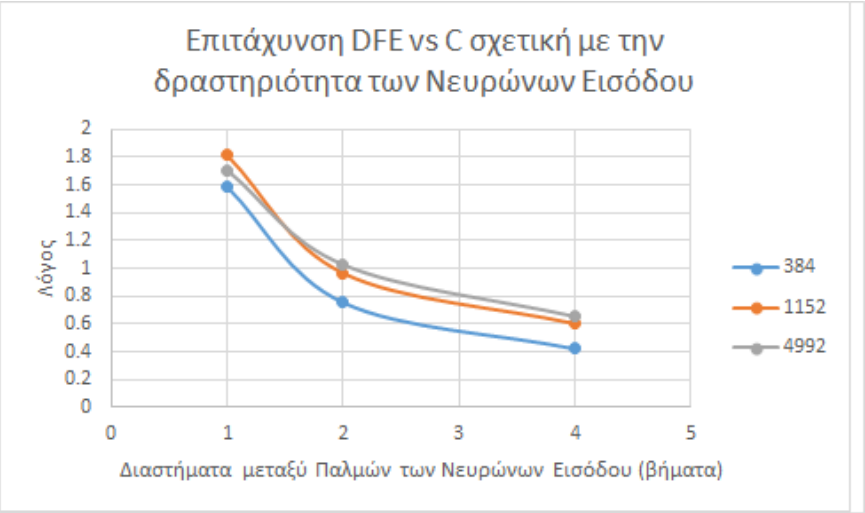
Σχήμα 7: Επιτάχυνση DFE vs CPU για μεγάλα δίκτυα

Ξανά ο αριθμός Νευρώνων Εισόδου και AdEx είναι ο ίδιος και φαίνεται στο οριζόντιο άξονα. Στη συνέχεια έχουμε τα πειράματα σχετικά με το ποσοστό συνδεσιμότητας των Συνάψεων. Για αυτές τις μετρήσεις χρησιμοποιήθηκαν οι ίδιες μεταβλητές με το πρώτο διάγραμμα με μόνη διαφορά το ποσοστό των συνδέσεων και μετριέται η επιτάχυνση σε DFE σε σύγκριση με τον επεξεργαστή.



Σχήμα 8: Επιτάχυνση DFE vs CPU εξαρτώμενη από την Συνδεσιμότητα

Τέλος βλέπουμε τον ρόλο που παίζει η δραστηριότητα των παλμών που παράγουν οι Νευρώνες Εισόδου στην επιτάχυνση της προσομοίωσης.



Σχήμα 9: Επιτάχυνση DFE vs CPU εξαρτώμενη από τα διαστήματα κατά τα οποία οι Νευρώνες Εισόδου στέλνουν Πλαμούς

Από τα προηγούμενα γραφήματα παρατηρούμε ότι η μέγιστη επιτάχυνση δεν είναι ιδιαίτερα μεγάλη σε σχέση με την υλοποίηση σε C, όντας περίπου 2 φορές. Αυτό συμβαίνει λόγω του ότι στη DFE χρειάζεται να εξεταστούν όλα τα ενδεχόμενα παλμών και εξαιτίας αυτού να καταναλωθεί χρόνος για την επίλυση των διαφορικών και της προσυναπτικής έκφρασης και της μετασυναπτικής. Αυτό όμως λόγω των διακλαδώσεων στην ροή εκτέλεσης του προγράμματος της C μέσω των if-statements γλυτώνει πράξεις και χρόνο από τον επεξεργαστή. Στην NxM προσομοίωση είμαστε σίγουροι μόνο για την δραστηριότητα των προσυναπτικών νευρώνων και μετά από παρατηρήσεις βλέπουμε ότι δεν υπάρχουν πολλοί παραγώμενοι παλμοί από τους μετασυναπτικούς εξαιτίας του μοντέλου τους. Έτσι εκτελούνται περισσότερες πράξεις στο FPGA χωρίς απαραίτητα να χρειάζονται, αφού εκτελούνται οι πράξεις και της μετασυναπτικής έκφρασης για κάθε ενδεχόμενο, σε αντίθεση με την C. Για τον ίδιο λόγο υπάρχει διακύμανση στην επιτάχυνση και στις περιπτώσεις μεταβολής της συνδεσιμότητας και της δραστηριότητας των προσυναπτικών Νευρώνων Εισόδου.

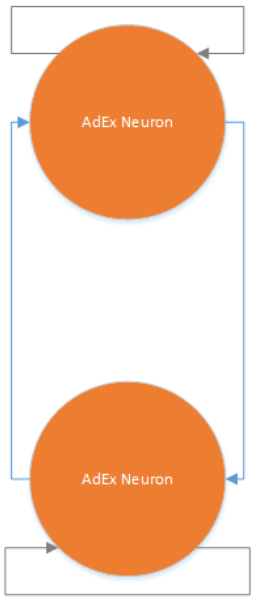
Αποκλίσεις

Τέλος όπως αναφέρθηκε προηγουμένως, λόγω της χρήσης αριθμών κινητής υποδιαστολής μονής ακρίβειας στη DFE αντί για διπλής στον Brian και στη C, υπάρχουν κάποιες αποκλίσεις στις τιμές που υπολογίζονται. Ένα χαρακτηριστικό παράδειγμα προέρχεται από την μεγαλύτερη προσομοίωση που έτρεξε για 1000 βήματα και 4992 Νευρώνες Εισόδου και 4992 Νευρώνες AdEx. Σε αυτό το πείραμα, οι διαφορές στα αποτελέσματα ήταν πάνω από 100000 μικρότερες από τις τιμές των μεταβλητών, άρα αναπαίσθητες και ασήμαντες.

MxM Προσομοίωση

Σε αυτόν τον τύπο προσομοιώσεων δεν υπάρχουν Νευρώνες Εισόδου συνδεδεμένοι με το

δίκτυο. Το δίκτυο αποτελείται μόνο από AdEx Νευρώνες συνδεδεμένους μεταξύ τους. Μια αναπαράσταση του δικτύου φαίνεται παρακάτω:
Αριθμός Νευρώνων Εισόδου: 0
Αριθμός Νευρώνων AdEx σαν Πηγές: 2
Αριθμός Νευρώνων AdEx σαν Στόχοι: 2
Συνδεσιμότητα: 100%
Αριθμός Συνάψεων: 4

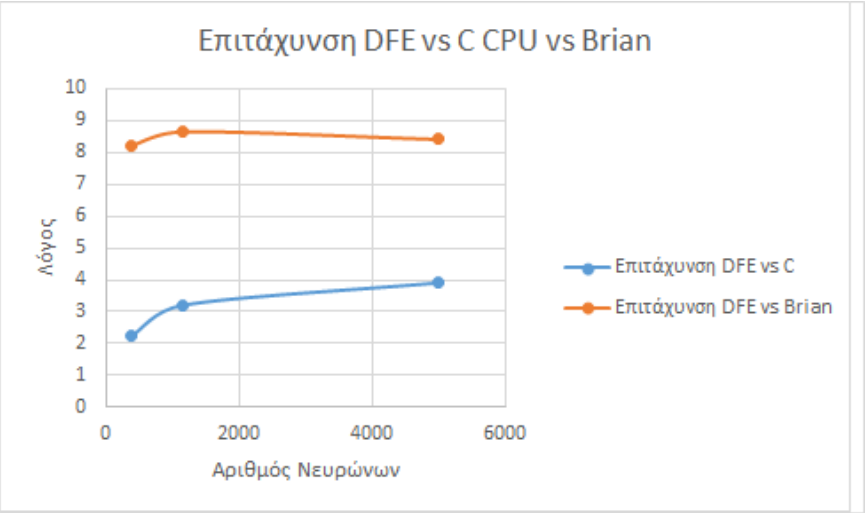


Σχήμα 10: MxM Δίκτυο Νευρώνων

Οι γραμμές αναπαριστούν τις συνάψεις. Οι γκρι γραμμές δεν έχουν φυσική σημασία μάλλον αφού αναπαριστούν συνάψεις προς τον ίδιο νευρώνα. Για την αρχικοποίηση των Νευρώνων AdEx χρησιμοποιήθηκαν οι ίδιες τιμές με την NxM προσομοίωση, εκτός από την μεταβλητή τους Vm, η οποία τέθηκε 5mV πάνω από την Vt, ώστε στο πρώτο βήμα να παράγουν παλμούς και κατ' επέκταση να κάνουν παλμούς σε κάθε χρονική στιγμή.

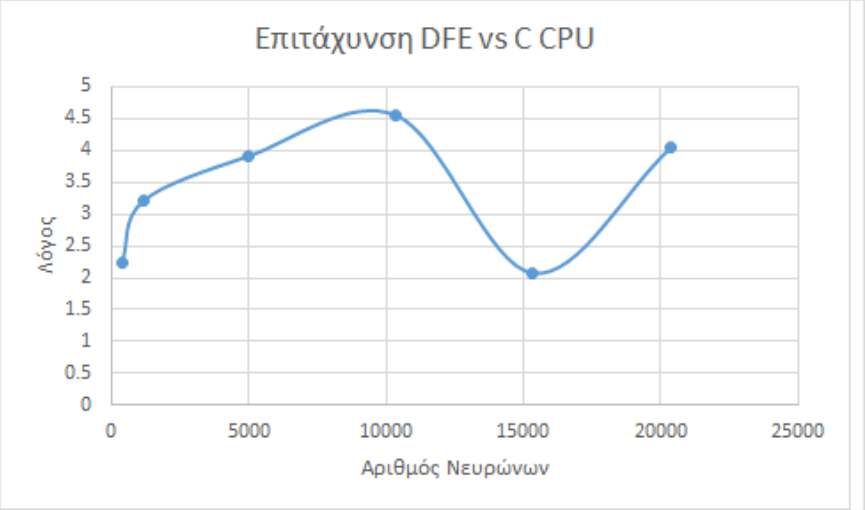
Αποτελέσματα

Αρχικά γίνεται μια σύγκριση ανάμεσα σε όλες τις υλοποιήσεις. Οι σταθερές των μετρήσεων είναι:
Βήματα Προσομοίωσης: 1000
Βήμα προσομοίωσης: 1ms
Συνδεσιμότητα: 100%



Σχήμα 11: Επιτάχυνση μεταξύ DFE, C και Brian

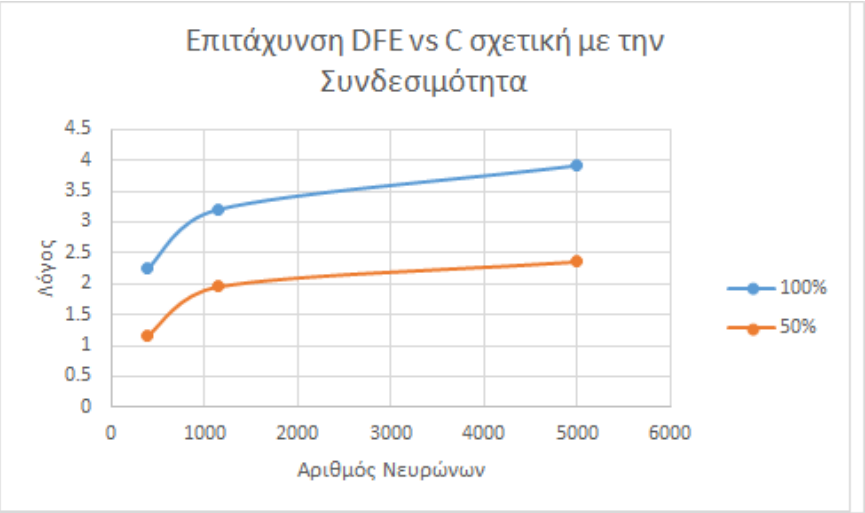
Στη συνέχεια για μεγαλύτερα πλήθη Νευρώνων γίνεται μια σύγκριση της προσομοίωσης σε DFE με την C. Για πάνω από 4992 Νευρώνες η προσομοίωση έτρεξε για 100 βήματα λόγω μεγάλου χρόνου εκτέλεσης.



Σχήμα 12: Επιτάχυνση DFE vs CPU για μεγάλα δίκτυα

Παρατηρούμε σε αυτό το διάγραμμα ότι για 15360 νευρώνες δεν υπήρχαν πολλοί παλμοί από τους νευρώνες λόγω του τρόπου με τον οποίο έχουν αρχικοποιηθεί οι συνάψεις και γι αυτό υπάρχει πτώση στην επιτάχυνση.

Τέλος παρατηρούμε την επίδραση της Συνδεσιμότητας στην επιτάχυνση της υλοποίησης σε DFE σε σχέση με την C.



Σχήμα 13: Επιτάχυνση DFE vs CPU εξαρτώμενη από την Συνδεσιμότητα

Συμπερασματικά, η προσομοίωση MxM όπου υπάρχει δραστηριότητα σε όλο το δίκτυο για κάθε στιγμή λόγω του τρόπου με τον οποίο έχει αρχικοποιηθεί παρουσιάζει σχεδόν διπλάσια επιτάχυνση σε σύγκριση με την NxM προσομοίωση για ίδιο μέγεθος δικτύου. Αυτό οφείλεται στις περισσότερες πράξεις που καλείται να εκτελέσει ο επεξεργαστής μέσω του C προγράμματος, πράξεις τις οποίες η DFE εκτελεί και στην περίπτωση της NxM και της MxM προσομοίωσης.

Αποκλίσεις

Σχετικά με τις αποκλίσεις, και σε αυτό το πείραμα είναι μικρές. Για την μεγαλύτερη σε διάρκεια προσομοίωση που έτρεξε με 1000 βήματα και 4992 Νευρώνες AdEx, οι αποκλίσεις ήταν πάνω από 100000 φορές μικρότερες από τις τιμές, άρα αναπαίσθητες και ασήμαντες.

Chapter 1

Introduction

The motivation for this Diploma Thesis has been the need of neuroscientists to run simulation of neuronal networks fast. Like in many disciplines of science, modern research in Neuroscience is also based on computer modelling and simulation. The vast complexity of the brain, the organization and the behaviour of the neurons leads to huge simulations that take a lot of time to run, delaying research. The goal of the thesis is to try and find the optimal way to accelerate a particular Neuron Model that is widely used in neuroscientific experiments and simulations. This model is called Adaptive Exponential Integrate-and-fire model. The particular simulation implemented encapsulates also the phenomenon of Spike-timing-dependent plasticity of the connections of Neurons.

The faster execution of the simulation was made possible by using a plethora of methods. Initially, the model was imported by the BRIAN Simulator, an open source and widespread solver for neuron simulations that runs on Python. The step towards accelerating the simulator was achieved by importing the simulation in C, which has a better performance than Python, which is a higher level programming language. Afterwards, the simulation was imported in Maxeler DFEs, implemented with FPGAs, to use the Maxeler Dataflow Programming model to accelerate it.

1.1 Neuroscience

An average adult brain weighs almost 1.4 kilos[Har94], which accounts for 2% of the person's weight, however it consumes 20% of the sum of energy its body uses throughout a relaxing day, without activities. Moreover, it is the most complicated organ of the neuronal system of the human, with 100 billion nerve cells and 100 trillion neural synapses which have evolved from the start of life to their current state[Gio14]. It is a safe to conclude that the science that studies the brain and the whole nervous system will have a similar evolution. This science is called Neuroscience and it is supposed to be a subcategory of Biology, which combines other branches of Biology, like physiology, anatomy, molecular and developmental biology and cytology, but not only. It also needs statistics, math and branches of medicine like psychology.

More precisely, Neuroscience studies the human nerve system, how it works, how it is organized and how it develops. The biggest effort of neuroscientists is focused on the study of the brain and how it deflects on the consciousness and the behavior of the person. However, Neuroscience research is not only focused in the normal operation, but it also tries to interpret brain damage and how it operates on people with neuronal problems or psychological disorders. The development of Neuroscience has produced solutions for patients with brain damage, problems in their spinal cord and psychological disorders.

The main targets of Neuroscience are:

- Understanding the human brain and how it functions
- Understanding and describing how the central nerve system develops, matures and maintains itself.
- Analysis and understanding of the neuronal and psychological disorders and finding ways of prevention and treatment.

1.1.1 History

The first records for the study of human brain come from Ancient Egypt and of course through the years has made huge steps forward. The Edwin Smith papyrus, which dates back to 17th century BC, describes the symptoms, diagnosis and prognosis of two patients which have fragmented parts of their skull. This battlefield doctor's papyrus contains descriptions of the brain and the symptoms of the patients. This was also the first record with logical conclusions and observation, as in the past people lacked complete understanding of how the brain works and were based on myths and superstition. [ERKJ81; AG87]

During the second millennium BC in Ancient Greece people started to occupy themselves with the study of brain, expressing different opinions. However, due to the fact that human body was considered sacred, hippocratic doctors didn't operate on bodies to study the nerve systems and therefore, didn't use anatomy. The first that conceived that the mind is in the brain was Alcmaeon of Croton (6th and 5th century BC). More precisely, he stated that the brain is what distinguishes human from animals. That people can understand through the brain what in the same time animals just feel. The brain for him was the center of the senses and after anatomic studies discovered the sensory nerves and named them pores. He stated that with those pores, the human brain absorbed the information that it takes through the senses and that is how cognition, which supports studying, phantasy, memory and crisis, is created. In the 4th century BC, Hippocrates believed that brain is also the seat of intelligence based on the knowledge produced by Alcmeon. Contrasting these views, in 4th century BC, Aristoteles stated that heart is responsible for intelligence in the human beings and that brain cooled the blood and was the part that added to people more reason than animals, which had smaller brain.

Egyptians, in contrast with Greeks who believed that human body was sacred, embalmed their dead for centuries and consequently studied systematically the human body. During the

hellenistic period, Herophilus of Chalcedon (330-250 BC) and Erasistratus of Ceos (304-250 BC) contributed vastly, not only in the study of human brain, the nervous system, anatomy and physiology, but in numerous other branches of biology. Herophilus defined the differences between the cerebrum and the cerebellum. In addition, he gave the first detailed description of ventricles. Erasistratus experimented on living brains. The original works of both of them were not saved and all these information come through secondary sources. During the Roman Empire, the Greek anatomist Galen did detailed dissections of the brains of sheeps, monkeys and dogs. He concluded that the cerebellum was more dense than the brain that it must be responsible for muscle control, while the brain was processing the senses. Furthermore, he described seven of the cranial nerves and defined by running experiments the operation of most of them. He discovered that specific specific spinal nerves controlled specific muscles and conceived the idea of reciprocal action of muscles.[AG87] Living during the Middle Ages, an Arab surgeon named Abul-Qasim Al-Zahrawi [NRARJLF84] made a significant contribution to neuroscience. His efforts focused on neurosurgery. He not only described neurosurgical diagnosis and treatment related to injuries and skull fractures, spinal injuries and dislocations, hydrocephalus and subdural effusions, headache and many other medical afflictions, but also the tools needed such as cranial drills that avoided puncture of the dura mater. In the 11th century, Ibn Sina also contributed to neurosurgery. It was up until the 13th to 14th century that Europe started recording the knowledge about anatomy and the brain. Andreas Vesalius wrote books about anatomy in general and consequently for the brain. He studied the peripheral nervous system and concluded that there are seven pairs of brain-nerves. [VL93] René Descartes was a philosopher and developed the theory of dualism. He believed that the pineal gland was what connected the mind with the body. [Zal17] Thomas Willis contributed to the study of the brain, the nerves and behavior to develop neurologic treatments.

In the 18th century Luigi Galvani discovered the function of electricity in dissected frogs. Marie Jean Pierre Flourens in 1820s was the founder of experimental brain science and developed anesthesia. He was the one that shattered the belief that the mind is in heart, with scientific evidence. The study concerning the function served by electricity in the brain continued in rabbits and monkeys by Richard Canton. Then the discovery of the microscope brought a revolution in the study of the brain. Camillo Golgi in the 1890s was able to picture with detail the structure of a single neuron applying a silver chromate salt to them. This enabled Santiago Ramón y Cajal to imagine that the functional unit of the brain is the neuron in the hypothesis of neuron doctrine. Emil du Bois-Reymond, Johannes Peter Müller and Hermann von Helmholtz presented in the 19th century the fact that neurons were electrically excitable and their activity affected their neighbor neurons. Paul Broca, occupying himself with brain-damaged patients supported that particular parts of the brain were responsible for certain actions. John Hughlings Jackson contributed in the diagnosis and understanding of epilepsy and by his observations in the focal motors, he supported the Broca's hypothesis. On this topic, Carl Wernicke focused on research of brain diseases on speech and language, based on Broca's previous work. From the 20th century, up until

now, Korbinian Brodmann's cytoarchitectonic anatomical definitions stand up, presenting the activation of distinct areas of the cortex for specific tasks. [ERKJ81]

During the 20th century, Neuroscience has started to be recognised as a distinct academic discipline and not as part of other sciences. David Rioch was a research scientist and neuroanatomist, pioneering brain research and leading the interdisciplinary neuropsychiatry division at the Walter Reed Army Institute of Research in the 1950s. This program helped the establishment of the neuroscience. At that time, Francis O. Schmitt created a neuroscience research program in the Biology Department of MIT, combining many disciplines of Science. James L. McGaugh founded the first distinct neuroscience department, that was firstly called Psychobiology, in 1964 in the University of California, Irvine. [ERKJ81; Cow00]

All these years of research where did they end up? It follows an overview on the basic anatomy of the human brain and what we know up until now for its operation. To begin with, the human brain is the central organ of the human nervous system and with the spinal cord makes up the central nervous system. The brain can be very easily compared to a CPU, talking with computer science terms. It gathers all the information coming from the senses and the nerves across the body, processes them and takes decisions that are transferred to the corresponding part of the body. These decisions are taken in different parts of the brain, based on what function they are related to. Consequently, the brain has been divided in three different areas. It consists of the cerebrum, the brainstem and the cerebellum.

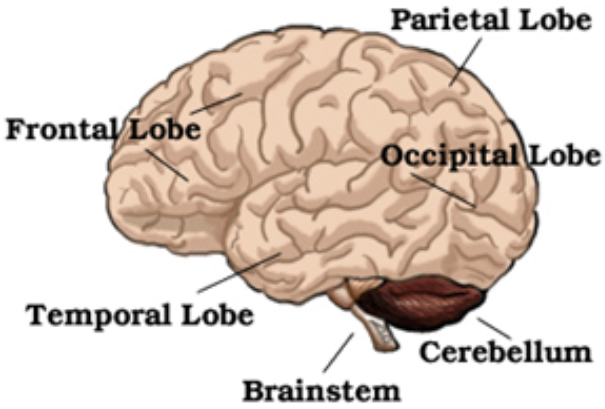


Figure 1.1: Human Brain Organization
[GSJ13]

The cerebrum is the biggest part of the human brain. The cerebral cortex covers the largest region of the cerebrum, is the most anterior brain region and has an outer layer of gray matter, which is a neural tissue that contains neuronal cell bodies. The cerebral cortex is divided into the left and right cerebral hemispheres that are interconnected by commissural nerve tracts, which are groups of nerve fibers (axons), the largest being the corpus callosum. These two hemispheres are further divided into the frontal, temporal, parietal and occipital lobes. [Dav11]

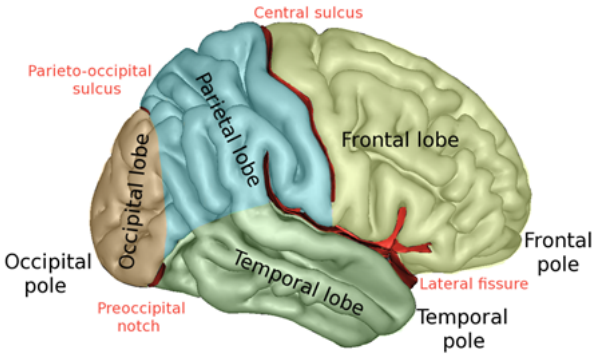


Figure 1.2: Hemisphere Divisions
[Seb12]

The frontal lobe undertakes the tasks related to self-control, planning, reasoning and abstract thought. The temporal lobe corresponds to visual memory, language comprehension and emotion association. The parietal lobe combines information coming from different sensor centers, like spatial sense and navigation, sense of touch and the dorsal stream of the visual system. The occipital lobe focuses on processing vision. The two hemispheres share similar shape and function, however each one is focused in different functions. The left in language and the right in visual-spatial ability. The brainstem is used to connect the cerebrum to the spinal cord. It includes the midbrain, the pons and the medulla oblongata. Its responsibilities are controlling the actions and sensations of the face and neck via the cranial nerves, helping regulate the cardiac and respiratory function, regulates the central nervous system and the sleep cycle. It is also decisive in maintaining consciousness. Moreover, through the brainstem pass all the nerve connections for the motor and sensory systems of the main part of the brain to the rest of the body.

The cerebellum is mainly responsible for motor control but it also takes part in cognitive functions like attention, language, regulating fear and pleasure responses. The cerebellum in people, doesn't initiate the moves but it help with coordination, precision and accurate timing, receiving input from sensory systems of the spinal cord and other parts of the brain and integrating these inputs to fine-tune motor activity.

All these parts of the brain consist of two basic cells. The neuron and the supportive glial cells. The latter ones are non-neuronal cells in the central and peripheral nervous system. Their responsibilities are maintaining homeostasis, forming myelin and providing support and protection for the neurons. More precisely, they surround neurons and keep them in place, the supply nutrients and oxygen to them, insulate one from another and destroy pathogens and remove dead neurons. [Jes80]

1.1.2 Neuron

Neurons are the principal cellular elements that defines the function of the whole nervous system and its parts:the brain, spinal cord, peripheral sensory systems and enteric nervous

system. The neuron is characterized by its central cell body (soma) that comes in different variations in shape. The central body encapsulates the cell nucleus and most of the genomic expression and synthetic machinery that elaborates the proteins, lipids and sugars that contribute to the cytoplasm and membranes. The bounds of the soma are defined by the membranous system that also defines intracellular compartments. [Lli08]

The different functions of the nerve cells (voltage and ligand activated ionic channels, ionic pumps, non-gated "leakage" channels) and the processes of taking up and replacing the molecular modules that constitute the cell's functional matrix are ordered by transmembrane macromolecules that are associated with the plasmalemma and other intracellular elements. Neurons can be conceived as a two port element. They have an input and an output, however in the past there were thought that there are neuron cells that didn't have any plasmalemmal extensions that we know now as receptor cells. Neurons in general can have many types of branching or no branching. Most of them include however include an input and an output pole.

The receiving or input pole is called dendrites, from the word "dendro" in Greek which means tree, due to its tree-like branches extensions of the soma membrane. In most neurons the whole body is a receiving site. In vertebrate neurons the dendrites come directly from the soma and in invertebrate neurons most commonly come from the axon. The output pole is called axon is a single structure that arises from the some or a dendrite. It sends propagating electrochemical signals called action potentials outside of the soma. These signals are most commonly initiated at the axon hillock. In some neurons the dendrites can function as outputs.

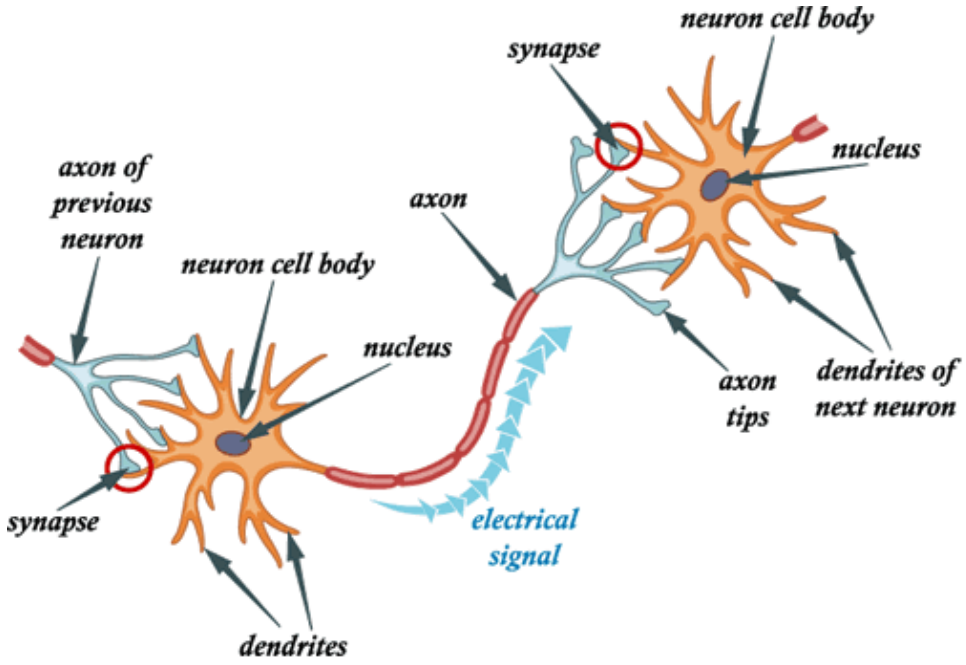


Figure 1.3: Neurons Organization

The neuron has a huge variety of electrophysiological properties that adds to it a vast set of electrical properties and functional styles. The voltage and ligand dependent ionic conductances that generate and modulate such excitability can implement autorhythmic properties as single cell oscillators or resonators that ultimately dictate network oscillatory properties. The study of motoneurons concluded that the basic function of neurons in integrate and fire. From then up until now only few parameters have been added to the neuron, such as Plasticity which is expressed as Long Term Potentiation and Long Term Depression and intrinsic electrical properties.

The four main functional properties that characterize the neurons are:

- Electrical Excitability
- Secretion
- Molecular Synthesis
- Growth and Plasticity

The main of them that concerns us most is Electrical Excitability and has numerous properties.

The passive electrical properties are related to the capacity and resistance of the the neuronal membranes and the resistance of the cytoplasm and the extracellular milieu. The combination of these properties forms an electrical resemblance between the neuronal processes and the axons and dendrites of the neuron with the electrical properties of cables. Selective ion pumps dictate the electric field and the voltage difference of the membrane. The membrane potential is not uniformly distributed throughout the membrane and it is related to the density of the ionic conductances which again are not uniformly distributed. However, for simplification the membrane potential is assumed to have a resting value uniformly distributed, called resting potential. The value of the electrical field (mV) is related to the driving force (emf) for each of the ionic species that can move across the membrane and the magnitude of the conductance for each ionic species. The membrane potential is passively conducted on the membrane processes as the result of currents that flow through the longitudinal resistance or across the plasmalemmal membrane as resistive or capacitive current.

The active electrical properties are related to the activation of voltage, ligand or second messenger gated transmembrane ionic channels that have as a result variation of the electrical potentials across the plasma membrane. The electric field across the membrane acts on the voltage sensors of the transmembrane ionic channels. Specifically on voltage-gated channels, the inflow of sodium or calcium ions depolarizes the plasma membrane. The opening of the voltage-gated channels results in current flow that repolarizes the plasma membrane. In general, the conductance of voltage-gated channels is increased by membrane depolarization. However, there are some channels that increase their conductance when the membrane is hyperpolarized. Another active electrical property is related to the ligand-gated

ionic conductance, where the binding of a neurotransmitter will gate ionic conductances allowing the generation of excitatory or inhibitory synaptic potentials. Moreover, there are subthreshold oscillations, meaning that the excitability of the cell is gated in such a way that the membrane potential is not uniform but follows a continuous fluctuation.

The superposition of passive and active electrical properties in an active cell result the possibility to allow the cell to sum the transmembrane potential linearly or nonlinearly and reach depolarization levels sufficiently high to trigger action potentials. These are conducted either along the axon or the dendritic tree, in an all-or-none continuous manner, in saltatory fashion or in decremental mode.

Neurons have only one axon that can start from the some or the dendrite of the neuron. Axons branch collaterally along their length or at their terminal (telodendrion). They are the presynaptic connections of the synapses that connects the neuron with another neuron, muscles or glands. Axons send very similar spike sequences to all their branches because the start from a single segment. However, due to spike failure at branch connections or changes in conduction velocity and changes in axonal diameter after branching can alter conduction patterns and conduction time.

Apart from axons, action potentials in axons, regenerative events can also come from dendrites. These potential in general decrement with distance, but it is not impossible to reach the most distant dendritic branches. The actions can move toward the soma or propagate away of the soma depending on the dendritic morphology and the distribution and density of voltage gated ionic channels over the dendritic tree.

In addition to action potentials and synaptic transmission there is also electrical activity generated autonomously by the neurons. Most of the time, the autonomous intrinsic activity leads to modulation of the resting potential and consequently the state of the whole neuron, in the sense not only of synaptic modulation or the modulation produced by peptidergic, hormonal and metabolic activity, but also of other parameters such as pH and free radical activity modulation. Apart from modulation, the most relevant parameter that defines intrinsic activity, other than resting potential, is the types and distribution of plasma membrane channels and second messenger modulation of channels.

Finally, the electrical signature of neurons is defined by two factors. The first one is the passive integrative properties of the dendrites and soma. The second one is the non-linear electrical properties superimposed by the presence of voltage, ligand, second messenger, and metabotropic conductances supported by specialized ionic plasma membrane bound channels. These modulate excitability by their number, functional phenotype, and distribution over the dendritic, somatic and axonal neuronal segments.

While certain characteristic properties can be assigned to given cellular phenotypes, the fact is that every neuron is unique both in its individually detailed shape and its connectivity. Perhaps it is the diversity of such parameters that allow the CNS to be as reliable as it actually is. It was von Neumann who first realized the genesis of reliability from unreliable elements as one of the central character of “neuron-ness”.

1.1.3 Synapses

A synapse is a structure that enables a neuron to transmit an electrical or chemical signal to another neuron or to the target effector cell. In a synapse, the plasma membrane of the signal source neuron (presynaptic neuron) comes into close apposition with the membrane of the target (postsynaptic) cell. The sites of presynaptic and post postsynaptic neurons that are connected have extensive arrays of molecular machinery that connect the two membranes together and carry out the signalling process. In most cases the presynaptic site is located in the axon, while the postsynaptic in the dendrites. Glia also exchange information with the synaptic neurons, having an impact in neurotransmission. Chemical synapses are kept in place by synaptic adhesion molecules projecting from pre- and post-synaptic neurons, connecting the overlapping part, while playing a part in the generation and function of synapses. [Fos97; Eli06; Sch11]

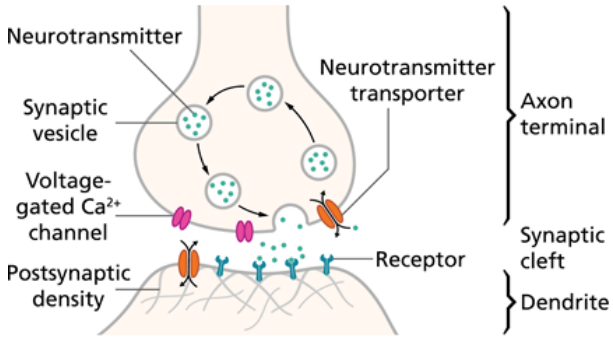


Figure 1.4: Synapse
[Spl15]

The synapses are divided into two categories, chemical and electrical synapses. In the chemical synapses electrical activity in the presynaptic neuron is converted via the activation of voltage-gated calcium channels into the release of a chemical called neurotransmitter that binds to the receptors located in the plasma membrane of the postsynaptic cell. There is a probability that the neurotransmitter initiates an electrical response or secondary messenger pathway that excites or inhibits the postsynaptic neuron. Chemical synapses are further classified based on the neurotransmitter released. There are glutamatergic (often excitatory), GABAergic (often inhibitory), cholinergic (e.g. vertebrate neuromuscular junction), and adrenergic (releasing norepinephrine). Because of the complexity of receptor signal transduction, chemical synapses can have complex effects on the postsynaptic cell. In the electrical synapses, pre- and post-synaptic cell membranes are connected by special channels called gap junctions or synaptic cleft that are capable of passing an electric current, causing voltage changes in the presynaptic cell to induce voltage changes in the postsynaptic cell. The main advantage of the electrical synapses is the rapid transfer of signals to the neighbor cell. Synaptic communication is distinct from an ephaptic coupling, in which communication between neurons occurs via indirect electric fields. An autapse is a chemical or electrical synapse that forms when the axon of one neuron synapses into dendrites of the

same neuron.

1.1.4 Levels of Analysis in Neural Modeling

The qualitative and quantitative models that represent natural phenomena in Neuroscience are as important as other scientific fields. For neuroscience however, this task is even more difficult, as these models serve two functions. Firstly, to represent the experimental data and secondly, to interpret the underlying computations that are processed in the brain. Neuroscientists have to understand the neural models, how they are connected to the experimental data and how to use them in practice. [Day06]

Brain modelling has four different levels of organization. The first one is scientific reduction, meaning the description of observations with qualitative and quantitative detail and explaining them by describing them in lower and less abstract levels. The second one is based on the duality of the first one. It is the implementation of systems for a particular purpose. Many times this is possible by dividing the problem into smaller ones that are easier to be solved. The third level is the algorithmical one, which means that the processes on the brain are described by algorithms and then there is a implementational level which describes how these algorithms are implemented by neurons. The final one is about processing as a way to manipulate and extract information from the input.

1.1.5 Types of Neural Modeling

1.1.5.1 Conventional reductive models

The idea of reductive modeling is applied to neurons too. The practical benefit of this approach is the ability to describe the phenomena and provide the means to reductive explain them, by appealing to the mechanisms that might actually be responsible for the generation of these phenomena. This modelling technique is also most of the time recursive, as most of these models are described by models too. To be able to have the required accuracy on the quantitative representation of a model, usually there are needed mathematical models. Of course, there are different levels of a model that correspond to different levels of reduction of a phenomenon. Especially in neuroscience, the difference is dictated by the anatomical detail that is contained in every model. The descriptive models of a level encapsulate only the behavior without any substrate, while explanatory models encapsulate the behavior by unraveling it in lower levels. The combination of those two approaches generate the quantitative models that allow proof, in the form of numerical demonstration, that the behaviour that is represented by a model is truly a description of the real phenomenon that it tries to explain. The models of a certain level almost always are more abstract than the models of lower levels, that encapsulate more details of the phenomenon they represent.

1.1.5.2 Computational interpretive models

These models are based on the idea that the brain does computations to perform various tasks. These computations involve parts of the the visual and the motor systems of the brain, that based on the input it has, previous behavior and experience, computes the best way to perform the task it needs. Computational modeling is about imputing and interpreting the overall operations of the neural systems of the brain to perform a specific action. The key aspects of computations are representation, storage and transformation or algorithmic manipulation. Computational modeling tries to interpret how neural systems represent and store information that take as input and how they process them to create new ones that help the completion of a given task. Similarly to standard computers, the semantics of the computation are implemented by the syntax of the physical substrate.

Computational and Conventional models have a lot of things in common. To begin with, there are for both of them different levels of abstraction. Furthermore, there are descriptive and explanatory computation models. Finally, for an single task of the same abstraction level, there can be many interpretations and representations for the same computation. The analysis of neural systems needs the combination of computational and conventional modelling. At a single level, the computational model is able to represent exactly the the experimental phenomena that the conventional reductive model has interpreted. The reductions on the complexity of the natural phenomenon comprises the lower level of the conventional model and it must comprise also the implentational plane of the lower level of the computational model. The algorithmic and computational planes of the computational level should be consistent with those of the conventional level. And they must be consistent throughout all the levels of abstraction. This only will enable us to have a complete understanding and representation of the operation of the human brain.

1.1.6 Degrees of Modeling Detail of Neurons

Based on the level of abstraction and based on the data that come from the same level of abstraction, there are three main classes of quantitative models in common use.

1.1.6.1 Conductance-based models

These models encapsulate a high level of detail, giving emphasis in a small number of neurons. They approximate structure of a neuron by multiple, interconnected, compartments, each of which is treated as being electrically compact. The whole set of compartments is designed to be faithful to the geometry of the neuron, including facets as branching points of dendrites and the diameters and lengths of different parts. To be make the computations of this model quick enough, usually a complex neuron with multiple compartments is represents by only one compartment. In standard conductance-based models, each compartment is given an assortment of active channels, such as voltage sensitive or synaptic channels. They are ideal for explaining phenomena related with spikes and threshold for initiating spikes, the precise effects of synaptic input, bursting, spike adaption, spikes that propagate

backwards up the dendritic tree and the like. The main issue with these models is that in order to have faithful results they need numerous variables that are very difficult to be calculated accurately by experiments.

1.1.6.2 Integrate-and-fire models

These models lie an abstraction level higher than conductance-based models. They make an approximation for action potentials using a symbolic model of spike generation coupled with a leaky integrator model of a cell that initiates a spike when the voltage is above a threshold. The also radically simplify the geometry of cells, elimination the compartmentalization. They are ideal for simulating large and recurrently connected networks of neurons. Their usage has helped with exploration of many mathematical issues about networks, such as the synchronization and desynchronization of spiking across the whole population and the effects of different sorts and sources of noise. The details of such phenomena as synaptic plasticity are dependent on such phenomena as precise time differences between pre-synaptic and post-synaptic activity. The integrate-and-fire is the simplest form that output spikes and can be used to address such issues.

1.1.6.3 Firing-rate models

This is the most abstract model of neuron representation, abandoning neuron firing and instead having a continuous-valued, timing varying firing rate for neuron output. It can become an abstraction of the integrate-and-fire model with some assumptions about the time-constants of processes inside cells. Networks of this model can be constructed, where the influence of a neuron to another is given by the product of the pre-synaptic neuron firing rate and the synaptic strength of the connection. The main advantages of this model is its empirical and analytical tractability. Firing-rate models involve a mild non-linearity, turning an internal continuous variable, like somatic voltage or current, into a (positive) firing rate. Consequently, networks of these neurons can be treated as coupled, non-linear differential equations that can be shown to exhibit dynamical behaviors. The regularities that are implied by attractor and oscillatory dynamical behaviors make them ideal as substrates on which to hang analysis of network computation. Most work on computational analyses was made using firing-rate models, due to the simplicity of the analysis of non-recurrent, feedforward network models.

1.2 FPGA

The term FPGA derives from the initial letters of the words Field Programmable Gate Array. As this phrase suggest, FPGA is an integrated circuit that its connections and configuration are programmable by the user. To describe this configuration the user has to use a hardware description language (HDL), which describes with high detail how exactly the algorithm, all the operations and operators, the pipeline and the whole timing of a

program design should be run. Due to the fact that this is quite complex for a non-expert developer, various higher level compilers and IDEs have been proposed to write programs in more high level languages but still be translated to HDL and run on FPGAs. The term Gate Array indicates that FPGAs contain a structured set of programmable logic blocks and a hierarchy of reconfigurable interconnects that allow particular blocks to connect with others. The logic blocks contain Look-Up-Tables that are connected with other elements like Full Adders, Multiplexers, simple logic gates, Flip-Flops, etc to implement from complex functions to simple logic gates like AND and XOR.

Logic Blocks

An example Logic Block (or Logic Cell) of a Xilinx FPGA is:

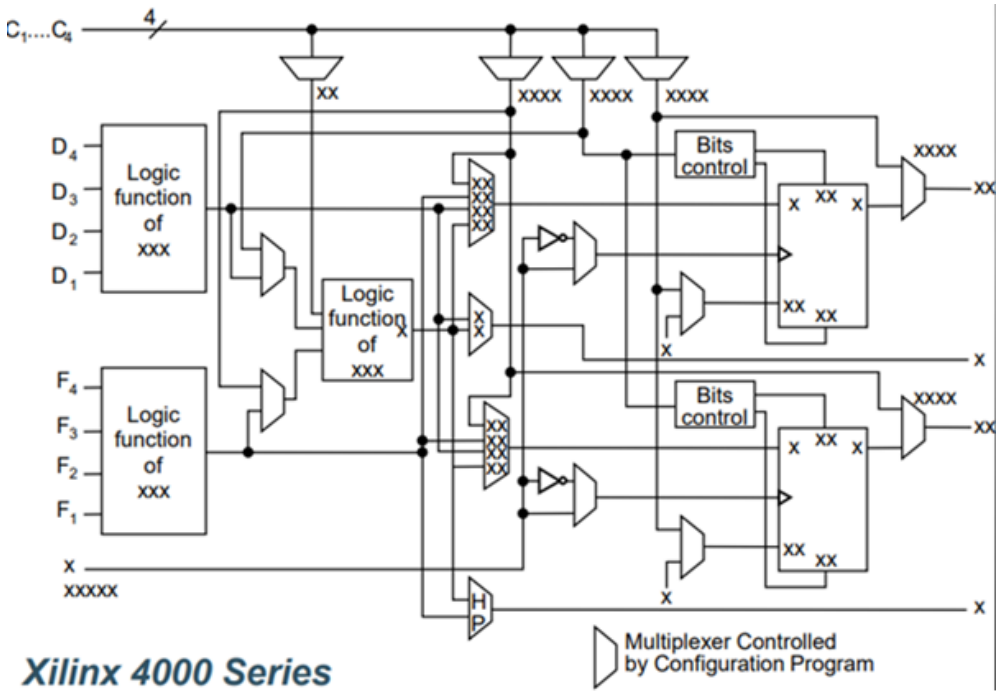


Figure 1.5: Logic Block
[Sou17]

The LUTs are configured in such way that they can implement whatever function. In each configuration, a Table of Truth is saved on them, so for a given input there is a given output. This makes the delay of calculation of a function irrelevant of the function that is implemented. The use of memory to implement complex functions to gain efficiency is a typical methodology in Digital Design. In the above example, two 4 input LUTs are connected with configured Multiplexers that combine them to create an 8bit input Logic Block to execute the function needed. The storage elements are configured as edge-triggered D-Type flip-flops or level-sensitive laches. [Xil08; Sou17]

Hard Blocks

On top of the Logic blocks described above, modern FPGAs include also Hard Blocks that are designed to perform a specific action and are not configurable by the user. Those

blocks are usually multipliers, generic DSP blocks, embedded processors, high speed I/O logic and embedded memories. The fact that are embedded and not configurable enables them to require less space in the chip and increased speed, compared to the same blocks designed by the user.

Clock

As seen in the picture above there is a clock needed, as most of the parts of the FPGA are digital and need to be synchronous. In most FPGAs on top of the global clock there are local clocks that are used to drive regional networks for clock and reset to have a minimal skew. Complex designs can have more that one clock to serve different operations such as RAM.

Area

The functions that can be implemented in a FPGA are constrained by the logic available on the FPGA. So, the FPGA vendors try to create more space by new techniques such as 3D Integrated Circuits, where different FPGA chips are stacked to create a bigger chip [Wu15] , FPGA chips combination [Sab12] and implementing chips with smaller transistors (7nm and less) to stack more of them in a single chip.

Programming

There are two ways for the user to program or define the design that will be implemented on the FPGA. The first one in Hardware Description Languages, such as Verilog and VHDL and the second one the schematic design. However, due to the fact that programmers want to make development easier, the schematic design is nearly obsolete as it is very difficult to draw on paper the number of transistors or gates needed to implement a complex design. Schematic design is only used for verification and design understanding and is automatically produced after the HDL development. The workflow for the implementation of a program in FPGAs consists of 4 steps. To begin with, the developer must develop the program on a HDL or a higher level language that then generates a technology-mapped netlist by an electronic design automation. This netlist is then fitted into the FPGA by a process called place-and-route that uses most of the time a FPGA company's proprietary place-and-route software. In this step there must be defined the model of the FPGA. Next, the user can evaluate the results of the place-and-route based on the timing analysis, simulation or other verification techniques. If and when this process is completed and verified, there is generated a binary file, which then is transferred to the FPGA via a serial interface or to an external memory device and is used to configure the FPGA. To make the program development more efficient, as it is the biggest drawback of the FPGA technology, there are many libraries of predefined complex functions and circuits, which have been verified for speed. These circuits are called IP Cores and are available from FPGA vendors and other third-party IP suppliers. Moreover, there are IDEs that are mostly developed by FPGA vendors that the programmer can use a higher level language to use a more abstract program design and make the development simpler. [Xil18] Finally, Amazon has created a marketplace, where a developer can subscribe, use Xilinx's high level development tools to run programs on Amazon's FPGAs or rent IP Cores or FPGA Programs for specific tasks and time and pay

in relevance to the time he uses the FPGAs. [Ama18]

1.2.1 History

FPGAs were a development of programmable read-only memory (PROM) and programmable logic devices (PLDs) which where programmed in batches in factory or in the field as the connections between their elements where hard-wired. [Cra15] The first FPGA was created in 1983 at Burroughs ASG and patented. [DWP83; Pag83] Altera delivered the first reprogrammable logic device in 1984 called the EP300. This device was programmed by shining in its die which had an quartz window an ultraviolet light that erased the needed EPROM cells and configured the device. [YFCW09] In 1992, Steve Casselman developed a computer that could implement 600.000 reprogrammable gates. After the implementation of FPGAs from Altera, which was acquired by Intel in 2015, and Xilinx, these companies continued to thrive and became the biggest FPGA vendors in the market. In 1990s there was a rapid growth in the FPGA industry and back then these devices were mostly used in telecommunication and networking applications, while in the end of the decade the were introduced to a wider set of applications such as consumer automotive and industrial applications. In parallel with hardware, hardware description languages were developed. Verilog, the first modern HDL was introduced in 1985 by the Gateway Design Automation. In 1987, VHDL was introduced by the U.S. department of Defense and was based on the Ada programming language. Those two languages were used in the beginning to document and simulate circuit designs that were already defined in other forms, such as schematic files. The HDL simulation enabled engineers to work in a higher level that schematic simulations and increased the design capacity of transistors at least a level of gratitude.

1.2.2 Applications

FPGAs due to their versatility, latency, performance, connectivity, engineering cost and energy efficiency are used in numerous sectors and for various reasons. Examples of the sectors that use FPGAs are Aerospace and Defense, Consumer Electronics, Industrial Electronics, etc. There are particular sectors such as military, space and astronomy applications that FPGAs are the best solution for low latency. The other main FPGA application is prototyping and testing a design before it is implemented in an ASIC which will be more energy efficient and less expensive to fabricate. The latest years there have been created new ways to utilize the FPGA's advantages. Those contain embedded systems, high performance computing and AI/ML applications. The usage of FPGAs has a steady increase the latest years but the FPGA market is expected to grow in its highest level due to the introduction of Heterogeneous FPGA systems for acceleration of enterprise workloads. The energy efficiency, the improvements in performance, as well as, the special demands of some applications for fixed point operations make FPGAs a very competitive solution to GPUs for computing. [res18; vdP18; Kat18]

1.3 Maxeler

The solution proposed by this thesis utilizes the Maxeler products and the Multiscale Dataflow Programing model that is introduced by the company. Maxeler is a company based in UK and has created products that utilize FPGA and specific interconnection between FPGA chips to create the platform that is used for accelerating applications in multiple sectors, such as Finance, Government, Science, Health Engineering and Security.

Maxeler's Multiscale Dataflow Computing combines the traditional synchronous dataflow with vector and array processors. Loop level parallelism is exploited in a spatial, pipelined way, where large streams of data flow through multiple arithmetic units, connected in such way to compute a specified task.

1.3.1 Dataflow Engines (DFEs)

On the Maxeler architecture, the Memory is decoupled from the logic. The part that is responsible for the computations is called Dataflow Engine. Dataflow Engines are in fact FPGAs that are programed by the program the developer designs and implement the dataflow data graph described in the kernel. DFEs consist of multiple Dataflow Cores which are basically blocks that perform certain computations. The data that are fed to the DFE by memory data streams pass through these Cores to create the stream of output data produced by the DFE.
Figure 1.6 attempts to depict the architecture of a DFE, as well as the flow of a program running on a DFE, an an illustrative manner.

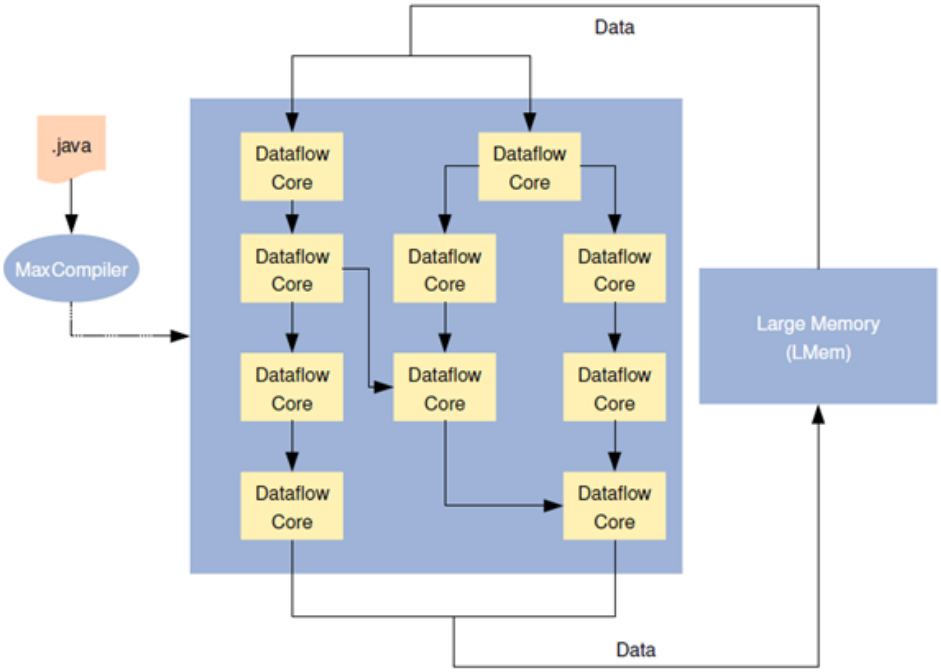


Figure 1.6: Dataflow program in action
[Tec17]

DFEs have two types of memory to use. The first one is a big one is called LMEM and is a DDR3 off-chip RAM close to the FPGA chip with gigabytes class of storage. The second one is called FMEM and is a small on-chip memory that consists from FPGA BRAMs with size of some megabytes and terabytes/second access bandwidth. The FMEM is a key factor that enables the Maxeler Dataflow Architecture to achieve great performance as there is a big versatility on how it can be used, either for pipelining or for saving data that are used multiple times inside a kernel. This hierarchy has a lot similarities with computer RAM and CPU caches. Furthermore, it is possible to connect multiple DFEs in a supercomputing system with MaxRing interconnect. The MaxRing interconnect allows applications to scale linearly with multiple DFEs in the system while supporting full overlap of communication and computation.

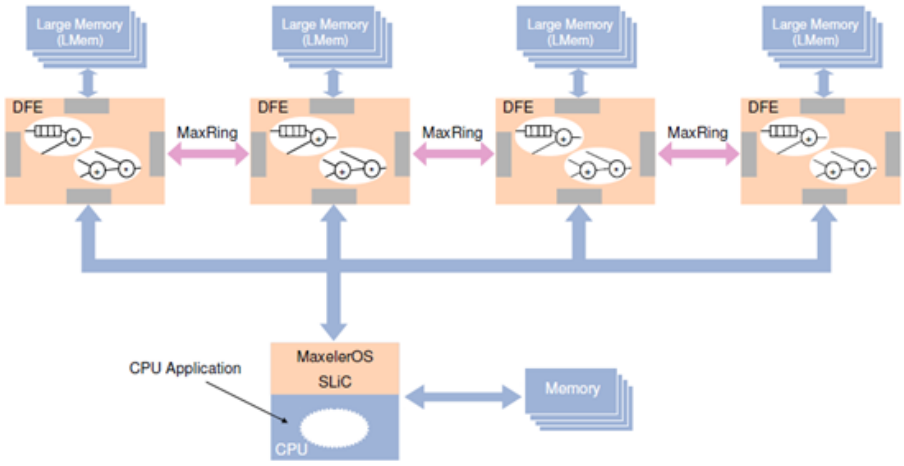


Figure 1.7: Maxeler Dataflow System Architecture
[Tec17]

1.3.2 Maxeler Dataflow Programing

When creating a Dataflow program it is needed to have in mind that what is basically created is a graph that has as input data, does computations based on them and creates data as output based on the inputs. This design has a lot of differences with the classical computer architecture where there is a list of instructions that should be serial executed (or almost serial executed) and some of them may not executed also based on the inputs of the program, which can be randomly accessed in the multiple level of memory in the computer (caches, RAM, disks, etc). In DFEs the computing in time is transformed to a computation in space. All the data that need processing are fed into the DFE in every tick (not exactly every clock cycle) and results are created by the DFE in every tick, being processed in a pipeline that is created by the graph generated by the design of the program. This means that there is no need for instruction decode logic, so all the logic inside the FPGA is utilized for data processing. An example kernel and the produced kernel graph is shown below:

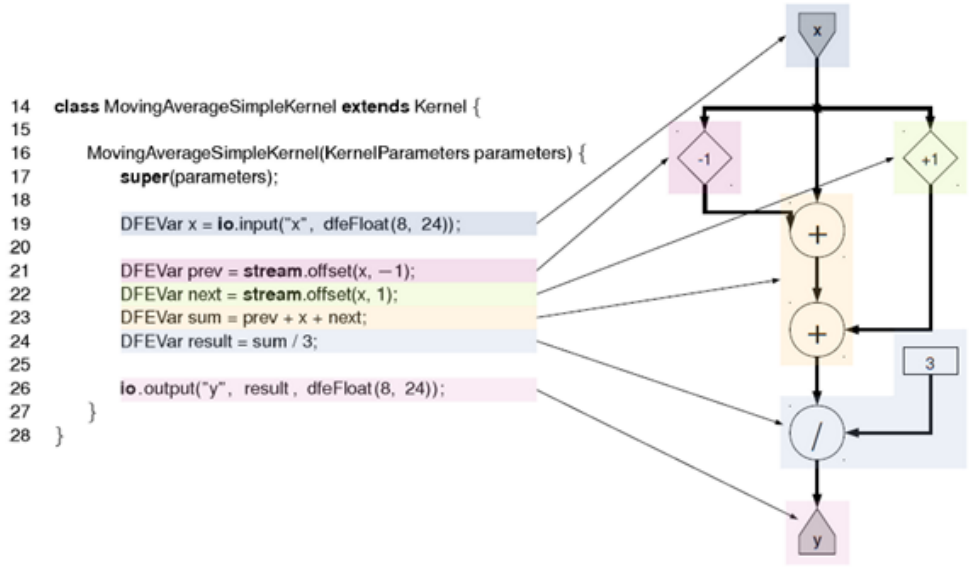


Figure 1.8: Moving Average Kernel
[Tec17]

The pipeline of this kernel is visualised in the next image.

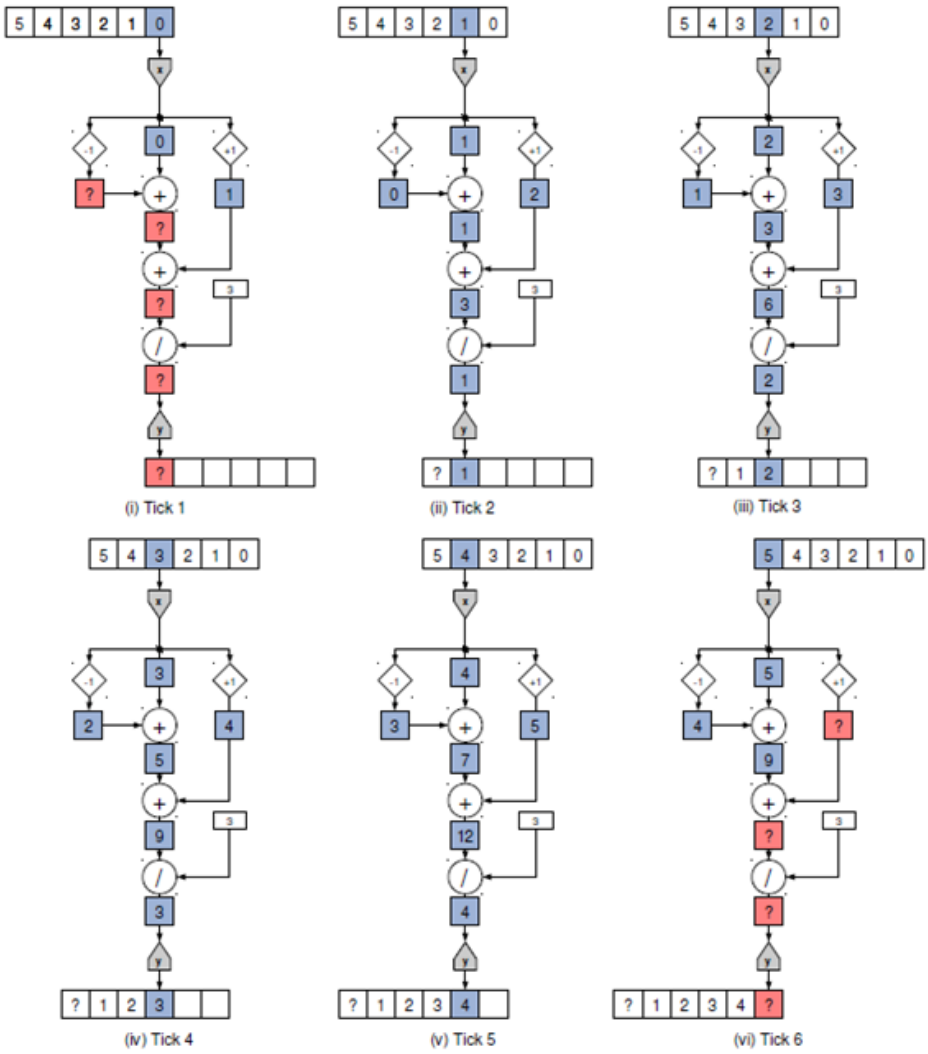


Figure 1.9: Moving Average Kernel Pipelining
[Tec17]

A Maxeler program consists of a CPU Code which runs on the cpu (C, MATLAB, Python, R), a Manager and the Kernel (Maxeler Java API). The CPU Code is responsible to loading the dataflow implementations via the SLiC (Simple Live CPU) interface. The SLiC is automatically generated to the corresponding dataflow program. The Manager is related to a given Kernel and defines the inputs and outputs of this kernel, as well as, the methods of the SLiC interface and the DFE characteristics such like optimizations, clock and memory frequency. The Kernel is the design which is translated into FPGA Design. The Kernel is written in a Java-like API created by Maxeler. The code in this high level language is then translated to code that can be used as input to the High Level Synthesis tool of the vendor of the FPGA chip that is used to configure the FPGA. In this way, the development time is decreased a lot as a lot of details of Verilog and VHDL are encrypted and automatically generated by the MaxCompiler.

Chapter 2

Related Work on Accelerated Computational Neuroscience

Computational Neuroscience is a branch of Neuroscience which uses mathematical models, theoretical analysis and abstractions of the brain to understand the principles that govern the development, structure, physiology and cognitive abilities of the nervous system. Due to the fact that many of the theorems, experiments and scientific hypotheses cannot be confirmed and checked through in vivo or in vitro experiments, many experiments are needed to run in silico, meaning simulations that run in computers.

As it was mentioned before, the brain and in general the nervous system is very complex and contains hundreds of billions of cells that are organized in various ways in three dimensions. These facts, in addition to the complexity of the description of their activity lead to very computationally heavy simulations. As the neuroscience domain develops and new, more detailed neuron models are being designed, these simulations become more and more heavy. This means that the research in neuroscience that is based in silico experiments is constrained by the computational power and the simulation tools available to the researchers.

The high computational demands and the simulation tools lead to close cooperation of neuroscience research groups with computer scientists, as the first ones need the expertise of the latter ones to be able to run the experiments they want and continue their research in more detailed and complex networks and neuron models. A lack of a generalized or unified simulation tool creates a vast variety of simulation tools and architectures that support different models, contain more or less details about the connections between neurons, and analyse different neuron's behavior. The urge for high performance tools and simulation techniques has widen the range of the available tools, frameworks and technologies of simulations. There are numerous improvements suggested for existing simulators or new ones with acceleration done utilizing multiple CPUs, threads, GPGPUs and even more specialized systems like ASICs and FPGAs.

To address the needs of neuroscientists to have easy to use simulators and not depend on computer engineers to run their experiments, there have been developed versatile frameworks that are able to simulate numerous neuron models and with different connections

between them. The big advantage in the usage of these simulator frameworks is that they need little programming knowledge, as they encrypt a lot of implementational details, they can run to every personal computer or workstation and they have a universal way to define the parameters of more or less detail neuron models, as well as their connections and the input to their network. This enables neuroscientists to get used to a specific framework and accelerate the time needed to setup their experiments. The drawbacks of this type of simulators is that due to the huge variety of supported neurons and networks, they do not have the optimal performance for every type of neuron network and they are not optimized for particular high performance computer architectures in their standard distributions. Some of the most famous neuron simulator frameworks are: NEURON, NEST, BRIAN, MOOSE and GENESIS. On top of them small experiments or new experimental models are simulated in general programing tools such like MATLAB.

Due to the high computational needs of large neuron networks some of those simulators, such as NEURON and NEST, have been extended to utilize multiprocessor architectures or GPUs to accelerate the simulations run on them, while they keep all the advantages of standardization of the frameworks that neuroscientists currently work on.

Furthermore, the latest years there are more examples of specific HPC cloud services that are used by neuroscientists to run their experiments on. These services serve specific neuron models and networks simulations and are based on their own frameworks. However, this means that each one of them runs optimized simulations based on the hardware that they utilize.

2.1 FPGA Simulators

To address the needs of further acceleration as the simulations of neuron networks become more complex, there are initiatives to try and utilize more specialized platforms to extract from them greater performance. A promising technology that is used for specified simulations and in which this thesis focuses on is FPGAs, that Maxeler DFEs utilize. Some examples of usage of FPGAs for accelerated simulations are referenced below.

The Artificial Intelligence and Machine Learning development have generated a need for accelerated Spiking Neural Networks (SNNs) which are used in AI/ML algorithms. A team consisting from people from the key Laboratory of OptoElectronic Science and Technology for Medicine of Ministry of Education and the College of Photonic and Electronic Engineering of the Fujian Normal University has developed an FPGA toolbox which is used as a library in the MATLAB environment that can implement different types of synaptic plasticity, Neuron Integrate-and-fire models, Dynamic threshold functions, Encoding blocks and Learning rule blocks to be used in SNN simulations. All those capabilities can be utilized in a simulation by importing them from a library as black boxes. These black boxes are then translated by the Xilinx System Generator to HDL Code and then implemented into the FPGAs. [QWJC15]

A team consisted by the Department of Electrical and Computer Engineering of Sungky-

unkwan University of Suwon, Korea and the DMC RD Center of Samsung Electronics in Suwon, Korea has built a simulator for 1000 spiking Neurons and 1 million synaptic connections in real time. The great achievement of this simulator is that it uses the Izhikevich Neuron model, which is almost as accurate as the Hodgkin-Huxley model which can biologically represent every type of biological neurons. However, the Izhikevich model needs less hardware to be utilized due to the simpler differential equations that is defined by. The results of this simulator had shown that a large level Izhikevich model simulation is possible. [JCK15]

Another large-scale simulation with conductance-based spiking neural networks using a real-time digital neuromorphic system has also been developed. This system utilizes a scalable 3-D network-on-chip topology with six Altera Stratix III FPGA chips to simulate 1 million neurons of a detailed large-scale cortico-basal ganglia-thalamocortical loop. The novel router architecture and the cost-efficient conductance based neuron model that has saved a large amount of HW resources is what enabled this system to outperform other state of the art systems, using CPUs, GPUs or other neuromorphic systems. The advantages of this proposal to the others is higher computational speed, better scalability and superior biological accuracy and reconfigurability in contrast to other neuromorphic systems. [SYKAL17]

Neuromorphic systems using FPGA have a potential of very high efficiency and computational power. Another suggestion was made for an FPGA-Based Massively Parallel Neuromorphic Cortex Simulation with a neuromorphic architecture that is based on the structural connectivity of neocortex to store all the required connections and parameters in on-chip memory using minicolumns and hypercolumns. This simulator can be easily reconfigured to simulate different neural networks and using the Altera Stratix V FPGA chips it could simulate 20 million to 2.6 billion leaky-integrate-and-fire neurons in real time.[RMWS18]

Apart from straight FPGA implementations, there have also been made implementations of different neuron models and simulations in Maxeler DFEs.

One of them is a simulation of the Inferior-Olivary nucleus brain region. Due to the complexity of these neurons, simulations can become rapidly intractable when there are biophysically plausible models and meaningful network sizes. A suggestion to address this problem is the usage of a Maxeler Dataflow Computing Machine. To achieve great performance the FMEM of the system is used to store the values of the Neurons and also there is parallelization in the level of neurons. This system has enabled the simulation in real-time speed of a 330-cell network and an acceleration of x92-107 in comparison to a Xeon processor and x2-8 to a pure Virtex-7 FPGA implementation due to the high memory bandwidth and max throughput of 24.7 GFLOPS. Moreover, due to the fact that multiple DFEs can be interconnected, the acceleration in comparison with a single FPGA can be even greater. Furthermore, in this particular real-time simulation the neuron network was x3.4 larger than the FPGA port of the simulation. [GS14]

Another simulator that uses Maxeler Dataflow Engines is Neuroflow. This simulator uses Izhikevich model for neurons and STDP. It also uses a PyNN interface to configure

the processor. The results of this simulator is simulating a 600k network and achieving an acceleration of x33.6 in comparison with an 8-core processor and x2.83 in comparison with GPU. All the spikes, the neuronal parameters and the synaptic weights are read and written to the LMEM of the DFE. [KCL16]

2.2 BrainFrame

This thesis was generated due to my interest in Computational Neuroscience and my initiative to help with the Brainframe project. Brainframe is a node-level accelerator platform for neuron simulations. The more and more complex neuron models and the bigger and bigger neuron networks make simulations require a lot of time and computational power. To cover these demands, the Brainframe utilizes an Intel Xeon-Phi CPU, a NVidia GP-GPU and a Maxeler Dataflow Engine to accelerate neuron simulation workloads. More precisely, the platform chooses between them, which one is the most efficient platform to run a simulation defined by neuroscientists. This accelerates simulations while keeping the energy footprint as low as possible. Moreover, the unification of the definition of the simulations' parameters through PyNN makes the platform usage very familiar to neuroscientists and accelerates further the process of deploying a simulation in the platform. PyNN is a simulator-independent language for building network models. In other words, a developer or neuroscientist can write the code for a model once using the PyNN API and the Python programming language and then run it without any modification on not only in Brainframe but also in numerous other simulators, such as NEURON, NEST and Brian. Last but not least, Brainframe is a platform on development to include more neuron models, platforms and features to improve its usability by neuroscientists. [GS17; APDY09]

Chapter 3

Problem Statement

In the process of enriching Brainframe with new neuron models, there was a need to import an Adaptive Exponential Integrate-and-fire (AdEx) neuron model in a network with Spike-timing Dependent Plasticity (STDP). The models of AdEx neurons and STDP where imported by a paper about the ability of synapses to memorize a specific behaviour and quickly recall said behaviour if similar spikes activate them. [RPC15] This paper has used the Brian Simulator and the PyNN interface to define the models of the neurons and the synapses and to simulate this system. To use this model to Brainframe and to have some acceleration, it was needed in the beginning to import the simulation with these particular models to a custom C program. To do this me and a fellow student have decoded how the Brian Simulator, which is based in Python, worked and designed a simulator on C to solve this particular type of simulation. After creating the C program, then it was my responsibility to try and accelerate it using Maxeler DFEs. During this process there where various architectures tried that will be analyzed later. First, the models of AdEx Neurons and STDP should be defined, as well as, how the Brian simulation was translated into a C program.

3.1 Adaptive Exponential Integrate-and-fire (AdEx) Neuron Model

The Adaptive exponential integrate-and-fire model, also called AdEx, is a spiking neuron model with two variables. The first equation describes the dynamics of the membrane potential and includes an activation term with an exponential voltage dependence. Voltage is coupled to a second equation which describes adaptation. Both variables are reset if an action potential has been triggered. The combination of adaptation and exponential voltage dependence gives rise to the name Adaptive Exponential Integrate-and-Fire model. The adaptive exponential integrate-and-fire model is capable of describing known neuronal firing patterns, e.g., adapting, bursting, delayed spike initiation, initial bursting, fast spiking, and regular spiking. Introduced by Brette and Gerstner in 2005, [BG05] the Adaptive exponential integrate-and-fire model AdEx builds on features of the exponential integrate-and-fire

model [NFTB03] and the 2-variable model of Izhikevich [Izh03]. The differential equations describing the AdEx model are:

$$\frac{dV_m}{dt} = \frac{g_L(E_L - V_m) + g_L\Delta_T \exp(\frac{V_m - V_T}{\Delta_T}) + I - x}{C}$$

$$\frac{dV_T}{dt} = -\frac{V_T - V_{Trest}}{\tau_{V_T}}$$

$$\frac{dx}{dt} = \frac{c(V_m - E_L) - x}{\tau_w}$$

V_m: Membrane potential

x: Adaptation Variable

I: Input Current

C: Membrane Capacitance

g_L: Leak Conductance

E_L: Leak Reversal Potential

V_T: Threshold

Δ_T: Slope Factor

c: Adaptation Coupling Parameter

τ_w: Adaptation Time

3.2 Spike-Timing Dependent Plasticity (STDP)

Spike-Timing Dependent Plasticity (STDP) is a temporally asymmetric form of Hebbian learning induced by tight temporal correlations between the spikes of pre- and postsynaptic neurons. As with other forms of synaptic plasticity, it is widely believed that it underlies learning and information storage in the brain, as well as the development and refinement of neuronal circuits during brain development. [BP01; SP08] The neural substrate of learning is believed to be long-term synaptic plasticity and after years of research and debate, it has become more clear that it can be expressed as pre- or postsynaptic or both. The functional consequences of the division between pre- and postsynaptic plasticity are yet to be studied and the paper in which this thesis is based on tries to do exactly that. To be more precise, there was developed a biologically tuned spike-timing dependent plasticity model that involves both parts of the stdp expression. [RPC15]

Inspired by earlier work, this phenomenological model relies on exponentially decaying traces of the pre- and postsynaptic trains, X and Y.

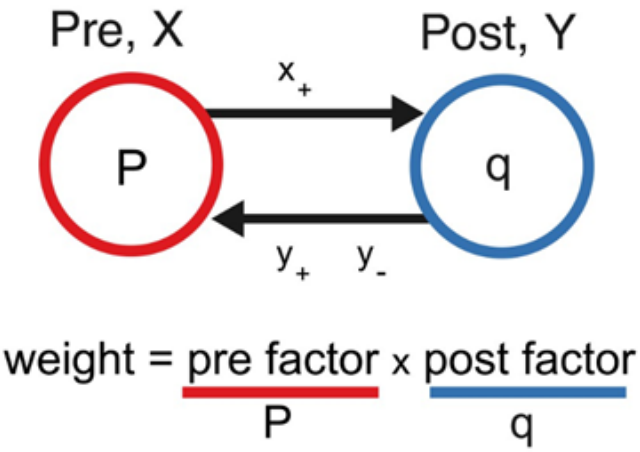


Figure 3.1: STDP weight model
[RPC15]

The synaptic weight is the product of a presynaptic factor P and a postsynaptic factor q. The presynaptic trace x_+ tracks past presynaptic activity, for example, glutamate binding to postsynaptic NMDA receptors. When presynaptic activity x_+ is rapidly followed by postsynaptic spikes, unblocking NMDA receptors, postsynaptically expressed long-term potentiation (LTP) is triggered and increases the postsynaptic factor q, which can be interpreted as the quantal amplitude. Conversely, the postsynaptic trace y_+ represents prior postsynaptic activity, for example, retrograde nitric oxide (NO) signalling, which when paired with presynaptic spikes leads to presynaptically expressed LTP. Finally, the trace y_- tracks postsynaptic activity such as endocannabinoid (eCB) retrograde release and elicits presynaptically expressed long-term depression (LTD) when coincident with presynaptic spikes. Presynaptically expressed plasticity is conveyed by long-term changes in the presynaptic factor P, which can be interpreted as the presynaptic release probability. The differential equations that describe this model are divided into Short- and Long-term plasticity. The equations of the Short-term plasticity are the following:

$$\frac{dr(t)}{dt} = \frac{1 - r(t)}{D} - p(t)r(t)X(t)$$

$$\frac{dp(t)}{dt} = \frac{P - p(t)}{F} + P[1 - p(t)]X(t)$$

$$X(t) = \sum_{t_{pre}} \delta(t - t_{pre})$$

r: (Normalized) Number of Vesicles

p: Presynaptic Factor

D: Depression Time Constant

P: Baseline Presynaptic Factor

F: Facilitation Constant

The equations for the Long-term plasticity model are:

$$\frac{dy_-(t))}{dt} = \frac{-y_-(t))}{\tau_{y_-}} + Y(t)$$

$$\frac{dy_+(t))}{dt} = \frac{-y_+(t))}{\tau_{y_+}} + Y(t)$$

$$\frac{dx_+(t))}{dt} = \frac{-x_+(t))}{\tau_{x_+}} + X(t)$$

y_+, y_-: Postsynaptic Traces

x_+: Presynaptic Traces

The postsynaptic factor q is modified with every postsynaptic spike Y according to:

$$\Delta q = c_+ x_+(t) y_-(t - \epsilon) Y(t)$$

The presynaptic factor p is modified with every presynaptic spike X according to:

$$\Delta P = -d_- y_-(t) y_+(t) X(t) + d_+ x_+(t - \epsilon) y_+(t) X(t)$$

The total synaptic strength is a product of both pre- and postsynaptic factors

$$w(t) = qp(t)r(t)$$

For a synapse that has not stimulated recently this simplifies to

$$w = Pq$$

3.3 Brian Simulator

Brian is a free, open source simulator for spiking neural networks. The goal of this simulator is to encrypt the details of the implementation of a simulation from neuroscientists, so that they can focus on the details of the models of neurons and synapses that are used in a simulation. The simulator is written in Python and this makes it even more friendly to entry level researchers or students that occupy themselves with computational neuroscience. To run a simulation in Brian, all they have to do is define using the Brian API the input of the network, the network of Neurons, the differential equations that describe them in mathematical form, how the neurons are interconnected (synapses) and what differential equations govern the behavior of synapses, and finally what variables of the system the want to plot, if needed. It also uses vector-based computation for efficient simulations. Furthermore, it supports the PyNN interface to keep a unified interface with other neuroscientific simulations and models. [GB09]

3.4 Import to C program from Python

To further accelerate the simulation but also to add it to the brainframe platform there was a need to create a C program that produced the exact same results with the Brian simulation. For this part of the thesis, we cooperated with a fellow student to understand how Brian simulator handles the network and solves the differential equations that describe the parts of the network.

3.4.1 Brian Architecture

The main parts of the Brian simulation are the definition of the Input Neurons which only produce spikes and don't follow a specific physical model, the definition of the Neurons that needed studying and the Synapses that connect those two.

Input Neurons

To define the Input Neurons of the simulation there have been studied two classes. The first one is PoissonGroup and is a Neuron Group class. This class is used to create the network of input neurons that spike in random moments based on a poisson distribution. The parameters for instantiating this class were:

- N : (int) Number of Input Neurons
- rates : (Quantity) Single rate, array of rates of length N, or a string expression evaluating to a rate

The second class used was SpikeGenerator which produces a network of input neurons that spike based on an array given as input that defines specifically which input neurons create spikes and exactly when. The parameters for instantiating this Neuron Group class are:

- N : (int) Number of Input Neurons
- spiketimes : array of (int, int) Array of tuples that contain in the first part the ID of a neuron and in the second the timestep that it produces a spike

Neurons

The class that creates the instances of Neurons that follow a specific model is called NeuronGroup and is a Neuron Group class. The parameters that is called with are:

- N : (int) Number of Neurons
- model : (Equations/string/StateUpdater) This parameter states the model that defines the behavior of the neurons that are created by this class initializer. It can be either an Equations object that has created by a string of mathematical equations given in a particular form, a string of equations or a State Updater class

- threshold : (Threshold object/function/scalar quantity/string) This parameter defines the state of the neuron that when it becomes true there is a spike generated by a particular neuron
- reset : (Reset object/function/scalar quantity/string) This parameter defines what happens when a neuron spikes
- freeze : (True/False) If True, parameters are replaced by their values at the time of initialization

Synapses

The Synapses class is used to set the synapses between two neuron groups and is able to encapsulate the plasticity phenomena. The way this class works is that connects two Neuron Groups based on the parameters that are given to it. Then, during the simulation, it updates the values of the produced Synapses based on the activity of the two Neuron Groups. The parameters that this class is instantiated with are:

- source : (Neuron Group) The source Neuron Group
- target : (Neuron Group) The target Neuron Group
- model : (Equations object/string) The equations that defined the synaptic variables. Same syntax as the Neuron Group
- pre : (list/tuple of strings) The code executed when presynaptic spikes arrive at the synapses
- post : (list/tuple of strings) The code executed when postsynaptic spikes arrive at the synapses

Concluding, the code needed to run the simulation in Brian is:

```
eqs_neuron = """
        dvm/dt=(gL*(EL-vm)+gL*DeltaT*exp((vm-vt)/DeltaT)+I-x)/C : volt
        dvt/dt=-(vt-vtrest)/tauvt : volt
        dx/dt=(c*(vm-EL)-x)/tauw : amp #In the standard formulation x is w
        I : amp
    """
neurons = NeuronGroup(M, model=eqs_neuron, threshold='vm>vt',
                      reset="vm=Vr;x+=b;vt=VTmax", freeze = True)
InitializeNeurons(neurons)
my_input = SpikeGeneratorGroup(N, spiketimes)
model='''w : 1
         FFp : 1
         FBp : 1
         FBn : 1
         R : 1
         u : 1
         U : 1
         A : 1
         dFFp/dt=-FFp/tau_FFp : 1 (event-driven)
```

```
            dFBp/dt=-FBp/tau_FBp : 1 (event-driven)
            dFBn/dt=-FBn/tau_FBn : 1 (event-driven)
            dR/dt=(1-R)/tau_r : 1 (event-driven)
            du/dt=(U-u)/tau_u : 1 (event-driven)
        '''
syn = Synapses(my_input, neurons, model,
               pre='''I=s*A*R*u;
                   U=clip(U+etaU*(-AFBn*FBn*FBp + AFBp*FBp*FFp),Umin,Umax);
                   w=U*A;
                   FFp+=1; R-=R*u; u+=U*(1-u)''',
               post='''A=A+etaA*(AFFp*FFp*FBn);
                   A=A-etaA*0.5*mean(AFFp*FFp*FBn);
                   A=clip(A,Amin,Amax);
                   w=U*A;
                   FBp+=1.;FBn+=1.''')
InitializeSynapses(syn)
run(stime)
```

After creating and initializing the network, the simulator solves automatically the differential equations of the models that define synapses and neurons and creates the code that needs running in every timestep or every time an event is generated and keeps it as a string in every class. Then this code is run for every part of the class has to based on the model of each one. The Neuron Group class keeps all the IDs and the variables of neurons in addition to the code that solves its DEs. The same is happening with Synapses. Furthermore, the Neuron Group Classes keep the neurons that have spiked and need to be propagated to the synapses class to update the needed synapses.

3.4.2 C Program Architecture

To begin with, there must be made some declarations that will help understanding the next chapters. The simulation coming from the ModelDB project that was given to us for learning the AdEx model and the specific STDP model to use was a simulation that connected **N** Input-Dummy Neurons (Neurons that produce only spikes in specific times, given by the user) with **M** AdEx Neurons (Neurons that are governed by the AdEx model equations). From now on, when number **N** is referred, it means the Input Neurons and when number **M** is referred, it will mean the AdEx Neurons. Furthermore, we have added to this type of simulation (**NxM**) the ability to run an experiment connecting **M** AdEx Neurons to **M** AdEx Neurons. These two type of simulations are different and the selection of which one of the will run is done by the inputs given by the user. The user has to define 3 variables that have to do with the number of Neurons:

- **N_S**: Number of Input Neurons
- **N_Group_S**: Number of AdEx Neurons used as source (presynaptic)
- **N_Group_T**: Number of AdEx Neurons used as target (postsynaptic)

So, the **NxM** simulation is a **N_SxN_Group_T** simulation (N_Group_S = 0) and the **MxM** simulation is a **N_Group_SxN_Group_T** (N_S = 0) simulation.

From the study of how the Brian simulator works, we came up with 3 functions that are run in every timestep and each one represents a different action-phenomenon of the AdEx model and the STDP.

To be able to show the specific code of each function in C, we will have to mention first the data structures that keep all the data of the Neurons and the Synapses.

- Input Neurons are saved as an array of ints with length **N** which represent if the Input Neuron i produced a spike in the timestep **t**.

- AdEx Neurons are saved as an array of structs Neuron with length **M**. The struct Neuron consists of all the variables an AdEx Neuron has, based on our model.

```
1    typedef struct {
2      double vt;     /**< Voltage threshold. */
3      double vm;     /**< Membrane potential.   */
4      double I;    /**< Neuron input current. */
5      double x;    /**< Adaption variable (w). */
6      int Spike;     /**< Variable to show if the neuron has spiked in a
   specific moment. */
7    } Neuron;
8
```

- Synapses are saved in an adjacency array of size **NxM** or **MxM**, depending on the type of simulation, of structs Synapse that consists of all the variables of a Synapse. This means that the Neurons that represent the lines are connected with the Neurons representing the columns. If a connection between these two doesn't exist, the variable conn is zero. The connections are read by an input file and the initialization of all the variables is done accordingly.

```
1    typedef struct {
2        int conn; /**< Variable that expresses if a given synapse
   between two neurons exists. */
3        double w; /**< Weight of a synapse. (Was present in BRIAN but
   never used for the ADEX with STDP simulation.) */
4        double FFp;    /**< FFp variable of a synapse. (x+) */
5        double FBp;    /**< FBp variable of a synapse. (y+) */
6        double FBn;    /**< FBn variable of a synapse. (y-) */
7        double R;    /**< R value of a synapse. (r)  */
8        double u;    /**< u value of a synapse. (p)  */
9        double U;    /**< U value of a synapse. (P)  */
10       double A;    /**< A value of a synapse. (q)  */
11       double lastupdate;    /**< Last time a synapse was updated   */
12       double target_I;  /**< The I value for the postsynaptic neuron.
   */
13   } Synapse;
14
```

The first function of the simulation is responsible for the update of the values of AdEx Neurons only, by solving their differential equations.

```
1 void SolveNeurons(Neuron* neurons, int N, int *SpikeArray)
```

The second function of the simulation is responsible for updating the values of the synapses based on the differential equations describing the presynaptic expression of STDP and the value of the current of the postsynaptic neuron.

```
void UpdateSynapses_pre(Synapse** Synapses, Neuron* neurons, int N_S, int
    N_Group_S, int N_Group_T, int* SpikeArray, double t)
```

In this function there are two nested for-loops. This is due to data dependencies, as the way the 2d array of synapses is accessed(by rows) is different of the way the neuron values are updated(by columns). In the DFE implementation however, this issue is solved due to a different access to the synapses array. A more abstract view of this function is described like this: For every Input/Source Neuron (i) that has generated a spike, the Synapses that start from it and the Neuron (j) that these Synapses end on are updated.

The third function of the simulation is responsible for updating the values of the synapses based on the differential equations describing the postsynaptic expression of STDP.

```
void UpdateSynapses_post(Synapse** Synapses, int N_S, int N_Group_S, int
    N_Group_T, int* SpikeArray, double t)
```

Here there are three nested for-loops. The first has to calculate the values FFp and FBn, before the value mean is calculated that is used to calculate the value A that later is used to calculate other values in the final nested loop. A more abstract view of this function is described like that: For every Target Neuron (i) that has generated a spike, the Synapses that end to it are updated.

The loop of the simulation looks like that:

```
for(int t = 0; t < timesteps; t++){
    SolveNeurons(neurons, N_Group_T, SpikeArray);
  InitializeSpikeArray(SpikeArray, N_S);
    UpdateSynapses_pre(syn, neurons, N_S, N_Group_S, N_Group_T, SpikeArray, t
   *defaultclock_dt);
    UpdateSynapses_post(syn, N_S, N_Group_S, N_Group_T, SpikeArray, t*
   defaultclock_dt);
}
```

Data Dependencies

For every function above I have mentioned the data dependencies and why there are different loops in each one of them. Moreover, the values of AdEx neurons should be calculated first to get the neurons that generate a spike. Then the spikes of the Input neurons should be calculated to get the input spikes, used by the UpdateSynapses functions.

Chapter 4

Implementation on DFEs

After having implemented the simulation in C, the next step was developing the simulation to run on DFEs and checking the performance gains of this implementation. As stated earlier, the Maxeler kernel generation is based in a creation of a Kernel Graph by the Kernel the programmer designs. This means that the Kernel Graph is seeded with data that pass through the operators defined in the Kernel and in every tick (different to clock cycle), there is an output produced by the Kernel. To make this possible, the Graph implements a pipeline, so that in every tick, new data are seeded into the pipeline and generated in the output. This special architecture means that the data structures that have been used in the C program must be altered a bit to be able to address the DFE architecture and feed the kernel graph with data.

4.1 Data Structures in DFE

This architecture demands an alternate representation of our data in the memory to be able to efficiently feed the kernel with data. To do this, we have to access the array of Synapses by rows, exactly as it is saved in the memory by the C language. This means that in the case of the UpdateSynapses_pre function, the first nested for should access memory not by columns but by rows, the same way with the UpdateSynapses_post function. However to do this, it is needed a change between the columns and the rows of the synapses array.

When in C the representation was the following with access by rows:

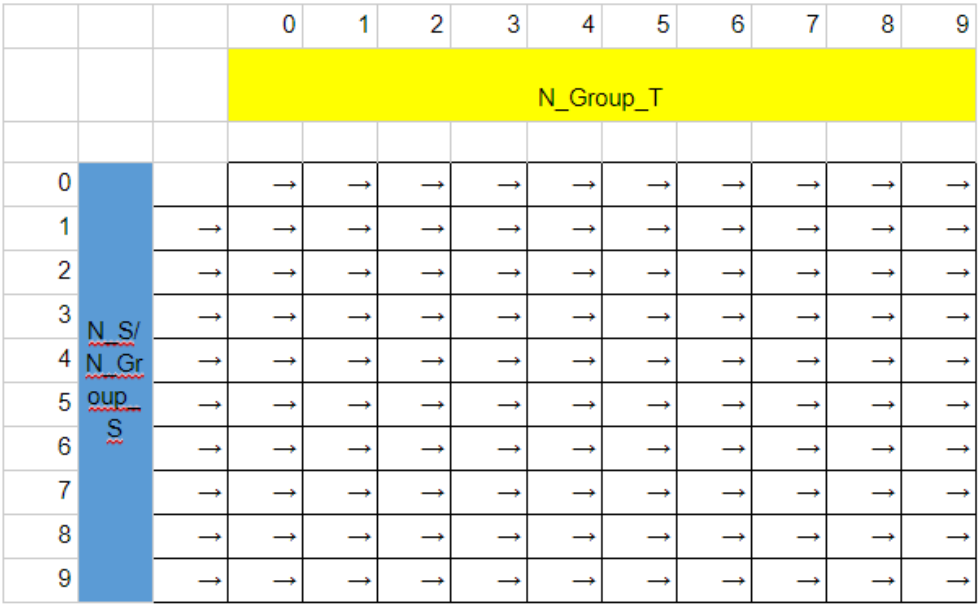


Figure 4.1: C Synapses Array Access

in the DFE implementation, we have to save to the DFE LMEM the synapses data as it is shown below

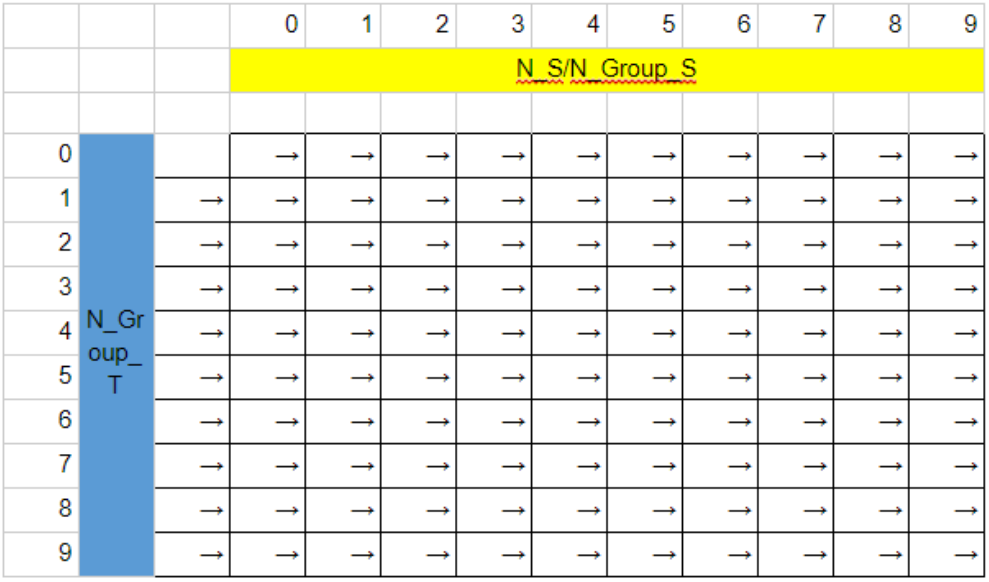


Figure 4.2: DFE Synapses Array Access

Furthermore, due to the difficulty of passing structs in the DFE the struct of AdEx Neurons was alternated to a vector of 6 variables, that however is still an array of doubles/floats.

For example:
If we have 2 AdEx Neurons and we want to access the vm variable of the 2nd one we would

have:
C implementation pseudocode:

```
struct Neuron Neurons[2];
double vm2nd = Neurons[1].vm;
```

DFE implementation pseudocode:

```
double Neurons[2*6];
double vm2nd = Neurons[1*6+1];
```

The indexing of the new array organization is based on the order of variables in the Neuron struct. The vector is abstractly defined by the way we access the variables in the program. The number 6 is used for word alignment in the DFE, even if the variables of the Neuron Struct are 5 and are all doubles or floats, depending on the implementation. The difference in C and DFE Implementation for the organization of Neuron data is shown below:

	N_Group_T									
	0	1	2	3	4	5	6	7	8	9
C	struct Neuron	struct Neuron	struct Neuron	struct Neuron	struct Neuron	struct Neuron	struct Neuron	struct Neuron	struct Neuron	struct Neuron
DFE	array[6]	array[6]	array[6]	array[6]	array[6]	array[6]	array[6]	array[6]	array[6]	array[6]

Figure 4.3: Neuron Representation in C and DFE

The same representation of the struct Synapses was used in the Synapses array. Every Synapse Struct is transformed in an array of 12 doubles/floats, again to be word aligned, even if they are 11 variables in each struct.
C:
Size: N_S*N_Group_T*sizeof(struct Synapse)

		N_Group_T									
		0	1	2	3	4	5	6	7	8	9
0	N_S/N_Group_S	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse
1		struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse
2		struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse
3		struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse
4		struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse
5		struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse
6		struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse
7		struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse
8		struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse
9		struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse	struct Synapse

Figure 4.4: Representation of Synapses in C

DFE Implementation:
Size: N_Group_T*N_S*12*sizeof(double or float)

			0	1	2	3	4	5	6	7	8	9
			N_S/N_Group_S									
0	N_Group_T		array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]
1			array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]
2			array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]
3			array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]
4			array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]
5			array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]
6			array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]
7			array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]
8			array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]
9			array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]	array[12]

Figure 4.5: Representation of Synapses in DFE

The array for the Input Neurons is transformed to an array of length N*timesteps, which has a byte in each element and if it is 1 then that means that the input neuron has generated a spike a particular timestep. This array is generated in the beginning of the simulation in the CPU Code for the whole simulation. If for example we want to see if Input Neuron 5 generates a Spike in the 2nd timestep we have to access the element SpikeArray[1*N_S+4]. Finally, the DFE has 2 types of memory. A big one (DDR3 RAM) in the MAX4 with 48GB called LMEM and a small one that is in the FPGA chip (BRAMs) which has a size of some megabytes only. In the DFE, all the arrays are saved in the LMEM in the begging of the CPU Code and fed from the LMEM to the Kernel due to their large size. To save time, the Input Neuron array is saved when needed in the FMEM, to be faster accessed. Moreover, the fixed variables of the STDP and AdEx differential equations are saved in the FMEM in the beginning of the kernel.

There were more thought about how the network could be represented in the memory, however for the given sizes and the functionality needed, this is the most efficient in terms of performance and memory footprint. An improvement would be saving the AdEx Neurons to the DFE FMEM. However, this would make the network of AdEx neurons a lot smaller and the gains would only come from faster access to them from the Kernel, with the same representation for the Synapses.

4.2 Kernel Architecture

The most important part of the thesis is the Kernel design. For the Kernel, there have been various considerations regarding the design of the whole simulation and what is possible due to all the dependencies that are imposed by the model of the Neurons and the Synapses and the connectivity of the network. All the design choices were made having in mind the

best functionality of the program, meaning that it would be able to simulate a large Neuron network and all the possible connections between them and the Input Neurons. Furthermore, the kernel design was done in such way that takes advantage of the DFE architecture and the parallelization that is possible in the simulation. Taking into consideration all these, the main design choices were between using one kernel or multiple kernels that would be loaded and unloaded in each step, similarly to the C program and the usage of float variables for all the data of the Neurons and Synapses or double variables, as the C program and the Python simulator.

4.2.1 Multiple Kernels

The first step to start creating a working simulation on DFEs was to develop three different kernels that implement the 3 distinct functions of the CPU Code that are executed in every timestep. This means that the whole simulation in the DFEs would have the same design as in the CPU Code but instead of the calls to the functions SolveNeurons, UpdateSynapses_pre and UpdateSynapses_post, there would be calls to the corresponding kernels and some data transfers to the main memory between these calls. While it was known that this probably wouldn't be the most efficient solution, it was a good begging to get familiar with the Maxeler tools and Dataflow Programming before trying to create a kernel that would run the whole simulation.

The development started with creating a kernel for each function to better understand the way DFEs work and get familiar with MaxCompiler and Maxeler Dataflow Programing. The first attempts were about just calculating the correct values and find the best ways to pass data into the dataflow graphs that are defined by the kernels. After finding the most efficient way to do all the calculations inside the kernel and checking the kernels for errors, then the main program that instrumentates the whole simulation should be developed. The flowchart of the simulation, in this implementation, follows:

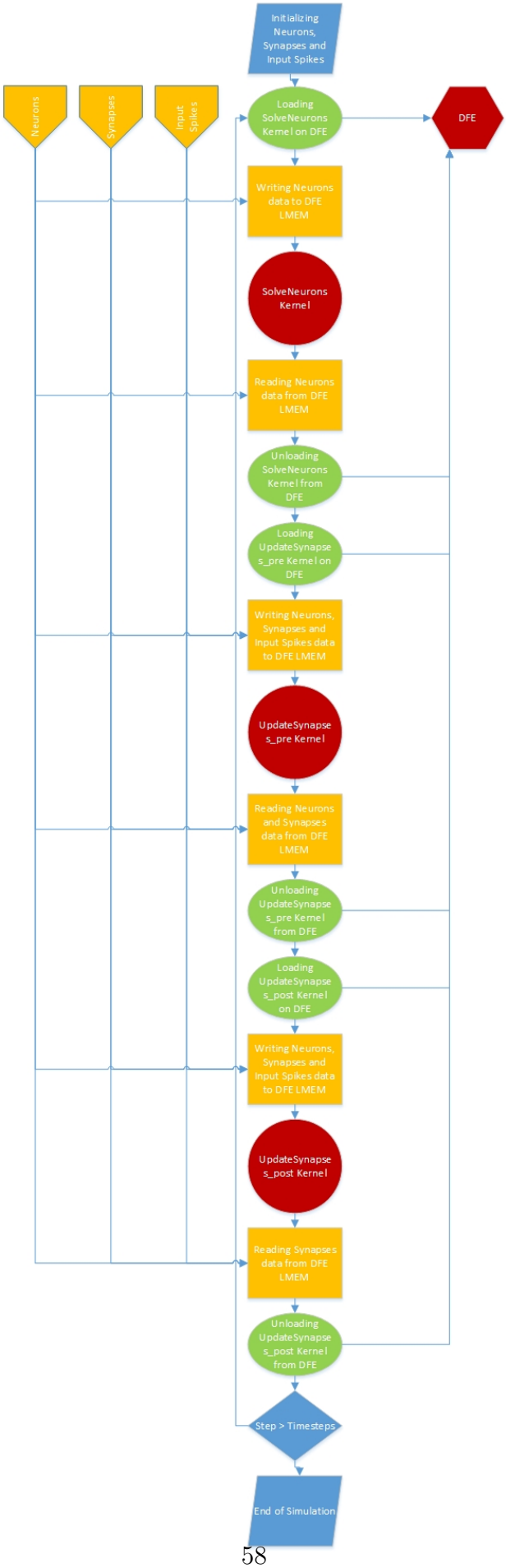


Figure 4.6: Multiple Kernels Flowchart

In this chapter I will elaborate on the kernel's design and the optimizations that were tried in each of them. For every kernel I will mention the needed code for setting up the Manager and the key points of the Kernel. The Manager provides a predictable input and output streams interface to the Kernel. Manager provides a Java API for configuring connectivity between Kernels and external I/ and controls the build process.

SolveNeurons Kernel

As the name suggests, this kernel is responsible for updating the values of the neurons' variables. This kernel is executed in every timestep, before any other function. As in the C function, in the first implementation of this Kernel, all the Neurons are passed into the created Kernel Graph from the LMEM, one by one, and their updated values are created and saved back again to the LMEM.

Manager

For this implementation there was used a Standard Manager. Standard Manager supports only one Kernel. In the beginning, there is created a Kernel Block that includes the Kernel, which is instantiated by the the Engine Parameters the developer has defined. These parameters have to do with the Kernel configuration.

```
1 SolveNeuronsEngineParameters params = new SolveNeuronsEngineParameters(args);
2 Manager manager = new Manager(params);
3 Kernel kernel = new SolveNeuronsKernel(manager.makeKernelParameters(
    s_kernelName));
4 manager.setKernel(kernel);
```

Then the streams that the Kernel utilizes are set. More precisely, the Stream names and their destination is set. There are some standard Destinations such as, IODestination.CPU, which creates a stream from/to the CPU Code, IODestination.LMEM_LINEAR_1D, which creates a stream to the LMEM. In this particular Kernel, there are used 4 streams. Two streams from the CPU that provide the Kernel with the AdEx model parameters and the STDP model parameters. This is the snipper of this part of the code:

```
1 manager.setIO(
2   link("adex_params", IODestination.CPU),
3   link("stdp_params", IODestination.CPU),
4   link("x", IODestination.LMEM_LINEAR_1D),
5   link("x_o", IODestination.LMEM_LINEAR_1D));
```

Then, the SLiC Interface of the manager is set. The SLiC Interface contains the definitions of the functions that are present in the CPU Code. Basically, it creates the API needed to call the Kernel from the CPU Code or make specific calls to write to or read from the LMEM.

```
1 manager.createSLiCinterface(interfaceDefault());
```

The interfaceDefault defines all the parameters that are passed from the CPU Code to the Kernel or from the Manager to the Kernel. In the beginning there must be created a new EngineInterface with the EngineInterface() constructor.

To add a new parameter in the interface there is a standard call engine_interface.addParam("name", VariableType) that adds the parameter named "name" of type VariableType to the engine_interface and consequently to the Kernel call. Manager provide a set of predefined variable types that can be used. Those are defined by the CPUTypes class and can be INTs of 8,16,32 or 64 bits, UINTS 8,16, 32 or 64 bits, FLOATs, DOUBLEs or VOID. The call of the addParam function creates an InterfaceParam object. This object apart from a parameter that comes from the CPU Code, it can be a constant that is created with the function addConstanstant of the engineInterface class or simply an operation on other InterfaceParam objects.

To pass the variables from the Manager to the Kernel, there must be used the engine_interface.setScalar(s_kernelName, "Name", interfaceParameter) function. This passes a single variable to the Kernel that must be in the Kernel read in a similar way. To connect the Kernel to a CPU Stream there is the function setStream("stream_name", CPUTypes.Type,size_of_stream_in_Bytes). This stream is unidirectional and the direction is defined later in the Kernel.

To read or write to the LMem there are numerous functions that read the LMem in different ways. The function setLMemLinear("name", start_addr, size_in_Bytes) uses a simple linear pattern to access the LMem for writing or reading. The function setLMemWrapped("name", start_addr, arr_size, whole_size, offset) wraps around the LMem starting from start_addr and continues for arr_size, before returning back to start_addr+offset until whole_size bytes are read from the LMem. The function setLMemBlocked is a more detailed function and is called to access in a 3D way the memory. To access a Memory as a 2D array there is the function setLMemStrided("name", start_addr, size_Fast, size_Slow, stride_Mode). For this implementation the functions that were used to read and write to the LMem the Neuron data are:

```
1 engine_interface.setLMemLinear("x", zero, M * 6 * size);
2 engine_interface.setLMemLinear("x_o", zero, M * 6 * size);
3 and for the Adex and Stdp variables:
4 engine_interface.setStream("adex_params", CPUTypes.DOUBLE, N_adex * CPUTypes.
    DOUBLE.sizeInBytes());
5 engine_interface.setStream("stdp_params", CPUTypes.DOUBLE, N_stdp * CPUTypes.
    DOUBLE.sizeInBytes());
```

These functions are standard and they take care of when all the data are written or read from the data and how to create interrupts to notify every needed system.

To define the number of ticks a kernel runs for, the function setTicks("name", ticks); is used. Finally, there is a function used in the EngineInterface that is used to ignore on the Kernel call from the CPU Code every other variable defined by the interface apart from the ones we defined ourselves.

After the creation of the SLiC interface there are some definitions that need to be done to configure the Kernel build.

```
configBuild(manager, params);
manager.build();
```

For this implementation there were defined the stream status debugger as false, the number of cables run to create the executable, the parallelism in the build process and the parameter for retrying near cable misses. The API for these operations is:

```
manager.setEnableStreamStatusBlocks(false);
BuildConfig buildConfig = manager.getBuildConfig();
buildConfig.setMPPRCostTableSearchRange(params.getMPPRStartCT(), params.
    getMPPREndCT());
buildConfig.setMPPRParallelism(params.getMPPRThreads());
buildConfig.setMPPRRetryNearMissesThreshold(params.getMPPRRetryThreshold());
```

The setEnableStreamStatusBlocks lets the kernel keep debugging data for the kernel if it is true. All the other parameters take their values from the KernelNameEngineParameters class object.

Kernel

To begin with, as described before, it is needed to define the datatype of the Neuron data. This is a 6 double variable array.

```
DFEVectorType<DFEVar> vectorVars =
        new DFEVectorType<DFEVar>(dfeFloat(11,53), 6);
```

dfeFloat(exponent, mantissa) is used to define a particular floating point variable.

After this definition, a counter is created to address the FMEM which keeps the variables of the AdEx and the STDP model that are read from the CPU and are of Double type. The AdEx variables are in reality 14, however for byte alignment when passing them from the CPU Code to the DFE, we have to create a stream of 16*sizeofdouble size. The same applies for the STDP variables. Instead of 24, the stream size is 32*sizeofdouble. The first 16 ticks of the kernel one variable of the AdEx model and one variable of the STDP model are read, while for the next 16, the remaining STDP model variables are read and saved to the FMEM. After the wrap of this counter, the stream of Neuron data are passed to the kernel. The 6 variables of a Neuron are read in each tick, their differential equations are solved and then the updated values are written back to the LMem in the same address that they were read from.

The code responsible for solving the DEs of the AdEx model is the following:

```
DFEVar _vm, _vt, _x;
_vm = (gL*(EL-vm)+gL*DeltaT*KernelMath.exp((vm-vt)/DeltaT)+I-x_var)/C;
_vt = -(vt-vtrest)/tauvt;
_vt.simWatch("vt_c");
_x = (c*(vm-EL)-x_var)/tauw;
DFEVar vm_o = vm + _vm * defaultclock_dt;
DFEVar vt_o = vt + _vt * defaultclock_dt;
DFEVar x_o = x_var + _x * defaultclock_dt;
DFEVar I_o = I; // I must be read from memory in every timestep
DFEVar Spike_o = Spike;
```

```
DFEVar control = vm_o > vt_o;
vm_o = control ? Vr : vm_o;
 x_o = control ? x_o + b : x_o;
vt_o = control ? vtmax : vt_o;
Spike_o = control ? constant.var(1).cast(double_type) : constant.var(0).cast(
    double_type);
```

All these operations are translated to a data flow graph by the MaxCompiler and the MaxCompiler creates automatically a pipeline for every one of these variables, so that in each tick the 6 variables of the neuron enter the pipeline and in every tick the 6 updated variables are produced. All the calculations are done in parallel on the hardware and this is one of the reasons why the FPGA throughput is high.

CPU Code

To run the simulation there is also needed a CPU Code which is written in C. This code runs on the CPU of the Maxeler platform and is responsible for all the initializations of the data structures that are passed to the Kernel, as well as, loading and unloading the DFE and calling the Kernel.

To run this particular Kernel, the needed code is:

```
x = InitializeNeurons(N); // Initializatin of Neurons array (x) with size N
SolveNeurons_writeLMem(0, NeuronSizeBytes, x);  // Writing the Neuron array
    to LMEM
SolveNeurons(M, N, adex_param_size, stdp_param_size, steps, adex_params,
    stdp_params); // Kernel call
SolveNeurons_readLMem(0, NeuronSizeBytes, x); // Reading the updated values
    of the Neurons
```

As it is implied from all the above, this call solves the neuron differential equation for a single timestep. This means that this Kernel must be called in every timestep to solve this part of the simulation.

Optimization

The optimization introduced to this Kernel was parallelization in the level of Neurons. This means that in every tick, there where 2 or more Neurons processed by the Kernel. As a result, the Kernel ticks where divided by the Unroll Factor used and less time was needed for the same simulation.

For an Unroll Factor=2, it is needed to read in the beginning of each tick the variables for two neurons, instead of one, and then do the calculations for both of them in parallel in hardware. Of course, this is only possible when there is hardware in the FPGA of the DFE available. For checking if this optimization was possible, it was needed to check the hardware utilization of the simple kernel.
FINAL RESOURCE USAGE
Logic utilization: 99743 / 262400 (38.01%)

Primary FFs: 156000 / 524800 (29.73%)
Secondary FFs: 6772 / 524800 (1.29%)
Multipliers (18x18): 100 / 3926 (2.55%)
DSP blocks: 50 / 1963 (2.55%)
Block memory (M20K): 596 / 2567 (23.22%)
As seen from the resource usage report from the compilation of the Kernel, there is room for at least an Unroll Factor of 2. The report for HW usage from this Kernel is:
FINAL RESOURCE USAGE
Logic utilization: 148481 / 262400 (56.59%)
Primary FFs: 231194 / 524800 (44.05%)
Secondary FFs: 10328 / 524800 (1.97%)
Multipliers (18x18): 200 / 3926 (5.09%)
DSP blocks: 100 / 1963 (5.09%)
Block memory (M20K): 772 / 2567 (30.07%)
While it seem like there is room for an Unroll Factor of 4, when tried the HW usage was almost 100% and even if the Kernel compiled, the performance was worse than that of Unroll Factor 2.

This meant that the optimal Kernel for this function calculated two Neurons per tick. The transformation of the part of the code that calculate the differential equations of Neurons is:

```
DFEVectorType<DFEVar> VarVector =
      new DFEVectorType<DFEVar>(dfeFloat(11,53), UnrollFactor);

DFEVector<DFEVar> _vm = VarVector.newInstance(this);
DFEVector<DFEVar> _vt = VarVector.newInstance(this);
DFEVector<DFEVar> _x = VarVector.newInstance(this);
for(int i = 0; i < UnrollFactor; i++){
_vm[i] <== (gL*(EL-vm[i])+gL*DeltaT*KernelMath.exp((vm[i]-vt[i])/DeltaT)+I[i
    ]-x_var[i])/C;
  _vt[i] <== -(vt[i]-vtrest)/tauvt;
  _x[i] <== (c*(vm[i]-EL)-x_var[i])/tauw;
}
DFEVector<DFEVar> vm_o = VarVector.newInstance(this);
DFEVector<DFEVar> vt_o = VarVector.newInstance(this);
DFEVector<DFEVar> x_o = VarVector.newInstance(this);
DFEVector<DFEVar> I_o = VarVector.newInstance(this);
DFEVector<DFEVar> Spike_o = VarVector.newInstance(this);
for(int i = 0; i < UnrollFactor; i++){
  vm_o[i] <== vm[i] + _vm[i] * defaultclock_dt;
  vt_o[i] <== vt[i] + _vt[i] * defaultclock_dt;
  x_o[i] <== x_var[i] + _x[i] * defaultclock_dt;
  I_o[i] <== I[i];
  Spike_o[i] <== Spike[i];
}
DFEVector<DFEVar> control = VarVector_int.newInstance(this);
for(int i = 0; i < UnrollFactor; i++){
  control[i] <== vm_o[i] > vt_o[i];
}
```

```
28 DFEVector<DFEVar> vm_o_res = VarVector.newInstance(this);
29 DFEVector<DFEVar> x_o_res = VarVector.newInstance(this);
30 DFEVector<DFEVar> vt_o_res = VarVector.newInstance(this);
31 DFEVector<DFEVar> Spike_o_res = VarVector.newInstance(this);
32 for(int i = 0; i < UnrollFactor; i++){
33         vm_o_res[i] <== control[i] ? Vr : vm_o[i];
34         x_o_res[i] <== control[i] ? x_o[i] + b : x_o[i];
35         vt_o_res[i] <== control[i] ? vtmax : vt_o[i];
36         Spike_o_res[i] <== control[i] ? constant.var(1).cast(double_type) :
    constant.var(0).cast(double_type);
37 }
```

Apart from the input and output this is the only part of the kernel that needs changes. In general, what changes is that instead of applying a computation in a single variable, computations are applied in an array of variables of length Unroll Factor. This is also why the resource usage in Unroll Factor = 2 is not exactly double from the simple Kernel.

The difference of performance between these two versions may not be substantial but it is measurable for even a very small number of neurons (384):
Simple Kernel: 0.0199640 s
Unroll Factor = 2: 0.0195800 s

UpdateSynapses_pre Kernel

This Kernel implements the UpdateSynapses_pre function that is written in the CPU Code. This Kernel has to read and update not only the Synapses, but also the AdEx Neurons. In addition, the Kernel should run for both the NxM and the MxM simulations. This means that there should be a way to choose if it has to read the values of Input Spikes or the AdEx neurons. All these features impose a big complexity in the development of the Kernel.

Manager

The differences from the previous kernel is that this one uses a Custom Manager. The use of the Custom Manager is to link manually all the streams to the Kernel and to be able to create new interfaces of the SLiC interface to read and write particular parts of the memory that store different data types.

As before, there are used LMemLinear Streams to read the array of Synapses and the neurons that correspond to every row of the Synapses' array. Apart from them, there is used another LMemLinear Stream to read the Input Spikes array from the LMem. Furthermore, to read the AdEx Neurons multiple times to check their spikes in the MxM simulation, there is used another LMemWrapped stream that reads multiple times the address where the Neurons are saved in the LMem.

The sizes of the streams that are read and if it is needed to stream the Neurons for the MxM simulation or the Input Spikes for the NxM array is defined by the number of input neurons and the number of AdEx neurons that are passed from the CPU Code. If one of them is zero, the streams don't pass any data to the Kernel, and the Kernel is designed not

to read anything from the related stream.
Kernel

The Kernel processes one Synapses at each step of the kernel. One step of the Kernel is calculated by the stream.makeOffsetAutoLoop("loopLength") function, which automatically calculates the ticks needed to do all the calculations inside the Kernel and update the needed values to produce outputs at every tick. The reason why this offset in this Kernel is greater than 1 is the update of the AdEx Neuron that needs to be read, updated and saved again to the FMEM. During the runtime, the array of synapses is accessed by rows. Each row corresponds to a Target (Postsynaptic) AdEx Neuron which is read in the beginning of the row and is kept to the FMem by an internal stream that is produced to update its I value. For this feature, there was used a function stream.offset(x_o,-loopLength), to read the value of the Neuron loopLength ticks earlier, which is the value needed to be updated. Another, way to do the same thing is to save the needed to value to a specific address of the FMEM and then to read it from there, change it and save the new one to the same address, to be updated in the same way again if needed. However, this was was checked to be slower, as the loopLength was greater and this meant that each step of the Kernel would take more time. To synchronize all the streams of the Kernel there is used a chain counter with two variables. The slow one is the number of Target AdEx Neurons (the number of rows of the Synapses array) and the fast one (the number of columns of the Synapses array) is the number of the Input Neurons for the NxM simulation or the number of AdEx Source (Presynaptic) Neurons. These variables are used to choose which one of the streams should be read. Furthermore, to try and make less accesses to the slower LMem, the Input Spikes which are represented by an Int variable which is 1 if there is a spike and 0 else, are saved when they are first read from the LMEM to the FMEM to be quickly read when needed. The Synapses, as stated before, are represented by an array of 12 doubles, which are read in every step of the kernel.
CPU Code

As this Kernel needs different types of data, there must be distinct functions to write data to the LMem. To be more precise, there is a new function writeLmemInt that writes the Input Spikes to the LMem. In general, the Neurons' array, the Synapses' array and the Input Spikes' array are written in continuous addresses of the LMem, starting from 0.
Optimization

In this Kernel there weren't any more optimizations possible, as while loop unrolling was possible as there were FPGA resources available, then the Kernel needed a bigger loopLength, as the streams could not be synchronized and some of them stalled and stopped the kernel run, which made the unrolling less efficient.
UpdateSynapses_post Kernel

This Kernel is responsible to make the computations of the UpdateSynapses_post function of the C program. This is the most time consuming part of the whole simulation, as this Kernel reads the whole Synapses Array 2 times. The first one to update some of the values of Synapses and calculate a mean of an expression based on some of the Synapses' variables and the second one to update the values of the Synapses based on this mean. Due to the complexity and the importance of this Kernel there were multiple architectures tries to find the optimal implementation. The exploration of multiple architectures was not only done to check the most optimal one, but also to try different techniques and get more familiar with Maxeler Dataflow Programming and DFEs.

Version 1

The first version is probably the less efficient. It is based on the thought of having two Kernels which would be serially loaded and run in the DFE. The first one would make all the computations needed in the first pass of the Synapses' array, including calculating the mean and updating the values of the Synapses. The second one would have as input the mean that was previously calculated and would use it to do all the remaining calculations on the Synapses. The two Kernels were distinctly developed and were later joined to implement the UpdateSynapses_post function.

The first Kernel uses a Standard Manager. The Neurons, Input Spikes and Synapses are read from the LMem in the same way as the UpdateSynapses_pre Kernel. This Kernel has also got a loop due to the sums that are calculated and needed for the final calculation of the mean variable. The loopLength is again calculated automatically by the MaxCompiler. This means, that there is a Synapse processed every loopLength ticks. The addition on the sum and num variables is done like before, using the stream offset architecture.

```
1 DFEVar carriedSum = double_type.newInstance(this);
2 DFEVar carriedNum = double_type.newInstance(this);
3 DFEVar sum_t = m === 0 & n === 0 ? constant.var(0) : carriedSum;
4 DFEVar num_t = m === 0 & n === 0 ? constant.var(0) : carriedNum;
5 sum_t = x[4] === 1 & syn_0[0] === 1 ? sum_t + AFFp * FFp * FBn : sum_t;
6 num_t = x[4] === 1 & syn_0[0] === 1 ? num_t + 1 : num_t;
7 carriedSum <== stream.offset(sum_t, -loopLength);
8 carriedNum <== stream.offset(num_t, -loopLength);
```

carriedSum and carriedNum are two variables saved inside the FPGA and this code creates a pipeline of length loopLength that is used to keep the corresponding values of these two variables. Due to the fact that these variables are of floating type, there is no other way to make the needed additions (there is an accumulator API but supports only fixed point variables and ints and does all the calculations in only 1 tick). Floating Point additions need 16 ticks, so the loopLength is 16 ticks. The variable mean is calculated in every tick and passed as a stream to the CPU Code. The last value of this stream is the correct one. The second Kernel uses again a Standard Manager and the Input Spikes, Neurons and Synapses are read in the same way as the first Kernel. This Kernel also takes as input a single scalar double value from the CPU Code, which is the mean that was calculated from the first Kernel. This Kernel doesn't have any loop, so each Synapse take

only one tick to be updated.

The interesting part of this version is how to load both of these Kernels to the DFE from the single CPU Code. To do this, the following steps must be followed to build the executable: Let's say that UpdateSynapses_post_1 is the name of the project for the first Kernel, UpdateSynapses_post_2 is the name of the project for the second Kernel and UpdateSynapses_post is the name of the combined project that implements the UpdateSynapses_post C function.

1. Build for DFE both Kernels
2. Create a new project in MaxIDE for the UpdateSynapses_post function
3. Copy all the files from UpdateSynapses_post_1_kernel/RunRules/DFE/maxfiles and UpdateSynapses_post_2_kernel/RunRules/DFE/maxfiles to UpdateSynapses_post/RunRules/max
4. Edit the UpdateSynapses_post/RunRules/Makefile.settings file, so that the following values are defined as below:

```
1    RUNRULE_MAXFILES     := UpdateSynapses_post_1.max
    UpdateSynapses_post_2.max
2    RUNRULE_MAXFILES_H   := UpdateSynapses_post_1.h UpdateSynapses_post_2
    .h
3
```

5. Copy UpdateSynapses_post_1/RunRules/DFE/include/UpdateSynapses_post_1.h and UpdateSynapses_post_2/RunRules/DFE/include/UpdateSynapses_post_2.h to UpdateSynapses_post/RunRules/DFE/include and include those two header files in UpdateSynapses_post/RunRules/DFE/include/Maxfiles.h
6. Include Maxfiles.h file in CPU Code
7. Run the following commands in terminal:

```
1    cd UpdateSynapses_post/CPUCode
2    source ''{$MaxCompilerDirectory}/settings.sh\''
3    make RUNRULE=''DFE''
4
```

Apart from those steps, the Kernels should be distinctively loaded and unloaded to and from the DFE. To do this, there are some functions that need to be called from the CPU Code.

```
1 // Initialization of DFE
2 max_engine_t *myDFE;
3 // Initialization of UpdateSynapses_post_1_Kernel
4 max_file_t *Post1KernelMaxFile = UpdateSynapses_post_1_init();
5 // Load UpdateSynapses_post_1 Max File to the DFE
6 myDFE = max_load(Post1KernelMaxFile,"*");
7 // Creation of an actions object as stated in the UpdateSynapses_post_1
     header file for writing to the LMem
8 UpdateSynapses_post_1_writeLMem_actions_t usp1_wL_Action;
```

```
usp1_wL_Action.param_address = 0;
usp1_wL_Action.param_nbytes = sizeBytes_d_var;
usp1_wL_Action.instream_cpu_to_lmem = x;
// Write to LMem the data stated in the actions object defined above
UpdateSynapses_post_1_writeLMem_run(myDFE, &usp1_wL_Action);
// Do the same for other data
usp1_wL_Action.param_address = sizeBytes_d_var;
usp1_wL_Action.param_nbytes = syn_size;
usp1_wL_Action.instream_cpu_to_lmem = syn;
UpdateSynapses_post_1_writeLMem_run(myDFE, &usp1_wL_Action);
usp1_wL_Action.param_address = sizeBytes_d_var + syn_size;
usp1_wL_Action.param_nbytes = InputSpikes_size;
usp1_wL_Action.instream_cpu_to_lmem = InputSpikes;
UpdateSynapses_post_1_writeLMem_run(myDFE, &usp1_wL_Action);
// In the same way definition of action object for running the
   UpdateSynapses_post_1 Kernel
UpdateSynapses_post_1_actions_t usp1Action;
usp1Action.param_M_S = N_Group_S;
usp1Action.param_M_T = N_Group_T;
usp1Action.param_N = N;
usp1Action.param_N_adex = adex_param_size;
usp1Action.param_N_stdp = stdp_param_size;
usp1Action.param_Steps = steps;
usp1Action.instream_adex_params = adex_params;
usp1Action.instream_stdp_params = stdp_params;
usp1Action.outstream_mean = mean;
// Running the action object for running the UpdateSynapses_post_1 Kernel
UpdateSynapses_post_1_run(myDFE, &usp1Action);
// Creation of an actions object as stated in the UpdateSynapses_post_1
   header file for reading from the LMem
pdateSynapses_post_1_readLMem_actions_t usp1_rL_Action;
usp1_rL_Action.param_address = 0;
usp1_rL_Action.param_nbytes = sizeBytes_d_var;
usp1_rL_Action.outstream_lmem_to_cpu = x;
// Running the reading from memory action object
UpdateSynapses_post_1_readLMem_run(myDFE, &usp1_rL_Action);
// Do the same for other data
usp1_rL_Action.param_address = sizeBytes_d_var;
usp1_rL_Action.param_nbytes = syn_size;
usp1_rL_Action.outstream_lmem_to_cpu = syn;
UpdateSynapses_post_1_readLMem_run(myDFE, &usp1_rL_Action);
// Unloading the UpdateSynapses_post_1 Kernel Max File from the DFE to load a
    new one
max_unload(myDFE);
// The same process is followed to load the DFE with the
   UpdateSynapses_post_2 Kernel, read and write to the LMem with the
   functions defined in its header file, run the UpdateSynapses_post_2 Kernel
    and unload it from the DFE
max_file_t *Post2KernelMaxFile = UpdateSynapses_post_2_init();
myDFE = max_load(Post2KernelMaxFile,"*");
UpdateSynapses_post_2_writeLMem_actions_t usp2_wL_Action;
usp2_wL_Action.param_address = 0;
usp2_wL_Action.param_nbytes = sizeBytes_d_var;
usp2_wL_Action.instream_cpu_to_lmem = x;
UpdateSynapses_post_2_writeLMem_run(myDFE, &usp2_wL_Action);
```

```
58 UpdateSynapses_post_2_actions_t usp2Action;
59 usp2Action.param_M_S = N_Group_S;
60 usp2Action.param_M_T = N_Group_T;
61 usp2Action.param_N = N;
62 usp2Action.param_N_adex = adex_param_size;
63 usp2Action.param_N_stdp = stdp_param_size;
64 usp2Action.param_Steps = steps;
65 usp2Action.instream_adex_params = adex_params;
66 usp2Action.instream_stdp_params = stdp_params;
67 usp2Action.param_mean = mean_fin;
68 UpdateSynapses_post_2_run(myDFE, &usp2Action);
69 usp2_wL_Action.param_address = sizeBytes_d_var;
70 usp2_wL_Action.param_nbytes = syn_size;
71 usp2_wL_Action.instream_cpu_to_lmem = syn;
72 UpdateSynapses_post_2_writeLMem_run(myDFE, &usp2_wL_Action);
73 usp2_wL_Action.param_address = sizeBytes_d_var + syn_size;
74 usp2_wL_Action.param_nbytes = InputSpikes_size;
75 usp2_wL_Action.instream_cpu_to_lmem = InputSpikes;
76 UpdateSynapses_post_2_writeLMem_run(myDFE, &usp2_wL_Action);
77 UpdateSynapses_post_2_actions_t usp2Action;
78 usp2Action.param_M_S = N_Group_S;
79 usp2Action.param_M_T = N_Group_T;
80 usp2Action.param_N = N;
81 usp2Action.param_N_adex = adex_param_size;
82 usp2Action.param_N_stdp = stdp_param_size;
83 usp2Action.param_Steps = steps;
84 usp2Action.instream_adex_params = adex_params;
85 usp2Action.instream_stdp_params = stdp_params;
86 usp2Action.param_mean = mean_fin;
87 UpdateSynapses_post_2_run(myDFE, &usp2Action);
88 UpdateSynapses_post_2_readLMem_actions_t usp2_rL_Action;
89 usp2_rL_Action.param_address = 0;
90 usp2_rL_Action.param_nbytes = sizeBytes_d_var;
91 usp2_rL_Action.outstream_lmem_to_cpu = x;
92 UpdateSynapses_post_2_readLMem_run(myDFE, &usp2_rL_Action);
93 usp2_rL_Action.param_address = sizeBytes_d_var;
94 usp2_rL_Action.param_nbytes = syn_size;
95 usp2_rL_Action.outstream_lmem_to_cpu = syn;
96 UpdateSynapses_post_2_readLMem_run(myDFE, &usp2_rL_Action);
97 max_unload(myDFE);
```

The timing results of this code when run on DFE for a 384x384 network for a single tiemstep are: DFE Runtime = 1.3056500 seconds
Kernel 1 init Time = 0.0015750 seconds
Kernel 1 load Time = 0.1491460 seconds
Write LMem Time = 0.0255410 seconds
Kernel 1 Time = 0.0455760 seconds
Read LMem Time = 0.0193310 seconds
Kernel 1 unload Time = 0.4137920 seconds
Kernel 2 init Time = 0.0014230 seconds
Kernel 2 load Time = 0.1474960 seconds
Write LMem Time = 0.0264010 seconds

Kernel 2 Time = 0.0190920 seconds
Read LMem Time = 0.0197460 seconds
Kernel 2 unload Time = 0.4363570 seconds
Sum Kernel Time = 0.0646680 seconds
Time for loading and unloading kernels = 1.149789 seconds

It is obvious that the time needed for loading and unloading the DFEs is much greater than the time that DFEs do computations. Due to the fact that loading and unloading the DFEs takes almost 1 second for only 2 Kernels suggest that this would not be a good solution for the final simulation, where there are 3 Kernels and the Simulation runs for thousands of timesteps.

Version 2

This version is based again in two Kernels, almost the same as before. However, in this version they are not loaded and unloaded from the CPU Code but they are working in parallel in the DFE. The only change in the Kernels is in the second Kernel, where the streams start to input data to the Kernel after the first Kernel has finished its computations. The connection between the two Kernel is done by a Custom Manager. The Custom Manager enables the creation of two Kernel Blocks that include the two Kernels mentioned above. This is done through the following API:

```
1 KernelBlock k1 = addKernel(
2   new UpdateSynapses_post_1Kernel(makeKernelParameters("
      UpdateSynapses_post_1Kernel")));
3 KernelBlock k2 = addKernel(
4   new UpdateSynapses_post_2Kernel(makeKernelParameters("
      UpdateSynapses_post_2Kernel")));
```

Then all the streams are manually connected to the kernel blocks. The Memory Streams in both Kernels point to the same LMem address. There is one additional stream for the mean that is calculated in the first Kernel to the second Kernel.

While this would be an interesting solution, it never worked for these two Kernels, as some streams stalled during runtime. However, this type of connection between Kernels was successful in other, less complex cases of Kernels.

Version 3

This version combines the two Kernels previously used in a single one. This saves time from loading and unloading kernels and is the most efficient solution. There is still the need to do a first pass of the Synapses array to update their values and calculate the mean and then pass the array again to update again the values of Synapses. This means that the Kernel contains a bigger loop were, in the first step the computations of the first Kernel are done and in the second step there are done the computations of the second Kernel. To do this and update the values two times from the same Kernel, the LMemWrapped type

of memory access sounded well. However, there were problem with that standard memory access patterns and the data that were read in the second loop were not updated from the first loop. This lead to using the Custom Memory Command Stream, which let the developer precisely choose which address of the LMem will be accessed.

Custom Memory Command Streams are command streams generated by the Kernel and passed to the Memory controller. They are responsible for addressing the LMem and the interrupts generated by the Memory Controller. The Memory Controller can read or write from or to the LMem in Bursts. Each DFE model has a particular Burst Size in Bytes. This also means that all the data in the LMem must be burst alligned. This burst size is passed to the Manager of the Kernel by the CPU Code. To define the Burst Size and be able to run an executable in various DFEs, there is an SLiC function called by the CPU. After the initialization of the Max File of the Kernel for running, the function that returns the Burst Size is max_get_burst_size(maxfile, NULL). The first argument is the specified Max File and the second is the number of the DFE. If it is NULL this means that it takes the value of the default DFE, as defined by the system variables. For the MAIA DFE that is used, the Burst Size is 384 Bytes. The Manager is also responsible for calculating the total number of bursts for accessing a part of the LMem, as well as, the number of words of a specified data type that can be represented in a single burst. These values will be used in the kernel to create the counters to synchronize the whole implementation, the Custom Command Memory Streams and the inputs and outputs. For example, the values that are passed to the Kernel for reading the Synapses and AdEx Neurons respectively are:

```
1 engine_interface.setScalar(s_kernelName, "totalBurstsSyn", (12*M_T*(M_S+N)*(
    double_size))/burstSize);
2 engine_interface.setScalar(s_kernelName, "wordsPerBurstSyn", burstSize/(
    double_size*12));
3 engine_interface.setScalar(s_kernelName, "totalBurstsX", M_T*6*double_size/
    burstSize);
4 engine_interface.setScalar(s_kernelName, "wordsPerBurstX", burstSize/(
    double_size*6));
```

In addition to this, to read the Synapses for example, it is needed to create a counter that will provide the Custom Memory Command Stream for the stream of the synapses with addresses.

```
1 CounterChain chain = control.count.makeCounterChain();
2 DFEVar burstCount = chain.addCounter(totalBurstsSyn,1);
3 DFEVar wordCount = chain.addCounter(wordsPerBurstSyn,1);
```

This counter chain is used to read a single Synapse array in every tick. Every wordsPerBurstSyn ticks, there is issued a Memory Command by the Custom Memory Command Generator that is passed to the Memory controller and returns to the Kernel buffer a burst of 384 bytes, coming from the Synapses data. The data in this way are accessed linearly, which is the most efficient pattern in the Maxeler architecture. After writing a burst of data to the Kernel buffer, then for wordsPerBurstSyn ticks, the data that are read from the Synapses stream are read from there. One Synapse array each tick.

The API of the Custom Command Generator to read the stream of the Synapses (Input

stream) is the following:

```
LMemCommandStream.makeKernelOutput("SynIncmdStream",
    control,                    // control
    burstCount,          // address
    constant.var(dfeUInt(8), 1),       // size
    constant.var(dfeUInt(1), 0),       // inc
    constant.var(dfeUInt(1), 0),       // stream
    constant.var(false));
```

The Command Stream generates a command when the control variable is true. The control variable is defined in such way to be true when the wordCount is 0.

For example:

```
DFEVar control = wordCount.eq(0);
```

The burstCount attribute of the makeKernelOutput function is the address of the LMem that the Memory Controller reads.

The stream to input the real data to the Kernel is similar with before. For example, for reading the Synapses' array:

```
DFEVectorType<DFEVar> SynapsesVector =
  new DFEVectorType<DFEVar>(dfeFloat(11,53), 12);
DFEVector<DFEVar> syn = io.input("SynIn", SynapsesVector, streamControl);
```

To correlate the data stream with the stream that is controlled by the corresponding Custom Memory Command Stream it is needed to define the connection in the Manager Interface.

```
KernelBlock k = addKernel(new UpdateSynapses_postKernel(makeKernelParameters(
    s_kernelName)));
LMemInterface iface = addLMemInterface();
DFELink SynIn = iface.addStreamFromLMem("SynIn",k.getOutput("SynIncmdStream")
    );
k.getInput("SynIn") <== SynIn;
```

The same things stand when there is need for writing to the LMem. The only thing that changes is the io.output for the data and the last attribute of the makeKernelOutput function, which generates an interrupt to the CPU Code, when all the bursts are read or written to the LMem.

AdEx Neurons are read from the LMem in the same way.

Of course, due to the complexity of the Kernel, the synchronization of the Data Streams and the Custom Memory Command Streams need more complex counters. Furthermore, due to time needed to access the LMem there was introduced in the second part of this Kernel which corresponds to the second Kernel of Version 1 a delay in the form of a loop. After experimenting, it was needed a loop of 4 ticks, between reading Synapses. To achieve this, there is a new set of counters introduced, that runs on the second part of the Kernel runtime and has different wrap points from the first counters which are used in the first part of the Kernel runtime that makes the computations of the first Kernel of the Version 1 implementation. The final runtime of the Kernel is: N_stdp + loopLength*(N+M_S)*M_T + loop2Length*(N+M_S)*M_T ticks.

This was the Version of the UpdateSynapses_post Kernel that was used in this implementation.

CPU Code of Multiple Kernels Implementation

After creating and building all the Kernels mentioned above, copying the produced files as described earlier to create a project that can call them all, it is time to work on the CPU Code that calls the kernels and controls the LMem reads and writes to implement the simulation. The pseudocode for the CPU Code is:

```
SolveNeurons_init();
UpdateSynapses_pre_init();
UpdateSynpases_post_init();
for(timesteps){
  load(SolveNeuronsMaxFile);
  writeToLMem(Neurons);
  SolveNeurons();
  readFromLMem(Neurons);
  unload(SolveNeuronsMaxFile);
  load(UpdateSynapses_preMaxFile);
  writeToLMem(Neurons);
  writeToLMem(Synapses);
  writeToLMem(InputSpikes);
  UpdateSynapses_pre();
  readFromLMem(Neurons);
  readFromLMem(Synapses);
  unload(UpdateSynapses_preMaxFile);
  load(UpdateSynapses_postMaxFile);
  writeToLMem(Neurons);
  writeToLMem(Synapses);
  writeToLMem(InputSpikes);
  UpdateSynapses_post();
  readFromLMem(Synapses);
  unload(UpdateSynapses_postMaxFile);
}
```

Of course, all the loads and unloads of the DFE, the writes to the LMem and the calls of the Kernels are done with the API shown above.

4.2.2 Single Kernel

To begin with, the Kernel has 3 different sections that need to be run sequentially in every timestep of the simulation due to data dependencies. I will distinguish each section in the C code and the DFE kernel by commenting where it starts and ends with a special colour to be able to understand more clearly the corresponding parts. In each section there is only one neuron or synapse read and updated.

Sections:

1. Reads the AdEx Neuron array, one neuron at a time, solves their differential equations and updates their values in the same addresses of memory that were read from. To do

this update in the same memory addresses, it is needed to create a Custom Memory Command Generator that generates the commands that are passed to the read/write commands of the kernel (io.input/io.output) for when to read/write something and in what address it should read/write from/to. This operation cannot be executed in a single tick, as the kernel stalls, so after experiments I have found out it needs 2 ticks. This means that the kernel should wait for 2 ticks in a loop, to be able to send and receive all the memory commands and to read and write the data. This section could be parallelized in the level of neurons, however this is not possible, due to high hardware utilization of the whole kernel that uses doubles. The whole length of this section in a single timestep is: N_Group_T*2 ticks, because data are read and written every 2 ticks.

2. Reads an AdEx neuron from the Neuron Array one by one, the Synapses array one by one and the Input Neurons array one by one and solves the differential equations of the UpdateSynapses_pre and a part of the UpdateSynapses_post function. To be more precise, in this section in the first tick of each row there are read a Neuron, a Synapse and the corresponding Input Neuron Spike or another Source AdEx Neuron. For the next N_S or N_Group_S loops of this section, the whole row of the Synapses' array that corresponds to the AdEx Neuron that was read in the beginning is processed. The process of this section contains not only the UpdateSynapses_pre function, but also a part of the UpdateSynapses_post function. If there is a Spike from an Input Neuron or an AdEx neuron that is used as source in the MxM simulation(presynaptic neuron) then the UpdateSynapses_pre is executed. Then, if the Target Neuron has generated a Spike, the UpdateSynapses_post part is also executed. This section takes 16 ticks for every Synapse because it needs to calculate a double or float accumulator that is used to calculate the mean variable of the UpdateSynapses_post function. In the beginning of each row, the Input Neuron spikes are saved in the FMEM, so that they can be used in the following part of the section. This section takes the most time in the Simulation and to be precise: (N_S + N_Group_S)*N_Group_T*16 ticks. This section cannot be parallelized in terms of Synapses due to the update of the accumulator.

3. Reads the array of Synapses again in the same way as the second section and executes the last part of the UpdateSynapses_post function. This section needs 4 ticks for every Synapse, to read, execute the calculations and write the data back to the LMEM. This section could be parallelized in terms of Synapses, but it isn't possible due to high utilization of HW in the kernel using doubles. This section takes N_Group_T*2 ticks.

This sections are run in every timestep. It becomes clear that all the elements of the 2d array of synapses must be run in every timestep to check which synapses and neurons need to be updated. The ticks needed to run the whole simulation are: Number_of_STDP_variables + timesteps*(2*N_Group_T+16*(N_S+N_Group_S)*N_Group_T+4*(N_S+N_Group_S)*N_Group_T).

The Number_of_STDP_variables ticks are used to save to the FMEM the fixed variables

of the STDP and AdEx models' differential equations.

To synchronize which section runs and which address of the memory to read or write, there are used a set of counters. There is a counter called worldCounter that wrapps in 2*N_Group_T+16*(N_S+N_Group_S)*N_Group_T+4*(N_S+N_Group_S)*N_Group_T ticks that is used to control the different sections and keep track in which section the kernel is by checking if the counter is less than Section 1 ticks, Section 2 ticks or Section 3 ticks. The distinct counters for this purpose are needed due to the fact that a counter cannot change wrap point dynamically and each one of them wraps to the corresponding Section's length. The representation of the kernel in a graph is the following:

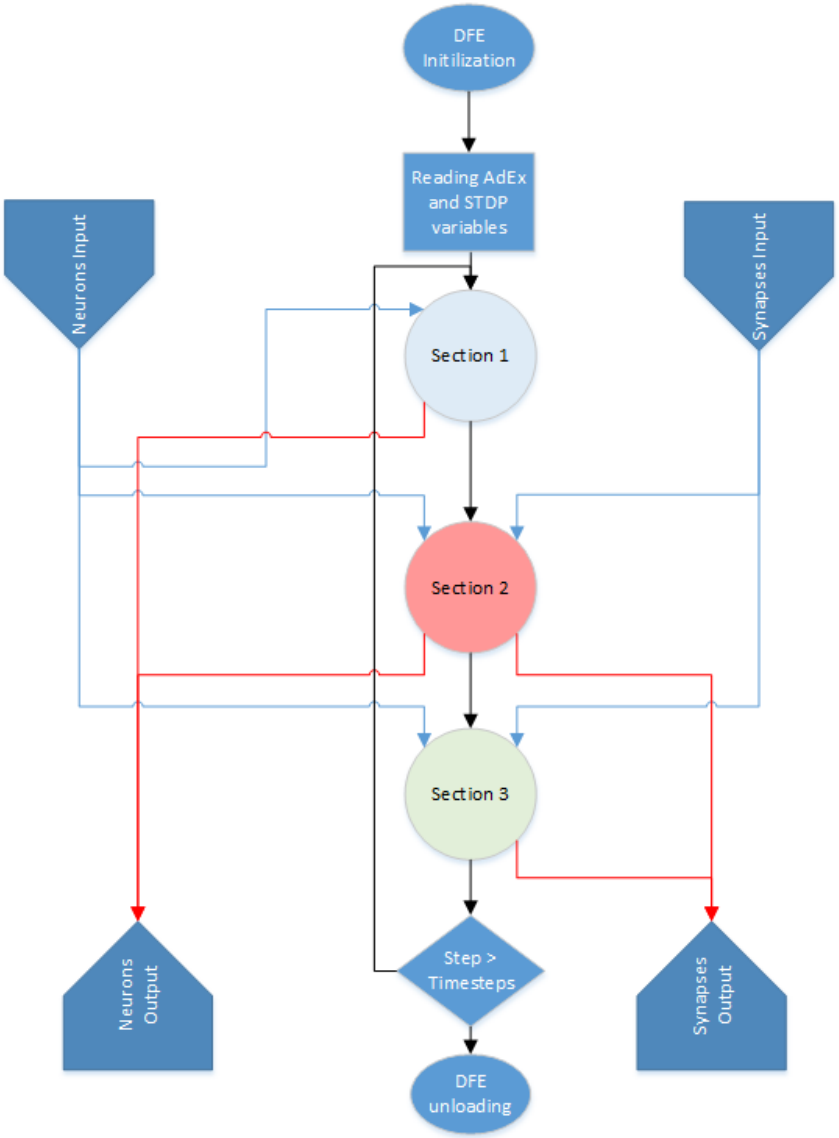


Figure 4.7: Single Kernel Flowchart

Due to the complexity of this Kernel the resource usage of this Kernel was very high.

To be more precise:
FINAL RESOURCE USAGE
Logic utilization: 252278 / 262400 (96.14%)
Primary FFs: 364674 / 524800 (69.49%)
Secondary FFs: 67155 / 524800 (12.80%)
Multipliers (18x18): 776 / 3926 (19.77%)
DSP blocks: 388 / 1963 (19.77%)
Block memory (M20K): 1440 / 2567 (56.10%)

The High Resource usage imposes a big difficulty to the synthesis tool of the FPGA. Especially the one related to the Logic Utilization. To try and build successively the Kernel there was needed an exploration of the optimization parameters of the MaxCompiler.

To begin with, to accelerate the build process, there was used parallelization in the level of the cables that run. To do this in the Engine Parameters of the Simulation, there were defined the values i_startMPPRCT, i_endMPPRCT and i_MPPRThreads as:

```
declareParam(i_startMPPRCT, DataType.INT, 1);
declareParam(i_endMPPRCT, DataType.INT, 4);
declareParam(i_MPPRThreads, DataType.INT, 4);
```

Then, in the configuration of the Manager these values were used in the functions

```
buildConfig.setMPPRCostTableSearchRange(params.getMPPRStart(), params.
    getMPPREnd());
buildConfig.setMPPRParallelism(params.getMPPRNumThreads());
```

The first one sets the number of cables of the build process, or to state it better, the number of tries to build the Kernel.

The second one defines the parallelism of the build process. For a parallelization of four, all the cables are run simultaneously.

Furthermore, there are two other parameters in the configuration of the kernel: buildConfig.setBuildEffort(Effort.VERY_HIGH) This one tries the best to build the Kernel. The other is buildConfig.setOptimizationGoal(OptimizationGoal.AREA) and optimize the building of the Kernel to use less Area on the FPGA, as this is the big problem of this Kernel.

All this optimizations however were not enough to build the Kernel. To do it, there was needed a decrease in the frequency of the FPGA. The FPGA clock frequency is defined again in the Manager by creating a clock using the Manager API:

```
ManagerClock myClk = generateStreamClock("myClk", 145)
Then this clock is connected to the Kernel needed building:
KernelBlock k = addKernel(new SimulationKernel(makeKernelParameters(
    s_kernelName)));
k.setClock(myClk);
```

The optimal frequency of the clock was selected after many tries of different frequencies. The bigger the frequency, the biggest the timing error of the kernel was. To reach a timing error of 0, there was needed to set the Frequency to 145MHz.

4.2.3 Comparison between Multiple and Single Kernels

The main focus on the comparison between the two implementations will be the load and unload times of the DFE. To check them, there will be some experiments with a small amount of Neurons, so that the DFE runtime is not very big.

Test Parameters
Simulation: NxM
N_Group_S = 0
N_Group_T = 384
N_S = 384
Timesteps = 1
Spike Interval = 1 step

Multiple Kernels:
init_time = 0.006994 s
load_time = 8.250590 s
unload_time = 1.262563 s
memory_reads = 0.045017 s
memory_writes = 0.066136 s
DFE_time = 0.123352 s

Single Kernel:
memory_reads = 0.020573 s
memory_writes = 0.028627 s
DFE_time = 0.079686 s
load_time = 4.864762 s
unload_time = 0.370028 s

This is a representative test of the timings for loading and unloading DFEs. These timing show that the Multiple Kernel architecture is not feasible for many timesteps, as they impose a huge delay in every timestep. For example for simulating 1 second with a step of 1ms, there would be needed 8000 seconds only for loading and unloading the Kernels, a lot more than the DFE Time.

In the Single Kernel there is a big loading time too. However, the loading of this Kernel happens only one time, in the beggining of the Simulation, so it doesn't matter in the final runtime.

Furthermore, it is seen that the DFE Time for the Single Kernel is less than the sum of the time that the Multiple Kernels run, so not only the Single Kernel Simulation is faster due to less loading time, but also the runtime is a lot less.

4.2.4 Double vs Float Data Type

Up until this point, the design of the Kernel was based on zero error from the C program. However, this means that in the Kernel all the data were doubles, which are represented in 64 bits. Consequently, they take more space in the LMem, but more importantly, their computations take almost double the hardware resources of the FPGA. An optimization on those factors is to used float type floating point variables, which are represented in 32 bits and take half the LMem space and a lot less resources (not exactly half). Furthermore, the high HW utilization of the Single Kernel with double type variables has as a result the need to lower the default Clock Frequency of the FPGA and the suboptimal performance of the FPGA. Moreover, the lower HW utilization by the Kernel with Float variables, makes able the parallelization of the processing of Synapses, gaining a lot more performance with that. The only drawback of this implementation is some errors that are being imposed to the results of the Simulation. There will be a more detailed reference to this problem later.

Float Kernel with parallelization

The only changes from the Single Kernel with doubles is that the variables of the Kernel are represented as floats, when there was only one Synapse processed in every step now there are two Synapses processed simultaneously and there are only double variables for the calculation of the mean, to avoid overflow of float variables. The update of the Neuron variables stayed as before due to complexity problems in building the kernel. To begin with, to change the variables from double to float it was a very easy job. In the beginning of the kernel there is a definition for the double type.

```
1 private static final DFEType double_type = dfeFloat(11,53);
```

After that, all the variables and the arrays are based on this type. By changing this type to dfeFloat(8,24) which represent a float variable, all the variables inside the Kernel were transformed to floats. The only variables that are represented as double variables are the ones related to the accumulation of the results needed to calculate the mean in the second section of the Kernel.

The second most important change of the Kernel is the parallelization in the level of Synapses. While it would be very easy to parallelize the kernel if there was no summation between values of two Synapses, now this task becomes more difficult. To solve this problem, there was suggested a type of reduce tree. As stated before, at every tick there are two Synapses processed. The values of these two Synapses that need to be calculated are added together and this sum is then added to the general sum. In this way, the delay for the Section 2 stays the same, while there are two Synapses calculated in the same tick.

A data flow of this computation in the Section 2 of the Kernel is visualised below:

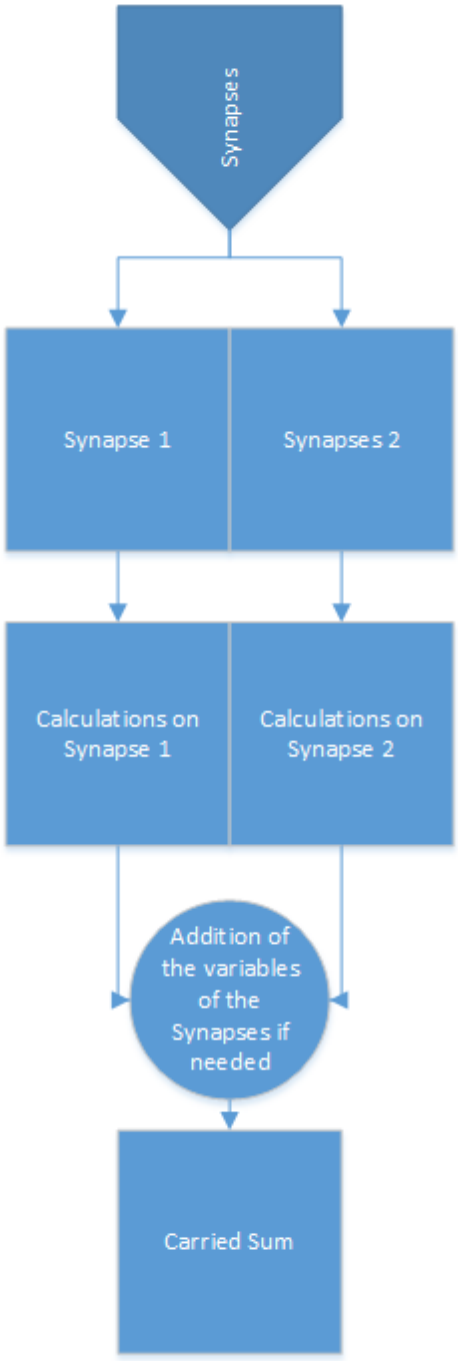


Figure 4.8: Synapses' Parallel Computation

The calculations on the Synapses include not only solving their differential equations but also create for each one the part that need to be accumulated to the global sum. If a postsynaptic Neuron has produced a spike, the a value that is calculated based on the values of the synapses must be added to the sum. Due to the fact that there must be always a number summed to the carried sum, if the Synapse value must be added then it is and if

not only zero is added to the sum. The same thing is happening in the calculation of the added Synapses to calculate the mean.

The Resource usage of this Kernel is show below:
FINAL RESOURCE USAGE
Logic utilization: 181875 / 262400 (69.31%)
Primary FFs: 323784 / 524800 (61.70%)
Secondary FFs: 12602 / 524800 (2.40%)
Multipliers (18x18): 393 / 3926 (10.01%)
DSP blocks: 207 / 1963 (10.55%)
Block memory (M20K): 1012 / 2567 (39.42%)

While it may seem that there could be a greater Unroll Factor, in reality an Unroll Factor of 4 was not possible, as the HW utilization was a lot bigger and the Kernel could not be compiled.
PRELIMINARY RESOURCE USAGE
Logic utilization: 300843 / 262400 (114.65%)
Multipliers (18x18): 681 / 3926 (17.35%)
Block memory (bits): 9954694 / 52572160 (18.94%)

Furthermore, the ticks needed for the sections 2 and 3 of the Kernel due to the delays of accessing the LMem were greatly increased, so an Unroll Factor greater than 2 was not feasible.

Acceleration Results

The results of the acceleration achieved by the float kernel is shown below:

Table 4.1: Acceleration Float with Unrolling vs Doubles

Run	1.1	1.2	1.3	2.1	2.2	2.3	3	4
Source Neurons	384	384	384	384	384	384	0	0
Source AdEx Neurons	0	0	0	0	0	0	384	1152
Target AdEx Neurons	384	384	384	1152	1152	1152	384	1152
Source Neurons Spike Interval (timesteps)	1	2	5	1	2	5	-	-
CPU Time (s)	2.38	1.24	0.56	7.83	4.56	2.24	4.10	52.21
DFE Run Time Float (s)	2.34	2.34	2.34	6.98	6.98	6.98	2.54	22.70
Acceleration Float vs CPU	1.02	0.53	0.24	1.12	0.65	0.32	1.61	2.30
DFE Run Time Double (s)	3.02	3.02	3.02	9.03	9.02	9.02	3.47	31.07
Acceleration Double vs CPU	0.79	0.41	0.19	0.87	0.50	0.25	1.18	1.68
Acceleration Float vs Double	1.29	1.29	1.29	1.29	1.29	1.29	1.37	1.37

It is obvious from this small networks that the float kernel achieves an average x1.8 acceleration in comparison with the double kernel. This is the reason why this Kernel was the one used to show the final acceleration in the DFEs.

Finally, due to the fact that there is no way to run a kernel for more than 1 hour in the DFE, there was needed a tiny tweak to the Kernel to be able to run the simulation for more than 1 hour, as needed in big networks. The tweak in the Kernel was passing a start time, which was added to the steps that run for, so that it would be instantiated in different time steps. To break the runs in timestep, there was implemented an algorithm in the CPU Code, which calculated how many 1 hour time frames the kernel needed to run and then called the Kernel for every one of them, breaking the simulation into steps.

Chapter 5

Experimental Results and Analysis

In this chapter there is going to be a presentation of the results of the DFE Implementation and the acceleration versus the Brian Simulator and the C Simulation. There are going to be discussed the reasons why certain parameters of the simulations were chosen and how their impact on the acceleration. The most interesting part of this chapter is the reasons why the acceleration of the DFE implementation follows the patterns described below.

The Maxeler model that was used to implement the DFE design was MAIA. The MAIA DFE is implemented with an Altera Stratix V FPGA. The FPGA chip, contains 262400 High-Performance Adaptive Logic Moduls, 3926 Variable-Precision DSP Blocks (18x18), 2567 M20K memory blocks and supports 14.1 Gbps transceivers. Furthermore, it supports up to 6 x72bit DIMM DDR3 memory interfaces up to 933Mhz. The Kernel used for taking the timing report was running on the default 150MHz clock and 400MHz DDR3 memory clock. The MAIA DFE had available a 48 GB LMem. The CPU that was used to run the Brian Simulator and the C Simulation was an Intel Xeon E5-2658A v3. Its architecture is based on the Haswell family of products, following a 22nm Lithography process and was launched on the 1st quarter of 2015. It has 12 cores and supports HyperThreading (24 Threads) and has a frequency of 2.20 Ghz and a Turbo Frequency of 2.90 Ghz. Moreover, it has a 30 MB SmartCache and 128 GB of DDR4 RAM.
The source code of the executable used for the following runs is on this GitHub repository: `https://github.com/iomaganaris/AdexSimMaxeler`

The variables of the runs are:

- Number of Input and AdEx Neurons: The number of Input and AdEx Neurons changes the overall runtime of the simulation. These two variables are also sweeped to check if there is any difference in acceleration in relation to the size of the problem, while keeping the other variables stable
- Input Neurons Spike Frequency: In the NxM simulation, where there are only Input Neurons connected to the AdEx Neurons, it is very important to check the differences in runtime based on the activity of Input Neurons
- Neuron Connectivity: It is also important to see the differences in run time due to connectivity variation

- Timesteps: The simulation steps

5.1 NxM Simulation

This is the type of Simulation were Input Neurons, that only produce spikes based on a given distribution by the user, are connected with "real" neurons, described by the AdEx Model. A graphical representation of such a network is given below:
N_S = 2
N_Group_S = 0
N_Group_T = 3
Connectivity = 100%
Number of Synapses = 6

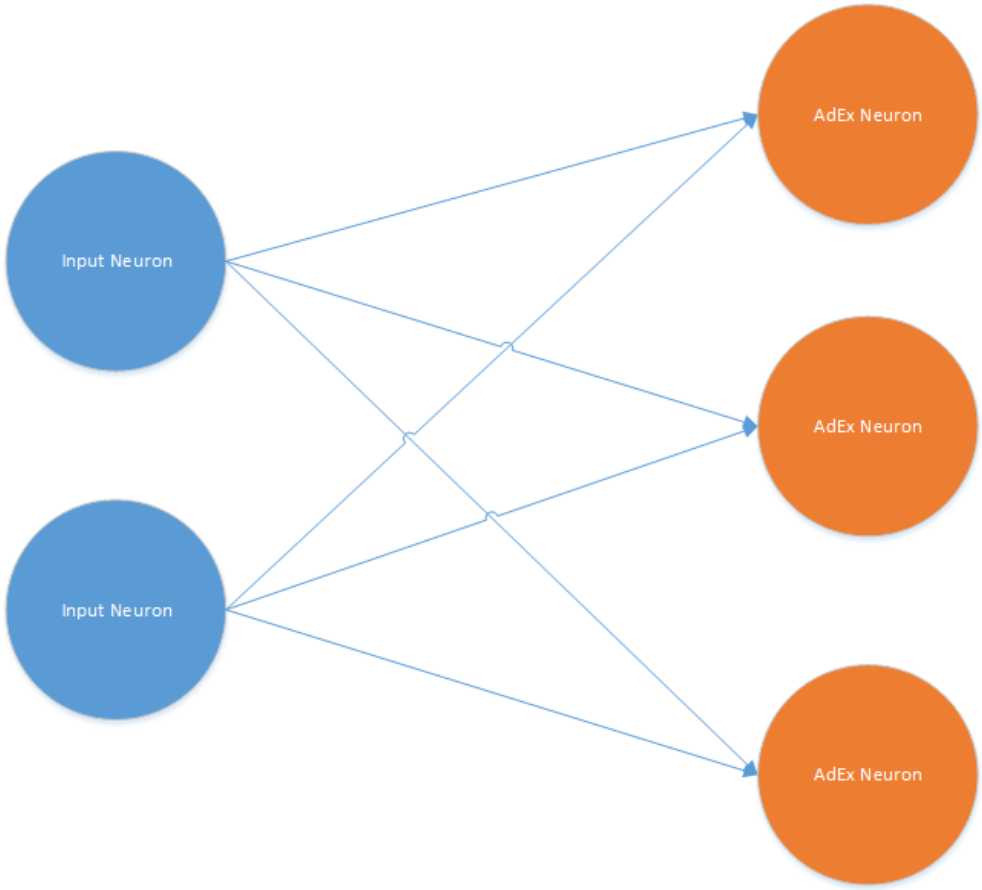


Figure 5.1: NxM Simulation Network

The straight lines represent Synapses

For every experiment of this type, the initialization of the AdEx Neurons were the same and similar to the paper that the thesis is based on. To be more precise:
vtrest = -45 mV

EL = -70.6 mV
Neuron.vt = vtrest
Neuron.vm = EL
Neuron.I = 0
Neuron.x = 0
Neuron.Spike = 0

The initialization of Synapses was based again on the paper. The same variables (FBp, FBn, R, U, A) were changed but the values on the synapses was based on an ascending number that wrapped around 16777216. This number was chosen because float numbers can represent all integers up until it. Different numbers on the synapses also meant that there would be different values on the variables of the synapses and neurons. This way, the debugging was easier and more sure and also there was a good amount of Spikes on the AdEx Neurons produced. The experiments that was run for the performance measurements, don't contain any artificial spikes on AdEx Neurons to keep the faithfulness.

5.1.1 Results

Brian vs C vs DFE

The following table shows the acceleration between the different implementations. The range of Neurons is 384 which is the smallest network simulated by the DFE, and 4992, as networks bigger, took a lot of time to be calculated (> 10h on Brian). The same conclusions from the given runs can be reached for bigger networks. The variables that are kept stable are:
Timesteps of the simulation: 1000
Simulation step: 1 ms
Input Spike Interval: 1 step (Spiking frequency: 1 kHz)
Connectivity: 100%

Table 5.1: Acceleration DFE vs C CPU vs Brian

N_S	384	1152	4992
N_Group_S	0	0	0
N_Group_T	384	1152	4992
Input Spike Interval	1	1	1
Connecticity (%)	100	100	100
CPU Time (s)	52.48	539.41	9694.05
LMEM Read Time (s)	0.02	0.06	1.07
LMEM Write Time (s)	0.45	0.23	4.39
DFE Time (s)	33.11	297.59	5685.43
Acceleration DFE vs CPU	1.58	1.81	1.71
Brian Time (s)	215.56	1719.79	32700.92
Acceleration DFE vs Brian	6.51	5.78	5.75

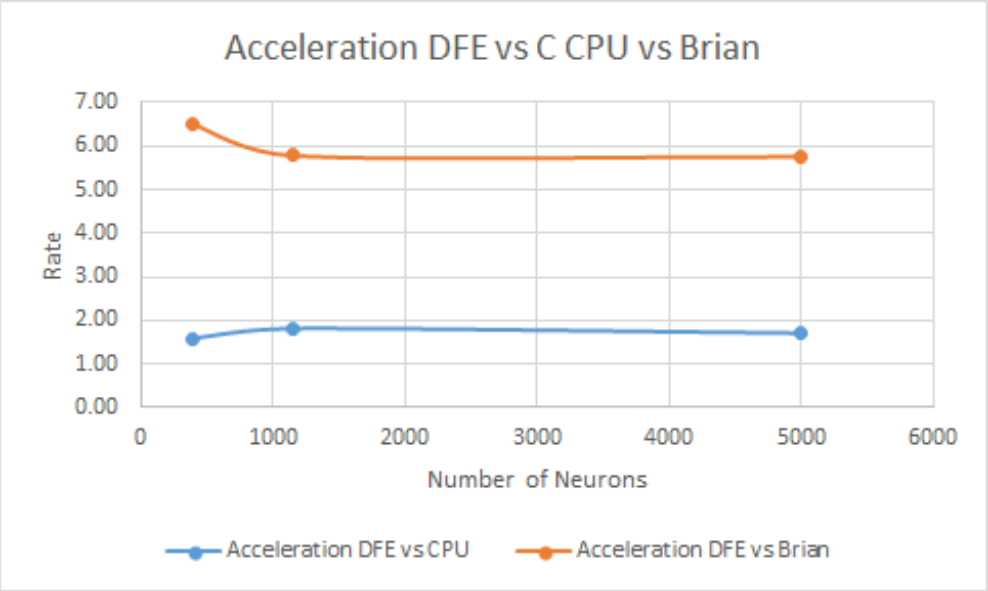


Figure 5.2: Acceleration DFE vs C CPU vs Brian

From the diagram, it is obvious that there is a stable acceleration for experiments containing more that 1152 Neurons. While the acceleration between DFE and Brian is quite large, around 6 times, the acceleration in relation with the C Simulation is not great. This is due to the architecture of the C program running in a CPU and the DFE architecture. In the NxM simulation, in every timestep the Input Neurons produce a spike. This means that all the Synapses should be updated based on the pre-synaptic STDP expression. However, due to the fact that there is not a huge activity in the post-synaptic (AdEx) Neurons, the Synapses should not be updated based on the post-synaptic STDP expression. This creates a difference in the time needed for the calculations of a single Synapse in the CPU and in the DFE program. Due to the deterministic definition of the FPGA runtime, there is no way to avoid checking if a synapse has pre- or post-synaptic activity, and if not pass clock

cycles and thus make the simulation faster. However, this is what happens in the CPU with the branches of if statements. In general, this is the biggest difference between the CPU instruction flow architecture and the FPGA data flow architecture. This difference has an impact on this simulation due to the fact that it is event-driven and there is no deterministic way to compute the steps or runtime needed to solve the simulation beforehand. So, the FPGA must be programmed to do all the calculations needed and always runs for the worst scenario, which in our case is activity in every Neuron of the network (Input or AdEx) for every timestep.

To check better how Input activity and Network size impacts performance, there were some more tests run with the C Simulation and the DFE implementation. Brian was not measured due to very high run times. In this experiment the Number of AdEx Neurons was kept stable.

Table 5.2: Acceleration DFE vs C

N_S	384	1152	2304	3840	4992
N_Group_S	0	0	0	0	0
N_Group_T	4992	4992	4992	4992	4992
Input Spike Interval	1	1	1	1	1
Connecticity (%)	100	100	100	100	100
CPU Time (s)	721.03	2179.19	4438.31	7418.39	9694.05
LMEM Read Time (s)	0.09	0.24	0.48	0.78	1.07
LMEM Write Time (s)	3.65	0.38	0.68	0.99	4.39
DFE Time (s)	416.32	1248.03	2495.64	4159.15	5685.43
Acceleration DFE vs CPU	1.73	1.75	1.78	1.78	1.71

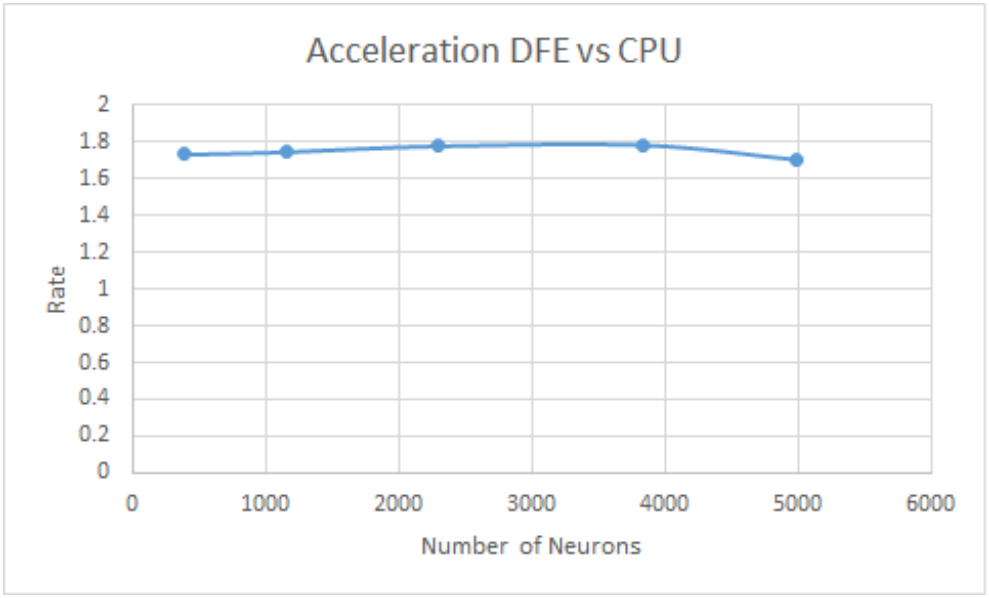


Figure 5.3: Acceleration DFE vs CPU

It is obvious that the bigger the Network, the most acceleration is achieved from the DFE. The smaller acceleration in the biggest network size is probably random and related to CPU traffic probably, as the absolute differences between the acceleration rates is again small. Another useful graph from this simulation is Runtime scaling of both of the architectures.

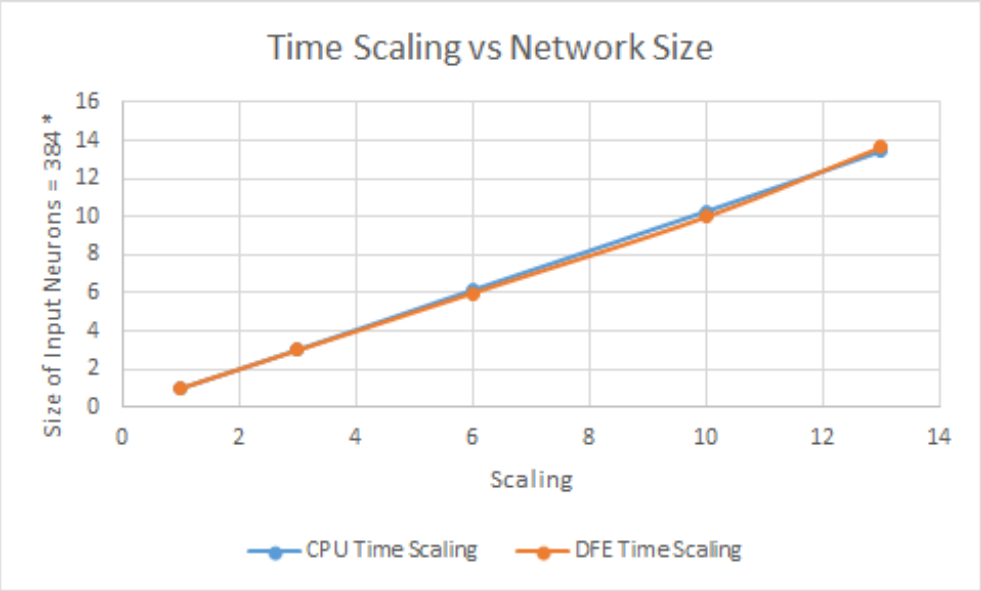


Figure 5.4: Acceleration DFE vs CPU

As we can see the run times is scaling almost linearly, as expected, as the network size is growing linearly too.

Large Number of Neurons

To simulate large networks in smaller time, there were done experiments for less timesteps but same everything else. The maximum number of Neurons simulated by the DFE Simulation is 20352.
Timesteps of the simulation: 100
Simulation step: 1 ms
Input Spike Interval: 1 step (Spiking frequency: 1 kHz)
Connectivity: 100%

Table 5.3: NxM Large Networks Acceleration DFE vs C

Timesteps	1000	1000	1000	100	100	100
N_S	384	1152	4992	10368	15360	20352
N_Group_S	0	0	0	0	0	0
N_Group_T	384	1152	4992	10368	15360	20352
Input Spike Interval	1	1	1	1	1	1
Connecticity (%)	100	100	100	100	100	100
CPU Time (s)	52.48	539.41	9694.05	4261.53	9726.64	16839.28
LMEM Read Time (s)	0.02	0.06	1.07	4.19	9.22	16.59
LMEM Write Time (s)	0.45	0.23	4.39	4.74	12.19	18.76
DFE Time (s)	33.11	297.59	5685.43	2380.65	5224.94	9449.30
Acceleration	1.58	1.81	1.71	1.79	1.86	1.78

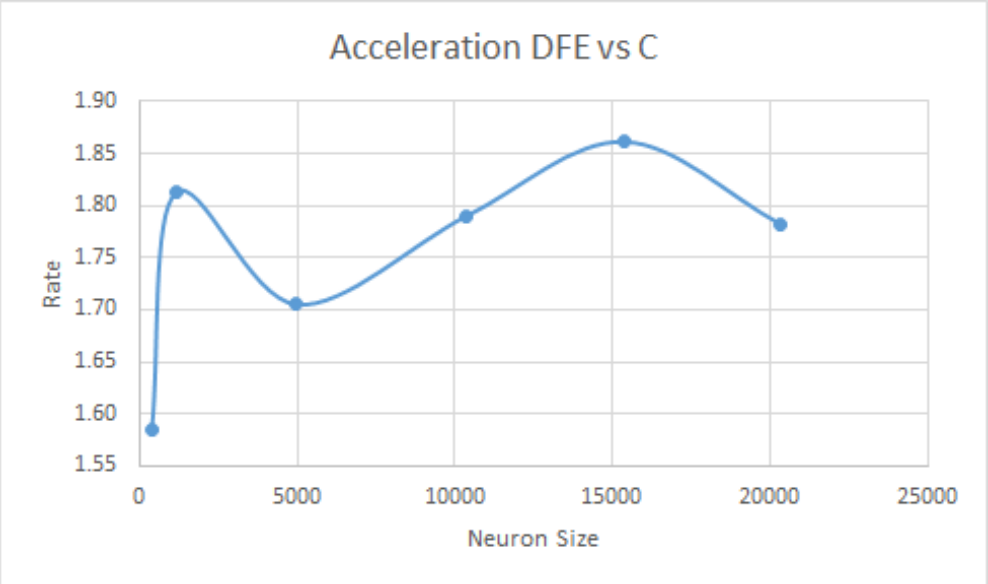


Figure 5.5: Acceleration DFE vs CPU

In addition to the larger size experiments there are shown the smaller experiments run for more timesteps. It is obvious that acceleration grows bigger as the network size becomes bigger but the average acceleration for large size experiments is x1.81. Similar acceleration rates were observed for every size between them, so they didn't need to be attached.

Connectivity

To check the impact of connectivity, there was simulated a network with 50% connectivity apart from 100%. This means that an Input Neuron is connected with half the AdEx Neurons. If the ID of an Input Neuron is odd then this Neuron is connected only with odd AdEx Neurons and the similar connection is happening with even ID Neurons.
Timesteps of the simulation: 1000
Simulation step: 1 ms

Input Spike Interval: 1 step (Spiking frequency: 1 kHz)

Table 5.4: Acceleration based on Network Connectivity

N_S	384	384	1152	1152	4992	4992
N_Group_S	0	0	0	0	0	0
N_Group_T	384	384	1152	1152	4992	4992
Input Spike Interval	1	1	1	1	1	1
Connecticity (%)	100	50	100	50	100	50
CPU Time (s)	52.48	25.35	539.41	284.32	9694.05	5875.74
LMEM Read Time (s)	0.02	0.02	0.06	0.06	1.07	0.99
LMEM Write Time (s)	0.45	3.18	0.23	0.23	4.39	1.18
DFE Time (s)	33.11	32.70	297.59	293.99	5685.43	5519.11
Acceleration	1.58	0.78	1.81	0.97	1.71	1.06

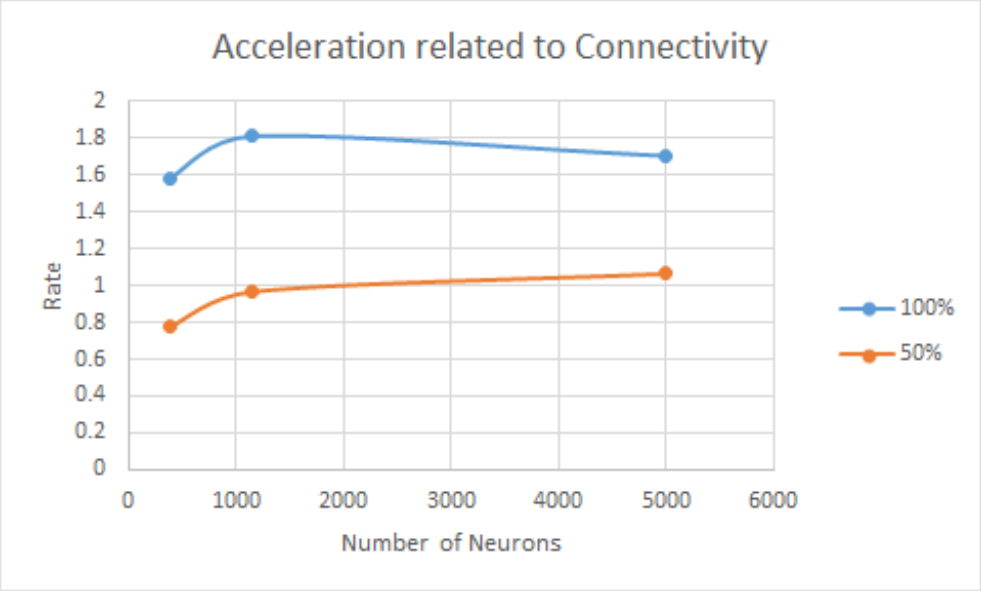


Figure 5.6: Acceleration DFE vs CPU

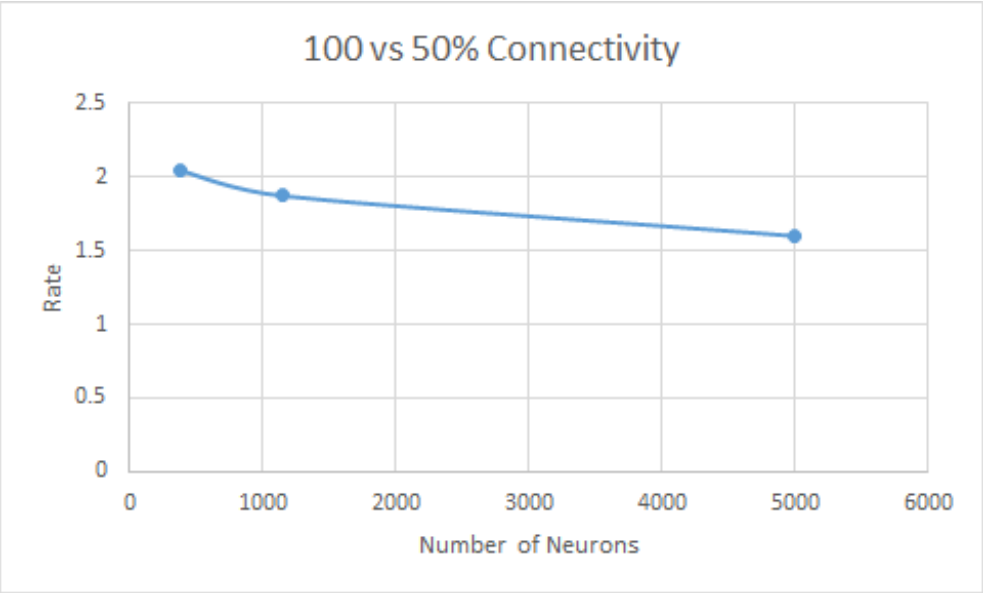


Figure 5.7: Runtime Scaling of 100 vs 50% Connectivity

From the graphs it is obvious that due to less connections between the Neurons the acceleration rate falls. The reasons behind this difference is again due to the CPU program instruction run flow. While traversing the 2D adjacency matrix of Synapses, if the connection flag is zero in one of them, then there are no calculations done for this Synapse. However, in the DFE simulation, when traversing this array, the same time must be passed to process a single synapses, whether it is connected or not in the Neuron Network. The smaller than 2 ratio in the bigger experiments is due to higher efficiency of the FPGA when reading big amounts of data. The acceleration is greater when the problem size gets greater.

Input Spiking Frequency

The following experiments were run to check the impact of Input Spiking Frequency to the simulation run time. Input Neurons are programed to generate Spikes every 1,2 or 4 simulation steps (1 kHz, 0.5 kHz, 0.25 kHz).
Timesteps of the simulation: 1000
Simulation step: 1 ms
Connectivity: 100%

Table 5.5: Acceleration based on Network Input Activity

N_S	384	384	384	1152	1152	1152	4992	4992	4992
N_Group_S	0	0	0	0	0	0	0	0	0
N_Group_T	384	384	384	1152	1152	1152	4992	4992	4992
Input Spike Interval	1	2	4	1	2	4	1	2	4
CPU Time (s)	52.48	25.38	14.15	539.41	287.17	182.27	9694.05	5832.15	3712.92
LMEM Read Time (s)	0.02	0.02	0.02	0.06	0.07	0.07	1.07	1.08	1.05
LMEM Write Time (s)	0.45	3.54	0.49	0.23	3.40	0.54	4.39	4.46	4.39
DFE Time (s)	33.11	33.70	33.70	297.59	297.69	302.90	5685.43	5685.43	5685.43
Acceleration	1.58	0.75	0.42	1.81	0.96	0.60	1.71	1.03	0.65

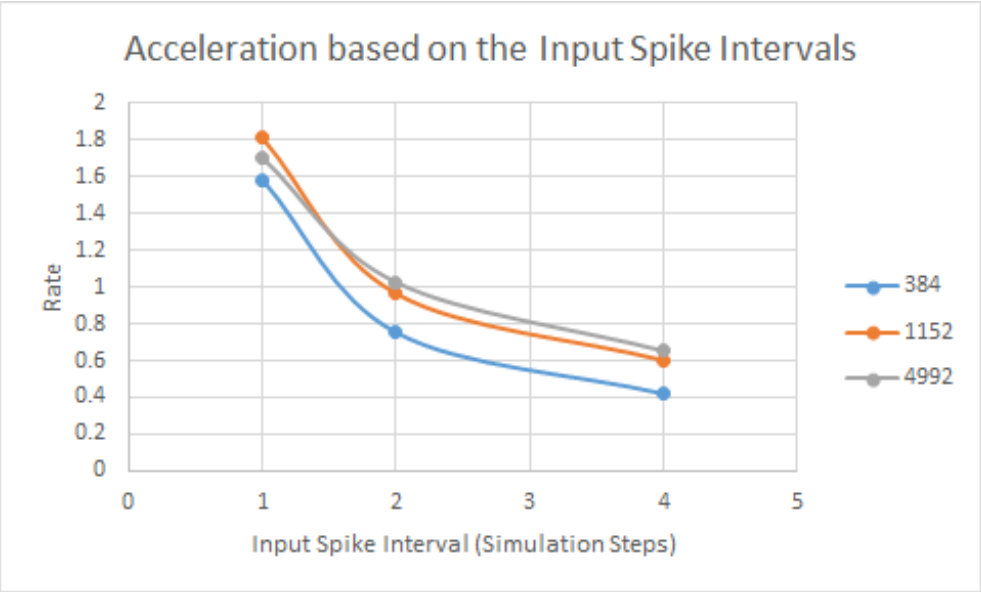


Figure 5.8: Acceleration DFE vs CPU

In this experiment, the difference between the CPU and the DFE architecture is again visible. The more activity there is on the network, the more acceleration there is in the DFEs. To make this more clear I will make a comparison between the traversing of the 2D Adjacency Matrix in the CPU and the DFE.

C Architecture

```
for (int i = 0; i < N_Group_T; i++){
    if (SpikeArray[i] > 0){ // if there is Presynaptic activity for the
   Presynaptic STDP expression
        for (int j = 0; j < N_S + N_Group_S; j++){
            if (Synapses[j][i].conn){ // if the Synapse is connected
                // Do computations on the Synapse
            }
        }
  }
}
```

DFE Architecture

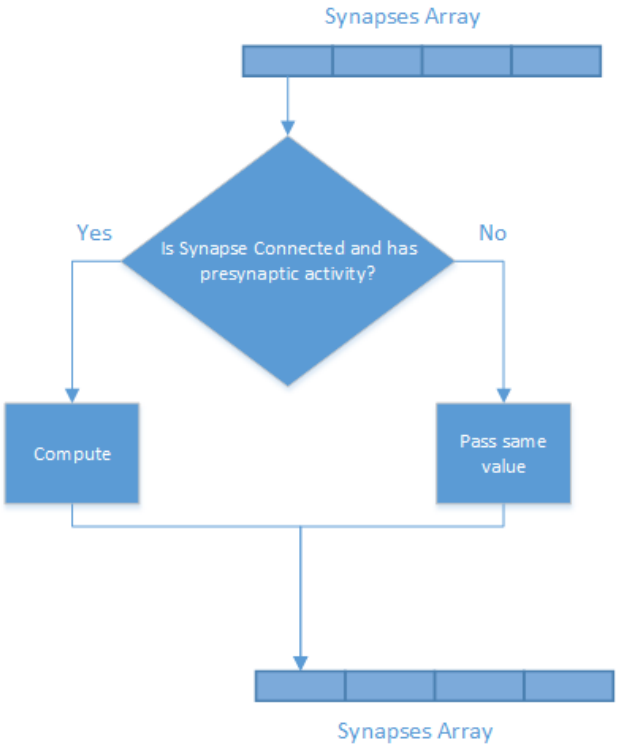


Figure 5.9: Dataflow Synapses Processing

"Compute" and "Pass same value" boxes have to take the same time, due to the deterministic run time of the FPGA.

5.1.2 Error

As stated in the DFE Implementation chapter, the float data type used for the DFE implementation is different to the double data type used by the C program and the Brian simulation. This means that there will be slight differences in the values that are calculated by the DFE Implementation. However, these errors are very small and don't intervene with the simulation run. The reason why they exist, is the smaller accuracy of floats from doubles. To check these errors there were introduced 2 measures.

The first one is the Average Error of an array of data. This means that all the errors between the corresponding values of the CPU and DFE simulation are summed and then divided by the number of Synapses or Neurons, depending on the array of values that is checked. The second one is the Relative Average Error of an array of data. This variable is the division between the Average Error and the Average of the values of the data that we want to check. This is the most important error measure, as it shows if the error is big enough to concern us, based on the range of values of the Neuron's or Synapse's variable.

In general, the errors in all the simulations run were very similar. I will include in this section the errors of the greatest simulation run, which is:

Type of Simulation: NxM
Timesteps = 100
N_S = 20352
N_Group_S = 0
N_Group_T = 20352
Spike Interval = 1
Connectivity = 100%

The relative error for the x variable might be 1 but the average error is very small, not

Table 5.6: Neuron Variables' Errors

Neuron Variable	Average Error	Relative Average Error
Vt	0.00E+00	
Vm	5.21E-09	-7.38E-08
I	0.00E+00	
x	1.00E-17	1.00E+00

representable by floats.

Table 5.7: Synapses Variables' Errors

Synapse Variable	Average Error	Relative Average Error
w	9.55E-08	3.72E-06
FFp	1.52E-06	2.91E-08
FBp	8.19E-01	1.51E-07
FBn	-4.16E-01	-1.03E-06
R	-2.14E+01	-4.23E-06
u	1.03E-10	1.86E-06
U	1.03E-10	1.86E-06
A	0.00E+00	

And for the biggest with most timesteps simulation:
Type of Simulation: NxM
Timesteps = 1000
N_S = 4992
N_Group_S = 0
N_Group_T = 4992
Spike Interval = 1 Connectivity = 100%

Table 5.8: Neuron Variables' Errors

Neuron Variable	Average Error	Relative Average Error
Vt	0.00E+00	
Vm	1.62E-08	-2.39E-07
I	6.69E-15	6.71E-05
x	9.38E-17	7.19E-06

Table 5.9: Synapses Variables' Errors

Synapse Variable	Average Error	Relative Average Error
w	3.78E-07	4.78E-07
FFp	-6.06E-05	-1.80E-06
FBp	-2.38E-01	-5.85E-06
FBn	-5.16E-12	-2.85E-05
R	0.00E+00	
u	0.00E+00	
U	0.00E+00	
A	3.78E-07	4.78E-07

5.2 MxM Simulation

In this type of Simulation there is no Input Spikes connected to the Network. The Network only contains AdEx Neurons connected to each other. A representation of this Network is shown below:
N_S = 0
N_Group_S = 2
N_Group_T = 2
Connectivity = 100%
Number of Synapses = 4

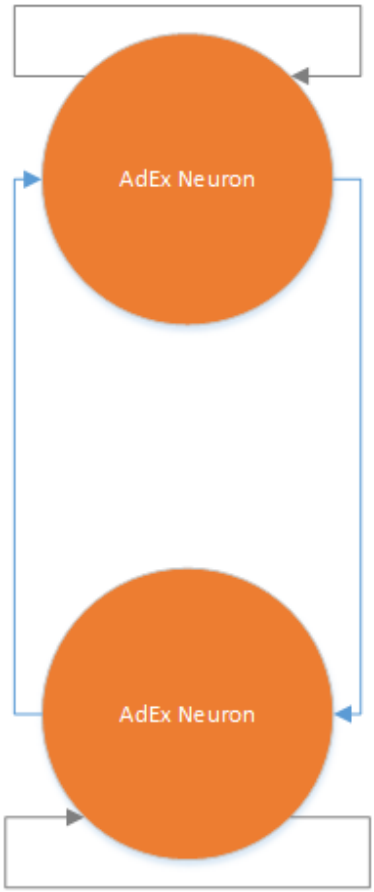


Figure 5.10: MxM Simulation Network

The gray lines are Synapses that don't have much physical meaning but can be represented in the Adjacency Matrix.

This initialization of Neurons in this type of experiments was done in such way that the AdEx Neurons spiked in every timestep. This was done, not only because there was no physical way to produce specific spikes in the AdEx Neurons in given intervals and to see the acceleration of the DFE when processing the same data with exactly the same computations with the CPU.

Neuron Initialization:

vtrest = -45 mV

Neuron.vt = vtrest

Neuron.vm = vtrest + 5mV

Neuron.I = 0

Neuron.x = 0

Neuron.Spike = 0

Synapses were initialized in the same way as before, apart from the u variable, which was set to 1 from 0 in all the Synapses.

5.2.1 Results

Brian vs C vs DFE

In this part, there will be a comparison between the different architectures. The range of networks will be 384 to 4992, as for the Brian simulator bigger networks require more than 15 hours to run. The key observation in this chapter is the efficiency of the FPGAs, when running exactly the same computations with the C program and the Python Brian Simulator. The variables of the simulation kept stable are:
Timesteps of the simulation: 1000
Simulation step: 1 ms
Connectivity: 100%

Table 5.10: Acceleration DFE vs C vs Brian

N_S	0	0	0
N_Group_S	384	1152	4992
N_Group_T	384	1152	4992
CPU Time (s)	83.28	1073.47	24797.61
LMEM Read Time (s)	0.02	0.06	1.01
LMEM Write Time (s)	0.19	0.44	4.69
DFE Time (s)	37.24	334.83	6337.73
Acceleration DFE vs CPU	2.24	3.21	3.91
Brian Time (s)	306.09	2898.11	53330.79
Acceleration DFE vs Brian	8.22	8.66	8.41

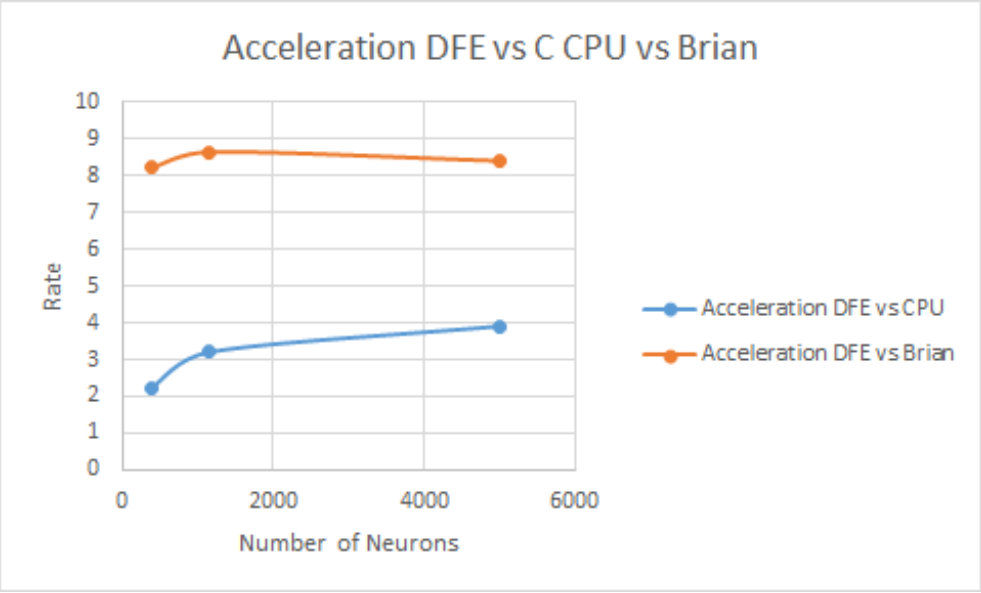


Figure 5.11: Acceleration DFE vs C CPU vs Brian

It is pretty obvious that there is a huge jump in the acceleration rate from the NxM

Simulation, which is very reasonable, as there are almost double the computations done in this experiment in the C Simulation but also Brian, due to both Pre- and Post-synaptic activity in the network, as when an AdEx Neuron Spikes this means that all the synapses that begin from it must be updated based on the presynaptic STDP expression, but other synapses must also be updated based on the postsynaptic STDP expression. Consequently, in every timestep, all the synapses are updated based on all their STDP expressions. The greatest acceleration ratio between Brian and DFE is x8.65. As the network grows bigger, this is very important, as Simulations that could take days when using the Brian Simulator, may take only a working day in DFEs.

Large Number of Neurons

To simulate large networks in smaller time, there were done experiments for less timesteps but same everything else. The maximum number of Neurons simulated by the DFE Simulation is 20352.
Timesteps of the simulation: 100
Simulation step: 1 ms
Connectivity: 100%

Table 5.11: Acceleration DFE vs C

Timesteps	1000	1000	1000	100	100	100
N_S	0	0	0	0	0	0
N_Group_S	384	1152	4992	10368	15360	20352
N_Group_T	384	1152	4992	10368	15360	20352
CPU Time (s)	83.28	1073.47	24797.61	12033.40	12033.40	41129.04
LMEM Read Time (s)	0.02	0.06	1.01	4.18	9.30	16.09
LMEM Write Time (s)	0.19	0.44	4.69	7.50	12.54	17.01
DFE Time (s)	37.24	334.83	6337.73	2642.48	5799.60	10186.02
Acceleration	2.24	3.21	3.91	4.55	2.07	4.04

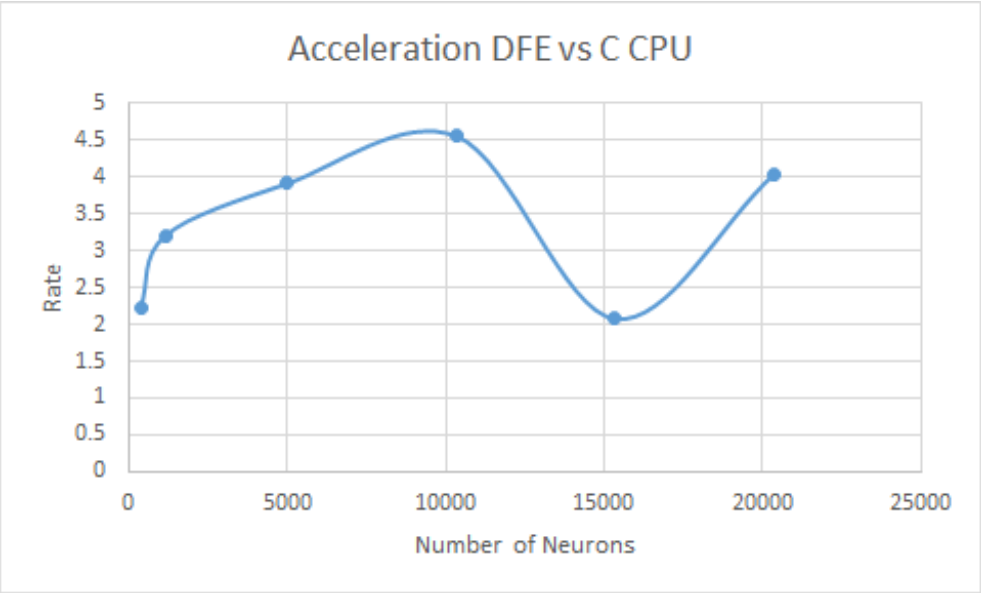


Figure 5.12: Acceleration DFE vs CPU

As seen in the graph the acceleration is almost double than the corresponding experiment in the NxM simulation, reaching up to x4.5. The big drop in the acceleration in the 15360 simulation is due to not producing spikes in some of the AdEx neurons, due to the initialization used in the Synapses. Similar acceleration rates were observed for every size between them, so they didn't need to be attached.

Connectivity

To check the impact of connectivity, there was simulated a network with 50% connectivity apart from 100%. This means that an Input Neuron is connected with half the AdEx Neurons. If the ID of an Input Neuron is odd then this Neuron is connected only with odd AdEx Neurons and the similar connection is happening with even ID Neurons. This type of connection between neurons in the MxM simulations is the only reason why the CPU code does less computations, and this is why the acceleration is not as great as the fully connected network.
Timesteps of the simulation: 1000
Simulation step: 1 ms

Table 5.12: Acceleration DFE vs C related to Connectivity

N_S	0	0	0	0	0	0
N_Group_S	384	384	1152	1152	4992	4992
N_Group_T	384	384	1152	1152	4992	4992
Connectivity (%)	100	50	100	50	100	50
CPU Time (s)	83.28	41.80	1073.47	636.93	24797.61	14421.14
LMEM Read Time (s)	0.02	0.02	0.06	0.06	1.01	1.00
LMEM Write Time (s)	0.19	0.21	0.44	0.24	4.69	1.17
DFE Time (s)	37.24	36.29	334.83	326.31	6337.73	6126.04
Acceleration	2.24	1.15	3.21	1.95	3.91	2.35

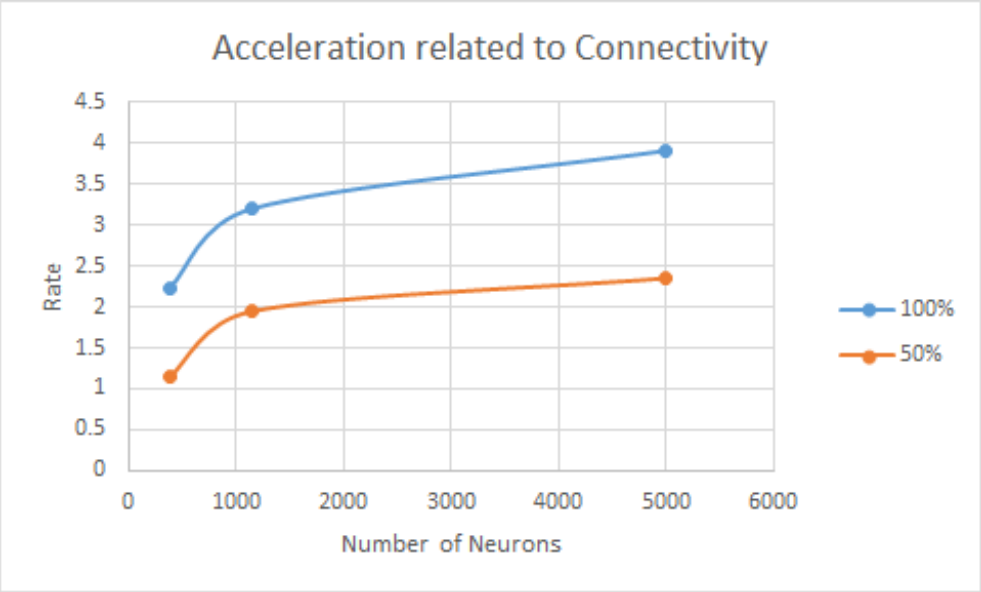


Figure 5.13: Acceleration DFE vs CPU

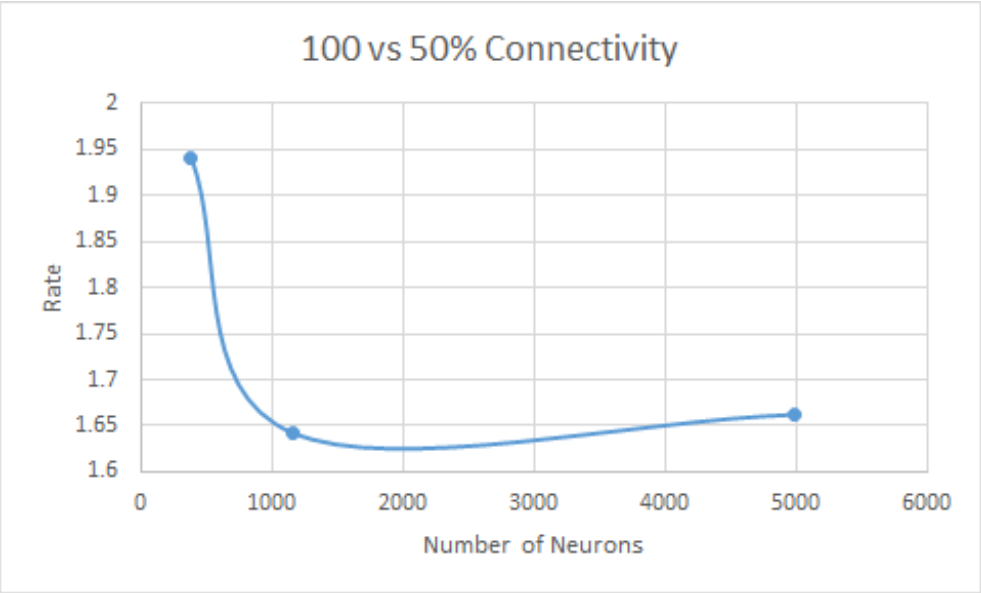


Figure 5.14: Runtime Scaling of 100 vs 50% Connectivity

Due to less less active connections in the simulation, the acceleration decreases for the same reasons as the NxM Simulations. The ratio of difference between the 100% and 50% connectivity is due to the efficiency of passing data to the DFE.

5.2.2 Error

As stated in the NxM simulation, the errors in the MxM Simulation was again irrelevant. The errors for the greatest Network simulation are:
Simulation Type: MxM
Timesteps = 100
N_S = 0
N_Group_S = 20352
N_Group_T = 20352
Connectivity = 100%

Table 5.13: Neuron Variables' Errors

Neuron Variable	Average Error	Relative Average Error
Vt	0.00E+00	
Vm	0.00E+00	
I	3.10E-02	7.20E-06
x	-2.76E-15	-4.74E-07

Table 5.14: Synapses Variables' Errors

Synapse Variable	Average Error	Relative Average Error
w	0.00E+00	
FFp	-1.23E-06	-2.35E-08
FBp	6.56E-01	1.21E-07
FBn	-5.01E-01	-1.24E-06
R	3.40E-07	1.27E-05
u	-5.27E-07	-3.82E-06
U	4.15E-11	2.38E-06
A	2.83E-08	1.89E-08

And for the biggest with most timesteps simulation:
Simulation Type: MxM
Timesteps = 1000
N_S = 0
N_Group_S = 4992
N_Group_T = 4992
Connectivity = 100%

Table 5.15: Neurons Variables' Errors

Neuron Variable	Average Error	Relative Average Error
Vt	0.00E+00	
Vm	0.00E+00	
I	3.35E-01	6.71E-05
x	-8.25E-15	-7.12E-07

Table 5.16: Synapses Variables' Errors

Synapse Variable	Average Error	Relative Average Error
w	0.00E+00	
FFp	-1.21E-04	-1.80E-06
FBp	-5.33E-01	-5.84E-06
FBn	-6.72E-05	-2.02E-06
R	0.00E+00	
u	0.00E+00	
U	0.00E+00	
A	0.00E+00	

Chapter 6

Conclusion

6.1 Remarks

Modern Neuroscience research is significantly dependent on computer simulations. From my occupation with Computational Neuroscience through this diploma thesis, I have found out that the complexity of the problems that Neuroscientists face are huge and the Simulations needed to push Brain Research further are a big burden. This Diploma Thesis tried to ease this setback by suggesting an efficient solution to the problem of big network simulations. The first step towards this goal was done with accelerating the simulation of the Brian Simulator for the AdEx Model with STDP by implementing it using the C programming language. This lead to an acceleration of x2. Furthermore, the acceleration using the Maxeler Dataflow Architecture was able to accelerate the simulation an additional x4 times, reaching an over 8 times acceleration in comparison to the Brian Simulator.

Maxeler Dataflow Programming is for sure a very interesting platform that can outperform other architectures for given applications. On top of that, the encryption of many implementation details of the FPGA platform from the developer is able to accelerate a lot the development of the program, while having same or even better results when the Maxeler DFE architecture is fully utilized. The Dataflow architecture is greatly assisted by the LMem and FMem features of the DFE that help a lot of program architectures to be accelerated on that platform. Furthermore, another big problem of FPGAs, the fast reconfiguration of the chip, is also solved, as the DFE can be loaded and unloaded in a matter of seconds. Unfortunately, the problem of slow compilation still exists, however the simulation tools and the MaxCompiler Manager help a lot with the synthesis process.

As described in the thesis, the acceleration results were not exactly what was anticipated, based on other Maxeler applications. However, there was a great effort put into trying to understand the reasons why this happened in detail. The simulation was event-driven due to the fact that the AdEx model is a spiking neuron model. This meant that the runtime and run flow of the program could not be predicted beforehand to save any time or optimize any aspect of the simulation without losing accuracy of the results. The event-drive simulation could not be accelerated in the DFEs, due to the deterministic runtime of the FPGAs that always run for the same time as the worst case scenario. In C however, due to the branches

in the run flow, there are a lot of computations omitted if there is no need to be run, so a lot of time is gained there. Apart from this, the biggest problem that arise was the lack of ability to widely parallelize the calculation of multiple Synapses or Neurons due to the complex differential equation models that described the very detailed STDP model that was used.

Concluding, the Maxeler platform advantages were not able to be shown in all the simulation scenarios, as the high memory bandwidth and the FPGA throughput wasn't possible to be achieved due to occasional waste of precious clock cycles that network inactivity imposed. However, all the accelerators are not appropriate for every problem and this diploma thesis was able to investigate DFE's most fertile field of applications.

6.2 Future work

The two main aspects that could be further explored based on this implementation is the usage of more modern Maxeler DFE platforms and the integration of the DFE executable in Brian.

Maxeler has recently introduced its new MAX5 DFEs that provide a lot more hardware resources and faster clock due to the utilization of more modern FPGA chips. This means that not only there will be existing more space for greater loop unrolling but also higher running clocks of the FPGAs would ensure higher throughput.

Furthermore, this DFE executable could be integrated to the Brian simulator, making it natively accelerated. This could accelerate research of Neuroscientists even further, as they wouldn't need to learn a new framework to run the accelerated simulation. The ease of usage of a simulator is a very important factor for the Neuroscientists.

Bibliography

[Fos97] C.S. Foster M. & Sherrington. *Textbook of Physiology. Volume 3*. London : Macmillan, 1897.

[Jes80] & Mirsky R. Jessen K. R. "Glial cells in the enteric nervous system contain glial fibrillary acidic protein." In: (1980). DOI: 10.1038/286736a0.

[ERKJ81] James H. Schwartz Eric R. Kandel and Thomas M. Jessell. *Principles of Neural Science (4th ed.)* New York: McGraw-Hill, 1981. ISBN: 0-8385-7701-6.

[DWP83] LuVerne R. Peterson David W. Page. *Re-programmable PLA*. 1983. URL: https://patents.google.com/patent/US4508977?oq=4508977.

[Pag83] David W. Page. *Dynamic data re-programmable PLA*. 1983. URL: https://patents.google.com/patent/US4524430?oq=4524430.

[NRARJLF84] M.B.B.S. Nayef R.F. AI-Rodhan and M.D. John L. Fox. *Al-Zahrawi and Arabian Neurosurgery, 936-1013 AD*. Tech. rep. Department of Neurosciences, King Faisal Specialist Hospital and Research Centre, Riyadh, Saudi Arabia, 1984. DOI: 10.1016/0090-3019(86)90070-4.

[AG87] Gross Charles G. Adelman George. *"Neuroscience, Early History of" in "Encyclopedia of Neuroscience"*. Birkhauser Verlag AG, 1987, 843–847. ISBN: 3764333332.

[VL93] J. Van Laere. "Vesalius and the nervous system". In: (1993).

[Har94] Adrian & Gudat F & J Mihatsch M & Polasek-W. Hartmann P & Ramseier. "Normal weight of the brain in adults in relation to age, sex, body height and weight". In: *Der Pathologe (Pathologe)* (1994).

[Cow00] D.H.; Kandel E.R. Cowan W.M.; Harter. "The emergence of modern neuroscience: Some implications for neurology and psychiatry". In: (2000). DOI: 10.1146/annurev.neuro.23.1.343.

[BP01] Guo qiang Bi and Mu ming Poo. "SYNAPTIC MODIFICATION BY CORRELATED ACTIVITY: Hebb's Postulate Revisited". In: (2001). DOI: https://doi.org/10.1146/annurev.neuro.24.1.139.

[Izh03] Eugene M. Izhikevich. *Simple Model of Spiking Neurons*. 2003. URL: https://www.izhikevich.org/publications/spikes.pdf.

[NFTB03] C. van Vreeswijk N. Fourcaud-Trocme D. Hansel and N. Brunel. "How spike generation mechanisms determine the neuronal response to fluctuating inputs". In: (2003). URL: https://neurophys.biomedicale.parisdescartes.fr/~carl/papers/jns2003.pdf.

[BG05] Romain Brette and Wulfram Gerstner. "Adaptive exponential integrate-and-fire model as an effective description of neuronal activity". In: *Journal of neurophysiology* 94.5 (2005), pp. 3637–3642.

[Day06] P. Dayan. "Levels of Analysis in Neural Modeling. Encyclopedia of Cognitive Science." In: (2006). DOI: Dayan,P.(2006).LevelsofAnalysisinNeuralModeling EncyclopediaofCognitiveScience.doi:10.1002/0470018860.s00363.

[Eli06] Deborah M. Elias Lorin J. & Saucier. *Neuropsychology: Clinical and Experimental Foundations*. Boston: Pearson/Allyn & Bacon, 2006. ISBN: 978-0-20534361-4.

[Lli08] Rodolfo Llinas. *Neuron*. 2008. URL: http://www.scholarpedia.org/article/Neuron.

[SP08] Roth A Häusser M. Sjöström PJ Rancz EA. "Dendritic excitability and synaptic plasticity". In: (2008). DOI: https://doi.org/10.1152/physrev.00016.2007.

[Xil08] Xilinx. *Xilinx User Guide*. 2008. URL: https://www.xilinx.com/support/documentation/user_guides/ug070.pdf.

[APDY09] Jochen Eppler Jens Kremkow Eilif Muller Dejan Pecevski Laurent Perrinet Andrew P. Davison Daniel Brüderle and Pierre Yger. "PyNN: a common interface for neuronal network simulators". In: (2009). DOI: https://doi.org/10.3389/neuro.11.011.2008.

[GB09] Dan F. M. Goodman and Romain Brette. "The Brian simulator". In: (2009). DOI: https://doi.org/10.3389/neuro.01.026.2009.

[YFCW09] Robert Hartmann Clive McCarthy Yiu-Fai Chan Robert Frankovich and Don Wong. *Altera EP300 Design Development Oral History Panel*. 2009. URL: http://archive.computerhistory.org/resources/access/text/2012/10/102702147-05-01-acc.pdf.

[Dav11] G. Davey. *Applied Psychology*. John Wiley & Sons, 2011. ISBN: 1444331213.

[Sch11] Daniel T. & Wegner Daniel M. Schacter Daniel L. & Gilbert. *Psychology (2nd ed.)* New York: Worth Publishers, 2011. ISBN: 978-1-4292-3719-2.

[Sab12] Kirk Saban. *Xilinx Inter*. 2012. URL: https://www.xilinx.com/support/documentation/white_papers/wp380_Stacked_Silicon_Interconnect_Technology.pdf.

[Seb12] Sebastian023. *Brain lobes, main sulci and boundaries*. 2012. URL: https://en.wikipedia.org/wiki/Frontal_lobe#/media/File:LobesCaptsLateral.png.

[GSJ13] Jr. Gordon S. Johnson. *About Brain Injury: A Guide to Brain Anatomy*. 2013. URL: https://waiting.com/brainanatomy.html.

[GS14] Christos Strydis Ioannis Sourdis Cătălin Ciobanu Oskar Mencer Chris I. De Zeeuw Georgios Smaragdos Craig Davies. "Real-Time Olivary Neuron Simulations on Dataflow Computing Machines". In: (2014). DOI: https://doi.org/10.1007/978-3-319-07518-1_34.

[Gio14] Panagiotis Giotakos. *Educational Neuroscience a bibliographic review*. Tech. rep. University of Ioannina, 2014.

[Cra15] Alexa Crawls. *History of FPGA*. 2015. URL: https://web.archive.org/web/20070412183416/http://filebox.vt.edu/users/tmagin/history.htm.

[JCK15] Minwook Ahn Jungmin Choi and Jong Tae Kim. "Implementation of Hardware Model for Spiking Neural Network". In: (2015).

[QWJC15] Xi Huang Rongtai Cai QingXiang Wu Xiaodong Liao and Jinqing Liu Jianyong Cai. "Development of FPGA Toolbox for Implementation of Spiking Neural Networks". In: (2015). DOI: 10.1109/CSNT.2015.216.

[RPC15] P Jesper Sjöström Mark CW van Rossum Rui Ponte Costa Robert C Froemke. "Unified pre- and postsynaptic long-term plasticity enables reliable and flexible learning". In: (2015). DOI: https://doi.org/10.7554/eLife.09457.001.

[Spl15] Thomas Splettstoesser. *Schematic of a synapse*. 2015. URL: https://commons.wikimedia.org/w/index.php?curid=41349083.

[Wu15] Xin Wu. *Xilinx 3d IC*. 2015. URL: https://www.xilinx.com/publications/white-papers/3d-ic-in-3d-fpgas.pdf.

[KCL16] Simon R. Schultz Kit Cheung and Wayne Luk. "NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors". In: (2016). DOI: https://doi.org/10.3389/fnins.2015.00516.

[GS17] Rahul Kukreja Harry Sidiropoulos Dimitrios Rodopoulos Ioannis Sourdis Zaid Al-Ars Christoforos Kachris Dimitrios Soudris Chris I. De Zeeuw Christos Strydis Georgios Smaragdos Georgios Chatzikonstantis. "BrainFrame: A node-level heterogeneous accelerator platform for neuron simulations". In: (2017). DOI: https://doi.org/10.1088/1741-2552/aa7fc5.

[SYKAL17] Bin Deng Chen Liu Member IEEE Huiyan Li Chris Fietkiewicz Shuangming Yang Jiang Wang and IEEE Kenneth A. Loparo Life Fellow. "Real-Time Neuromorphic System for Large-Scale Conductance-Based Spiking Neural Networks". In: (2017). DOI: 10.1109/TCYB.2018.2823730.

[Sou17] Dimitrios Soudris. *Xilinx Logic Block*. 2017. URL: http://mycourses.ntua.gr/courses/ECE1207/document/dsoudris/lecture-0-soudris-methodolofies-intro.pdf.

[Tec17] Maxeler Technologies. "Maxcompiler 2017.2.1 Tutorial, Multiscale Dataflow Programming". In: (2017).

[Zal17] Edward N. Zalta. *Descartes and the Pineal Gland*. 2017. URL: https://plato.stanford.edu/archives/win2017/entries/pineal-gland/.

[Ama18] Amazon. *Amazon AWS FPGA Marketplace*. 2018. URL: https://aws.amazon.com/marketplace/pp/B06VVYBLZZ.

[Kat18] Konstantinos Katsantonis. *Accelerating recommender systems on multiple FPGA platforms*. Tech. rep. 2018. URL: http://dspace.lib.ntua.gr/handle/123456789/46783.

[res18] researchandmarkets.com. *Data Center Accelerator Market by Processor Type (CPU, GPU, FPGA, ASIC), Type (HPC Accelerator, Cloud Accelerator), Application (Deep Learning Training, Public Cloud Interface, Enterprise Interface), and Geography - Global Forecast to 2023*. Tech. rep. 2018. URL: https://www.researchandmarkets.com/research/lbcg9s/data_center?w=4.

[RMWS18] Chetan S. Thakur Runchun M. Wang and André van Schaik. "An FPGA-Based Massively Parallel Neuromorphic Cortex Simulator". In: (2018). DOI: 10.3389/fnins.2018.00213.

[vdP18] Atze van der Ploeg. *Why use an FPGA instead of a CPU or GPU?* 2018. URL: https://blog.esciencecenter.nl/why-use-an-fpga-instead-of-a-cpu-or-gpu-b234cd4f309c.

[Xil18] Xilinx. *Xilinx Vivado*. 2018. URL: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.