Εθνικο Μετσοβιο Πολυτεχνειο
Τμημα Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Implementation and Acceleration of Neuron Simulator with CUDA C

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αλέξανδρου Νεοφύτου

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής

Αλέξανδρος Νεοφύτου

Αθήνα, Ιούνιος 2019

(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

Implementation and Acceleration of Neuron Simulator with CUDA C

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αλέξανδρου Νεοφύτου

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την Τρίτη 18 Ιουνίου 2019.

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

Αθήνα, Ιούνιος 2019

....
Αλέξανδρος Νεοφύτου
Διπλωματούχος Φοιτητής του Εθνικού Μετσόβιου Πολυτεχνείου

Περίληψη

Η νευροεπιστήμη είναι η επιστημονική μελέτη του νευρικού συστήματος και της σχέσης των νευρώνων με τη συμπεριφορά και τη μάθηση. Η μεγαλύτερη προσπάθεια για τους νευροεπιστήμονες επικεντρώνεται στον εγκέφαλο, καθώς η αυξημένη κατανόησή του οδηγεί σε πιο σαφή κατανόηση της ανθρώπινης συνείδησης. Ενώ τα περισσότερα πειράματα στο παρελθόν διεξήχθησαν σε εξειδικευμένα εργαστήρια που μελετούσαν τμήματα πραγματικών εγκεφαλικών νευρώνων, στη σύγχρονη νευροεπιστήμη οι υπολογιστές χρησιμοποιούνται ευρέως για την προσομοίωση βιολογικών νευρωνικών δικτύων με μεγάλη λεπτομέρεια και πολυπλοκότητα. Αυτές οι προσομοιώσεις επιτρέπουν την απεικόνιση δικτύων μεγαλύτερου μεγέθους, βοηθώντας τους περαιτέρω στην έρευνά τους.

Προκειμένου να συνεχιστεί η εξέλιξη του συγκεκριμένου επιστημονικού κλάδου, χρησιμοποιούνται ακόμα πιο ευρεία και σύνθετα δίκτυα νευρώνων σε προσομοιώσεις, δημιουργώντας την ανάγκη για επιτάχυνση προσομοίωσης σε διαφορετικές πλατφόρμες και αρχιτεκτονικές που χρησιμοποιούνται από τα αντίστοιχα εργαστήρια. Ενώ υπάρχουν αρκετοί προσομοιωτές για μια ευρεία επιλογή προσομοιώσεων μοντέλων νευρώνων, η πλειονότητα δεν είναι βελτιστοποιημένη για σύγχρονα συστήματα υπολογιστών και συνεπώς δεν επιτυγχάνει βέλτιστη απόδοση, καθυστερώντας τους νευροεπιστήμονες από σημαντικά συμπεράσματα που προκύπτουν από τα αποτελέσματα προσομοίωσης.

Αυτή η διπλωματική εργασία στοχεύει να κατευνάσει την ανάγκη για επιταχυνόμενη προσομοίωση χρησιμοποιώντας το CUDA API για επιτάχυνση σε ένα σύστημα NVIDIA GPU. Το μοντέλο στο οποίο επικεντρώνεται αυτή η εργασία είναι το μοντέλο νευρώνα Adaptive Exponential Integrate-and-Fire με Spike-Timing Dependent Plasticity στις συνάψεις του, που χρησιμοποιείται ευρέως στη σύγχρονη έρευνα. Η αρχική προσομοίωση εισήχθη αρχικά από τον προσομοιωτή Brian σε ένα νέο προσομοιωτή γραμμένο στη γλώσσα προγραμματισμού C και στη συνέχεια αναπτύχθηκε για την αποτελεσματική επιτάχυνση σε GPU. Η πλατφόρμα CUDA είναι ένα στρώμα λογισμικού που παρέχει άμεση πρόσβαση στο σύνολο εικονικών εντολών της GPU και σε παράλληλα υπολογιστικά στοιχεία για γενικές εφαρμογές προγραμματισμού.

Αυτή η υλοποίηση ήταν επιτυχής στην επιτάχυνση της προσομοίωσης νευρώνων πάνω από 100 φορές σε σύγκριση με τον προσομοιωτή Βριαν, περιστασιακά φτάνοντας ακόμα και ένα ρυθμό επιτάχυνσης 1000x διατηρώντας παράλληλα την ίδια λειτουργικότητα με τον προσομοιωτή Brian. Δεν υπάρχει θεωρητικό όριο στην ποσότητα νευρώνων που περιέχονται στο δίκτυο, αν και η απόδοση παρατηρήθηκε να μειώνεται σημαντικά καθώς τα όρια της μνήμης των GPU ξεπεράστηκαν. Το γεγονός αυτό αποτέλεσε σημείο ενδιαφέροντος και διερευνήθηκε περαιτέρω μαζί με άλλες παρατηρήσεις σε αυτή τη διπλωματική εργασία.

Λέξεις Κλειδιά

NVIDIA, CUDA, Parallel Programming, Προσομοίωση Νευρώνων, Brian, GPGPU, Παραλληλοποίηση, Adaptive Exponential Integrate-and-Fire model, STDP, in-silico experiment

Abstract

Neuroscience is the scientific study of the nervous system and the relation of nerves to behaviour and learning. The biggest effort for neuroscientists is focused on the brain as its increased understanding results in more clear knowledge about human consciousness. While most experiments in the past were conducted in specialised laboratories studying portions of actual brain neurons, in modern neuroscience computers are widely utilised to simulate biological neural networks in great detail and complexity. These simulations enable the visualization of networks of greater size, aiding them further in their research.

In order for this discipline to continue advancing, even more sizeable and complex neuron networks are being used in simulations, generating the need for simulation acceleration in different platforms and architectures used by the corresponding laboratories. While plenty of simulators are available for a wide selection of neuron model simulations, the majority is not optimized for modern computer systems and subsequently doesn't achieve optimal performance, delaying neuroscientists from important conclusions drawn from simulation results.

This diploma thesis aims to appease the need for accelerated simulation by utilizing the CUDA API for acceleration on an NVIDIA GPU system. The model this thesis is focused on is the Adaptive Exponential Integrate-and-Fire neuron model with Spike-timing Dependent Plasticity on its synapses, widely used in modern research. The original simulation was firstly imported from the Brian Simulator into a new simulator written in the C programming language and then developed for efficient GPU acceleration. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements for general purpose programming applications.

This implementation was successful in accelerating the neuron simulation a factor of over 100x times in comparison to the Brian Simulator, occasionally reaching even a 1000x acceleration rate while keeping the same functionality with the Brian Simulator. There is no theoretical limit in the amount of neurons contained in the network, though performance was observed to drop significantly as GPU memory limits were surpassed. This fact constituted a point of interest and was investigated further along with other observations in this diploma thesis.

Keywords

NVIDIA, CUDA, Parallel Programming, Neuron Simulation, Brian, GPGPU, 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 για την πολύτιμη βοήθειά του.

Contents

Εκτεταμένη Περίληψη **1**
Εισαγωγή . . . 1
Σχετικό Έργο στην Επιταχυνόμενη Υπολογιστική Νευροεπιστήμη . . . 5
Περιγραφή Προβλήματος . . . 5
Υλοποίηση του Προσομοιωτή σε C . . . 6
Υλοποίηση Παράλληλου Προγραμματισμού . . . 7
Αποτελέσματα . . . 12

1 Introduction **20**
1.1 Neuroscience . . . 20
1.1.1 Computational Neuroscience . . . 21
1.1.1.1 History . . . 21
1.1.2 Neurons . . . 22
1.1.3 Synapses . . . 25
1.1.4 Levels of Analysis in Neural Modeling . . . 25
1.1.5 Types of Neural Modeling . . . 26
1.1.5.1 Conventional reductive models . . . 26
1.1.5.2 Computational interpretive models . . . 26
1.1.6 Degrees of Modeling Detail of Neurons . . . 27
1.1.6.1 Conductance-based models . . . 27
1.1.6.2 Integrate-and-fire models . . . 27
1.1.6.3 Firing-rate models . . . 28
1.2 Accelerated Computation via GPU . . . 28
1.2.1 Graphics Processing Unit . . . 28
1.2.2 General-Purpose Computing on GPU . . . 29
1.2.3 Development Environment of GPGPU . . . 30
1.2.3.1 Early stages . . . 30
1.2.3.2 The CUDA programming model . . . 31
1.2.3.3 Other GPGPU frameworks . . . 32

2 Related Work on Accelerated Computational Neuroscience **34**
2.1 Popular Neural Simulation Frameworks . . . 34
2.2 GPU-accelerated Simulators . . . 35

2.3 BrainFrame 37

3 Problem Statement **38**

3.1 Adaptive Exponential Integrate-and-fire (AdEx) Neuron Model 39

3.2 Spike-Timing Dependent Plasticity (STDP) 40

4 Initial Approach **42**

4.1 Brian Architecture 42

4.1.1 Input Neurons 42

4.1.2 Model Neurons 43

4.1.3 Synapses 44

4.1.4 Brian Code 44

4.2 C Simulator Architecture 46

5 Parallel Programming Optimization **51**

5.1 CUDA implementation 51

5.1.1 General Design Features 51

5.1.2 Data Structures and Memory Handling 52

5.1.3 Kernel Design 55

5.1.3.1 Device code 55

5.1.3.2 CPU code 60

5.1.4 Large Scale Experiments Support 61

5.1.4.1 Single GPU 61

5.1.4.2 Dual GPU 63

5.1.5 CUDA Optimizations 65

5.1.5.1 Float vs Double Datatype 65

5.1.5.2 Kernel Merge 66

5.1.5.3 Parallel Mean Calculation 66

5.1.5.4 AoS vs SoA 68

5.1.5.5 Shared Memory 69

5.1.5.6 Other Attempts 71

5.2 OpenMP implementation 73

6 Performance Analysis **76**

6.1 Simulation Environment 76

6.2 Experimental Results 78

6.2.1 Comparisons based on network size 78

6.2.2 Comparisons based on connectivity 83

6.2.3 Comparisons based on firing percentage 84

7 Conclusion **87**

7.1 Remarks 87

7.2 Future work 88

List of Figures

1 Επιτάχυνση GPU vs CPU vs OpenMP, MxM 13
2 Επιτάχυνση GPU vs CPU vs Brian, MxM 13
3 Επιτάχυνση GPU vs CPU vs Brian, NxM 14
4 Επιτάχυνση GPU vs CPU vs OpenMP, NxM 14
5 Επιτάχυνση GPU vs CPU, MxM 17
6 Επιτάχυνση GPU vs CPU, NxM 17
7 Επιτάχυνση GPU vs CPU, MxM 18
8 Επιτάχυνση GPU vs CPU, NxM 19

1.1 Common parts of a neuron 23

4.1 Neuron architectures 46

6.1 Acceleration GPU vs CPU vs OpenMP, MxM 79
6.2 Acceleration GPU vs CPU vs Brian, MxM 80
6.3 Acceleration GPU vs CPU vs Brian, NxM 80
6.4 Acceleration GPU vs CPU vs OpenMP, NxM 81
6.5 Acceleration GPU vs CPU, MxM 83
6.6 Acceleration GPU vs CPU, NxM 84
6.7 Acceleration GPU vs CPU, MxM 85
6.8 Acceleration GPU vs CPU, NxM 86

List of Tables

1 Επιτάχυνση ως προς Μέγεθος Δικτύου, Μεγαλύτερης Κλίμακας, MxM 16

6.1 Acceleration by Network Size, MxM 78
6.2 Acceleration by Network Size, NxM 79
6.3 Acceleration by Network Size, Large Scale, MxM 82
6.4 Acceleration by Amount of Synaptic Connections, MxM 83
6.5 Acceleration by Amount of Synaptic Connections, NxM 83
6.6 Acceleration by Amount of Firing Neurons, MxM 85
6.7 Acceleration by Amount of Firing Neurons, NxM 85

Εκτεταμένη Περίληψη

Εισαγωγή

Αυτό που ενέπνευσε αυτή τη διπλωματική εργασία ήταν η αυξανόμενη ανάγκη να μειωθεί ο χρόνος προσομοίωσης πειραμάτων που σχετίζονται με νευρωνικά δίκτυα για νευροεπιστήμονες. Παρόμοια με αρκετούς άλλους κλάδους, η έρευνα στη νευροεπιστήμη περιλαμβάνει σήμερα προσομοιώσεις σύνθετων μαθηματικών μοντέλων που προσπαθούν να εξηγήσουν τη λειτουργικότητα του εγκεφάλου. Η μεγάλη πολυπλοκότητα του εγκεφάλου, καθώς και η πολυπλοκότητα συμπεριφοράς και οργάνωσης των νευρώνων οδηγούν σε τεράστιες προσομοιώσεις που χρειάζονται πολύ χρόνο για να τρέξουν, καθυστερώντας την έρευνα. Στόχος της παρούσας εργασίας είναι να προσπαθήσει να ανακαλύψει τον βέλτιστο τρόπο επιτάχυνσης της προσομοίωσης του AdEx (Adaptive Exponential Integrate-and-Fire) μοντέλου νευρώνων, η οποία χρησιμοποιείται ευρέως σε νευροεπιστημονικά πειράματα. Αυτή η προσομοίωση περιλαμβάνει το φαινόμενο της πλαστικότητας εξαρτώμενης από τις στιγμές πυροδότησης νευρώνων για τις συνδέσεις των νευρώνων που ονομάζονται συνάψεις.

Χρησιμοποιήθηκαν πολλές μέθοδοι για την επίτευξη της τελικής επιτάχυνσης. Από την αρχή του πειράματος, η προσομοίωση ήταν υλοποιημένη για τον προσομοιωτή BRIAN Simulator, έναν φημισμένο λύτη νευρωνικών μοντέλων ανοιχτού κώδικα που λειτουργεί με Python. Προκειμένου να επιταχυνθεί αποτελεσματικά το μοντέλο, πρώτα η προσομοίωση έπρεπε να μεταφερθεί στη γλώσσα προγραμματισμού C, η οποία παρουσιάζει συνολικά καλύτερη απόδοση από την Python, καθώς είναι μια γλώσσα προγραμματισμού χαμηλότερου επιπέδου. Ακολούθως, χρησιμοποιήθηκε η διεπαφή προγραμματισμού εφαρμογών CUDA προκειμένου να εφαρμοστεί μια παράλληλη υπολογιστική έκδοση που επιτυγχάνει υψηλότερα επίπεδα επιτάχυνσης χρησιμοποιώντας μια GPU της Nvidia. Τέλος, το OpenMP API χρησιμοποιήθηκε για να επιταχύνει την προσομοίωση σε πολλαπλά θέματα της CPU και να διερευνήσει την κλίμακα επιτάχυνσης σε σύγκριση με την έκδοση GPU.

Νευροεπιστήμη

Η νευροεπιστήμη είναι η επιστημονική μελέτη του νευρικού συστήματος, ειδικά η σχέση των νεύρων με τη συμπεριφορά και τη μάθηση. Η μεγαλύτερη προσπάθεια των νευροεπιστημόνων επικεντρώνεται στον εγκέφαλο, καθώς η αυξημένη κατανόησή του, μαζί με τις βελτιωμένες μεθόδους μελετών, οδηγούν σε πιο σαφή γνώση σχετικά με τη φυσιολογική ανθρώπινη συμπεριφορά και την ψυχική ευεξία. Η γνώση του πώς ακριβώς λειτουργεί το νευρικό σύστημα μπορεί να βοηθήσει τους ερευνητές να βρουν τρόπους για την πρόληψη ή τη θεραπεία προβλημάτων που επηρεάζουν τον εγκέφαλο, το νευρικό σύστημα και το σώμα. Οι ανακαλύψεις του επιστημονικού κλάδου επιτρέπουν στους επιστήμονες να αναπτύξουν θεραπείες για νευροεκφυλιστικές νόσους (όπως η νόσος του Αλτσχάιμερ) και ψυχικές ασθένειες, βοηθώντας επίσης στην ανάπτυξη της Τεχνητής Νοημοσύνης. Αυτές οι εξελίξεις είναι πιθανό να προσφέρουν σημαντικά οφέλη για την κοινωνία και να έχουν επιπτώσεις σε ένα ευρύ φάσμα τομέων δημόσιας πολιτικής όπως η υγεία, η εκπαίδευση, το δίκαιο και η ασφάλεια.

Υπολογιστική Νευροεπιστήμη

Η υπολογιστική νευροεπιστήμη είναι ένας κλάδος της νευροεπιστήμης που στοχεύει στην κατανόηση του πληροφοριακού περιεχομένου των νευρικών σημάτων χρησιμοποιώντας μαθηματικά μοντέλα του νευρικού συστήματος σε πολλές διαφορετικές δομικές κλίμακες, συμπεριλαμβανομένων των βιοφυσικών, του κυκλώματος και των συστημάτων. Είναι μια διεπιστημονική επιστήμη που συνδέει τα διάφορα πεδία της νευροεπιστήμης, της γνωστικής επιστήμης και της ψυχολογίας με την ηλεκτρολογία, την επιστήμη των υπολογιστών, τα μαθηματικά και τη φυσική. Ο τελικός στόχος της υπολογιστικής νευροεπιστήμης είναι να εξηγήσει πώς χρησιμοποιούνται ηλεκτρικά και χημικά σήματα στον εγκέφαλο για να αντιπροσωπεύουν και να επεξεργάζονται πληροφορίες. Η καταγωγή της υπολογιστικής νευροεπιστήμης βρίσκεται κατά την κοινή επιστημονική γνώμη στο μαθηματικό μοντέλο που οι νικητές του Βραβείου Νόμπελ Allan Hodgkin και Andrew Huxley ανέπτυξαν για το δυναμικό δράσης μέλους ενός καλαμαριού το 1952. Αυτό είναι ιστορικά το πρώτο μοντέλο βιολογικών νευρώνων που εξηγεί τους ιονικούς μηχανισμούς στους οποίους βασίζεται η εκκίνηση και η διάδοση των δυνατοτήτων δράσης στον μεγάλο άξονα του καλαμαριού. Το μοντέλο Hodgkin-Huxley ισχύει για όλους τους νευράξονες και εξακολουθεί να χρησιμοποιείται μέχρι σήμερα:

$$i_m = g_{Na} m^3 h(V - E_{Na}) + g_K n^4 (V - E_K) + g_L (V - E_L)$$

Νευρώνες

Οι νευρώνες είναι ηλεκτρικά διεγερτικά κύτταρα που επικοινωνούν με άλλα κύτταρα μέσω εξειδικευμένων συνδέσεων που ονομάζονται συνάψεις. Αυτά είναι τα κύρια κυτταρικά στοιχεία που αποτελούν τη βάση της λειτουργίας του νευρικού συστήματος. Σχεδόν όλοι οι πολυκύτταροι οργανισμοί έχουν νευρώνες. Ένας τυπικός νευρώνας αποτελείται από ένα κεντρικό κυτταρικό σώμα, δενδρίτες και ένα μόνο άξονα. Οι νευρώνες γενικώς χαρακτηρίζονται από το σώμα τους που έρχεται σε διαφορετικά σχήματα, καθώς μπορεί να στερούνται δενδριτών ή να μην έχουν άξονα. Είναι εξειδικευμένα για την επεξεργασία και μετάδοση κυψελοειδών σημάτων. Λόγω της ποικιλίας των λειτουργιών που εκτελούνται σε διάφορα μέρη του νευρικού συστήματος, υπάρχει μεγάλη ποικιλία στο σχήμα, το μέγεθος και τις ηλεκτροχημικές ιδιότητές τους. Το σώμα περιέχει τον πυρήνα του κυττάρου και το μεγαλύτερο μέρος της γονιδιακής έκφρασης και συνθετικού μηχανισμού που είναι υπεύθυνος για τη σύνθεση πρωτεϊνών. Γενικά, οι νευρώνες μπορούν να περιγραφούν ως έχοντες έναν πόλο εισόδου και εξόδου, ο οποίος δεν είναι απόλυτος, καθώς μπορεί να υπάρχει απουσία διακλάδωσης. Ωστόσο, για την πλειονότητα των νευρώνων ο πόλος λήψης (εισόδου) αποτελείται από νήματα που εξέρχονται από το σώμα και ονομάζεται δενδρίτης. Οι δενδρίτες εμφανίζονται σε νευρώνες σπονδυλωτών απευθείας στο κελί του σώματος και συνήθως διακλαδίζουν άφθονα, όλο και πιο λεπτοί με κάθε διακλάδωση. Ο πόλος μετάδοσης (εξόδου) εγκαταλείπει το σώμα σε οίδημα που ονομάζεται λοφίσκος του αξόνου, ταξιδεύει μέχρι 1 μέτρο σε ανθρώπους ή περισσότερα σε άλλα είδη και καλείται άξονας. Διεξάγει ηλεκτροχημικά σήματα πολλαπλασιασμού που ονομάζονται δυναμικά δράσης. Αν και αυτός είναι ο γενικός κανόνας, μπορεί να υπάρχουν εξαιρέσεις π.χ. σε περιφερειακούς αισθητήριους νευρώνες όπου η είσοδος γίνεται μέσω αξόνων.

Συνάψεις

Οι νευρώνες επικοινωνούν μεταξύ τους μέσω συνάψεων, όπου είτε ο άξονας τερματικού ενός κυττάρου έρχεται σε επαφή με τον δενδρίτη, το σώμα ή, λιγότερο συχνά, τον νευρώτη ενός άλλου νευρώνα. Είναι δομές που επιτρέπουν σε ένα νευρώνα να περάσει ένα ηλεκτρικό ή χημικό σήμα σε άλλο νευρώνα ή στο κύτταρο τελεστή στόχου. Η μεμβράνη πλάσματος του νευρώνα διέλευσης σήματος (προσυναπτικής) έρχεται σε συνεγκατάσταση με τη μεμβράνη του στοχευόμενου (μετασυναπτικού) κυττάρου σε μια σύναψη. Τόσο η προσυναπτική όσο και η μετασυναπτική θέση περιέχουν εκτεταμένες συστοιχίες μοριακής μηχανής που συνδέουν τις δύο μεμβράνες μεταξύ τους και διεξάγουν τη διαδικασία σηματοδότησης.

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

Προγραμματισμός Γενικού Σκοπού με GPU

Το 2003, ο Mark Harris αναγνώρισε τη δυνατότητα χρήσης γραφικών μονάδων επεξεργασίας (GPU) για εφαρμογές γενικού σκοπού. Δεδομένου ότι οι GPU έχουν σχεδιαστεί με πολύ περισσότερα τρανζίστορ που διατίθενται στην επεξεργασία δεδομένων παρά τον έλεγχο της ροής ή την προσωρινή αποθήκευση δεδομένων, είναι εξειδικευμένα για εξαιρετικά παράλληλους εντατικούς υπολογισμούς, με αποτέλεσμα πολύ υψηλότερα ποσοστά λειτουργιών κινητής υποδιαστολής και πολύ υψηλότερους ρυθμούς MFLOPs από τους πολυεπίπεδους CPUs. Έτσι γεννήθηκε ο όρος προγραμματισμού γενικού σκοπού σε μονάδες επεξεργασίας γραφικών (GPGPU), δηλαδή η χρήση μιας μονάδας επεξεργασίας γραφικών (GPU), η οποία συνήθως χειρίζεται τον υπολογισμό μόνο για γραφικά υπολογιστή, για τον υπολογισμό σε εφαρμογές που παραδοσιακά χειρίζονται από την κεντρική επεξεργασία μονάδας (CPU). Οι ιδιότητες της GPU οδηγούν σε μια πολύ διαφορετική αρχιτεκτονική επεξεργαστών από τις παραδοσιακές

CPU. Οι επεξεργαστές αφιερώνουν πολλούς πόρους, κυρίως από την περιοχή των τσιπ, για να εκτελούν γρήγορα μεμονωμένα ρεύματα οδηγιών, συμπεριλαμβανομένης της προσωρινής αποθήκευσης για την απόκρυψη της λανθάνουσας μνήμης και η εκτέλεση εντολών εκτός σειράς. Οι GPU, από την άλλη πλευρά, χρησιμοποιούν την περιοχή τσιπ για μεμονωμένα στοιχεία επεξεργασίας που εκτελούν ταυτόχρονα ένα μόνο ρεύμα εντολών σε πολλαπλά στοιχεία δεδομένων. Η λανθάνουσα μνήμη αποκρύπτεται με γρήγορη εναλλαγή περιβάλλοντος. Οι ιδανικές εφαρμογές GPGPU χαρακτηρίζονται από μεγάλα σύνολα δεδομένων, υψηλό παραλληλισμό και ελάχιστη εξάρτηση μεταξύ των στοιχείων δεδομένων. Επιπλέον, είναι σημαντικό οι εφαρμογές GPGPU να έχουν υψηλή υπολογιστική ένταση αλλιώς η λανθάνουσα πρόσβαση στην μνήμη θα περιορίσει την υπολογιστική ταχύτητα. Η αριθμητική ένταση ορίζεται ως ο αριθμός των πράξεων που εκτελούνται ανά μεταφερόμενη λέξη (word μνήμης.

Το Μοντέλο Προγραμματισμού CUDA

Η πλατφόρμα CUDA είναι ένα στρώμα λογισμικού που παρέχει άμεση πρόσβαση στο εικονικό σύνολο εντολών της GPU και σε παράλληλα υπολογιστικά στοιχεία για την εκτέλεση υπολογιστικών πυρήνων. Πρακτικά διευκολύνει τους ειδικούς στον παράλληλο προγραμματισμό χρησιμοποιώντας τους πόρους των GPU χωρίς να διαθέτει προηγμένες δεξιότητες στον προγραμματισμό γραφικών. Το μοντέλο παράλληλου προγραμματισμού CUDA έχει τρεις κύριες αφαιρετικές έννοιες - μια ιεραρχία ομάδων νήματος, κοινές μνήμες και φραγμούς συγχρονισμού. Κάθε μπλοκ νημάτων μπορεί να προγραμματιστεί για εκτέλεση σε οποιονδήποτε από τους διαθέσιμους πυρήνες επεξεργαστών, ταυτόχρονα ή διαδοχικά. Αυτό επιτρέπει σε ένα πρόγραμμα CUDA να εκτελεστεί σε οποιοδήποτε αριθμό πυρήνων επεξεργαστών. Υπάρχουν τρία επίπεδα ιεραρχίας για τα νήματα CUDA. Νήματα, μπλοκ νημάτων και πλέγματα. Κάθε νήμα εκτελεί μια ακολουθία οδηγιών ή πυρήνα, δηλ. Το 'πρόγραμμα' που όλα τα νήματα εκτελούν ταυτόχρονα. Επίσης έχει ένα μοναδικό αναγνωριστικό το οποίο μπορεί να χρησιμοποιηθεί για τον προσδιορισμό των λειτουργιών που αφορούν το νήμα μέσα στον πυρήνα. Τα νήματα ομαδοποιούνται σε μπλοκ νημάτων που περιέχουν οποιοδήποτε αριθμό των νημάτων μέχρι κάποιο όριο, το οποίο σε σύγχρονες συσκευές είναι 1.024 threads ανά μπλοκ. Τα μπλοκ εκτελούνται εντελώς ανεξάρτητα και μπορούν να οργανωθούν περαιτέρω σε δίκτυα. Ένα πλέγμα είναι ολόκληρη η συλλογή των νημάτων CUDA για την εκτέλεση ενός συγκεκριμένου πυρήνα. Στις GPU τρέχουσας γενιάς, υπάρχει μια προσωρινή μνήμη επιπέδου-2 (L2) που διαχειρίζεται υλικό και μοιράζεται μεταξύ όλων των πυρήνων πολλαπλών επεξεργαστών στη GPU και μνήμης cache επιπέδου-1 (L1) που είναι τοπική σε κάθε πυρήνα. Επιπλέον, υπάρχει η κοινόχρηστη μνήμη, η οποία είναι προγραμματισμένη και τοπική σε ένα μπλοκ νημάτων, που συνδυάζεται με την κρυφή μνήμη L1 για να έχει μέγεθος 64 KB. Τέλος, οι σύγχρονες μονάδες GPU προσφέρουν επιπλέον ποσότητα σταθερής μνήμης γενικής χρήσης μόνο 48KB, καθώς και μεγάλα αρχεία καταγραφής (register files) για την υποστήριξη hardware multithreading.

Σχετικό Έργο στην Επιταχυνόμενη Υπολογιστική Νευροεπιστήμη

Δεδομένου ότι η υπολογιστική νευροεπιστήμη βασίζεται σε μεγάλο βαθμό στη μαθηματική μοντελοποίηση προκειμένου να αναλύσει τη λειτουργικότητα του εγκεφάλου, τα πειράματα που σχετίζονται με τον τομέα βασίζονται σε βαριές αριθμητικές πράξεις που παράγονται για παράδειγμα με τις μεθόδους που απαιτούνται για την επίλυση των διαφορικών εξισώσεων που περιγράφουν το δεδομένο νευρικό μοντέλο. Επομένως, τα πειράματα πρέπει να εκτελούνται in silico, δηλ. προσομοιώσεις που εκτελούνται σε υπολογιστές. Το γεγονός ότι οι νευρώνες και οι συνάψεις τους είναι σύνθετες οντότητες που χρειάζονται έναν τεράστιο αριθμό παραμέτρων για την αποτελεσματική περιγραφή της δραστηριότητάς τους, καθώς και η ανάγκη διεξαγωγής πειραμάτων σχετικά με τη δραστηριότητα συστημάτων που αποτελούνται από εκατοντάδες ή χιλιάδες νευρώνες οδηγεί σε αυξανόμενη ζήτηση της υπολογιστικής ισχύος. Επιπλέον, προκειμένου να διευκολυνθούν οι επιστήμονες στη διεξαγωγή μεγάλου αριθμού πειραμάτων χωρίς τεχνογνωσία στον προγραμματισμό, ενώ δεν υπάρχει ενιαίο εργαλείο προσομοίωσης που να ανταποκρίνεται στο σύνολο των αναγκών της υπολογιστικής νευρολογίας, μια ποικιλία πλαισίων προσομοίωσης έχει καθιερώσει το ρόλο τους στην πειθαρχία. Μερικά από τα πιο ευρέως χρησιμοποιούμενα πλαίσια προσομοιωτή νευρώνων είναι το Neuron, το NEST, το BRIAN, το MOOSE και το Genesis. Ειδική αναφορά αξίζει στο Brainframe, από το οποίο προέκυψε αυτή η διπλωματική. Το Brainframe είναι μια online πλατφόρμα επιτάχυνσης με ιδιαίτερη χρηστικότητα, καθώς είναι προσαρμοσμένη στο σύστημα της PyNN και χρησιμοποιεί την ευρέως διαδεδομένη σε επιστημονικούς τομείς γλώσσα προγραμματισμού Python για τη διεπαφή της.

Περιγραφή Προβλήματος

Στη διαδικασία εμπλουτισμού του Brainframe με νέα μοντέλα νευρώνων, η ευθύνη που ανέλαβα για τη διπλωματική μου ήταν η προσθήκη ενός Προσαρμοστικού Εκθετικού Μοντέλου Συσσώρευσης-και-Πυροδότησης (Adaptive Exponential Integrate-and-Fire model) (AdEx), το οποίο χαρακτηρίζεται επίσης από την πλαστικότητα εξαρτώμενη από τις στιγμές πυροδότησης των νευρώνων (Spike-timing Dependent Plasticity, STDP). Αυτό το μαθηματικό μοντέλο εμπνεύστηκε από μια δημοσίευση νευροεπιστημών που διερεύνησαν την ικανότητα των συνάψεων να απομνημονεύσουν μια συγκεκριμένη συμπεριφορά και αν μπορούσαν γρήγορα να ανακαλέσουν αυτή τη συμπεριφορά αν ενεργοποιούνταν από παρόμοιες αιχμές. Ένας συμφοιτητής και εγώ αναλάβαμε το καθήκον να επιταχύνουμε το χρόνο προσομοίωσης αυτού του μοντέλου σε διαφορετικές πλατφόρμες πριν το εισάγουμε στο Brainframe. Και οι δύο πλατφόρμες απαιτούσαν να εισάγουμε την προσομοίωση του συγκεκριμένου μοντέλου σε ένα προσαρμοσμένο πρόγραμμα C. Αυτό οδήγησε σε εμένα και τον συνάδελφο να αποκωδικοποιήσουμε τον ακριβή τρόπο με τον οποίο εργάζεται ο προσομοιωτής Brian και σχεδιάζοντας έναν προσομοιωτή συγκεκριμένου μοντέλου στη γλώσσα προγραμματισμού C που μπορεί να διεξάγει πειράματα σχετικά μόνο με το μοντέλο αναφοράς, επιτρέποντας ωστόσο στον χρήστη να καθορίζει τιμές παραμέτρων και για τις δύο τους νευρώνες και τις συνάψεις τους. Στη συνέχεια, η δική μου ευθύνη ήταν να επιτύχω βέλτιστη απόδοση χρησιμοποιώντας την παράλληλη υπολογιστική πλατφόρμα CUDA, επιταχύνοντας τις προσομοιώσεις στις μονάδες GPU.

Το Προσαρμοστικό Εκθετικό Μοντέλο Συσσώρευσης-και-Πυροδότησης είναι ένα μοντέλο νευρώνα με δύο μεταβλητές. Η πρώτη εξίσωση περιγράφει τη δυναμική του δυναμικού της μεμβράνης και περιλαμβάνει έναν όρο ενεργοποίησης με μια εκθετική τάση εξάρτησης. Η τάση συνδέεται με μια δεύτερη εξίσωση η οποία περιγράφει την προσαρμογή του νευρώνα. Και οι δύο μεταβλητές επαναφέρονται αν ενεργοποιηθεί ένα δυναμικό δράσης. Αναλυτικά οι εξισώσεις του μοντέλου μπορούν να βρεθούν στο αγγλικό τμήμα της παρούσας διπλωματικής. Ο συνδυασμός της προσαρμογής και της εκθετικής τάσης αποτελεί την προέλευση του ονόματος του μοντέλου.

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 από 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 και με τον προ- και τον μετα-συναπτικό νευρώνα και τα σχετικά φαινόμενα. Οι αναλυτικές εξισώσεις επίσης βρίσκονται στο αγγλικό τμήμα της διπλωματικής.

Υλοποίηση του Προσομοιωτή σε C

Ένα σημαντικό μέρος της εφαρμογής ήταν η επιλογή των δομών δεδομένων που θα χρησιμοποιηθούν για να διατηρηθούν όλα τα απαραίτητα δεδομένα των νευρώνων και των συνάψεών τους. Ένας μονοδιάστατος πίνακας από ακεραίους μήκους **N_S** + **N_Group_S** χρησιμοποιείται για να αντιπροσωπεύει συμβάντα πυροδότησης στους αντίστοιχους νευρώνες, με το όνομα *SpikeArray*. Δεδομένου ότι ο μοναδικός σκοπός των εισερχομένων νευρώνων είναι η παροχή συμβάντων πυροδότησης, αντιπροσωπεύονται επαρκώς από τις **N_S** θέσεις του πίνακα. Ωστόσο, ο πίνακας συμπληρώνεται πρώτα από όλες τις απαραίτητες θέσεις για AdEx Νευρώνες πριν από αυτές που αντιπροσωπεύουν τους Νευρώνες Εισόδου.

Λόγω των διαφορετικών μεταβλητών του μοντέλου νευρώνων AdEx που χρειάζονται εξοικονόμηση προκειμένου να επιλυθούν σωστά οι εξισώσεις μοντέλων σε κάθε χρονική στιγμή, οι Νευρώνες AdEx αντιπροσωπεύονται από μια δομή που ονομάζεται Neuron η οποία περιέχει τις μεταβλητές που κρατούν την τιμή του κατωφλίου τάσης, το δυναμικό μεμβράνης, το ρεύμα και τη μεταβλητή προσαρμογής του μοντέλου, που όλα ορίζονται ως *double* για επαρκή ακρίβεια, καθώς και ένα πεδίο ακέραιων τιμών των οποίων η αξία εξηγεί εάν ο νευρώνας έχει πυροδοτηθεί στο τρέχον χρονικό διάστημα. Ενώ αυτό αντιπροσωπεύεται από το *SpikeArray*, έχει διατηρηθεί για να εξασφαλίσει πλήρη αντιστοιχία με τον αρχικό κώδικα Brian, όπως ζητήθηκε. Η ίδια αρχή εφαρμόζεται σε πολλά τμήματα κώδικα του προσομοιωτή μας χωρίς σημαντική επιβάρυνση μνήμης.

Το μοντέλο STDP και οι διαφορικές εξισώσεις που περιγράφουν τις συνάψεις οδηγούν στην υλοποίηση του struct *Synapse* για να αποθηκευτούν όλες οι σχετικές πληροφορίες που χρειάζονται οι συνάψεις. Αυτό περιλαμβάνει τις συναπτικές μεταβλητές, ένα πεδίο που διατηρεί την τελευταία φορά που μια τιμή της δομής ενημερώθηκε, το ρεύμα που πρόκειται να μεταδοθεί στον μετασυναπτικό νευρώνα και τέλος μια μεταβλητή που στην πραγματικότητα εκφράζει εάν η δεδομένη σύναψη υπάρχει από την άποψη του πειράματος. Η εξήγηση του τελευταίου είναι ότι τόσο τα σύνολα των νευρώνων όσο και των συνάψεων σώζονται ως δύο πίνακες δομών (AoS). Αυτή η διάταξη προτιμήθηκε αρχικά λόγω της χωρικής τοπικότητας μεταξύ των διασυνδεδεμένων μεταβλητών και προσέφερε πλεονεκτήματα απόδοσης σε σύγκριση με μία μοναδική δομή από πίνακες (SoA) για κάθε στοιχείο, στην αρχική σειριακή περίπτωση. Επομένως, οι συνάψεις αποθηκεύονται σε μια συστοιχία παρακέντησης (των δομών) μεγέθους **NxM** ή **MxM**, ανάλογα με την προσομοίωση. Η μεταβλητή *conn* ορίζει τη συνδεσιμότητα, η οποία μπορεί να διαβαστεί από ένα αρχείο εισόδου καθώς και να παραμείνει στο προκαθορισμένο χαρακτηριστικό 50 %.

Σε κάθε βήμα της προσομοίωσης χρειάζεται να εκτελεστούν 4 βασικές λειτουργίες με την σειρά με την οποία θα αναφερθούν, λόγω μεταξύ τους εξαρτήσεων. 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 έχουν ως στόχο να επιλύουν τις διαφορικές εξισώσεις των μοντέλων των νευρώνων και των συνάψεων.

Υλοποίηση Παράλληλου Προγραμματισμού

Μετά την υλοποίηση του προσομοιωτή σε C σε συνεργασία με έναν άλλο φοιτητή, ο προσωπικός μου στόχος σε αυτή την εργασία ήταν να αναπτύξω μια παράλληλη υπολογιστική έκδοση του προσομοιωτή για να συγκρίνω τα κέρδη απόδοσης όταν τρέχω σε (max) παραλληλία στις GPU χρησιμοποιώντας το API CUDA. Επιπλέον, προγραμματίστηκε μια βασική έκδοση επιτάχυνσης που χρησιμοποίησε τη διεπαφή προγραμματισμού OpenMP για επιτάχυνση πολλαπλών νημάτων CPU, έτσι ώστε να κατανοηθούν αποτελεσματικά τα πραγματικά πλεονεκτήματα και μειονεκτήματα της χρήσης των GPU για τα πειράματα του συγκεκριμένου προσομοιωτή. Καθώς η έκδοση OpenMP τελεί καθαρά βοηθητικούς σκοπούς στο πλαίσιο της διπλωματικής, η περιγραφή της συγκεκριμένης έκδοσης αφήνεται αποκλειστικά στο πλήρες, αγγλικό τμήμα της εργασίας.

Λαμβάνοντας υπόψη ότι λόγω της αρχιτεκτονικής του προγράμματος C που αναλύθηκε όλοι οι υπολογισμοί του μοντέλου είναι ειδικοί για το νευρώνα και τη σύναψη, που σημαίνει ότι υπάρχουν ελάχιστες ή καθόλου εξαρτήσεις μεταξύ δεδομένων σε μία μόνο χρονική στιγμή, η προσέγγιση που επιλέχθηκε για την επιτάχυνση σε GPU ήταν να εκχωρηστεί κάθε δομή που χρησιμοποιείται για υπολογισμούς σε διαφορετικό νήμα. Κάθε νευρώνας και κατά συνέπεια κάθε σύναψη - ανάλογα τον πυρήνα - χειρίζεται με ένα μόνο, ανεξάρτητο νήμα, επιτρέποντας τον μέγιστο δυνατό παραλληλισμό και το μικρότερο δυνατό φορτίο υπολογισμού για κάθε νήμα. Η επιλογή αυτή εξασφαλίζεται με τον καθορισμό ενός μεγέθους δικτύου ανάλογου με τον αριθμό των προς επεξεργασία στοιχείων του δικτύου.

Τροποποίηση δομών δεδομένων

Δεδομένου ότι το CUDA είναι μια διεπαφή προγραμματισμού που υποστηρίζει πλήρως τη γλώσσα C, χρειάζονται μόνο λίγες τροποποιήσεις στις υπάρχουσες δομές δεδομένων του πηγαίου κώδικα του προσομοιωτή, λαμβάνοντας υπόψη ότι όλες οι απαραίτητες υπάρχουσες δομές δεδομένων διατίθενται στη συσκευή και τα δεδομένα τους αντιγράφονται χρησιμοποιώντας μια αποκλειστική λειτουργία CUDA. Πρώτα από όλα, πρέπει να σημειωθεί ότι για να επεξεργαστούν δεδομένα σε μια GPU εκτελώντας έναν πυρήνα CUDA, τα απαραίτητα δεδομένα πρέπει να διανεμηθούν ή / και να περάσουν στη συσκευή. Στην περίπτωση των συνάψεων, το CUDA API πρόσφερε δύο δυνατότητες για τη μετάδοση των δεδομένων τους στη συσκευή. Αυτό οφείλεται στο γεγονός ότι η συστοιχία δομών (AoS) που ενεργεί ως μήτρα συγγένειας καθώς και αποθηκεύει τις πληροφορίες που απαιτούνται για κάθε σύναψη μπορεί να είναι είτε μια κλασική δισδιάστατη συστοιχία C (των δομών) είτε μια μονοδιάστατη αναπαράσταση της ίδιας συστοιχίας, δεδομένου ότι οι σειρές μιας δισδιάστατης συστοιχίας μπορούν να αποθηκευτούν σε συνεχείς θέσεις μνήμης και να έχουν πρόσβαση με τη χρήση αριθμητικών δεικτών. Για λόγους εξοικονόμησης μνήμης επιλέχθηκε ένας μονοδιάστατος πίνακας από δομές συνάψεων. Οι υπεύθυνες συναρτήσεις για την κατανομή και τη μεταφορά τμημάτων μνήμης σε αυτήν την περίπτωση είναι οι *cudaMalloc* και *cudaMemcpy*.

Η μόνη άλλη διαφορά σχετικά με τις δομές δεδομένων είναι η προσθήκη ενός *double* datatype πίνακα που ονομάζεται *testvar*. Ο σκοπός του *testvar* είναι να ελαχιστοποιήσει την ποσότητα της μνήμης που πρέπει να μεταβιβαστεί στον κεντρικό υπολογιστή, καθώς κάθε σύναψη - κατά συνέπεια κάθε νήμα - αποθηκεύει την τιμή που θα προστεθεί στη μεταβλητή *μεαν* στις αντίστοιχες θέσεις του *testvar*. Στη συνέχεια, μόνο το *testvar* αντιγράφεται στον κεντρικό υπολογιστή, ο οποίος καταναλώνει σημαντικά λιγότερη μνήμη ως *double* πίνακας παρά για μια σειρά *Synapses*, καθιστώντας την λειτουργία *cudaMemcpy* γρηγορότερη.

Σχεδιασμός Πυρήνων

Η αρχική προσέγγιση για τον ορισμό των πυρήνων ήταν να προσπαθήσει να σχεδιαστεί το πρόγραμμα GPU σε αντιστοιχία με τον πηγαίο κώδικα του αρχικού προσομοιωτή. Συγκεκριμένα, ολόκληρη η προσομοίωση θα έχει την ίδια ροή με την έκδοση της CPU, αλλά αντί των κλήσεων προς τις λειτουργίες *SolveNeurons*, *UpdateSynapses_pre* και *UpdateSynapses _post* θα υπάρχουν κλήσεις στους αντίστοιχους παράλληλους πυρήνες. Δεδομένου ότι το σύνολο των αριθμητικών υπολογισμών που σχετίζονται με τα μοντέλα των νευρώνων και των συνάψεων πραγματοποιείται στις προαναφερθείσες λειτουργίες, φάνηκε επίσης ότι, αν υποτεθεί ότι ο προσομοιωτής χρειάζεται να εκπέμπει δεδομένα στον χρήστη μόνο μετά την ολοκλήρωση της προσομοίωσης, χρειάστηκαν μόνο τα δεδομένα προσομοίωσης να μεταφερθούν στη συσκευή μία φορά στο σύνολό τους στο βήμα αρχικοποίησης και να αποσταλούν πίσω στον κεντρικό υπολογιστή μόνο αφού κάθε μεταβλητή μοντέλου έχει αποκτήσει την τελική της τιμή. Σε κάθε πυρήνα δόθηκε το ίδιο όνομα με τις αρχικές λειτουργίες για απλότητα.

Στην περίπτωση του πυρήνα *UpdateSynapses_pre*, εμφανίστηκε το εξής δίλημμα: όταν περισσότεροι από ένας προ-συναπτικοί νευρώνες συνδέονται ταυτόχρονα με τον ίδιο μετά-συναπτικό νευρώνα και πυροδοτούνται ταυτόχρονα, κάθε τιμή *target_I* αντικαθιστά την τιμή που έχει ήδη καταχωρηθεί στο μετασυναπτικό νευρώνα. Λόγω της επανάληψης **κατά στήλες**, η τελική τιμή που καταγράφεται είναι του νευρώνα που αντιστοιχεί στο μεγαλύτερου αναγνωριστικού νήμα μεταξύ εκείνων των προ-συναπτικών νευρώνων, που ανιστοιχεί στον 'τελευταίο' κατά σειρά νευρώνα που πυροδότησε εκείνη τη χρονική στιγμή. Δεδομένου ότι δεν υπάρχει λειτουργία CUDA για την επιβολή συγχρονισμού νημάτων σε όλο τον πυρήνα, η αναπαραγωγή αυτής της συμπεριφοράς απαιτεί την έξοδο από τον πυρήνα πριν από αυτόν τον τελευταίο βρόχο, τη μεταφορά των δεδομένων *target_I* και την εκτέλεση του στον κώδικα της ῝ΠΥ, μεταφέροντας τη *neurons* δομή πίσω στη συσκευή για τους απαραίτητους υπολογισμούς. Ωστόσο, μεταξύ των κλήσεων CUDA, δημιουργείται η μεγαλύτερη χρονική ποινή όταν χρησιμοποιείται το *cudaMemcpy* (και τα παράγωγά του) για τη μεταφορά δεδομένων μεταξύ συσκευής και κεντρικού υπολογιστή.Επιπλέον, η σειρά που έχει περάσει το ρεύμα είναι αυστηρά ένας περιορισμός κώδικα του Brian, καθώς δεν υπάρχει υποστήριξη του Brian για να καθορίσει ποιος προ-συναπτικός νευρώνας κυριαρχεί επί της ταυτόχρονης πυροδότησης. Ούτε η δημοσίευση στην οποία βασίστηκε το αρχικό πείραμα ούτε τα μοντέλα των νευρώνων AdEx ή των συνάψεων που χαρακτηρίζονται από STDP παρέχουν κάποια σχετική πληροφορία. Τέλος, δεν υπάρχει πρόβλημα να έχουμε πολλαπλά νήματα που να γράφουν μια ενιαία (κοινόχρηστη ή παγκόσμια) θέση μνήμης στο CUDA, ακόμη και 'ταυτόχρονα' δηλαδή από την ίδια γραμμή κώδικα, εφόσον δεν υπάρχουν συνθήκες ταυτόχρονης ανάγνωσης-εγγραφής. Ως εκ τούτου, η επιλεγμένη προσέγγιση ήταν να επιτραπεί απλώς σε νήματα να ενημερώσουν παράλληλα τη μεταβλητή νευρώνων παράλληλα, χωρίς να ελέγχεται η σειρά που θα έχουν πρόσβαση σε αυτό το πεδίο, με αποτέλεσμα ένα σημαντικό κέρδος απόδοσης.

Τέλος, ο πυρήνας **UpdateSynapses_post**. Όπως αναφέρεται στην ενότητα 4.2, η συνάρτηση *UpdateSynapses_post* αποτελείται από δύο βρόχους που περνάνε όλες τις δομές συνάψεων κατά στήλες, γεγονός που καθιστά ελάχιστη τη διαφορά στην τροποποίηση της λειτουργίας που εκτελείται παράλληλα. Ωστόσο, ο υπολογισμός της μέσης τιμής ορισμένων συναπτικών μεταβλητών μεταξύ αυτών των βρόχων αποδείχθηκε πρόκληση. Αυτή η τιμή, κοινή σε όλες τις συνάψεις και συνεπώς κάθε νήμα, είναι απαραίτητη για τις λειτουργίες που εκτελούνται στον δεύτερο βρόχο, οπότε πρέπει να βεβαιωθεί ότι κανένα νήμα δεν έχει μεταφερθεί στον δεύτερο βρόχο πριν η μέση μεταβλητή έχει οριστεί στη σωστή τιμή. Δυστυχώς, το μοντέλο προγραμματισμού CUDA ειδικεύεται στην επίλυση προβλημάτων διασπώντας τα σε μπλοκ και συγχρονισμό νήματος εντός πυρήνα μπορεί να γίνει μόνο μέσα σε ένα μπλοκ χρησιμοποιώντας τη συνάρτηση CUDA *__syncthreads*. Ο συγχρονισμός εκτός πυρήνα CUDA μπορεί να επιτευχθεί μόνο με μεθόδους που σειριοποιούν την εκτέλεση (π.χ. κλείδωμα mutex). Έτσι, η πιο απλή λύση ήταν να χωριστεί η συνάρτηση σε δύο χωριστούς πυρήνες, οι οποίοι είναι υπεύθυνοι για την παράλληλη εκτέλεση κάθε ένθετου for-loop, επιτρέποντας στον κεντρικό επεξεργαστή (CPU) να χειριστεί τον υπολογισμό του μέσου τους διαβιβάζοντας όλα τα απαραίτητα δεδομένα μέσω του *testvar*. Δεδομένου ότι οι πυρήνες εκτελούνται αυτόματα διαδοχικά εάν εκκινήσουν στην προεπιλεγμένη ροή, όπως συμβαίνει στο σύνολο του σχετικού κώδικα της παρούσας διπλωματικής εργασίας, δεν υπάρχει λόγος να συγχρονιστεί ρητά η συσκευή για να εξασφαλιστεί ότι ο μέσος όρος υπολογίζεται.

Πειράματα Μεγάλης Κλίμακας

Σε πειράματα μεγέθους που ξεπερνάει τα όρια που τίθενται από τη συσκευή GPU για να παραλληλοποιηθεί πλήρως η εκτέλεση ως προς τις συνάψεις, έχουν αναπτυχθεί προσαρμογές του προσομοιωτή για μία αλλά και για περισσότερες από μία GPU. Στην περίπτωση μονής συσκευής, η προσαρμογή του προσομοιωτή στην εκτέλεση μεγαλύτερων πειραμάτων με πρακτικά κανένα όριο μεγέθους είναι αρκετά εύκολη από την άποψη της ανάπτυξης, καθώς αντί να καλείται κάθε πυρήνας μία φορά, εφαρμόζεται ένας βρόχος για κάθε πυρήνα, όπου σε κάθε επανάληψη το σύνολο των τμημάτων των δομών που μπορούν να επεξεργαστούν ταυτόχρονα μεταβιβάζεται στη συσκευή, ο πυρήνας εκτελείται και τα δεδομένα μεταφέρονται ξανά στον κεντρικό υπολογιστή, καθώς η επόμενη επανάληψη θα αντικαταστήσει τα δεδομένα της συσκευής. Αν και αυτή είναι πράγματι μια έγκυρη μέθοδος για την παροχή θεωρητικά απεριόριστης κλιμάκωσης για τα πειράματα του προσομοιωτή, υπάρχει ένα σημαντικό κόστος απόδοσης που προκαλείται από το γενικό κόστος των υποχρεωτικών μεταφορών μνήμης των κατατμήσεων των δομών μεταξύ της συσκευής και του κεντρικού υπολογιστή σε κάθε χρονική στιγμή.

Ακολουθώντας τις αρχές που εξηγούνται παραπάνω ενότητα, υλοποιήθηκε και δοκιμάστηκε και μια έκδοση που υποστηρίζεται από πολλαπλές GPU σε έναν κόμβο με δύο συσκευές. Η εφαρμογή πολλαπλών GPU προσφέρει αυξημένο εύρος ζώνης μνήμης, καθώς η διπλή ποσότητα δεδομένων μπορεί να μεταφερθεί στη μνήμη της συσκευής σχεδόν την ίδια στιγμή και η φύση των νευρώνων και των συναπτικών μοντέλων επιτρέπει την ανεξάρτητη ταυτόχρονη εκτέλεση διαφορετικών κομματιών των δεδομένων σε ξεχωριστές συσκευές, προσφέροντας ακόμα καλύτερες επιδόσεις. Το μεγαλύτερο κέρδος απόδοσης σε σύγκριση με τη χρήση μίας μονάδας GPU επιτυγχάνεται προφανώς όταν εκτελούνται πειράματα μεγέθους που προσεγγίζει το όριο και των δύο συσκευών, ώστε να επιτευχθεί μέγιστο εύρος ζώνης και ελαχιστοποιούνται οι μεταφορές μνήμης.

Βελτιστοποιήσεις CUDA

Μετά τη βασική υλοποίηση της επιταχυμένης έκδοσης του προσομοιωτή, ερευνήθηκαν διάφορες βελτιστοποιήσεις ως προς το σχεδιασμό και τις τεχνικές λεπτομέρειες. Αρχικά, δοκιμάστηκε η χρήση του 4-byte τύπου δεδομένων float αντί του 8-byte double, ώστε να μειωθούν οι απαιτήσεις μνήμης και να επιταχυνθούν οι υπολογισμοί σε κάθε χρονική στιγμή. Παρόλο που ο καινούριος τύπος δεδομένων επέφερε μειωμένη ακρίβεια στους μαθηματικούς υπολογισμούς, δεδομένου ότι το μέγιστο παρατηρούμενο σφάλμα για οποιεσδήποτε παραμέτρους προσομοίωσης δεν υπερέβη ποτέ το **1.00E-13**, η έκδοση με *φλοατς* έγινε αποδεκτή για να αντικαταστήσει το πρωτότυπο και προσέφερε σημαντικότατη αύξηση επιδόσεων. Επιπρόσθετα, διερευνήθηκε ο συνδυασμός πυρηνών CUDA σε μεγαλύτερους ενιαίους πυρήνες με κάθε πιθανό τρόπο, ωστόσο η μετατροπή αυτή δεν επέφερε κάποιο κέρδος και απεδείχθη ανούσια ως επιβαρύνουσα.

Στον παράλληλο προγραμματισμό, οι λειτουργίες μείωσης είναι εκείνες που μειώνουν μια συλλογή τιμών σε μία μόνο τιμή. Η άθροιση των στοιχείων μιας συστοιχίας είναι ένα κοινό παράδειγμα μιας διαδικασίας μείωσης, η οποία μπορεί να εξηγηθεί ως εξής:

- Υποθέτοντας τον N ως τον αριθμό των στοιχείων σε μια συστοιχία, δημιουργούνται N

/ 2 νήματα, ένα νήμα για κάθε δύο (διαδοχικά) στοιχεία.

- Κάθε νήμα υπολογίζει το άθροισμα των αντίστοιχων δύο στοιχείων, αποθηκεύοντας το αποτέλεσμα στη θέση του πρώτου.
- Επαναληπτικά, σε κάθε βήμα:
  - Ο αριθμός των νημάτων μειώνεται στο μισό.
  - κάθε υπολειπόμενο νήμα είναι πλέον υπεύθυνο για το αθροισμένο στοιχείο του και για αυτό του παρακείμενου, αφαιρεθέντος στοιχείου.
  - το μέγεθος βήματος μεταξύ των αντίστοιχων δύο στοιχείων διπλασιάζεται.
- Όταν ένα νήμα παραμείνει, το αποτέλεσμα μείωσης αποθηκεύεται στο πρώτο στοιχείο του πίνακα.

Η CUDA παρέχει στον προγραμματιστή ορισμένες βιβλιοθήκες έτοιμων συναρτήσεων για εξειδικευμένους χειρισμούς. Μία τέτοια βιβλιοθήκη είναι η Thrust, η οποία χρησιμοποιήθηκε σε αυτή την περίπτωση για τον παράλληλο υπολογισμό της μέσης τιμής της συνάρτησης *UpdateSynapses_pre* χρησιμοποιώντας έναν βελτιστοποιημένο αλγόριθμο μείωσης, ώστε να εξαλειφθεί η καθυστέρηση που προκαλείται ανάμεσα στους δύο πυρήνες που προέκυψαν από τη συνάρτηση. Για να κληθεί η συνάρτηση, ο δείκτης σε πίνακα *testvar* που βρίσκεται ήδη στην GPU μετατρέπεται σε ειδικό δείκτη της Thrust, επιτρέποντας την επεξεργασία των στοιχείων του πίνακα χωρίς να χρειασθεί καμία ενέργεια στο χώρο μνήμης του κεντρικού υπολογιστή. Στη συνέχεια, καλείται η συνάρτηση *reduce* της βιβλιοθήκης ώστε να αθροίσει όλα τα στοιχεία του πίνακα *testvar*, και τέλος το αποτέλεσμα διαιρείται από το πλήθος των αντίστοιχων συνάψεων με μια παραπάνω εντολή στον κώδικα της CPU.

Επιπρόσθετα, ο πίνακας από δομές συνάψεων μετατράπηκε σε μια ενιαία δομή που περιέχει πίνακες που αντιστοιχούν στις μεταβλητές κάθε σύναψης, καθώς η νέα δομή δεδομένων απέφερε βελτιωμένα αποτελέσματα σε συνδυασμό με τη χρήση της Μοιραζόμενης Μνήμης των μπλοκ της CUDA. Δεδομένου ότι η κοινόχρηστη μνήμη της CUDA GPU βρίσκεται on-chip, προσφέρει σημαντικά πλεονεκτήματα ταχύτητας σε σχέση με την τοπική και 'παγκόσμια' (global) μνήμη της κάρτας γραφικών. Στην έκδοση προσομοιωτή που χρησιμοποιεί μια συναπτική δομή πινάκων, ένας πίνακας που αντιστοιχεί σε κάθε πίνακα της στρυςτ δηλώνεται με την εντολή *__shared__*, ορίζοντας κατανομή της κοινόχρηστης (διαμοιραζόμενης) μνήμης. Καθώς η κοινόχρηστη μνήμη είναι προσβάσιμη από όλα τα νήματα ενός μπλοκ, σε κάθε πίνακα δίνεται το μέγεθος του αριθμού των νημάτων μέσα σε ένα μπλοκ. Για να επιταχυνθεί η πρόσβαση στη μνήμη, τα νήματα αποθηκεύουν πρώτα τα δεδομένα στην κοινόχρηστη μνήμη, με κόστος μία ανάγνωση απ την παγκόσμια μνήμη ανά πρόσβαση, συνεχίζουν να εκτελούν τις λειτουργίες τους με πρόσβαση στις κοινόχρηστες συστοιχίες και τέλος αποθηκεύουν τα αποτελέσματα στην παγκόσμια μνήμη με κόστος μία εγγραφή ανά μεταβλητή ανά νήμα. Η αρχιτεκτονική της κοινόχρηστης μνήμης οδήγησε στη βελτίωση της απόδοσης στην περίπτωσης της ενιαίας δομής πινάκων (SoA), καθώς αυτή η δομή προσφέρει ισχυρότερη χωρική τοπικότητα για τα νήματα και επιταχύνει κι άλλο την πρόσβαση. Η προσθήκη της χρήσης της κοινόχρηστης μνήμης εφαρμόστηκε στους πυρήνες *UpdateSynapses_pre* και *UpdateSynapses_post_Part1*, καθώς ήταν οι μόνοι πυρήνες με αισθητά περιθώρια βελτίωσης.

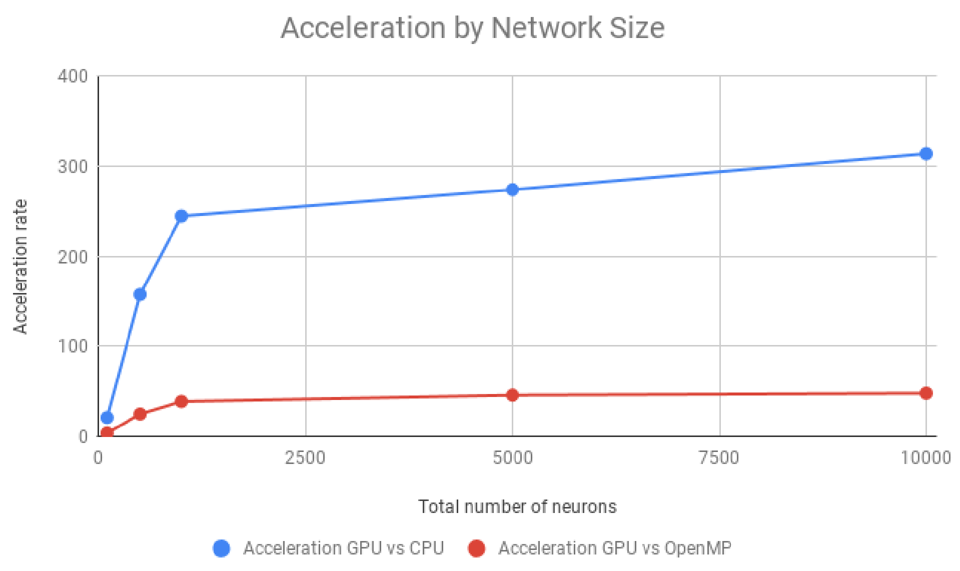
Αποτελέσματα

Οι μετρήσεις απόδοσης συλλέχθηκαν με τη διεξαγωγή πειραμάτων προσομοίωσης σε έναν κόμβο με δύο GPU του υπερυπολογιστή ARIS (Advanced Research Information System) που αναπτύχθηκε και λειτούργησε από το Ελληνικό Δίκτυο Έρευνας και Τεχνολογίας (ΕΔΕΤ). Οι κόμβοι GPU του ARIS περιλαμβάνουν από 2 μονάδες επεξεργασίας του τύπου Hasbell-Intel® Xeon® E5-2660v3, που η καθεμία διαθέτει 10 πυρήνες, 64 GB συνολικής μνήμης και επιταχυντές 2 GPU **NVIDIA Tesla K40** με μνήμη επιταχυντή 12 GB ανά καθένα. Τα αναλυτικά χαρακτηριστικά των επιταχυντών παρατίθενται στο αγγλικό τμήμα της εργασίας.

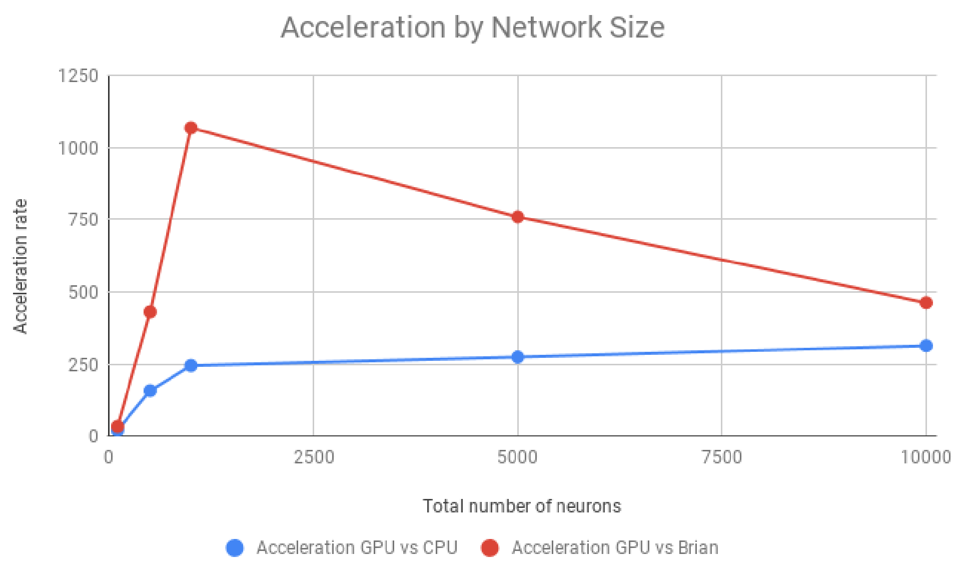
Η τελική έκδοση του προσομοιωτή που χρησιμοποιείται για την ανάλυση απόδοσης περιέχει μια συναπτική δομή με πίνακες για τις συναπτικές μεταβλητές που κατανέμονται τόσο στη μνήμη της συσκευής όσο και στο κεντρικό υπολογιστή, χρησιμοποιεί την κοινόχρηστη μνήμη της συσκευής ανά μπλοκ όταν λειτουργεί στην εν λόγω δομή και υπολογίζει τη μέση τιμή που απαιτείται από τον τελευταίο πυρήνα με τη βοήθεια της συνάρτησης που παρέχει η βιβλιοθήκη Thrust. όπως παρουσιάζεται στα τμήματα 5.1.5.4, 5.1.5.5 και 5.1.5.3 αντίστοιχα. Για την πλειονότητα των μεταβλητών προσομοίωσης επιλέχθηκε τύπος δεδομένων *float*, δεδομένου ότι το μέγιστο αναφερόμενο σφάλμα ήταν μικρότερο από 1,00E-13, όπως αναφέρθηκε στο 5.1.5.1, και θεωρήθηκε αμελητέο. Ο κώδικας των πυρήνων ως προς τις βασικές λειτουργίες παρέμεινε πανομοιότυπος με τις αρχικές υλοποιήσεις που εμφανίζονται στο 5.1.3.1. Όσον αφορά τη διαμόρφωση εκτέλεσης του πυρήνα, το μέγεθος μπλοκ 256 απέδειξε ότι ξεπέρασε όλες τις άλλες διαμορφώσεις σε όλα τα πειράματα, συνεπώς ήταν το απόλυτο μέγεθος μπλοκ που χρησιμοποιήθηκε σε όλα τα πειράματα. Οποιαδήποτε διαφορά από τα αναφερθέντα πρέπει να αναφέρεται στην ακόλουθη παρουσίαση των πειραματικών αποτελεσμάτων. Ο χρόνος πειράματος τέθηκε 1 δευτερόλεπτο.

Συγκρίσεις ως προς το Μέγεθος Δικτύου

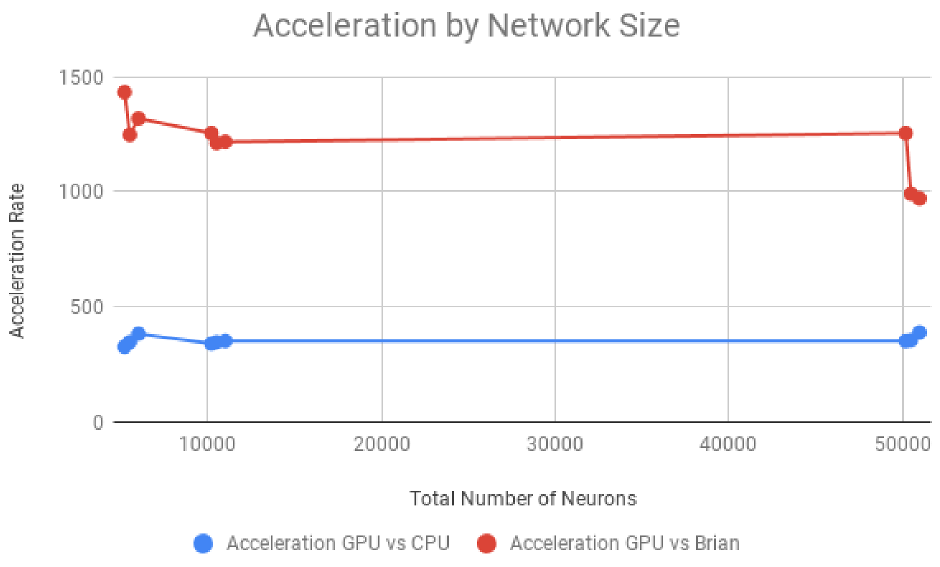
Οι αρχικοποιήσεις τόσο των νευρώνων όσο και των συνάψεων ήταν απολύτως αντίστοιχες με το έγγραφο στην οποία στηρίζεται η διπλωματική. Για αυτά τα πειράματα, η συναπτική συνδεσιμότητα ρυθμίστηκε στο 100%, πράγμα που σημαίνει ότι κάθε νευρώνας AdEx ή Εισόδου συνδέεται με κάθε άλλο νευρώνα του δικτύου. Επιπλέον, η συχνότητα πυροδότησης ρυθμίστηκε στο 1 κΗζ και το εύρος πυροδότησης είναι 100 τοις εκατό, που σημαίνει ότι κάθε μεμονωμένος νευρώνας έχει ρυθμιστεί να πυροδοτείται συνεχώς και στις δύο περιπτώσεις NxM και MxM.



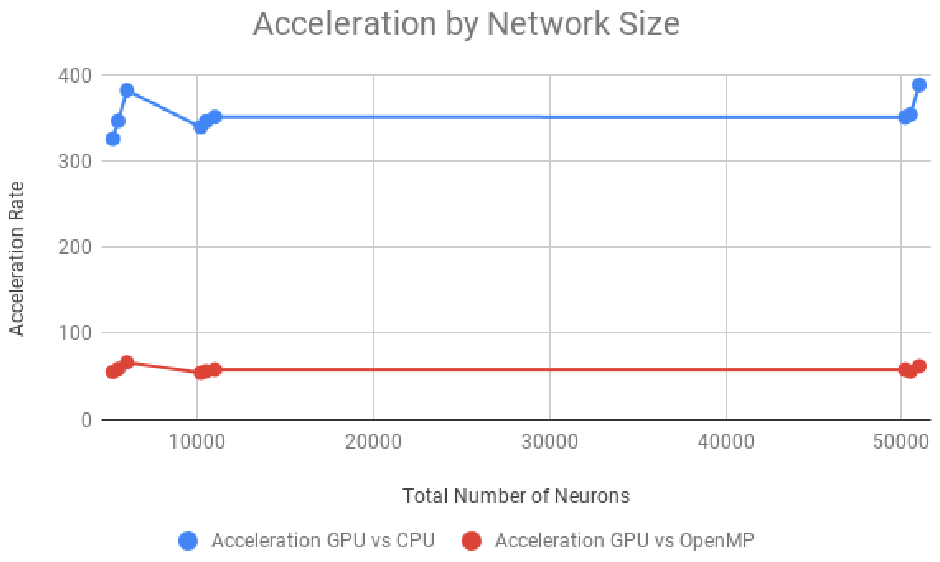
Σχήμα 1: Επιτάχυνση GPU vs CPU vs OpenMP, MxM



Σχήμα 2: Επιτάχυνση GPU vs CPU vs Brian, MxM



Σχήμα 3: Επιτάχυνση GPU vs CPU vs Brian, NxM



Σχήμα 4: Επιτάχυνση GPU vs CPU vs OpenMP, NxM

Πρώτον, πρέπει να σημειωθεί ότι σε όλες τις περιπτώσεις, ο πυρήνας *SolveNeurons* καταναλώνει ένα ελάχιστο ποσοστό του συνολικού χρόνου προσομοίωσης. Αυτό έχει ως αποτέλεσμα οι συναπτικές λειτουργίες να καθορίζουν την απόδοση του προσομοιωτή. Επιπλέον, καθώς οι προσομοιώσεις του Brian είναι σημαντικά χειρότερες σε απόδοση από την σειριακή υλοποίηση του προσομοιωτή γλώσσας C, παραλείφθηκε από την πλειονότητα των πειραμάτων καθώς δεν υπήρχε κέρδος στη σύγκριση των επιπέδων επιτάχυνσης με εκείνα που μετρήθηκαν λαμβάνοντας υπόψη το χρόνο της ῝ΠΥ.

Η εφαρμογή OpenMP προσφέρει μια βασική επιτάχυνση που κυμαίνεται γύρω από 6x επιτάχυνση από τον αρχικό χρόνο CPU. 8 νήματα OpenMP δημιουργήθηκαν σε κάθε εκτέλεση και ο παραλληλισμός έφθασε αρκετά κοντά στο βέλτιστο χωρίς εκτεταμένη έρευνα. Χρησιμοποιήθηκε ως βασική επιτάχυνση και υστερεί σε μεγάλο βαθμό σε σχέση με την εφαρμογή της CUDA.

Παρατηρώντας τους χρόνους της GPU ενάντια στους χρόνους CPU των πειραμάτων MxM, καθίσταται σαφές ότι το βαρύτερο υπολογιστικό φορτίο έχει ως αποτέλεσμα μεγαλύτερους ρυθμούς επιτάχυνσης, αρκεί να υπάρχουν αρκετά νήματα CUDA για το πλέγμα ώστε να καλύψουν ολόκληρο το συναπτικό πλέγμα της προσομοίωσης, αφού σε κάθε νήμα ανατίθεται μία σύναψη και η κλιμάκωση μιας τέτοιας διάταξης υπερνικά σε μεγάλο βαθμό τη σειριακή επεξεργασία του συνόλου των συνάψεων. Η μέγιστη απόδοση της GPU στις προσομοιώσεις MxM έφθασε σχεδόν σε 314x επιτάχυνση έναντι του σειριακού αντίστοιχου.

Παρατηρώντας τα αποτελέσματα προσομοίωσης NxM με παρόμοια κλίμακα με τις προσομοιώσεις MxM, τα ποσοστά επιτάχυνσης είναι παρόμοια και ακόμη και λίγο υψηλότερα στην περίπτωση NxM. Η εξήγηση έγκειται στο γεγονός ότι η διαδικασία προσομοίωσης διαφέρει στο φορτίο υπολογισμού θεωρώντας την πλήρη δραστηριότητα πυροδότησης μεταξύ των δύο αρχιτεκτονικών νευρώνων. Όταν υπάρχουν Νευρώνες Εισόδου, ενημερώνονται οι συναπτικές μεταβλητές μόνο των συνάψεων που συνδέουν τους Νευρώνες Εισόδου και AdEx, ενώ στην περίπτωση των Νευρώνων AdEx μόνο, το σύνολο της συναπτικής δομής ενημερώνεται τακτικά. Επομένως, στην περίπτωση NxM η ενημέρωση των συνάψεων με βάση τις μετασυναπτικές εκφράσεις STDP εξαλείφεται, καθώς οι Νευρώνες Εισόδου δεν αντιστοιχούν στο πραγματικό μοντέλο νευρώνων και δεν απαιτούν καμία επεξεργασία σε όλη τη διάρκεια της προσομοίωσης, προσφέροντας καθαρά αιχμές στο δίκτυο. Καθώς οι βαρύτεροι πυρήνες της CUDA είναι οι μετασυναπτικοί, οι προσομοιώσεις NxM είναι ελαφρύτερες για τη GPU, προσφέροντας ελαφρώς αυξημένες επιδόσεις. Ο μεγαλύτερος λόγος επιτάχυνσης που επιτεύχθηκε ανάμεσα στις εκδόσεις GPU και CPU ήταν 388x στο μεγαλύτερο πείραμα NxM.

Πειράματα Μεγάλης Κλίμακας

Για να μειωθεί ο χρόνος εκτέλεσης για ένα μεγαλύτερο πείραμα, ο χρόνος προσομοίωσης μειώθηκε σε **0,1 s** αντί για κανονικό χρόνο προσομοίωσης 1 δευτερολέπτου, έτσι τα πειράματα αντιστοιχούσαν σε 100 χρονικές στιγμές - βήματα. Ο μέγιστος αριθμός προσομοιωμένων νευρώνων για τους σκοπούς της παρούσας εργασίας είναι 20000. Τα ποσοστά σύνδεσης και πυροδότησης ρυθμίζονται σε 100 και διεξήχθησαν πειράματα μονής και διπλής GPU.

Όπως φαίνεται στον πίνακα, η επιτάχυνση ακολουθεί παρόμοια κλίμακα και αυξάνεται ακόμη και σε σχέση με τα μικρότερα πειράματα σε σύγκριση με την απόδοση της CPU χρησιμοποιώντας έναν κόμβο με δύο συσκευές επιτάχυνσης, καθώς και οι δύο συσκευές λειτουργούν ταυτόχρονα και το μέγεθος του προβλήματος διαιρείται στο μισό για την πλειοψηφία των λειτουργιών της προσομοίωσης. Η σημαντική ποινή στην επιτάχυνση όταν χρησιμοποιείται μία GPU προκαλείται από την ανάγκη να αντιγράφονται συνεχώς τμήματα δεδομένων προσομοίωσης μεταξύ της μνήμης της συσκευής και του κεντρικού υπολογιστή, κάτι που είναι χειρότερο όταν χρησιμοποιείται μια συναπτική δομή πεινάκων λόγω της επιβάρυνσης των ξεχωριστών *cudaMemcpyAsync* κλήσεων για κάθε πίνακα μεταβλητών. Η επιτάχυνση έφθασε στο ακραίο

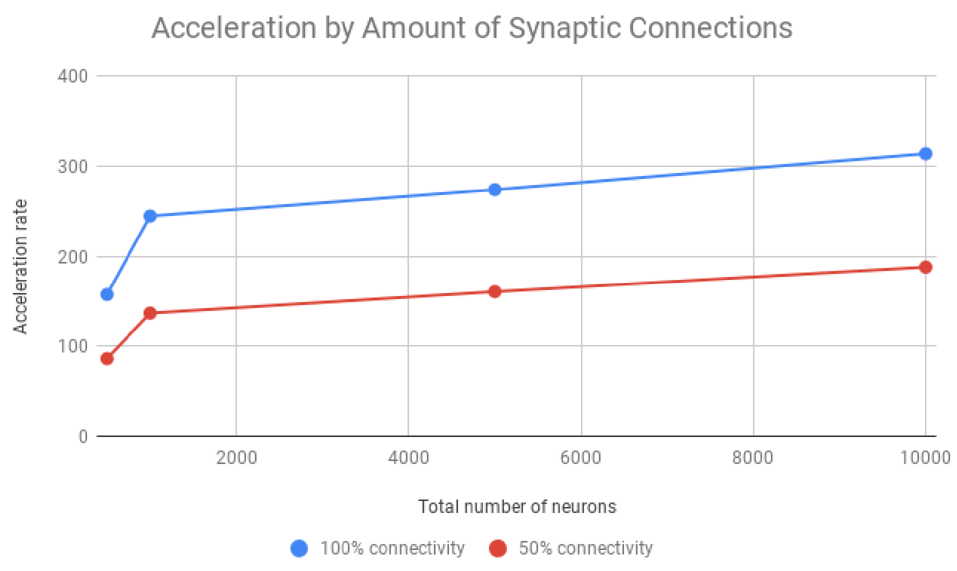
Πίνακας 1: Επιτάχυνση ως προς Μέγεθος Δικτύου, Μεγαλύτερης Κλίμακας, MxM

N_S	0
N_Group_S	20000
N_Group_T	20000
Βήματα	100
Συνδεσιμότητα(%)	100
Χρόνος CPU	27962,93
Χρόνος OpenMP	3967,63
Χρόνος Μιας GPU	890,17
Χρόνος Δύο GPU	43,43
Επιτάχυνση Μιας GPU vs CPU	31,41
Επιτάχυνση Μιας GPU vs OpenMP	4,45
Επιτάχυνση Δύο GPU vs CPU	643,86
Επιτάχυνση Δύο GPU vs OpenMP	91,36
Επιτάχυνση Δύο GPU vs single GPU	20,5

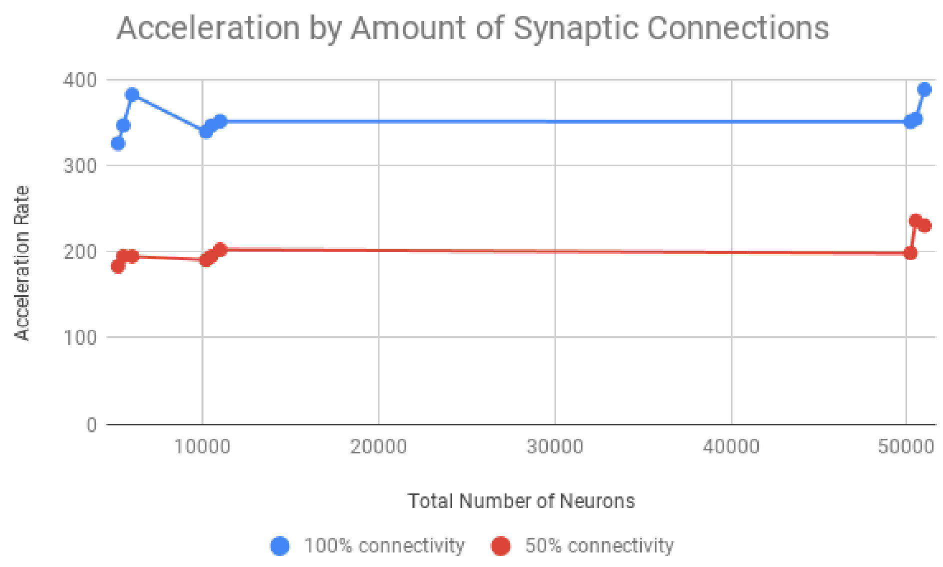
επίπεδο των 643x για την έκδοση με δύο GPU, ενώ η απλή έκδοση GPU επέτυχε μόνο 31x επιτάχυνση και βελτιώθηκε μόλις 4 φορές σε σύγκριση με το OpenMP, μια απόδοση πολύ χειρότερη από πριν.

Συγκρίσεις ως προς τη Συνδεσιμότητα

Δίκτυα με 50 % συνδεσιμότητα προσομοιώθηκαν για να ελεγθεί η επίδραση των αραιών συναπτικών συνδέσεων σε σχέση με την απόλυτη συνδεσιμότητα. Χρησιμοποιήθηκαν πανομοιότυπα μοτίβα σύνδεσης για να προκύψουν ασφαλή συμπεράσματα μεταξύ των προσομοιώσεων MxM και NxM. Μόνο οι νευρώνες με άρτιο αναγνωριστικό αριθμό αλληλοσυνδέονται και οι νευρώνες που έχουν περιττά αναγνωριστικά ήταν ουσιαστικά αδρανείς σε όλη τη διάρκεια της προσομοίωσης. Η πυροδότηση βρίσκεται για άλλη μια φορά σε συνεχή εμφάνιση σε όλο το δίκτυο.



Σχήμα 5: Επιτάχυνση GPU vs CPU, MxM

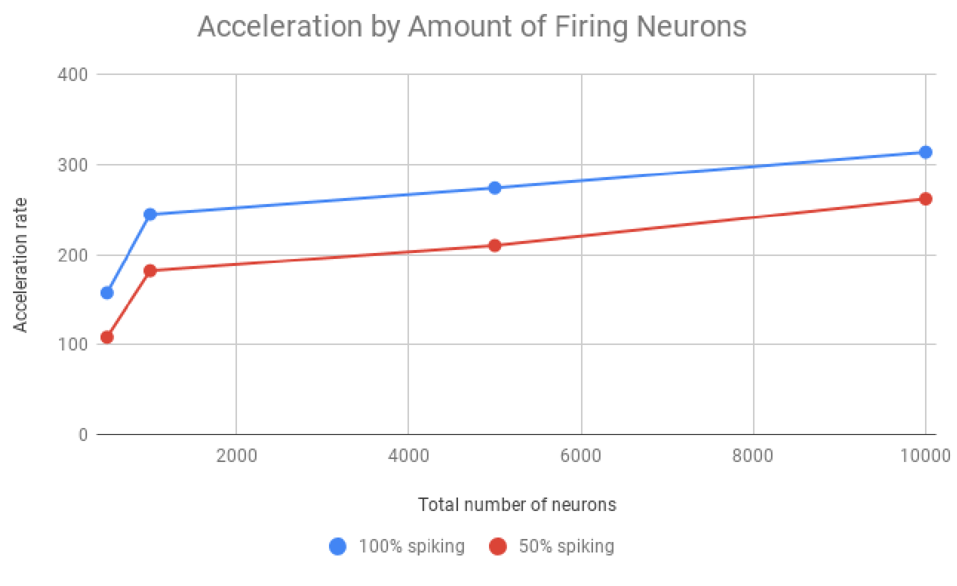


Σχήμα 6: Επιτάχυνση GPU vs CPU, NxM

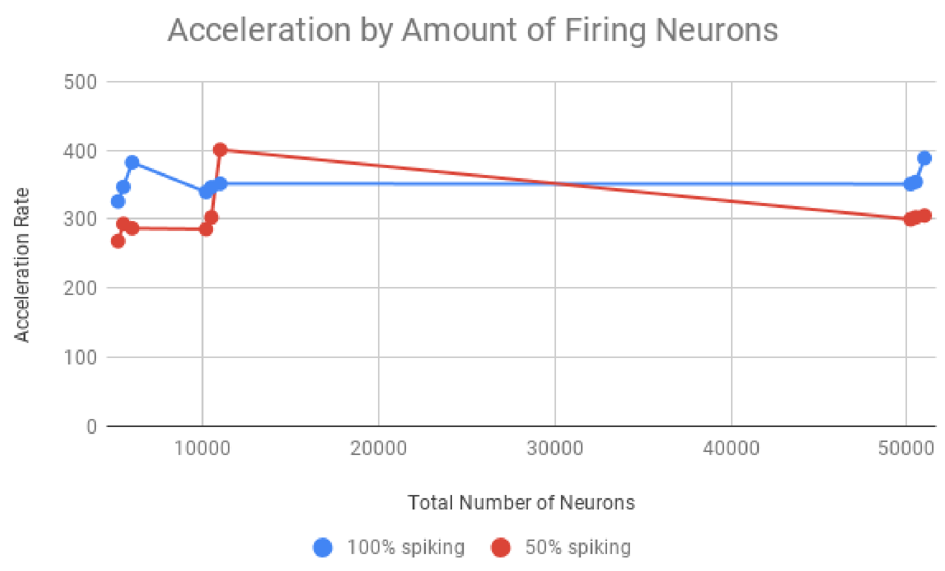
Τα αποτελέσματα από αυτόν τον πειραματισμό ήταν ιδιαίτερα ενδιαφέροντα. Στο σύνολο των διεξαγόμενων πειραμάτων, οι χρόνοι προσομοίωσης ήταν ίσοι - στην πραγματικότητα ελάχιστα χειρότεροι - με αυτούς της πλήρους συνδεσιμότητας. Δεδομένου ότι η έκδοση της CPU επιταχύνθηκε σημαντικά ως αποτέλεσμα του χαμηλότερου φόρτου υπολογιστικής εργασίας, τα ποσοστά επιτάχυνσης ήταν αρκετά χαμηλότερα. Ο λόγος για αυτά τα αποτελέσματα είναι κατά πάσα πιθανότητα ο τρόπος εκτέλεσης των νημάτων CUDA παράλληλα. Καθώς κάθε ομάδα 32 νημάτων εκτελεί ταυτόχρονα την ίδια εντολή, η διάταξη συνδεσιμότητας αναγκάζει τα μισά από τα νήματα της ομάδας να είναι αδρανή, ενώ το άλλο μισό εκτελεί όλες τις συναπτικές λειτουργίες. Στην ῝ΥΔΑ, διακλάδωση νημάτων σε διαφορετικά σημεία κώδικα οδηγεί σε σειριοποίηση της εκτέλεσης μέσα σε μια ομάδα νημάτων, οπότε κάθε πιθανή επιτάχυνση εξαλείφεται καθώς τα ενεργά νήματα υποβάλλονται σε παρόμοια ένταση εκτέλεσης και στις δύο περιπτώσεις συνδεσιμότητας και τα ανενεργά νήματα δεν βοηθούν στην επιτάχυνση της προσομοίωσης. Δεδομένου ότι η εφαρμογή OpenMP ακολουθεί όμοια πρότυπα επιτάχυνσης με αυτά των πρώτων παρουσιαζόμενων πειραμάτων στο σύνολο της διαδικασίας, οι χρόνοι OpenMP παραλήφθηκαν από εδώ και στο εξής. Τα μοτίβα επιτάχυνσης δεν άλλαξαν όσον αφορά την κλίμακα προσομοίωσης για τα πειράματα MxM, καθώς επιτεύχθηκε μεγαλύτερη επιτάχυνση στο μεγαλύτερο πείραμα, επιτυγχάνοντας ταχύτητα 188x. Ένα παρόμοιο μοτίβο μπορεί να παρατηρηθεί στα πειράματα NxM, παρόλο που το μεγαλύτερο - 236x - speedup επιτεύχθηκε στο δεύτερο μεγαλύτερο πείραμα.

Συγκρίσεις ως προς το Ποσοστό Πυροδότησης

Μια παρόμοια προσέγγιση με την προηγούμενη ήταν να διεξαχθούν πειράματα με μια διαφορά στον αριθμό των νευρώνων που παράγουν αιχμές σε κάθε χρονική στιγμή. Η δοκιμή της χαμηλότερης συχνότητας πυροδότησης με διατήρηση του αριθμού των νευρώνων που πυροδοτούνται θα ήταν άσκοπη, καθώς αυτό θα οδηγούσε σε πλήρη αδράνεια του συστήματος στις χρονικές στιγμές που δεν έχει πυροδοτηθεί κανένας νευρώνας. Οι παράμετροι αρχικοποιήθηκαν ως συνήθως και μόνο οι νευρώνες με άρτιο αναγνωριστικό αριθμό είχαν οριστεί να πυροδοτούν συνεχώς καθ ΄όλη τη διάρκεια της προσομοίωσης.



Σχήμα 7: Επιτάχυνση GPU vs CPU, MxM



Σχήμα 8: Επιτάχυνση GPU vs CPU, NxM

Στην περίπτωση της εμφάνισης πυροδοτούμενης πυρκαγιάς αυτών των συγκεκριμένων πειραμάτων, το σύνολο των γεννημένων νημάτων CUDA ακολουθεί τους ίδιους κανόνες από την άποψη της δραστηριότητας. Αυτό σημαίνει ότι ενώ κάθε δεύτερο χρονικό διάστημα όλα τα νήματα εκτελούν το σύνολο των λειτουργιών του πυρήνα, εξέρχονται επίσης από τους πυρήνες σχεδόν αμέσως μετά τη δημιουργία στο άλλο μισό των timesteps, προκαλώντας πολύ μικρότερη ποινή στη GPU σε σύγκριση με την εκτέλεση της CPU. Η μεγαλύτερη επιτάχυνση που παρατηρήθηκε σε αυτή την περίπτωση ήταν 262x για την έκδοση MxM και για άλλη μια φορά αυξήθηκε μαζί με την ποσότητα των νευρώνων που υπήρχαν στο πείραμα, ενώ τα πειράματα NxM εμφάνισαν παρόμοιους ρυθμούς επιτάχυνσης, εκτός από ένα υψηλό λάκτισμα έως 401x ταχύτητα που παρατηρήθηκε στη προσομοίωση 11000 νευρώνων. Αυτή η απομονωμένη αύξηση της επιτάχυνσης στην πραγματικότητα αποδίδεται στην CPU που δεν απέδιδε ως συνήθως σε αυτό το συγκεκριμένο σειριακό πείραμα, καθώς όλοι οι άλλοι χρόνοι CPU είναι παρόμοιοι με εκείνους της χαμηλότερης συνδεσιμότητας εκτός από αυτόν, έτσι δε μπορεί να βγει με ασφάλεια κάποιο συμπέρασμα όσον αφορά την απόδοση της GPU.

Chapter 1

Introduction

The raising need of decreasing the simulation time of experiments related to neuronal networks for neuroscientists motivated this Diploma Thesis. Similarly to several other disciplines, research in neuroscience nowadays includes running simulations of complex mathematical models that try to explain the functionality of the brain. The vast complexity of the brain as well as the behavioral and organizational complexity of the neurons leads to enormous simulations that take a lot of time to run, delaying research. This thesis' goal is to try to discover the optimal way to accelerate the Adaptive Exponential Integrate-and-fire neuron model simulation, which is widely used in neuroscientific experiments. This simulation encompasses the phenomenon of Spike-timing-dependent plasticity of the connections of neurons called synapses.

Plenty of methods were explored in order to attain the final acceleration. At the beginning of this research project, the simulation was coded for the BRIAN Simulator, a renowned open-source neuron model solver that runs on Python. In order to effectively accelerate the model, first the simulation needed to be ported to the C programming language, which showcases overall better performance than Python as it is a lower level programming language. Afterwards, the CUDA application programming interface (API) was used in order to implement a parallel computing version which achieves higher levels of acceleration utilizing an Nvidia GPU. Last but not least, the OpenMP API was used to accelerate the simulation on multiple CPU threads and explore the acceleration scaling compared to the GPU version.

1.1 Neuroscience

Neuroscience is the scientific study of the nervous system, especially the relation of nerves to behaviour and learning. The biggest effort for neuroscientists is focused on the brain as its increased understanding along with improved studying methods results in more clear knowledge relating normal human behaviour and mental well-being. Knowing how exactly the nervous system functions can help researchers find ways to prevent or treat problems that affect the brain, nervous system, and body. This discipline's breakthroughs enable scientists to develop treatments for neurodegenerative diseases (such as Alzheimer's disease) and mental illnesses, while also assisting in the development of Artificial Intelligence. These developments are likely to provide significant benefits for society and have implications for a diverse range of public policy areas such as health, education, law, and security.[Inv]

1.1.1 Computational Neuroscience

Computational neuroscience is a branch of neuroscience which aims to understand the information content of neural signals by utilizing mathematical models of the nervous system at many different structural scales, including the biophysical, the circuit, and the systems levels.[PD01] It is an interdisciplinary science that links the diverse fields of neuroscience, cognitive science, and psychology with electrical engineering, computer science, mathematics, and physics. The ultimate aim of computational neuroscience is to explain how electrical and chemical signals are used in the brain to represent and process information.

1.1.1.1 History

Evidence of scientific research which leads to what is now known as neuroscience dates back to ancient history civilisations, i.e. the Edwin Smith Surgical Papyrus, written in the 17th century BC, which contains the earliest recorded reference to the brain and describes the symptoms, diagnosis, and prognosis of two patients wounded in the head. Around the same period people in Ancient Greece also started to occupy themselves with the study of the brain, expressing different opinions while being limited by the fact that the human body was considered sacred by hippocratic doctors, forbidding themselves from operating for scientific reasons. Little progress was made until the AD 10th century, when an Arab neurosurgeon among plenty of neurosurgical diagnosis described the tools needed to perform surgery while avoiding puncture of important parts of the brain. Around the 14th century, European scientists started recording information about brain anatomy. From an experimental point of view, Marie Jean Pierre Flourens in the 1810s was the founder of experimental brain science and a pioneer in anesthesia. Through the study of ablations on animals, he was the first to prove that the mind was located in the brain, not the heart. [AG87; NRARJLF84].

However, the origin of computational neuroscience is commonly traced to the mathematical model that Nobel prize winners Allan Hodgkin and Andrew Huxley developed of the squid giant axon action potential in 1952. This is historically the first model of biological neurons explaining the ionic mechanisms underlying the initiation and propagation of action potentials in the squid giant axon. The Hodgkin–Huxley model applies to all axons and is still used to this day:

$$i_m = g_{Na}m^3h(V - E_{Na}) + g_K n^4(V - E_K) + g_L(V - E_L)$$

Their work in axon potentials was recognized by them receiving the Nobel Prize in Physiology and Medicine in 1963. The 3rd laureate was Sir John Eccles for his work on Synapses, an Australian neurophysiologist known for his pioneering work in neuroscience, especially the synaptic transmission and nature of reflexes. [Ben13] Furthermore, they developed an action potential theory representing one of the earliest applications of a technique of electrophysiology, known as the voltage clamp. [Hod52]. One could also argue that the origin of computational neuroscience was the introduction of the integrate-and-fire neuron by Louis Lapicque. [BR08]

The next big step was the work of Wilfrid Rall. Considered one of the founders of computational neuroscience, Rall originated the use of cable theory in neuroscience, and developed passive and active compartmental models of the neuron dendritic tree which made it possible to show that dendritic arborizations of neurons strongly affect processing of synaptic input. [Ral62] Before Rall, neurons were assumed to be isopotential and the electrophysiological importance of dendrites was ignored. Moreover, he pioneered the use of digital computers in neuroscience and developed the discretized version of cable theory, compartmental modeling, which forms the basis for some of the most widely used software packages in computational neuroscience (such as GENESIS [BB19] and NEURON [HC01]). His contribution is historically interesting for two additional reasons: his conflict with experimental neuroscientists (i.e. John Eccles) and the attention to the spatial domain. In general, the considerations introduced by Rall had only a limited impact on the thinking of contemporary neuroscientists. It wasn't until the early seventies that key concepts introduced by Rall, like spatial summation and dendritic attenuation of synaptic input [Ral62], which are now part of core curricula in neuroscience, became commonplace.

The actual term computational neuroscience didn't appear until the second half of the eighties. [SKC88] Around the same time, plenty of initiatives related to the discipline were started: Graduate programs (CNS program, Caltech, 1986), meetings (first Computational Neuroscience (CNS) meeting, University of California, 1992), summer courses (Methods in Computational Neuroscience, Woods Hole, 1988), as well as the appearance of the first computational neuroscience textbook. [Ooy00]

1.1.2 Neurons

Neurons are electrically excitable cells that communicate with other cells via specialized connections called synapses. They are the principal cellular elements that underlie the function of the nervous system. All multicellular organisms except sponges and Trichoplax have neurons. They are not the only cellular types of which the central nervous system consists, as several types of glial elements, essential in the maintenance of the neuronal network, in neuronal migration during development and in the generation of myelin, are also present. Scientists have distinguished neurons into three different types: sensory neurons, which respond to external stimuli that affect the cells of the sensory organs and send signals to the spinal cord or brain, motor neurons (in direct communication with muscles or glands) and interneurons, which shoulder the task of establishing contacts between sensory and motor neurons. The latter represent the vast majority of neurons in the brain and ganglia.

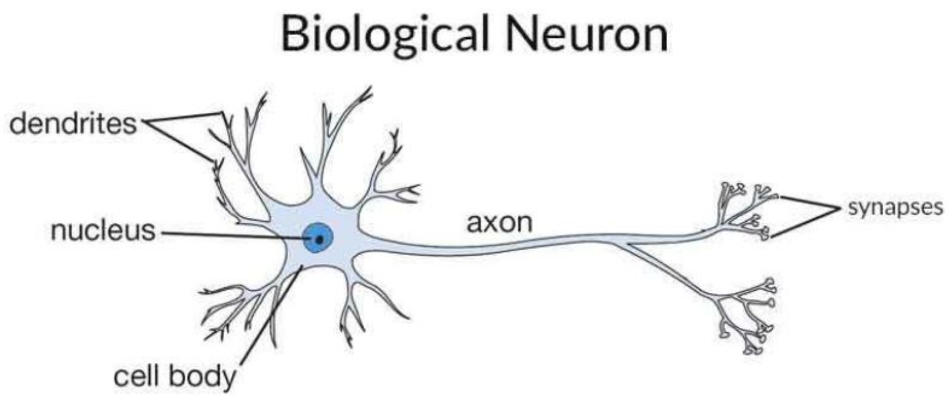


Figure 1.1: Common parts of a neuron

A typical neuron consists of a central cell body (soma), dendrites, and a single axon. Neurons are generally characterized by their soma that comes in different shapes, as they can lack dendrites, or have no axon. They are highly specialized for the processing and transmission of cellular signals. Given their diversity of functions performed in different parts of the nervous system, there is a wide variety in their shape, size, and electrochemical properties. The soma contains the cell nucleus and most of the genomic expression and synthetic machinery responsible for protein synthesis. In general, neurons can be described as having an input and an output pole, which is not absolute as there can be an absence of branching. However, for the majority of neurons the receiving (input) pole consists of filaments that extrude from the soma called dendrites. They arise in vertebrate neurons directly from the cell body and typically branch profusely, getting thinner with each branching. The transmitting (output) pole leaves the soma at a swelling called the axon hillock, travels for as far as 1 meter in humans or more in other species and is called the axon. It conducts propagating electrochemical signals termed action potentials. While this is the general rule, there can be exceptions e.g. peripheral sensory neurons where the input occurs via axons. [Lli08]

The cell body of every neuron is bordered by a plasma membrane, a bilayer of lipid molecules with many types of protein structures embedded in it. It is a powerful electrical insulator, though in neurons many of the protein structures embedded in the membrane are electrically active. These include ion channels that permit electrically charged ions to flow across the membrane and ion pumps that chemically transport ions from one side of the membrane to the other. Interactions between them produce a voltage difference across the membrane which provides a power source for protein elaboration aside from a basis for electrical signal transmission between parts of the membrane.

In most cases, neurons are generated by neural stem cells during brain development and childhood. The process of neuron generation is called Neurogenesis. Neurogenesis largely ceases during adulthood in most areas of the brain. However, strong evidence supports generation of substantial numbers of new neurons in the hippocampus and olfactory bulb. The extent to which adult neurogenesis exists in humans, and its contribution to cognition are controversial, with conflicting reports published. [Wikb] Regarding nerve regeneration, peripheral axons can regrow if they are severed, but one neuron cannot be functionally replaced by one of another type [Lli14]

Neurons communicate with other cells via specialized connections called synapses, further explained below. The signaling process is partly electrical and partly chemical. Neurons are electrically excitable, due to maintenance of voltage gradients across their membranes. If the voltage changes by a large enough amount over a short interval, the neuron generates an electrochemical pulse called an action potential. This potential travels rapidly along the axon, and activates synaptic connections as it reaches them. Synaptic signals may be excitatory or inhibitory, increasing or reducing the net voltage that reaches the soma. Their functional properties can be listed as such:

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

Regarding neuronal function, electrical excitability is by far the most important property, as neurons exhibit both passive and active electrical characteristics. Passive properties refer to the capacitative and resistive aspects inherent in neuronal membranes, along with the resistivity inherent in the cytoplasm and the extracellular milieu. They are often termed cable properties, due to the resemblance of neuronal processes and conduction in electrical cables. Across the membrane, an electric field and a voltage difference is maintained by the action of selective ion pumps. Neurons conduct waves of membrane potential passively (electrotonically) a short distance along their processes as the result of currents that flow intracellularly along the longitudinal resistance and simultaneously across the plasmalemmal membrane as resistive or capacitative current. On the other hand, active electrical properties refer to the effect of the activation of voltage, ligand, or second messenger gated transmembrane ionic channels on electrical potentials across the plasma membrane. An example of such a result is the generation of action potentials.

In an active neuron the superposition of passive and active electrical properties serves to allow the cell the possibility of summing the transmembrane potential either linearly or non-linearly and to reach depolarization levels sufficiently high to trigger action potentials, also known as “nerve impulses” or “spikes”, and the temporal sequence of action potentials generated by a neuron is called its “spike train”. Each excitable patch of membrane has two important levels of membrane potential: the resting potential, which is the value the membrane potential maintains as long as nothing perturbs the cell, and a higher value called the threshold potential. For a typical neuron, the resting potential is around –70 millivolts (mV) and the threshold potential is around –55 mV. Synaptic inputs may cause the membrane to depolarize or hyperpolarize by rising or decreasing the membrane potential.

1.1.3 Synapses

Neurons communicate with each another via synapses, where either the axon terminal of one cell contacts another neuron's dendrite, soma or, less commonly, axon. They are structures that permits a neuron to pass an electrical or chemical signal to another neuron or to the target effector cell. The plasma membrane of the signal-passing (presynaptic) neuron comes into collocation with the membrane of the target (postsynaptic) cell at a synapse. Both the presynaptic and postsynaptic sites contain extensive arrays of a molecular machinery that link the two membranes together and carry out the signaling process. [Fos97]

Synapses can be excitatory or inhibitory, either increasing or decreasing activity in the target neuron, respectively. There are two different types of synapses, termed chemical and electrical. In a chemical synapse, when an action potential reaches the axon terminal, it opens voltage-gated calcium channels, allowing calcium ions to enter the terminal. Calcium causes synaptic vesicles filled with neurotransmitter molecules to fuse with the membrane, releasing their contents into the synaptic cleft. The neurotransmitters diffuse across the synaptic cleft and activate receptors on the postsynaptic neuron. High cytosolic calcium in the axon terminal triggers mitochondrial calcium uptake, which, in turn, activates mitochondrial energy metabolism to produce ATP to support continuous neurotransmission. [VIM13] In an electrical synapse, special channels called gap junctions or synaptic cleft are responsible for connecting the presynaptic and postsynaptic cell membranes that are capable of passing an electric current, effecting in voltage changes in the presynaptic cell inducing voltage changes in the postsynaptic cell. These synapses present the advantage of rapid signal transfer between connected cells. [Sil07]

It is widely accepted that the synapse plays a role in the formation of memory. Connections between the two neurons are strengthened when both neurons are active at the same time, as a result of the receptor's signaling mechanisms when neurotransmitters activate receptors across the synaptic cleft. The strength of two connected neural pathways is thought to result in the storage of information, resulting in memory. This process of synaptic strengthening is known as long-term potentiation.[Lyn04]

By altering the release of neurotransmitters, the plasticity of synapses can be controlled in the presynaptic cell. The postsynaptic cell can be regulated by altering the function and number of its receptors. Changes in postsynaptic signaling are most commonly associated with a N-methyl-d-aspartic acid receptor (NMDAR)-dependent long-term potentiation (LTP) and long-term depression (LTD) due to the influx of calcium into the post-synaptic cell, which are the most analyzed forms of plasticity at excitatory synapses.

1.1.4 Levels of Analysis in Neural Modeling

Even though neuroscience follows the standard practice set by all other scientific disciplines, what distinguishes neural modeling is that brains are computational devices handling information to control their actions. The most comprehensive neural models must therefore play the dual role of accounting for experimental data and interpreting it in terms of underlying computations. Thus, understanding the meaning behind the existence of neural models and their effect in practice is troublesome, due to the complexity and multiple goals of modeling.

In order to be effective, one needs to understand the different levels of organization within the structure of modeling. The first thread to this idea is scientific reduction, describing observable phenomena in qualitative and quantitative detail, and explaining them in terms of descriptions of their underlying substrates at lower and less abstract levels. A second thread, parallel to the first, is the construction or synthesis of systems to execute some particular task, conventionally by utilizing a divide-and-conquer strategy. Thirdly, regarding computational modeling, one must be aware of the computational, algorithmic and implementational planes suggested by David Marr. Finally, the fourth thread is about levels of processing as a strategy for manipulating and extracting information from input. [Day06]

1.1.5 Types of Neural Modeling

1.1.5.1 Conventional reductive models

The first thread to the idea of levels concerns standard reductive modeling. Practically, this approach enables scientists to describe neural phenomena and provide reductionist explanations by appealing to the mechanisms that might actually be responsible for originally generating the phenomena. The modeling process is recursive, as these mechanisms are likewise represented by models. Quantifying these models in an elemental manner is essential in order to check the accuracy of capturing the phenomena. There are different levels of modeling, subsequently creating different levels of anatomical detail in neuroscience. There are descriptive models at a level, which capture the behavior without much regard to the substrate, and explanatory models, which capture the behavior by reducing it to models at lower levels. Quantitative models epitomize the different levels because they allow numerical demonstration that the behaviour that represented by a model is truly a description of the real phenomenon that it tries to explain.

1.1.5.2 Computational interpretive models

Several tasks for brains are best characterized as involving computations. For example, hand-eye coordination actions require transforming visual input into a sequence of motor commands timed correctly. Computational modeling is about imputing a computational task and interpreting the collective behavior of the neural components of the system in terms of this task. The key aspects of computations are representation, storage, and transformation or algorithmic manipulation. Similar to standard computers, the semantics of the computation are implemented by the syntax of the physical substrate.

Computational models share several properties with conventional models:

- There are different levels of abstraction, expressing a decomposition of the underlying computation.

- There are both descriptive and explanatory models.
- Finally, the same computation defined on equal abstraction levels can be interpreted and represented in diverse ways for the same computation.

The analysis of neural systems requires a combination of computational and conventional modeling. The implementational plane of the computational model, considering a single level, conceives exactly the experimental phenomena for which conventional reductive modeling provides an account. The explanatory reduction of these phenomena needs to comprise the lower level of the conventional model along with the implementational plane of the lower level of the computational model. The algorithmic and computational planes at multiple abstraction levels of the computational model as such will be forced to be consistent with the multiple levels of the conventional model, as it is necessary in order to have a complete understanding and representation of the operation of the human brain.

1.1.6 Degrees of Modeling Detail of Neurons

While not able to competently fit the ensemble of used models nowadays, there are three main classes of quantitative models in common use, corresponding to different levels of abstraction.

1.1.6.1 Conductance-based models

These models emphasize on describing a limited number of neurons with an increased degree of detail by approximating the structure of a neuron by multiple electrically compact interconnected subdivisions which combine to resemble neuron architecture. In standard conductance-based models, each subdivision is given an assortment of active channels, such as voltage sensitive or synaptic channels. Representing cells that have complex geometrical structures in terms of minimum possible compartments is often a necessary challenge in order to decrease computation time. Conductance-based models of single cells are ideal for explaining spike-related phenomena and the effects of synaptic input. The main issue of such models is the difficulty of obtaining useful experimental data critical to making the models faithful to the neural substrate. This is due to the vast number of parameters, plenty of whom's values cannot be determined from experiments. Secondly, even though compartmental models capture the electrical geometry of single cells, it is rather improbable of them to accurately capture the three dimensional surroundings of the cells.

1.1.6.2 Integrate-and-fire models

Lying at a level of abstraction above conductance-based ones, these models use a symbolic model of spike generation coupled with a leaky integrator model of a cell whenever the voltage drops below the threshold for spike initiation. Moreover, they eliminate the compartment-based approximation of cell geometry, including at best a time-course for synaptic input and other time-dependent factors such as spike-rate adaptation allowance.

These models are best suited for simulating large, recurrently connected, networks of neurons, enabling scientists to discover various mathematical issues about these networks. In addition, details of phenomena such as synaptic plasticity are proven to be dependent to other phenomena such as precise time differences between pre-synaptic and post-synaptic activity. As the integrate-and-fire model is the simplest form that outputs spikes, it is commonly used to address such issues.

1.1.6.3 Firing-rate models

These models compose the most abstract level of characterization of neurons, as they treat the output of cells as continuous-valued, time-varying firing rates, abandoning previous spike-related approaches. Networks of firing-rate models can be constructed, in which the influence of one cell on another is given by the product of the pre-synaptic cell's firing rate and the synaptic strength for the connection. The advantages these models offer are their empirical and analytical tractability. Firing-rate models usually 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 Accelerated Computation via GPU

As with all disciplines that have raised the need for simulations characterized by heavy mathematical computations, neuroscience has deemed computation speed a significant parameter that defines the amount of output data available in a predefined experimentation period. Thus, recent technological advances have enabled the use of accelerator devices to decrease simulation time by executing a higher number of operations compared to standard computers. An example of such devices is the GPU, further presented below for this thesis' purposes.

1.2.1 Graphics Processing Unit

A Graphics Processing Unit (GPU) is a specialized device designed to rapidly manipulate high amounts of graphical pixels and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. Historically, GPUs were born for being used in advanced graphics and videogames. They are used in various electronic systems such as mobile phones, embedded systems, personal computers, workstations, and game consoles. In a personal computer, a GPU can be present on a video card or embedded on the motherboard or even in the same chip as the CPU. [Atk07; Pcm]

The first documented use of the term GPU dates to the 1980s. [HHD86] Nvidia popularized the term in 1999 stating its technical definition as "a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second".

Modern GPUs use most of their transistors to do calculations related to 3D computer graphics. In addition to the 3D hardware, today's GPUs include basic 2D acceleration and framebuffer capabilities (usually with a VGA compatibility mode). [Ole18] Graphics cards often avoid hardware dedicated to 2D acceleration and prefer it to be emulated by 3D hardware. GPUs were initially used to accelerate the memory-intensive work of texture mapping and rendering polygons, later adding units to accelerate geometric calculations such as the rotation and translation of vertices into different coordinate systems. Recent developments in GPUs include support for programmable shaders which can manipulate vertices and textures with many of the same operations supported by CPUs, oversampling and interpolation techniques to reduce aliasing, and very high-precision color spaces. Because most of these computations involve matrix and vector operations, engineers and scientists have increasingly studied the use of GPUs for non-graphical calculations.

1.2.2 General-Purpose Computing on GPU

Parallel computing offer a great advantage in terms of performance for very large applications in different disciplines like engineering, physics, biology, chemistry, computer vision and econometrics. The rapid technological advances of the last three decades have led to an almost exponential increase in performance as the years go by, since modern off-the-shelf desktops offer FLOPS rates higher than supercomputers of the previous decade. Increased clock frequency led to the issue of overheating processors, which in turn led to the designers' selection of multicore processors in lieu of higher clock frequency. As such, multicore processors have turned normal desktops into truly parallel computers which were extensively used by professionals to decrease execution time for experiments of various disciplines.

In 2003, Mark Harris recognized the potential of using graphical processing units (GPU) for general purpose applications. [OAMD14] As GPUs are designed with a much higher number of transistors allocated to data processing rather than flow control or data caching, they are specialized for highly parallel intensive computations, resulting in much higher floating point operations rates and vastly higher MFLOPs rates than multicore CPUs. Thus, the term General-purpose computing on graphics processing units (GPGPU) was born, meaning the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). [Owe+07]

GPU properties lead to a very different processor architecture from traditional CPUs. CPUs devote a lot of resources, primarily from the chip area, to make single streams of instructions run fast, including caching to hide memory latency and complex instruction stream processing such as pipelining and out-of-order execution. GPUs, on the other hand, use the chip area for individual processing elements that execute a single instruction stream on multiple data elements simultaneously. Memory latency is hidden by swift context switching, since subsets waiting on a memory reference are set aside on free ones upon a memory fetch issue.

Algorithms well-suited to GPGPU implementation are those that exhibit the following properties: they are data parallel and throughput intensive. Data parallel means that a processor can execute the same operation on different data elements simultaneously. Throughput intensive means that the algorithm is going to process a vast amount of data elements, so there will be plenty to operate on in parallel. Ideal GPGPU applications are characterized by large data sets, high parallelism, and minimal dependency between data elements. Moreover, it is important for GPGPU applications to have high arithmetic intensity else the memory access latency will limit computational speedup. Arithmetic intensity is defined as the number of operations performed per word of memory transferred.

The following are example areas where GPUs for general-purpose computing has lately been noticeably popular:

- Physical based simulation and physics engines, cloth simulation, fluid incompressible flow by solution of Euler equations or Navier–Stokes equations [Har04]
- GPU learning – machine learning and data mining computations, the k-nearest neighbor algorithm [GDB08]
- Scientific computing - Monte Carlo simulation of light propagation [ASAE08], Molecular modeling on GPU [Has+15]
- Bioinformatics [Sch+07]
- Electronic design automation [Ler09]

1.2.3 Development Environment of GPGPU

1.2.3.1 Early stages

At the initial stage of GPGPU development, researchers had to write assembly instructions to conduct computation on GPU. General-purpose computing on GPUs became more practical and popular after about 2001, with the advent of both programmable shaders and floating point support on graphics processors. Notably, problems involving matrices and/or vectors – especially ones of higher dimension – were easy to translate to a GPU, which acts with native speed and support on those types. These early efforts to use GPUs as general-purpose processors required reformulating computational problems in terms of graphics primitives, as supported by the major APIs for graphics processors, OpenGL and Direct3D.

GPUs are designed specifically for graphics and thus are very restrictive in operations and programming. Due to their design, they are only effective for problems that can be solved using stream processing. [Bea18] GPUs can only process independent vertices and fragments, but can process a substantial amount in parallel, becoming most effective tasked to process multiple vertices or fragments in the same way. In this sense, GPUs are stream processors – processors that can operate in parallel by running one kernel on many records in a stream at once. A stream is simply a set of records that require similar computation. Streams provide data parallelism. Kernels are the functions that are applied to each element in the stream. In the GPUs, vertices and fragments are the elements in streams and vertex and fragment shaders are the kernels to be run on them. The most common form for a stream used in GPGPU is a 2D grid, since this fits naturally with the rendering model built into GPUs. This grid design facilitates applications like matrix algebra, image processing, physically based simulations etc. However, programmers still needed to map the problems to a graphics rendering procedure and understand the graphics pipeline until the early 2000s.

1.2.3.2 The CUDA programming model

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. [Wikc] It was first introduced in 2006 by NVIDIA and allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for GPGPU applications. At the time of its introduction CUDA supported only the C programming language, but nowadays it supports FORTRAN , C++ , Java, Python, etc. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. It basically facilitates specialists in parallel programming in using GPU resources without possessing advanced skills in graphics programming. [Zun18]

As explained before, GPUs are designed to solve problems that can be formulated as data-parallel computations – the same instructions are executed in parallel on many data elements with a high ratio between arithmetic operations and memory accesses. This is similar to the SIMD (Single Instruction, Multiple Data) approach of the parallel computers taxonomy. To fully utilize the available CUDA (GPU) cores on a GPU, CUDA adopts a variation of SIMD, which NVIDIA refers to as the single instruction multiple thread (SIMT) style. In SIMT, a program consists of a number of threads and all threads execute the same sequence of instructions. Therefore, if all threads execute the same instruction at the same time, just on different data per thread, a CUDA program would simply be a sequence of SIMD instructions. The difference between SIMT and SIMD lies in the fact that SIMT allows individual threads to execute different instructions (e.g. conditional statements where threads execute different branches). The SIMT model allows this flexibility at the cost of performance, since diverging threads' execution is serialized.[VC13]

The CUDA parallel programming model has three main key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions are exposed to the programmer as language extensions. They provide fine grain data parallelism and thread parallelism together with task parallelism that can be considered coarse grain parallelism. These abstractions also guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block, thus preserving language expressivity by allowing threads to cooperate when solving each sub-problem, while simultaneously enabling automatic scalability. Each block of threads can be scheduled for execution on any of the available processor cores, concurrently or sequentially. This allows a CUDA program to be executed on any number of processor cores.

There are three layers of hierarchy for CUDA threads; threads, thread blocks, and grids. Each thread executes a sequence of instructions, or kernel, i.e. the “program” that all threads execute simultaneously. Each thread has a unique ID which can be used to determine thread-specific operations within the kernel. Threads are grouped into thread blocks that contain any number of threads up to some limit, which on current hardware is 1,024 threads per block. Thread blocks execute completely independently and may be further logically organized into grids. A grid is the entire collection of CUDA threads to execute a given kernel.

The CUDA architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. Each SM contains a number of CUDA cores depending on the GPU model, and at any one time they’re executing a single warp of 32 threads. From a memory perspective, GPUs have multi-level memory hierarchy similar to traditional CPUs. On current generation GPUs, there is a level-2 (L2) cache that is hardware-managed and shared between all multiprocessor cores on the GPU and a level-1 (L1) cache that is local to each core. Additionally, there is the shared memory, which is programmer-managed and local to a thread block, combining with the L1 cache to make up 64 KB in size. Finally, modern GPUs additionally offer a 48KB read-only general purpose constant memory as well as large register files to support hardware multithreading.

Threads within a thread block may coordinate their activity. When a thread reaches a certain point within the kernel, it can be issued to wait until all threads within the same block are at the exact same instruction. This operation is called a barrier. This type of synchronization is only possible within a thread block. Global synchronization can only be achieved on the host, after the execution of a GPU kernel.

1.2.3.3 Other GPGPU frameworks

AMD Accelerated Parallel Processing (former ATI Stream technology) is a set of advanced hardware and software technologies that enable AMD graphics processors (GPU) to cooperate with the CPU in order to accelerate applications (AMD, 2011). The APP programming model resembles the CUDA paradigm. It supports data-parallel and task-parallel programming models.

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL provides a standard interface for parallel computing using task- and data-based parallelism.

Regarding GPU acceleration, OpenCL programs are divided in two parts: one that executes on the device (the GPU) and another that executes on the host (the CPU). The device program is the part of the code that uses GPU for parallel execution. Programmers have to write special functions called kernels which uses OpenCL Programming Language (an extension to the C programming language). These kernels are scheduled to be executed on GPU. The host program offers an API that allows programs running on the host to launch kernels on the compute devices and manage device memory, which is (at least conceptually) separate from host memory. It can be programmed in C or C++ and it controls the OpenCL environment. Programs in the OpenCL language are intended to be compiled at run-time, so that OpenCL-using applications are portable between implementations for various host devices.[SGS10]

OpenCL shares several concepts with the CUDA programming model. For example, it also uses the Single Instruction, Multiple Thread (SIMT) model of execution. Moreover, it shares equivalent thread hierarchy with CUDA. Work-items are the equivalent of the CUDA threads being the basic execution entity. Work-items cooperate between them within a work-group, which is the OpenCL equivalent of CUDA thread blocks. In addition, OpenCL follows the same synchronization model, as barriers can be set in kernel code which will stop all threads within the same work-group to wait until they all reach the barrier, but provide no work-group wide synchronization.

Chapter 2

Related Work on Accelerated Computational Neuroscience

2.1 Popular Neural Simulation Frameworks

Since computational neuroscience relies heavily on mathematical modeling in order to analyze brain functionality, experiments related to the discipline rely on heavy numerical operations generated for example by the methods needed to solve the differential equations that describe the given neural model. Therefore, experiments must be run in silico, i.e. simulations performed on computers. The fact that neurons and their synapses are complex entities which need a vast number of parameters for their activity to be described effectively, as well as the need to conduct experiments regarding the activity of systems comprised of hundreds or thousands of neurons leads to an increasing demand of computational power. Moreover, in order to facilitate scientists in conducting a grand number of experiments without programming expertise, while there is no unified simulation tool that complies with the entirety of computational neuroscience's needs, a variety of simulation frameworks have established their role in the discipline. Some of the most widely-used neuron simulator frameworks are presented below.

The NEURON Simulation Environment is designed for modeling individual neurons and networks of neurons, and is widely used by experimental and theoretical neuroscientists. Developed by Michael Hines, John W. Moore, and Ted Carnevale at Yale and Duke, it provides tools for conveniently building, managing, and using models that are numerically sound and computationally efficient. Though it began in the domain of single-cell models, since the early 1990s it has been applied to network models that contain large numbers of cells and connections, enhanced by the introduction of runtime-shortening features. [HC01]

The Neural Simulation Tool NEST is a computer program for simulating large heterogeneous networks of point neurons or neurons with a small number of compartments. NEST is best suited for models that focus on the dynamics, size, and structure of neural systems rather than on the detailed morphological and biophysical properties of individual neurons. It is developed by the NEST initiative, initially released in August, 2004. NEST possesses the trait of extensibility, meaning that new models for neurons, synapses, and devices can be added anytime. [GD07]

Brian is an open source Python package for developing simulations of networks of spiking neurons. The design is aimed at facilitating its understanding and use, enhancing its flexibility and expressiveness and, last in priority, increasing computational efficiency. Users specify neuron and synapse models by giving their equations in standard mathematical form. The intent is to make the process as flexible as possible, so that researchers are not restricted to using neuron and synapse models already built in to the simulator. The entire simulator is written in Python, using the NumPy and SciPy numerical and scientific computing packages. [GB09]

MOOSE (Multiphysics Object Oriented Simulation Environment) is an object-oriented C++ finite element framework for the development of tightly coupled multiphysics solvers from Idaho National Laboratory.[Wika] A key design aspect of MOOSE is the decomposition of weak form residual equations into separate terms that are each represented by compute kernels. The combination of these kernels into complete residuals describing the problem to be solved is performed at run time. This allows modifications such as toggling of mechanisms and the addition of new physics without recompilation.

GENESIS (the GEneral NEural SImulation System) is a general purpose software platform that was developed to support the biologically realistic simulation of neural systems, ranging from subcellular components and biochemical reactions to complex models of single neurons, simulations of large networks, and systems-level models. [Wikd] It was originally developed in the laboratory of Dr. James M. Bower at Caltech. The "building block" object-oriented approach taken by GENESIS and its high-level simulation language allows modelers to easily extend the capabilities of the simulator, and to exchange, modify, and reuse models or model components.

2.2 GPU-accelerated Simulators

While frameworks such as the ones mentioned above offer the advantages of little to no programming knowledge required coupled with the capacity to simulate user-defined networks without neuron architecture limitations, they exhibit important performance drawbacks. This is due to the fact that such extended simulators cannot offer optimal performance for every possible user-define supported experiment, in addition to lack of optimization for particular high performance computer architectures in their standard distributions. As neural network experiments become broader and more complex, initiatives that utilize specialized, often single model-specific platforms and attempt to significantly improve runtime performance have appeared.

A widely-used technology used for program runtime acceleration in general, which has proven to be as effective in several neuron related experiments, is acceleration via GPU, in which this thesis focuses. Due to significant memory (capacity for significantly large neural networks) and performance capabilities, GPUs accelerate simulated experiments to such a degree that computer-aided simulations can be conducted in a fraction of the time it would take to conduct a much more complicated in vivo experiment. Some examples of GPU usage for accelerated simulations is referenced below.

A team of scientists from the Computer Science and Cognitive Science departments of the University of California Irvine have implemented a biologically realistic Spiking Neural Network (SNN) simulator that runs on a single GPU. While the model they used for experimentation included Izhikevich spiking neurons, detailed models of synaptic plasticity and variable axonal delay, the simulator allows user-defined configuration of the exact model architecture by means of a high-level programming interface written in C++. The team used an NVIDIA GTX-280 with 1 GB of memory to simulate for the simulation of 100K neurons with 50 Million synaptic connections, firing at an average rate of 7 Hz, reaching up to 26 times faster runtime than the CPU version of the simulation. [Nag+09]

Due to its range of realistic spiking dynamics, in combination with relatively adequate computational efficiency, the Izhikevich neuron model has gained popularity in recent neuroscience research. Another team from the Imperial College of London developed NeMo, a platform which "achieves high performance through the use of highly parallel commodity hardware in the form of graphics processing units (GPUs)". The implemented GPU kernel delivers up to 400 million spikes per second, which corresponds to a real-time simulation of around 40000 neurons, each connected through 1000 synapses, with a mean firing rate of 10 Hz. [KF+09]

The Brain Computation Laboratory of the Department of Computer Science and Engineering of the University of Nevada presented the NeoCortical Simulator version 6 (NCS6), a free open-source scalable CPU/GPU simulation environment for large-scale biological networks. It is designed to run on clusters of multiple machines equipped with GPU devices. It expands from the Izhikevich and additionally supports the Leaky Integrate-and-Fire neuron model. However, users have the capability to design their own neuron model through the simulator interface, expanding this simulator's usefulness to any computational neuroscience experiment relating neurons and their synapses. NCS6 is currently able to simulate 1 million cells with 100 million synapses in total in quasi real time, distributing data across eight nodes, each having two video cards, in order to cover memory requirements. [Hoa+13]

Last but not least, scientists from the Department of Electrical and Computer Engineering of Clemson University implemented a simulator for the Hodgkin-Huxley and the Izhikevich model that supported both CPU and GPU acceleration. They exploited CUDA memory optimizations such as Coalesced Global Memory Accesses , Shared Memory and Texture Memory in order to get the best results possible on GPUs and compared them to equally researched CPU accelerations, the GPUs outperforming CPUs as the network gets larger. The simulated neuron architecture consisted of 2 levels of neurons, the latter being a token one compared to level 1's number of neurons. Thus, GPUs are exploited for the simulation of level 1's neurons and level 2's necessary calculations are performed on the host CPU, combining to result in significant runtime decrease for the conducted experiments. [BPS10]

2.3 BrainFrame

The project that sparked my interest in Computational Neuroscience and urged me to help in its development is called Brainframe. Brainframe is a heterogeneous node-level acceleration platform for neuron simulations that utilizes three distinct acceleration technologies, originally an Intel Xeon-Phi CPU (Intel Xeon Phi 5110P), a Maxwell-based GP-GPU (Nvidia Titan X) and a Maxeler Dataflow Engine (Maxeler Maia DFE), all of them using PCIe slots in order to communicate with the host system. This manner of communication ensures that Brainframe-enabled machines can be easily tailored on a per-case basis depending on the availability of funds and hardware resources of any research laboratory. Thus, different types of PCIe-based accelerators can also be selected. Thus, depending on the conducted experiment, the platform automatically chooses the best suitable accelerator in order to balance satisfactory acceleration and as low as possible energy consumption. Furthermore, the PyNN software framework is integrated into the platform, providing a familiar bridge to the vast number of models already available, as it is a network model-building API for the Python programming language that neuroscientists can use to run their experiments on Brainframe while being supported by other popular simulators such as NEURON and NEST without requiring modifications even on a single line of code. The PyNN frontend coupled with the PCIe-based host-device communication offers significant flexibility to the Brainframe platform, as both different devices and neural models can be added, the latter constituting part of the purpose of this thesis. [GS17; APDY09]

Chapter 3

Problem Statement

In the process of enriching Brainframe with new neuron models, the model I took on as a responsibility is the addition of an Adaptive Exponential Integrate-and-Fire (AdEx) neuron network model, which is also characterized by Spike-timing Dependent Plasticity (STDP). This mathematical model was inspired from a neuroscience paper which researched the ability of synapses to memorize a specific behaviour and whether they could quickly recall that behaviour if activated by similar spikes. [RPC15] The aim of the paper was to show that spike-timing-dependent plasticity with both pre- and postsynaptic expression develops receptive fields with reduced variability and improved discriminability compared to postsynaptic plasticity alone, leaving a hidden postsynaptic memory trace that enables fast relearning of previously stored information. Its results revealed the importance of including presynaptic plasticity so as to obtain significant characteristics that are missed when only postsynaptic expression of long-term plasticity is considered. From an implementation perspective, the experiments conducted for this paper's purposes used the PyNN interface and the already-mentioned Brian simulator [GB09] to define model parameters and simulate the system, respectively. However, due to the experiment's modest size and consequently limited resource requirements, there was no acceleration whatsoever and the Brian simulator showcased prohibitive runtime increase as the experiment scaled to larger dimensions. Thus, a fellow student and I undertook the task of accelerating this model's simulation time on different platforms before importing it to Brainframe. Both platforms required that we imported the simulation of the particular model to a custom C program. This led to me and the fellow student decoding the exact way that the Brian simulator works and designing a model-specific simulator on the C programming language that can currently conduct experiments regarding only the referenced model, though enabling the user to define parameter values for both neurons and their synapses. Afterwards, my responsibility was to achieve optimal performance using the CUDA parallel computing platform, accelerating the simulations on GPUs. The process of the optimization will be analyzed later. Henceforth, the models of AdEx Neurons and STDP will be defined, followed by a description of the process of importing the Brian simulation 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 model is especially reliable in high-conductance states, typical of cortical activity in vivo (experimentation using a whole, living organism), in which intrinsic conductances were found to have a reduced role in shaping spike trains. It has become popular in modern research due to its expressive power, enough to reproduce qualitatively several electrophysiological classes described in vitro (experimenting in a controlled environment outside of a living organism).

The following are the differential equations describing the AdEx model:

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

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

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

Parameters:

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] According to the Hebbian rule, synapses increase their efficiency if the synapse persistently takes part in firing the postsynaptic target neuron. [Heb49] Experiments that stimulated two connected neurons with varying interstimulus asynchrony confirmed the importance of temporal precedence implicit in Hebb's principle: the presynaptic neuron has to fire just before the postsynaptic neuron for the synapse to be potentiated. [CD08] In addition, it has become evident that the presynaptic neural firing needs to consistently predict the postsynaptic firing for synaptic plasticity to occur robustly. [BLN01] With STDP, repeated presynaptic spike arrival a few milliseconds before postsynaptic action potentials leads in many synapse types to Long-Term Potentiation (LTP) of the synapses, whereas repeated spike arrival after postsynaptic spikes leads to Long-Term Depression (LTD) of the same synapse. 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 paper this thesis is based on attempts to study the functional consequences of the division between pre- and postsynaptic plasticity by developing a biologically tuned spike-timing dependent plasticity model that involves both parts of the stdp expression. [RPC15]

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

From a mathematical point of view, plasticity in this model is divided into Short-term and Long-term, each having their seperate set of differential equations. Below are listed the equations related to Short-term plasticity:

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

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

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

r: (Normalized) Number of Vesicles

p: Presynaptic Factor

D: Depression Time Constant

P: Baseline Presynaptic Factor

F: Facilitation Constant

As for the Long-term plasticity model, the postsynaptic depression trace is defined as:

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

for the postsynaptic potentiation trace:

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

and for the presynaptic potentiation trace:

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

where:

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 whenever the presynaptic cell is active 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 been stimulated recently this simplifies to:

$$w = Pq$$

Chapter 4

Initial Approach

As mentioned before, in order to achieve an ideal acceleration environment compatible with multiple devices, a fellow student working on a related thesis and I implemented a C port of the Python-based Brian version of the simulation. As a matter of fact, we extended over replicating the specific experiment and coded a standalone simulator that currently supports only the AdEx neuron model with its synapses characterized by Spike-Timing Dependent Plasticity. However, several features such as neuron connectivity, spike frequency, parameter values and of course neuron and synapse quantity have been turned into user-defined attributes, allowing neuroscientists interested in this model to independently conduct personalized experiments. In order to successfully implement our simulator, a thorough analysis of Brian's characteristics, namely the code responsible for handling the particular network and solving differential equations that describe this model's different parts.

4.1 Brian Architecture

The following is an explanation of Brian's python code related to the model of Costa et al. and the experiment mentioned in their paper. [RPC15] The code can be divided into three segments, analogous to the definition of Input Neurons, which represent artificially induced neural spikes that are feeded directly into the AdEx neurons as input of the experiment, hence not following any specific physical model, the definition of the Neurons of the AdEx model, from now on often referred to as model neurons, and the Synapses used to connect both Input and AdEx neurons to each other. Regarding network connectivity, it shall be noted here that both Input and AdEx neurons have been made possible to connect to any other neuron created for the simulation, stating the fact that all synapses follow the same STDP model.

4.1.1 Input Neurons

Input Neurons are created by calling the constructor of a *PoissonGroup* class. This Python class' purpose is to create the network of neurons that feed the model neurons with artificial spikes in random moments. It requires passing the following parameters:

- N : (int) Number of Input Neurons
- rates : Either a constant rate, an array of rates (one rate per neuron), or a string expression evaluating to a rate as an argument. This parameter will be evaluated at every time step and therefore allows the use of time-dependent rates, i.e. inhomogeneous Poisson processes, thus the name PoissonGroup.

In order to replace passing spikes on a given rate as input with explicitly manually defined spikes at user-defined time steps, the class SpikeGenerator is used. It defines a quantity of Input Neurons that spike based on a given array that defines independent neuron- and time-specific spikes. The parameters required by this class are:

- N : (int) Number of Input Neurons
- spiketimes : (int, int) Array of tuples that contain the indices of Input Neurons to fire spikes and the timesteps these spikes will be produced and passed to the model neurons.

4.1.2 Model Neurons

Neurons corresponding to the model defined by the user are created as a *NeuronGroup* class, which contains all the necessary neuron variables for the simulation and is responsible for keeping record of the neurons that spike, propagating the list to the Synapses class in order to update the corresponding synaptic variables. Moreover, this class chooses automatically (unless explicitly shown else) the method that solves the differential equations of the models. It is called with the following parameters:

- N : (int) Number of Neurons
- model : (Equations class/string/StateUpdater class) This parameter states The differential equations defining the behavior of the neurons that are created by this class initializer. The first two possibilities are essentially a string representing the differential equations, while the latter is a class that updates the state variables of the neurons at every timestep depending on the defined method.
- threshold : (Threshold object/function/scalar quantity/string) This parameter defines the condition under which the neuron produces spikes. Usually a string describing a boolean expression between variables or a variable and a constant.
- reset : (Reset object/function/scalar quantity/string) This parameter defines what happens when a neuron spikes. It is basically The (possibly multi-line) string with the code to execute on reset.
- freeze : (True/False) If True, parameters are replaced by their values at the time of initialization

4.1.3 Synapses

Synaptic connections are created by calling the *Synapses* class constructor. This class connects two Neuron Groups (e.g. PoissonGroup, NeuronGroup) given some user-submitted synaptic model and updates the values of the produced synapses on every timestep. Apart from the Neuron Groups, this class also requires these parameters:

- model : (Equations object/string) The (differential) equations that define the synaptic variables and their relations. Same syntax as the Neuron Group
- pre : (list/tuple of strings) TThe code that will be executed after every pre-synaptic spike. Can be either a single (possibly multi-line) string, or a touple of strings defining seperate commands.
- post : (list/tuple of strings) The code that will be executed after every post-synaptic spike. Same conventions as for pre.

4.1.4 Brian Code

The code used to run the simulation in Brian is the following:

```
 1 eqs_neuron = """
 2         dvm/dt=(gL*(EL-vm)+gL*DeltaT*exp((vm-vt)/DeltaT)+I-x)/C : volt
 3         dvt/dt=-(vt-vtrest)/tauvt : volt
 4         dx/dt=(c*(vm-EL)-x)/tauw : amp #In the standard formulation x is w
 5         I : amp
 6     """
 7 
 8 F_input1 = ones(N)*Foff
 9 for i in range(0,N): #Define gaussian input
10     F_input1[i] = exp(-((((i+1)-input1_pos)**2)/(2.0*rad**2)))*(Fon-Foff)+Foff;
11 
12 input = PoissonGroup(N, rates=F_input1)
13 
14 neurons = NeuronGroup(M, model=eqs_neuron, threshold='vm>vt',
15                          reset="vm=Vr;x+=b;vt=VTmax", freeze = True)
16 InitializeNeurons(neurons)
17 
18 #user_input = SpikeGeneratorGroup(N, spiketimes)
19 
20 model='''w : 1
21          FFp : 1
22          FBp : 1
23          FBn : 1
24          R : 1
25          u : 1
26          U : 1
27          A : 1
28          dFFp/dt=-FFp/tau_FFp : 1 (event-driven)
29          dFBp/dt=-FBp/tau_FBp : 1 (event-driven)
30          dFBn/dt=-FBn/tau_FBn : 1 (event-driven)
31          dR/dt=(1-R)/tau_r : 1 (event-driven)
32          du/dt=(U-u)/tau_u : 1 (event-driven)
33      '''
34 syn = Synapses(my_input, neurons, model,
35             pre='''I=s*A*R*u;
36                 U=clip(U+etaU*(-AFBn*FBn*FBp + AFBp*FBp*FFp),Umin,Umax);
37                 w=U*A;
38                 FFp+=1; R-=R*u; u+=U*(1-u)''',
39             post='''A=A+etaA*(AFFp*FFp*FBn);
40                 A=A-etaA*0.5*mean(AFFp*FFp*FBn);
41                 A=clip(A,Amin,Amax);
42                 w=U*A;
43                 FBp+=1.;FBn+=1.''')
44 syn[:,:]=True
45 InitializeSynapses(syn)
46 run(stime)
```

A gaussian expression defines the rate at which Input Neurons will spike. In line 16, a user-input assignment is commented out which would be an alternative Input Neuron instance that could be passed to the Synapses class. The parameter called spiketimes would be an array of touples descripting spikes. In addition, in line 42 every synapse is defined as "connected" to its corresponding neurons by changing a value of a two-dimensional boolean matrix to 1. Creating a Synapses class alone does not connect any synapse with a neuron.

In order to connect neurons, it needs to be defined explicitly in the form of a connectivity matrix that the Synapses class offers.

4.2 C Simulator Architecture

After obtaining a thorough understanding of its backend and the exact process behind the simulations conducted in Brian, my fellow student and I implemented the simulator in the C programming language. The original simulation was a ModelDB project that used 100 Input Neurons to feed spikes to a single AdEx Neuron and monitor its behaviour, defining the synapses according to the STDP model. The extended simulator allows for user-defined experiments of **N** Input Neurons feeding spikes to **M** AdEx Neurons. Synapses also follow the STDP model, connecting both Input to AdEx Neurons as well as AdEx Neurons with one another, enabling the option of Model-only , **MxM** experiments apart from the original **NxM** ones. In the case of **MxM**, AdEx spikes cannot be caused automatically as a result of input, but users can freely choose spike density and frequency percentage-wise.

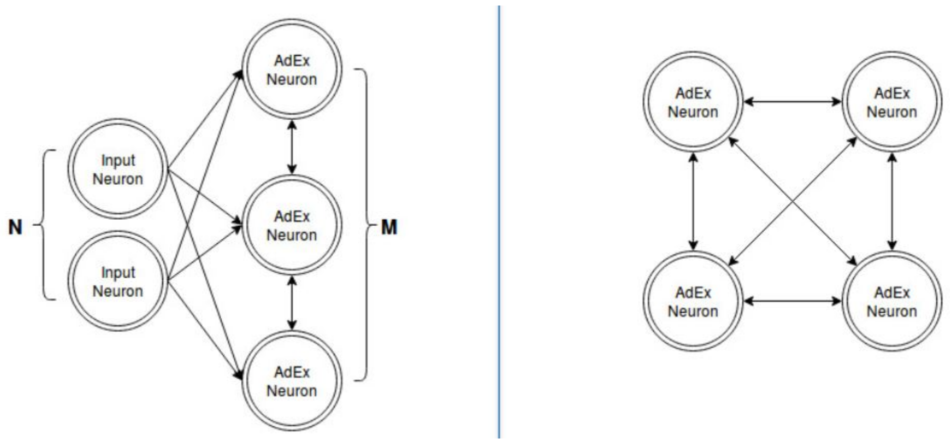


Figure 4.1: Neuron architectures

In order to select the network architecture to be simulated, users need to define 3 parameters. First is the number of Input Neurons (**N_S**, as of Number_Source, the ones feeding spikes to the network). As of the current version, AdEx neurons need to be defined twice in the form of Number of AdEx Group Source neurons (**N_Group_S**) and Number of AdEx Group Target neurons (**N_Group_T**), in order to match the original source (Input) - target (AdEx) template. When non-zero, the last two parameters must be set equal. To summarize, the following neuron architectures are allowed:

- **N_SxN_Group_T**, where **N_Group_S** is set to zero.
- **N_Group_SxN_Group_T**, in which case **N_S** is set to zero.

An important part of the implementation was the selection of the data structures to be used to keep all the necessary data of neurons and their synapses. An array of integers of length **N_S** + **N_Group_S** is used to represent spike occurrences on the corresponding neurons, named *SpikeArray*. As Input Neurons' sole purpose is spike feeding, they are sufficiently represented by **N_S** array slots. However, the array is first filled by all the necessary AdEx Neuron slots before the ones representing Input Neurons. Even though a boolean definition of *SpikeArray* would be more memory-efficient, much larger data structures lead to this overhead being negligible, hence we rest on this datatype in order to increase further development assistance e.g. debug codes, expanded Input Neuron functionality etc.

Due to the different variables of the AdEx neuron model that need saving in order to correctly solve the model equations on every timestep, AdEx Neurons are represented by a struct named Neuron which contains the variables holding the value of the voltage threshold, membrane potential, input current and the model's adaption variable, all of which are defined as *double* for sufficient precision, as well as an integer field whose value explains whether the neuron has spiked on the current timestep. While this is represented by *SpikeArray*, it has been preserved to ensure complete correspondence to the original Brian code, as asked. The same principle is applied on several code segments of our simulator with no significant memory overhead.

The STDP model and the differential equations that describe the synapses lead to the implementation of struct *Synapse* to save all the related information that synapses need. This includes the synaptic variables, a field that maintains the last time a value of the struct was updated, the current to be propagated to the post-synaptic neuron and lastly a variable that actually expresses whether the given synapse exists from the experiment's point of view. The explanation for the latter is that both the ensembles of neurons and synapses are saved as two arrays of structs (AoS). This layout was preferred because of spatial locality between interconnected variables and offered performance advantages compared to a single struct of arrays (SoA) for each element, in the original serial case. Therefore, synapses are stored in an adjacency array (of structs) of size **NxM** or **MxM**, depending on the simulation. The *conn* variable defines connectivity, which can be read by an input file as well as left to a default attribute of 50%.

The code for the datatype definitions is the following:

```c
int SpikeArray[N];

typedef struct {
    double vt;    /* Voltage threshold. */
    double vm;    /* Membrane potential.   */
    double I;   /* Neuron input current. */
    double x;   /* Adaption variable (w). */
    int Spike;    /* Spike occurence */
    } Neuron;

typedef struct {
    int conn; /* Synapse connectivity. */
    double w; /* Weight of a synapse. (not used in STDP) */
    double FFp;   /* FFp variable of a synapse. (x+) */
    double FBp;   /* FBp variable of a synapse. (y+) */
    double FBn;   /* FBn variable of a synapse. (y-) */
    double R;   /* R value of a synapse. (r)  */
    double u;   /* u value of a synapse. (p)  */
    double U;   /* U value of a synapse. (P)  */
    double A;   /* A value of a synapse. (q)  */
    double lastupdate;   /* Last time a synapse was updated.  */
    double target_I;  /* The I value for the postsynaptic neuron. */
    } Synapse;
```

When a simulation runs on Brian, the functions that occur in order to accurately update the variables defined by the model can be distinguished into three different subsets corresponding to the action-phenomena that happen on spike occurences. Thus, we have coded three main functions that shall be explained below along with their C code mildly simplified for clarity reasons.

- The first function is responsible for the update of the values of AdEx Neurons by solving the differential equations that define the model on every timestep.

  ```c
  void SolveNeurons(Neuron* neurons, int N, int *SpikeArray){
    for(int i = 0; i < N; i++){
      /*update variables according to solution*/
      if(neurons[i].vm > neurons[i].vt){
        resetNeuron(&neurons[i]);
        SpikeArray[i] = 1;
      }
      else SpikeArray[i] = 0;
    }
  }
  ```

  In the case that an AdEx neuron's membrane potential crosses the voltage threshold, it indicates that the neuron has fired. Therefore, the *SpikeArray* field that corresponds to the particular neuron is updated accordingly and the neuron's variables are reset to rest status.

- When a spike occurs, synaptic variables of the synapses connecting the neuron that fired to other neurons must be updated according to STDP and the value of the current that appears needs to be propagated to the post-synaptic neurons. Therefore, this function was implemented:

```
 1 void UpdateSynapses_pre(Synapse** Synapses, Neuron* neurons, int N_S,
     int N_Group_S, int N_Group_T, int* SpikeArray, double t){
 2   for (int i = 0; i < N_S + N_Group_S; i++){
 3     if (SpikeArray[i+N_Group_T-N_Group_S] > 0){
 4       for (int j = 0; j < N_Group_T; j++){
 5         if (Synapses[i][j].conn) update_variables();
 6       }
 7     }
 8   }
 9   for (j = 0; j < N_Group_T; j++){
10     for (i = 0; i < N_S + N_Group_S; i++){
11       if (Synapses[i][j].conn && SpikeArray[i+N_Group_T-N_Group_S]){
12         neurons[j].I = Synapses[i][j].target_I;
13       }
14     }
15   }
16 }
```

Here it should be noted that in the case of **NxM** experiments, changes to synaptic variables ensue only when Input Neurons fire (hence the conditions on lines 3 and 11), while in **MxM** experiments calculations are done on every AdEx Neuron spike. Moreover, target electric current propagation needs to be done on a seperate nested for-loop due to the different way neurons are accessed (column-wise instead of row-wise). For every Input Neuron (i) that has generated a spike, the synapses that start from it and the post-synaptic neuron (j) are updated.

- Last but not least, new values are calculated for synaptic variables when the post-synaptic AdEx Neurons fire, based on the differential equations describing the post-synaptic expression of STDP. Following is the function responsible for this phenomenon:

```
void UpdateSynapses_post(Synapse** Synapses, Neuron* neurons, int N_S,
    int N_Group_S, int N_Group_T, int* SpikeArray, double t){
  for (int i = 0; i < N_Group_T; i++){
    if (SpikeArray[i] > 0){
      for (int j = 0; j < N_S + N_Group_S; j++){
        if (Synapses[j][i].conn) update_variables_1();
      }
    }
  }

  calculate_mean_value();

  for (i = 0; i < N_Group_T; i++){
    if (SpikeArray[i] > 0){
      for (j = 0; j < N_S + N_Group_S; j++){
        if (Synapses[j][i].conn) update_variables_2();
      }
    }
  }
}
```

This function follows similar principles to the second one. Even though the for-loops are based on the same conditions, the average of two synaptic variables' values over all firing neurons of the particular timestep must be calculated in between so as to perform correct calculations.

To conclude, taking into consideration that *SpikeArray* needs to be reinitialized on every timestep after the solution of the AdEx model equations, the loop that encapsulates the simulation resembles the following:

```
for(int t = 0; t < timesteps; t++){
    SolveNeurons(neurons, N_Group_T, SpikeArray);

    InitializeSpikeArray(SpikeArray, N_S);
    UpdateSynapses_pre(synapses, neurons, N_S, N_Group_S, N_Group_T,
   SpikeArray, t*defaultclock_dt);

    UpdateSynapses_post(synapses, N_S, N_Group_S, N_Group_T, SpikeArray, t*
   defaultclock_dt);
}
```

Chapter 5

Parallel Programming Optimization

After having implemented the simulator in C in cooperation with another student, my personal task on this thesis was developing a parallel computing version of the simulator to compare the performance gains when running in (max) parallel on GPUs using the CUDA API. Moreover, a baseline acceleration version was coded that utilized the OpenMP programming interface for multi-threaded CPU acceleration, so as to effectively understand the actual advantages and drawbacks of using GPUs for the particular simulator's experiments.

5.1 CUDA implementation

5.1.1 General Design Features

Modern (CUDA-compatible) GPUs contain a multitude of processing units termed as CUDA cores, which in turn support plenty of threads running simultaneously on each. For example, calling CUDA's *cudaGetDeviceProperties()* on a PASCAL GeForce Titan X GPU lists the specs of the card as follows:

- Maximum number of threads per block: 1024
- Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
- Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

As mentioned before, a grid is the entire collection of CUDA threads to execute a given kernel. Therefore a Titan X can theoretically execute a program using up to (maxGridSize x maxThreadsPerBlock) threads, even though per thread resource limitation might restrict the total number of threads per block to less than this maximum. Therefore, taking into consideration that due to the C program architecture analyzed before all computations of the model are neuron- and synapse-specific, meaning there are little to no data dependencies between their structs on a single timestep, the approach chosen for the GPU acceleration was to assign every single struct used for computations to a different thread. Each neuron and consequentially every synapse -depending on the kernel- is handled by a single, independent thread, enabling maximum possible parallelism and the least possible calculation load for each thread. This choice is ensured by defining a grid size analogous to the number of elements to be processed, for example:

```
int blocksize = 256; /* chosen block size */
int gridsize = (N_Group_T + blocksize - 1) / blocksize; /* number of neurons
    divided by number of blocks */
neuron_kernel<<<gridsize,blocksize>>>(neurons); /* launch GPU kernel */
```

It shall be noted here that a CUDA kernel call is recognised by the **<<<...>>>** execution configuration syntax.

5.1.2 Data Structures and Memory Handling

Since CUDA is a programming interface that fully supports the C language, only a few modifications needed to be done to the existing data structures of the simulator's source code, taking into consideration that all necessary existing data structures are allocated on the device and their data is copied using a CUDA dedicated function. First of all, it should be noted that in order to process data on a GPU by running a CUDA kernel the necessary data must be allocated on and/or passed to the device. In the case of synapses, the CUDA API offered two possibilities for passing their data to the device. This is due to the fact that the array of structs (AoS) that acts as an adjacency matrix as well as stores the information needed for every synapse can be either a classic two-dimensional C array (of structs) or a one-dimensional representation of the same array, since the rows of a two-dimensional array can be stored in contiguous memory locations and accessed by utilizing pointer arithmetics.

eikona

eikona

eikona

eikona

Regarding the 2D version, for simplicity purposes we initially consider that the array is allocated statically:

```
typedef struct { /* synaptic variables */ } Synapse;
Synapse syn[N_S+N_Group_S][N_Group_T]; /* rows = N_S + N_Group_S; columns =
    N_Group_T */
```

This ensures that a contiguous block of memory is allocated for the two-dimensional array which is stored by rows in C and C++. Array allocation and/or copying requires every array item to be in adjacent memory positions with its neighboring ones, as there is no CUDA operation to handle discontinuity between array elements. When accessing 2D arrays in CUDA, memory transactions are much faster if each row is properly aligned. CUDA provides the *cudaMallocPitch* function to properly pad each row with extra bytes so to achieve the desired alignment. Device memory allocation for the array is therefore coded as such:

```
Synapse *dev_syn;
cudaMallocPitch((void**)&dev_syn, &dev_pitch, N_Group_T * sizeof(Synapse),
    N_S+N_Group_S);
```

Where *dev_syn* is a pointer to the allocated memory space, *dev_pitch* is a *size_t* output variable denoting the length, in bytes, of the padded row and the last two parameters define array size in columns and rows, noting that columns size is also defined in bytes. *cudaMallocPitch* will allocate a memory space of size, in bytes, equal to *N_S+N_Group_S * pitch*. However, only the first *N_Group_T * sizeof(Synapse)* bytes of each row will contain the array data. Accordingly, *cudaMallocPitch* consumes more memory than strictly necessary for the 2D array storage, but this is returned in more efficient memory accesses.

Obviously, the fact that I am allocating an array of structs instead of e.g. a *float* array has no consequences other than the extensive memory allocation needs. Assuming data is allocated on device memory using *cudaMallocPitch*, the function responsible for copying the data between host and device memory space is called *cudaMemcpy2D*, its syntax being as follows:

```
cudaMemcpy2D(devPtr, devPitch, hostPtr, hostPitch, Ncols * sizeof(float),
    Nrows, cudaMemcpyHostToDevice)
```

Where:

- *devPtr* and *hostPtr* are input pointers to the (source) device and (destination) host memory spaces, respectively;
- *devPitch* and *hostPitch* are *size_t* input variables denoting the length, in bytes, of the padded rows for the device and host memory spaces, respectively, *hostPitch* being equal to the number of columns allocated for the array on the host space
- array size is defined as in *cudaMallocPitch* and the last parameter defines source and target memory space.

Therefore, memory copies on my code are written as follows:

```
cudaMemcpy2D(dev_syn, pitch, syn, N_Group_T * sizeof(Synapse), N_Group_T *
    sizeof(Synapse), N_S+N_Group_S, cudaMemcpyHostToDevice);

synapses_kernel<<<gridsize,blocksize>>>(dev_syn);

cudaMemcpy2D(syn, N_Group_T * sizeof(Synapse), dev_syn, pitch, N_Group_T *
    sizeof(Synapse), N_S+N_Group_S, cudaMemcpyDeviceToHost);
```

describing host-to-device and device-to-host copies respectively. From a kernel standpoint, the access to elements of the 2D array allocated by *cudaMallocPitch* can be performed as in the following example:

```
int tidx = blockIdx.x*blockDim.x + threadIdx.x;
int tidy = blockIdx.y*blockDim.y + threadIdx.y;
if ((tidx < Ncols) && (tidy < Nrows))
{
    Synapse* row = (Synapse *)((char*)Synapses + tidy * pitch);
    if (row[tidx].conn){
        /* perform operations */
    }
}
```

The pointer to the first element of a row is calculated by offsetting the initial pointer *Synapses* by the row length *tidy * pitch* in bytes (*char ** is a pointer to bytes and *sizeof(char)* is 1 byte), where the length of each row is computed by using the *pitch* information.

Finally, below is the actual code used for dynamic memory allocation of the two-dimensional array of structs of synaptic data, ensuring that rows are stored in contiguous memory locations:

```
Synapse (*syn)[N_Group_T] = malloc(sizeof *Synapse * (N_Group_S + N_S));
if (syn)
{
  /* do stuff with syn[i][j] */
  free(syn);
}
```

In reality, the improvements arising from the use of *cudaMallocPitch* depend on the compute capability and are more significant for older ones. For GPUs with most recent compute capabilities, such as the ones used for this thesis, pitched memory allocation does not seem to lead to a relevant speedup. Moreover, using *cudaMallocPitch* leads to a slight increase in memory occupancy which, while usually negligible, is preferred to be avoided in the simulator's case due to the vastly memory-consuming data structure.

The above reasons subsequently led to choosing a one-dimensional representation of the 2D array. The code for memory allocation on the device and copies between host and device memory space is listed below:

```
//allocate device memory
Synapse *dev_syn;
cudaMalloc(&dev_syn, sizeof(Synapse) * (N_S+N_Group_S) * (N_Group_T));

//pass data to device
cudaMemcpy(dev_syn, syn, sizeof(Synapse) * (N_S+N_Group_S) * (N_Group_T),
    cudaMemcpyHostToDevice);

synapses_kernel<<<gridsize, blocksize>>>(dev_syn);

//get data back
cudaMemcpy(syn, dev_syn, sizeof(Synapse) * (N_S+N_Group_S) * (N_Group_T),
    cudaMemcpyDeviceToHost);
```

Access to array elements is achieved in regular C manner, using the thread ID as array subscript. This approach also offers the advantage of significantly simpler code.

Apart from adapting the synapses array of structs to a one-dimensional representation, the only other difference regarding data structures is the addition of a *double* datatype array called *testvar* ("test variable", the name comes from debugging purposes) of the same dimensions as *syn*. The third function mentioned in section 4.2 contains the calculation of a mean value by combining three synaptic variables. On the CUDA implementation, this operation initially couldn't be done in parallel by running a kernel (shall be explained later). Thus, the data needed to calculate this value had to be sent back to the host and the result had to be propagated back to the device for completing the function's operations. *testvar*'s purpose is to minimize the amount of memory to be passed to the host, as each synapse -subsequently each thread- stores the value to be added to the *mean* variable in the corresponding *testvar* element. Then, only *testvar* is copied back to the host, which consumes significantly less memory as a *double* array instead of a *Synapses* array, making the *cudaMemcpy* operation faster.

5.1.3 Kernel Design

The most important part of the GPU implementation is obviously designing the kernel architecture for the program to run optimally in parallel. There have been various considerations regarding kernel size, contents and quantity, taking into account the flexibility and limitations imposed by the process of solving the equations of the models of AdEx Neurons and their STDP-characterized synapses. Final design choices were made bearing in mind optimal program functionality, adequate acceleration and output flexibility, i.e. being able to output multiple variable values on every timestep, maintaining the possibility for user-defined loquaciousness levels. Moreover, datatype variances were explored as well as other minor features that could or could not have affected overall performance.

5.1.3.1 Device code

The initial approach towards kernel definition was to try and design the GPU program on a similar note to the original simulator's source code. Specifically, the entire simulation would have an identical flow to the CPU version but instead of calls to the functions *SolveNeurons*, *UpdateSynapses_pre* and *UpdateSynapses_post* there would be calls to corresponding kernels. Since the entirety of the arithmetic calculations relevant to the neuron and synapse models is performed in the aforementioned functions, it also seemed that, assuming the simulator needed to output data to the user only after the completion of the simulation, simulation data only needed to be transferred to the device once in its entirety on the initialization step and then sent back to the host after every model variable has acquired its final value. Each kernel was given the same name as the original functions for simplicity.

To properly indicate the kernel-solver function correspondence while presenting kernel implementations, the following general concept should be noted. As the final implementation used a 1D-representation for all arrays, the CUDA grid is also set to be one-dimensional along with the thread blocks. In this case a thread's unique ID can be retrieved by executing a simple command:

```
int id = blockIdx.x*blockDim.x + threadIdx.x;
```

This command sets *id* to a value that equals to the thread global index (and not the one within the block) and can be used to access data structures' positions that the particular thread is responsible for. As already stated, the key concept of the GPU implementation is to assign every single model "element" (i.e. neurons and synapses) to a different thread, meaning that each CUDA thread shall perform all necessary calculations for updating the variables of the specific struct of its corresponding element. The implemented kernels shall now be analyzed in the order of their execution in the simulation, same as the sequence of the original function calls in every timestep.

- The first kernel to be presented is the **SolveNeurons** kernel, which was also the simplest due to the lack of data dependencies other than a one-by-one connection between the *neurons* struct and the *SpikeArray* array. As already shown, due to these data structures' design, the same thread responsible for position *i* of *neurons* also handles the corresponding element *i* of *SpikeArray*. The SolveNeurons function encapsulates all commands within a loop that iterates the AdEx neurons, updates their variables according to the solution of the AdEx model's differential equations and, in case of a spike, resets the firing neurons and sets the matching *SpikeArray* position to 1. Since these two data structures are both logically and code-wise one-dimensional, the identical dimensions of the CUDA grid spawned on kernel launch lead to the complete removal of the loop and the replacement of array subscripts with the thread ID, ensuring correct functionality due to the fact that **all** array elements are independent of each other:

```
__global__
void SolveNeurons(Neuron* neurons, int N, int *SpikeArray){
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  double _vm, _vt, _x;
  if (id < N){
    _vm = (gL*(EL-neurons[id].vm)+gL*DeltaT*exp((neurons[id].vm-neurons[
    id].vt)/DeltaT)+neurons[id].I-neurons[id].x)/C;
    _vt = -(neurons[id].vt-vtrest)/tauvt;
    _x = (c*(neurons[id].vm-EL)-neurons[id].x)/tauw;
    neurons[id].vm += _vm * defaultclock_dt;
    neurons[id].vt += _vt * defaultclock_dt;
    neurons[id].x += _x * defaultclock_dt;
    if(neurons[id].vm > neurons[id].vt){
      /* reset neuron */
      neurons[id].vm = Vr;
      neurons[id].x += b;
      neurons[id].vt = VTmax;
      SpikeArray[id] = 1;
    }
    else SpikeArray[id] = 0;
  }
}
```

  The CUDA *__global__* keyword indicates that the function it accompanies runs on the device and is called from the host code (CUDA kernels). For GPUs with compute capability of 3.5 or higher, kernel calls can also be issued from within kernels, useful for dynamic parallelism. Another note is that kernels do not recognize global variables declared in CPU code, therefore every variable must be passed to the kernel as a parameter. This detail has been omitted from the presented kernel codes for pure simplicity reasons.

- Next is the **UpdateSynapses_pre** kernel. In this kernel's design a decision was needed to be made as behaviour correspondence to the original Brian simulator was weighed against significant performance increase. The original *UpdateSynapses_pre* function is also characterized by a loop that iterates the synapses' structs and updates all relevant variables. However, there is a subsequent loop that iterates the synapses **by columns** and propagates each synapse's *target_I* current -generated by the pre-synaptic neuron- to the post-synaptic neuron.

```
for ( j = 0; j < N_Group_T; j++){
  for ( i = 0; i < N_S + N_Group_S; i++){
    if (Synapses[i][j].conn && SpikeArray[i+N_Group_T−N_Group_S]){
      neurons[j].I = Synapses[i][j].target_I;
    }
  }
}
```

While this seemed to lead to a straightforward parallel implementation, the following case led to the dilemma: when more than one pre-synaptic neurons connected to the same post-synaptic neuron fire simultaneously, each synapse's *target_I* overwrites the value already written to the post-synaptic neuron's variable. Due to the iteration **by columns**, the final value passed is of the neuron corresponding to the largest thread ID among those of firing pre-synaptic neurons, that is the last neuron that fired if we iterated the *SpikeArray* struct **by rows**. Since there is no CUDA function to impose global thread synchronization inside a kernel, replicating this behavior requires exiting the kernel before this last loop, transferring the *target_I* data and executing it in the CPU code, passing the updated *neurons* struct back to the device for further calculations. However, among CUDA API calls, the greatest time penalty is generated when using *cudaMemcpy* (and its derivatives) to transfer data between device and host. Moreover, the order the current is passed is strictly a Brian source code limitation, as there is no Brian support to define which pre-synaptic neuron prevails on simultaneous spiking. Neither the paper the original experiment was based on nor the models of AdEx neurons or STDP-characterized synapses provide any relevant information. Finally, there is no problem having multiple threads writing a single (shared or global) memory location in CUDA, even "simultaneously" i.e. from the same line of code, as long as there are no read-write race conditions. Hence, the chosen approach was to simply allow threads to update the neuron variable in parallel, having no control over the order they will access that field whatsoever, resulting in a significant performance gain.

```
__global__
void UpdateSynapses_pre(Synapse* Synapses, Neuron* neurons, int* SpikeArray){
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (SpikeArray[id/N_Group_T + N_Group_T - N_Group_S] > 0){
        if (Synapses[id].conn){
            /* update variables */
            Synapses[id].FFp = Synapses[id].FFp * exp(-(-Synapses[id].lastupdate + t)/tau_FFp);
            Synapses[id].FBn = Synapses[id].FBn * exp(-(-Synapses[id].lastupdate + t)/tau_FBn);
            Synapses[id].u = Synapses[id].U + (-Synapses[id].U + Synapses[id].u) * exp(-(-Synapses[id].lastupdate + t)/tau_u);
            Synapses[id].FBp = Synapses[id].FBp * exp(-(-Synapses[id].lastupdate + t)/tau_FBp);
            Synapses[id].R = (Synapses[id].R - 1) * exp(-(-Synapses[id].lastupdate + t)/tau_r) + 1;
            Synapses[id].target_I = s * Synapses[id].A * Synapses[id].R * Synapses[id].u;
            Synapses[id].U = Synapses[id].U + etaU * (-AFBn * Synapses[id].FBn * Synapses[id].FBp + AFBp * Synapses[id].FBp * Synapses[id].FFp);
            if (Synapses[id].U < Umin) Synapses[id].U = Umin;
            else if (Synapses[id].U > Umax) Synapses[id].U = Umax;
            Synapses[id].w = Synapses[id].U * Synapses[id].A;
            Synapses[id].FFp += 1;
            Synapses[id].R -= Synapses[id].R * Synapses[id].u;
            Synapses[id].u += Synapses[id].U * (1 - Synapses[id].u);
            Synapses[id].lastupdate = t;
            /* pass current to neurons */
            neurons[id%N_Group_T + N_Group_T - N_Group_S].I Synapses[id].target_I;
        }
    }
}
```

Once again, the first double loop is essentially replaced with thread ID retrieval.

- Last, the **UpdateSynapses_post** kernel. As mentioned in section 4.2, the *UpdateSynapses_post* function consists of two loops iterating all synapse structs by columns, which makes little difference in modifying the function to run in parallel. However, the calculation of the mean value of certain synaptic variables between those loops proved to be a challenge. This value, common to all synapses hence every thread, is necessary for the operations performed in the second loop, so it has to be assured that no thread has moved on to the second loop before the mean variable is set at the right value. Unfortunately, the CUDA programming model specializes on solving problems by breaking them down onto blocks, and thread synchronization can be done within a block using CUDA's *__syncthreads* function. However, global thread synchronization within a CUDA kernel can only be achieved with methods that serialize execution (e.g. mutex locks). Thus, the most straightforward solution was to split the function into 2 seperate kernels responsible for running each nested for-loop in parallel, allowing the host (CPU) to handle the mean calculation in between by passing all the necessary data through the *testvar* array. Since kernels are automatically executed sequentially if launched in the default stream, as happens in the entirety of this thesis' related code, there is no need to explicitly synchronize the device to ensure the mean is calculated.

```
__global__
void UpdateSynapses_post_Part1( Synapse *Synapses, int* SpikeArray,
    double *testvar){
     int id = blockIdx.x*blockDim.x + threadIdx.x;
     if (id < (N_S + N_Group_S) * N_Group_T)
     {
          if (SpikeArray[id%N_Group_T + N_Group_T - N_Group_S] > 0){
               if (Synapses[id].conn){
                    Synapses[id].FFp = Synapses[id].FFp * exp(-(-Synapses[id
    ].lastupdate + t)/tau_FFp);
                    Synapses[id].FBn = Synapses[id].FBn * exp(-(-Synapses[id
    ].lastupdate + t)/tau_FBn);
                    Synapses[id].u = Synapses[id].U + (-Synapses[id].U +
    Synapses[id].u) * exp(-(-Synapses[id].lastupdate + t)/tau_u);
                    Synapses[id].FBp = Synapses[id].FBp * exp(-(-Synapses[id
    ].lastupdate + t)/tau_FBp);
                    Synapses[id].R = (Synapses[id].R - 1) * exp(-(-Synapses[
    id].lastupdate + t)/tau_r) + 1;
                    Synapses[id].A = Synapses[id].A + etaA * (AFFp *
    Synapses[id].FFp * Synapses[id].FBn);
                    /* save values for mean calculation */
                    testvar[id] = AFFp * Synapses[id].FFp * Synapses[id].FBn
    ;
               }
          }
     }
}



__global__
void UpdateSynapses_post_Part2(Synapse *Synapses, int* SpikeArray,
    double mean){
     int id = blockIdx.x*blockDim.x + threadIdx.x;
     if (id < (N_S + N_Group_S) * N_Group_T)
     {
          if (SpikeArray[id%N_Group_T] > 0){
               if (Synapses[id].conn){
                    Synapses[id].A = Synapses[id].A - etaA * 0.5 * mean;
                    if (Synapses[id].A < Amin) Synapses[id].A = Amin;
                    else if (Synapses[id].A > Amax) Synapses[id].A = Amax;

                 Synapses[id].w = Synapses[id].U * Synapses[id].A;
                 Synapses[id].FBp += 1;
                 Synapses[id].FBn += 1;
                 Synapses[id].lastupdate = t;
               }
          }
     }
}
```

5.1.3.2 CPU code

After the implementation of the CUDA kernels responsible to perform (most of) the work of the three main simulator functions in parallel, it is time to present the code executed on the host CPU that controls the flow of the simulation. The CPU code is:

```
Initialize_host_structures();
Initialize_device_structures();
for( t = 0; t < timesteps; t++){

    SolveNeurons<<<gridsize1,blocksize1>>>(d_neurons, N_Group_T, d_SpikeArray);

    /* spike generation */
    for(i = 0; i < M ; i++) if (i%2 == 0) SpikeArray[i] = 1;
    cudaMemcpy(d_SpikeArray, SpikeArray, sizeof(int) * (N_Group_T+N_S), cudaMemcpyHostToDevice);

    UpdateSynapses_pre<<<gridsize2,blocksize2>>>(d_syn, d_neurons, d_SpikeArray);

    UpdateSynapses_post_Part1<<<gridsize2,blocksize2>>>(d_syn, d_SpikeArray, d_testvar);

    /* mean calculation */
    cudaMemcpy(testvar, d_testvar, sizeof(double) * (N_S+N_Group_S) * (N_Group_T), cudaMemcpyDeviceToHost);
    for ( int k = 0; k < (N_S+N_Group_S) * N_Group_T; k++){
        if (syn[k].conn && SpikeArray[k%N_Group_T]){
            mean += testvar[k];
            num++;
        }
    }
    if (num > 0) mean = mean / num;

    UpdateSynapses_post_Part2<<<gridsize2,blocksize2>>>(d_syn, d_SpikeArray, mean);

    /* print variables of interest */
    if (verbose){
        cudaMemcpy(syn, d_syn, sizeof(Synapse) * (N_S+N_Group_S) * (N_Group_T), cudaMemcpyDeviceToHost);
        cudaMemcpy(SpikeArray, d_SpikeArray, sizeof(int) * (N_Group_T+N_S), cudaMemcpyDeviceToHost);
        cudaMemcpy(neurons, d_neurons, sizeof(Neuron) * N_Group_T, cudaMemcpyDeviceToHost);
        output_sim(syn, SpikeArray, neurons, t);
    }
}
```

Due to the models' nature, in the case of **MxM** experiments user-defined spikes need to be hard coded and passed to the device on every timestep. As of the time this thesis is composed, there is limited functionality related to users choosing spike properties, so they need to be explicitly coded for the CPU before compilation. Moreover, the option to output a plethora of significant variables on every timestep is available to the user, in addition to a standard simulation that outputs its results on finish.

5.1.4 Large Scale Experiments Support

As already mentioned, the simulator is an application limited by available memory, meaning that there are strict limitations to the amount of neurons and synapses that can be allocated and used on the device on a single experiment at once without surpassing device memory limits. The largest experiment carried out successfully by operating on the entirety of data simultaneously on the GPU comprised of 12500 interconnected AdEx neurons (**MxM**). Larger experiments using the already presented code led to a *not enough memory* error and as a result two different approaches were implemented for larger scale experiments, supporting single and dual GPU nodes correspondingly.

5.1.4.1 Single GPU

Since there is a limit in the amount of data that can be transferred and operated on by the device, all data dependencies needed to be carefully analyzed in order to ensure correct execution and output of “larger” simulations. The fact that for each synapse to be updated, its pre-synaptic or post-synaptic neuron needs to have fired on that timestep, leads to splitting the data to be elaborated at once based on a specific pattern and executing the original, unmodified kernels on independent portions of the data structures. Considering the following branch condition in *UpdateSynapses_pre*:

```
if (SpikeArray[tidx/N_Group_T + N_Group_T - N_Group_S]){
    /* operate on synapses[tidx] */
}
```

Passing a part of the synaptic structure of size **ROWS * COLUMNS** to the device requires to additionally pass a size **ROWS/(N_Group_T + N_Group_T - N_Group_S)** portion of *SpikeArray* that contains all pre-synaptic neurons that correspond to the synapses updated by the device. Following the same logic, due to the branch condition of *UpdateSynapses_post*, passing a **ROWS%(N_Group_T + N_Group_T - N_Group_S)** portion is necessary. In the *SolveNeurons* case, as *SpikeArray* is actually a spiking log of all neurons, there is a one-to-one correspondence between the array and the *neurons* structure and the same partition of both data structures must be passed to the device. Finally, the *testvar* array complies absolutely with the synaptic structure's partitioning. Thus, adjusting the simulator to executing larger experiments with practically no size limit is fairly easy from a development standpoint, as instead of calling each kernel once, a loop is implemented for each kernel where on each iteration the ensemble of structure partitions to be operated on get passed to the device, the kernel is executed and the data is transferred back to the host, as the next iteration will overwrite device data.

```
thrust::device_ptr<float> testvar_ptr = thrust::device_pointer_cast(d_testvar
    );
/* CPU code executed on each timestep */
for (i = 0; i < N_Group_T/neuron_tempsize; i++){
    cudaMemcpy(d_neurons, neurons + i * neuron_tempsize, sizeof(Neuron) *
    neuron_tempsize, cudaMemcpyHostToDevice);
    SolveNeurons<<<gridsize1,blocksize1>>>(d_neurons, N_Group_T, d_SpikeArray
    , neuron_tempsize);
    cudaMemcpy(neurons + i * neuron_tempsize, d_neurons, sizeof(Neuron) *
    neuron_tempsize, cudaMemcpyDeviceToHost);
}

for (i = 0; i < (N_S + N_Group_T)/(spike_tempsize); i++)
    cudaMemcpy(d_SpikeArray, SpikeArray, sizeof(int) * (N_Group_Ttemp +
    N_Stemp), cudaMemcpyHostToDevice);

for (i = 0; i < (N_S + N_Group_S)*N_Group_T/synapses_tempsize; i++){
    cudaMemcpy(d_syn, syn + i * synapses_tempsize, sizeof(Synapse) *
    synapses_tempsize, cudaMemcpyHostToDevice);
    UpdateSynapses_pre<<<gridsize2,blocksize2>>>(d_syn, d_neurons, N_S,
    N_Group_S, N_Group_T, d_SpikeArray, synapses_tempsize);
    cudaMemcpy(syn + i * synapses_tempsize, d_syn, sizeof(Synapse) *
    synapses_tempsize, cudaMemcpyDeviceToHost);
}

for (i = (N_S + N_Group_S)*N_Group_T/synapses_tempsize; i >= 0; i--){
    if (i < (N_S + N_Group_S)*N_Group_T/synapses_tempsize){
        cudaMemcpy(d_syn, syn + i * synapses_tempsize, sizeof(Synapse) *
    synapses_tempsize, cudaMemcpyHostToDevice);
    }
    UpdateSynapses_post_Part1<<<gridsize2,blocksize2>>>(d_syn, N_S, N_Group_S
    , N_Group_T, d_SpikeArray, d_testvar, synapses_tempsize);
    cudaMemcpy(syn + i * synapses_tempsize, d_syn, sizeof(Synapse) *
    synapses_tempsize, cudaMemcpyDeviceToHost);
    cudaMemcpy(testvar + i * synapses_tempsize, d_testvar, sizeof(float) *
    synapses_tempsize, cudaMemcpyDeviceToHost);
}

/* mean calculation, save result to average */
for (i = 0; i < (N_S + N_Group_S)*N_Group_T/synapses_tempsize; i++){
    mean = thrust::reduce(testvar_ptr, testvar_ptr + (synapses_tempsize));
    mean /= ((N_S+N_Group_S) * (N_Group_T));
    average += mean;
    cudaMemcpy(d_testvar, testvar + i * synapses_tempsize, sizeof(float) *
    synapses_tempsize, cudaMemcpyHostToDevice);
}

for (i = 0; i < (N_S + N_Group_S)*N_Group_T/synapses_tempsize; i++){
    if (i > 0){
        cudaMemcpy(d_syn, syn + i * synapses_tempsize, sizeof(Synapse) *
    synapses_tempsize, cudaMemcpyHostToDevice);
    }
    UpdateSynapses_post_Part2<<<gridsize2,blocksize2>>>(d_syn, N_S, N_Group_S
    , N_Group_T, d_SpikeArray, average, synapses_tempsize);
    cudaMemcpy(syn + i * synapses_tempsize, d_syn, sizeof(Synapse) *
    synapses_tempsize , cudaMemcpyDeviceToHost);
}
```

While this is indeed a valid method to provide theoretically unlimited scaling for the simulator's experiments, there is a vast performance cost caused by the overhead of the mandatory memory transfers of the structures' partitions between device and host on every timestep. However, there is still a significant performance increase compared to the original linear version, to be presented on the next chapter. Partitions are transferred by looping over parts of the data structures and constantly overwriting the device allocated structures in order to perform necessary calculations. Thus, by setting the loops on a first-to-last then last-to-first then first-to-last chain, two memory copies are omitted on every timestep by operating on data that were written on device buffers in the previous loop, on the first iteration of each loop.

5.1.4.2 Dual GPU

Following the principles explained in the above section, a multi-GPU supported version was also implemented and tested on a node with two devices. The multi-GPU implementation offers increased memory bandwidth, as the double amount of data can be transferred to device memory spending almost the same time, and the neuron and synaptic models' nature permits independent concurrent execution of different chunks of the data on separate devices, offering even greater performance than that of the version shown in 5.1.4.1. The greatest performance gain compared to using a single GPU is obviously achieved when running experiments of a size nearing both devices' limit of all-at-once execution, so that maximum bandwidth is achieved and memory transfers are minimized. Serving simplicity and clarity purposes, a relevant example of a non-parametrized code running a simulation of 20000 neurons is shown below, noting that all data structures first need to be separately allocated and copied to each GPU:

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
thrust::device_ptr<float> testvar_ptr0 = thrust::device_pointer_cast(
    d_testvar0);
thrust::device_ptr<float> testvar_ptr1 = thrust::device_pointer_cast(
    d_testvar1);

for( t = 0; t < timesteps; t++){
     gpuErrchk(cudaSetDevice(0));
     SolveNeurons<<<gridsize1,blocksize1>>>(d_neurons0, N_Group_T,
    d_SpikeArray0, N_Group_Tpergpu);
     gpuErrchk(cudaSetDevice(1));
     SolveNeurons<<<gridsize1,blocksize1>>>(d_neurons1, N_Group_T,
    d_SpikeArray1, N_Group_Tpergpu);

     for(i = 0; i < N_Group_S ; i++) SpikeArray[i] = 1;

     gpuErrchk(cudaSetDevice(0));
     cudaMemcpyAsync(d_SpikeArray0, SpikeArray, sizeof(int) * (N_Group_Tpergpu
    +N_Spergpu), cudaMemcpyHostToDevice);
     gpuErrchk(cudaSetDevice(1));
     cudaMemcpyAsync(d_SpikeArray1, SpikeArray, sizeof(int) * (N_Group_Tpergpu
    +N_Spergpu), cudaMemcpyHostToDevice);

     gpuErrchk(cudaSetDevice(0));
     UpdateSynapses_pre<<<gridsize2,blocksize2>>>(d_syn0, d_neurons0, N_S,
    N_Group_S, N_Group_T, d_SpikeArray0, N_Spergpu, N_Group_Spergpu,
    N_Group_Tpergpu);
     gpuErrchk(cudaSetDevice(1));
     UpdateSynapses_pre<<<gridsize2,blocksize2>>>(d_syn1, d_neurons1, N_S,
    N_Group_S, N_Group_T, d_SpikeArray1, N_Spergpu, N_Group_Spergpu,
    N_Group_Tpergpu);

     gpuErrchk(cudaSetDevice(0));
     UpdateSynapses_post_Part1<<<gridsize2,blocksize2>>>( d_syn0, N_S,
    N_Group_S, N_Group_T, d_SpikeArray0,  d_testvar0, N_Spergpu,
    N_Group_Spergpu, N_Group_Tpergpu);
     gpuErrchk(cudaSetDevice(1));
     UpdateSynapses_post_Part1<<<gridsize2,blocksize2>>>( d_syn1, N_S,
    N_Group_S, N_Group_T, d_SpikeArray1,  d_testvar1, N_Spergpu,
    N_Group_Spergpu, N_Group_Tpergpu);

     gpuErrchk(cudaSetDevice(0));
     cudaMemcpyAsync(testvar + 0 * (N_Spergpu+N_Group_Spergpu) *
    N_Group_Tpergpu, d_testvar0, sizeof(float) * (N_Spergpu+N_Group_Spergpu) *
     (N_Group_Tpergpu), cudaMemcpyDeviceToHost);
     gpuErrchk(cudaSetDevice(1));
     cudaMemcpyAsync(testvar + 1 * (N_Spergpu+N_Group_Spergpu) *
    N_Group_Tpergpu, d_testvar1, sizeof(float) * (N_Spergpu+N_Group_Spergpu) *
     (N_Group_Tpergpu), cudaMemcpyDeviceToHost);

     gpuErrchk(cudaSetDevice(0));
     mean0 = thrust::reduce(testvar_ptr0, testvar_ptr0 + ((N_S+N_Group_Spergpu
    ) * (N_Group_T)));
     mean0 /= ((N_Group_S+N_S) * N_Group_T);
     gpuErrchk(cudaSetDevice(1));
    mean1 = thrust::reduce(testvar_ptr1, testvar_ptr1 + ((N_S+N_Group_Spergpu
    ) * (N_Group_T)));
    mean1 /= ((N_Group_S+N_S) * N_Group_T);
    average = mean0 + mean1;

    gpuErrchk(cudaSetDevice(0));
    UpdateSynapses_post_Part2<<<gridsize2,blocksize2>>>(d_syn0, N_S,
   N_Group_S, N_Group_T, d_SpikeArray0, average, N_Spergpu, N_Group_Spergpu,
   N_Group_Tpergpu);
    gpuErrchk(cudaSetDevice(1));
    UpdateSynapses_post_Part2<<<gridsize2,blocksize2>>>(d_syn1, N_S,
   N_Group_S, N_Group_T, d_SpikeArray1, average, N_Spergpu, N_Group_Spergpu,
   N_Group_Tpergpu);
}
```

A notable part of the code is the usage of *cudaMemcpyAsync* for memory transfers. It is a *cudaMemcpy* variant that is asynchronous with respect to the host, so the call may return before the copy is complete, enabling memory transfers to and from different devices and even kernel calls to execute in parallel. Moreover, *gpuErrchk* is a standard error checking practice for debugging purposes, which is left on the device setting calls even in the final code as it is a critical CUDA call and there are no viable slowdowns from this addition.

5.1.5 CUDA Optimizations

5.1.5.1 Float vs Double Datatype

Up until the point of a functioning prototype, GPU-accelerated simulator design was based on achieving zero error compared to the original. However, this consideration leads to all data processed by the GPU being represented by 64-bit doubles, which has the following consequences: First of all, due to double precision arithmetics requiring 64-bit variables (on most devices), an excessive amount of memory is consumed by simulation data compared to a single precision (*float*) implementation. The main cause is the memory dominant array of synaptic structs, limiting scaling possibilities unless speedup is sacrificed for loop-based partial memory transfers to fit the entirety of data in the GPU. Moreover, since the models used for this simulator render it a memory-limited application, there is a vast difference in the number of bytes to be read or written in a kernel which are doubled when using double precision, resulting in a corresponding performance penalty. Taking into consideration the additional fact that mean calculation is performed on the CPU, resulting in device-to-host memory transfers - which are the most costly CUDA operations from a time perspective - of a considerably large array on every timestep, a significant amount of time is sacrificed on the altar of absolute zero error. Therefore, an implementation using mostly *floats* to represent simulation data was chosen. The new implementation is identical to its predecessor except for the datatype of the defined synaptic and neuron variables, both global and struct-restricted. The only variables represented as *double* are the one related to the summation of the values needed to calculate *mean* (for overflow prevention) and those related to simulation time representation, due to arithmetic accuracy issues.

In order to approve this modification, certain output variables of both neurons and synapses were analyzed in order to assess output inaccuracy compared to the original when using single precision variables. As maximum observed error for any output variables never surpassed **1.00E-13**, the *float* version was accepted to replace the original and offered vast performance increase that reached almost double acceleration rates.

5.1.5.2 Kernel Merge

Another explored optimization was to slightly alter the design of the CUDA kernels, so as to diminish kernel launch overhead and further accelerate the simulator. Specifically, the only operation taking place on the CPU between the *SolveNeurons* and *UpdateSynapses_pre* kernels is explicitly setting *SpikeArray*'s elements corresponding to firing neurons to 1. A modification was attempted by assigning spike setting to CUDA threads and merging the two kernels into a single, aggregate kernel launched with thread configuration analogous to the size of the synaptic AoS:

```
__global__
void SolveNeurons_UpdateSynapses_pre_merged(Neuron* neurons, int N, int *
    SpikeArray, Synapse* Synapses){
     int id = blockIdx.x*blockDim.x + threadIdx.x;
     /* SolveNeurons operations */
     /* ... */

     /* AdEx Neurons fire */
     if (id < N_Group_S) SpikeArray[i] = 1;

     /* UpdateSynapses_pre operations */
     /* ... */
}
```

However, even though a memory transfer operation is also eliminated by this modification as spikes do not need to be passed to the device anymore, the merged kernel proved to perform equally if not slightly worse than the two seperate ones. An equivalent performance decrease was presented with the merge of *UpdateSynapses_pre* and *UpdateSynapses_post_Part1*. A possible explanation would be the exceeding amount of branches combined as a result of the merge, considering that the original two kernels were meant to operate on separate data structures (*neurons* and *Synapses*). The absence of a practical speedup along with the preference of defining the firing neurons in the CPU code to facilitate future code modifications relevant to spike-related user request resulted in discarding the merged kernel and moving forward with the original design.

5.1.5.3 Parallel Mean Calculation

The most time consuming operation of the simulation is the calculation of the mean value of certain synaptic variables throughout all synapses connected to spiking neurons on the CPU. As already mentioned, this operation breaks down to passing the *testvar* array back to the host and performing a serial summation by going through every struct of the synaptic array by rows, the code of which is under the comment "mean calculation" in 5.1.3.2, and dividing by the number of the corresponding synapses. Therefore, methods to accelerate this code segment by also running it in parallel were investigated. In parallel programming, reduction operations are those that reduce a collection of values to a single value. Summing the elements of an array is a common example of a reduction operation, which can be explained as follows:

- Assuming N as the number of the elements in an array, N/2 threads are spawned, one thread for each two (sequential) elements.
- Each thread computes the sum of the corresponding two elements, storing the result at the position of the first one.
- Iteratively, each step:
  - the number of threads is halved.
  - each remaining thread is now responsible for its summed element and that of the adjacent, removed one.
  - the step size between the corresponding two elements is doubled.
- When one thread is left, the reduction result is stored in the first element of the array.

Part of the CUDA toolkit are NVIDIA GPU-accelerated libraries that provide highly-optimized functions for compute-intensive applications in different areas. One such GPU-accelerated library is **Thrust**, a library of parallel algorithms and data structures that provides a flexible, high-level interface for GPU programming by providing access to its algorithms through the standard template interface. [Thr] Thrust automatically manages low-level functionality like memory access and allocation, allowing the user to focus on algorithm development. Below is the code responsible for *mean* calculation in the case of all AdEx Neurons **constantly spiking** and **100% connectivity**:

```
thrust::device_ptr<float> testvar_ptr = thrust::device_pointer_cast(d_testvar
    );
mean = thrust::reduce(testvar_ptr, testvar_ptr + ((N_S+N_Group_S) * (
    N_Group_T)));
mean /= ((N_Group_S+N_S) * N_Group_T);
```

After including the necessary header files, the only requirement of the library is to use the *thrust* namespace when calling any function. The code used for the mean calculation shall be explained. The first line defines a thrust device pointer called *testvar_ptr* by casting the already set *d_testvar* pointer into a thrust-compatible. Then, the *reduce* function is called with two arguments which are the beginning and the end of the newly defined pointer to set the memory space the reduction will be applied. Normally, a third argument can be the binary operator that defines the kind of reduction to be performed. However, the default operator is the *plus sign*, hence there is no need to explicitly set it. Finally, the summed value is divided by the synaptic quantity to receive the desired mean value.

As expected, the replacement of 64-bit *doubles* with 32-bit single-precision *floats* led to an impressive total performance increase even with the elimination of the data transfer necessary for originally calculating the mean value on the CPU, as operations such as iterating through a grandiose data structure and transferring large amounts of memory (i.e. *SpikeArray*) on a timestep basis gain significant speedups when decreasing element size. However, the reduction approach requires noting a minor detail regarding simulator parameterization. The advantage of using a reduction method is the entirely parallel execution of the desired operation, avoiding serial iteration through the synaptic data. In order to actually avoid serial access, the number of both spiking neurons and connected synapses needs to be known from the beginning of the simulation, so as to calculate the divisor in line 3 of the above code, which is normally a variable set during program execution. This is obviously solved by a simple extra parameter set by the user before execution. In the **NxM** case, due to the possible occurrence of additional spikes on the AdEx Neurons, only part of the data structure is iterated to count additional significant synapses, causing a negligible time penalty.

5.1.5.4 AoS vs SoA

One of the most important targets during CUDA optimizations is to achieve memory coalescence. During execution, apart from grids and blocks, there is a finer grouping of threads into warps of 32 threads. Multiprocessors on the GPU execute instructions for each warp in SIMD (Single Instruction Multiple Data) fashion. Threads in the same warp execute in full parallel, that is executing the same instruction simultaneously on every step. Global memory loads and stores by threads of a warp are coalesced by the device into as few as one transaction when certain access requirements are met. Thus, when parallel threads running the same instruction access to consecutive locations in the global memory, the most favorable access pattern is achieved. On the contrary, memory accesses to distant locations by adjacent threads lead to a significant performance penalty because of hardware constraints.

A common optimization technique related to memory coalescing is turning any major array of structs (AoS) into a struct of arrays (SoA). As all threads of a single warp execute the same instruction at the same time, in the simplest case where an instruction refers to modification of one variable of the struct it is rather obvious that by utilizing a SoA every thread will access consecutive elements of the corresponding array of the variable, achieving optimal coalescence. Therefore, a new version of the simulator was implemented that showcased a synaptic struct of arrays instead of the original AoS. The neurons struct remained as is, since the synapse-focused kernels are dominant in terms of time consumption and there is no adequate potential for increased performance from a neuron perspective. The only differences in this new version's code are the declaration of the synaptic structure, where each synaptic variable is now declared as a pointer to this data type, and relevant memory allocation and management.

```
/* host structure */
Synapse *syn = (Synapse *)malloc(sizeof(Synapse));
syn->conn = (int *)malloc(sizeof(int) * (N_S+N_Group_S) * (N_Group_T));
/* ... */
syn->target_I = (float *)malloc(sizeof(float) * (N_S+N_Group_S) * (N_Group_T)
    );

/* host copy of the device structure */
h_syn = (Synapse*)malloc(sizeof(Synapse));
cudaMalloc(&(h_syn->conn), sizeof(int) * (N_S+N_Group_S) * (N_Group_T));
/* ... */
cudaMalloc(&(h_syn->target_I), sizeof(float) * (N_S+N_Group_S) * (N_Group_T))
    ;

/* transfer data to device */
cudaMemcpy(h_syn->conn, syn->conn, sizeof(int) * (N_S+N_Group_S) * (N_Group_T
    ), cudaMemcpyHostToDevice);
/* ... */
cudaMemcpy(h_syn->target_I, syn->target_I, sizeof(float) * (N_S+N_Group_S) *
    (N_Group_T), cudaMemcpyHostToDevice);

/* device structure */
cudaMalloc(&d_syn, sizeof(Synapse));
cudaMemcpy(d_syn, h_syn, sizeof(Synapse), cudaMemcpyHostToDevice);
```

As shown above, in order to declare and use a structure that contains pointers to other data structures on the device, first a copy of that structure needs to be declared and built on the host and then copied to the device copy that is allocated exclusively on the device (commonly termed "deep copy"). The SoA approach led to a rather insignificant speedup. While not absolute, a possible explanation for this similarity in execution time between SoA and AoS versions is the fact that most kernel instructions contain read or write operations on different fields of struct *Synapse*. Considering the following instruction as an example:

```
Synapses[id].FFp = Synapses[id].FFp * exp(-(-Synapses[id].lastupdate + t)/
    tau_FFp);
```

It is evident that both *FFp* and *lastupdate* values of the struct are loaded. Therefore, while a SoA approach may lead threads to load their *FFp* values from adjacent memory positions, the *lastupdate* values have been led to a further distance memory-wise, apparently obstructing the opportunity for a significant speedup presented by memory coalescence.

5.1.5.5 Shared Memory

Since a CUDA GPU's shared memory is located on-chip, it offers significant speed advantages related to local and global memory. This makes the use of shared memory a popular optimization choice for most algorithms running on GPUs. The impact of integrating shared memory in the simulator was investigated for both AoS and SoA versions. The code responsible for shared memory operations has no logical difference between the two, hence the SoA version's code for *UpdateSynapses_pre* is presented below, noting that there is no necessary modification of the CPU code at all:

```
__global__
void UpdateSynapses_pre(Synapse* Synapses, Neuron* neurons, int* SpikeArray){
    __shared__ float sharedFFp[BLOCKSIZE];
    __shared__ float sharedFBn[BLOCKSIZE];
    /* ... */
  __shared__ float sharedlastupdate[BLOCKSIZE];

  int tidx = threadIdx.x;                          /* local indexing */
  int id = blockIdx.x * blockDim.x + threadIdx.x;  /* global indexing */4

  if (SpikeArray[id/N_Group_T + N_Group_T - N_Group_S] > 0){
      if (Synapses->conn[id]){
          /* move data to shared memory */
          sharedFFp[tidx] = Synapses->FFp[id];
      sharedFBn[tidx] = Synapses->FBn[id];
      /* ... */
      sharedlastupdate[tidx] = Synapses->lastupdate[id];

      /* synaptic operations */

      /* save data back */
      Synapses->FFp[i] = sharedFFp[tidx];
      Synapses->FBn[i] = sharedFBn[tidx];
      /* ... */
      Synapses->lastupdate[i] = sharedlastupdate[tidx];
    }
  }
}
```

In the simulator version using a synaptic struct of arrays, an array corresponding to each array of the struct is declared with the *__shared__* keyword, instructing allocation of shared memory. As shared memory is accessible from all the threads of a thread block, each array is given the size of the number of threads within a block. In order to accelerate memory access, threads first store data in shared memory, costing one global memory read per access, proceed to execute their operations by accessing the block-shared arrays and finally store back the results to global memory, costing one global memory write per variable per thread. Since each thread is responsible for only a single synapse, there was little room available for performance gains due to minimal data re-use, and in the AoS version execution time for most experiments was almost identical to the pure global memory implementation. However, performance in the presented version presented an adequate increase that scaled along with experiment size.

The difference in the effect in execution speed is attributed to the elimination of bank conflicts in the synaptic struct of arrays version. To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules called banks that can be accessed simultaneously. Therefore, any memory load or store of **n** addresses that spans **n** distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is **n** times as high as the bandwidth of a single bank. If multiple addresses of a memory request map to the same memory bank, the accesses are serialized. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests. This leads to the deduction that coalesced memory access is critical to achieving optimal shared memory performance, and optimal coalescence in this model's case proves to be achieved with a synaptic struct of arrays.

An important point to be made regarding kernel code is that all shared memory accesses are performed only in the case the branch conditions regarding spiking and connectivity are met. While this presents a little performance cost due to the struct's *conn* variable being accessed in global memory in order to determine the branching outcome, it also ensures that there is no unnecessary workload on CUDA threads that correspond to an unconnected or non-firing neuron. Last but not least, the exact same modification was applied on *UpdateSynapses_post_Part1*, while the other kernels remained intact, as their execution times make up for a trivial percentage of total program runtime.

Since the shared memory optimization showed the best results on the Struct-of-Arrays version, the latter was also modified to address larger experiment scaling. Concerning running the simulator on a single device, even though the CPU code follows identical logic compared to the version presented in 5.1.4.1, in this version each array of the struct needs to be copied separately on every timestep instead of passing the entirety of *syn* and *d_syn* at once, leading to both -negligible- performance penalty due to the excessive *cudaMemcpy* calls and extensive repetitive code segments.

5.1.5.6 Other Attempts

The research process included several additional optimization attempts following different CUDA concepts presented below:

- Since the models supported by the simulator encapsulate a vast number of constant parameters, the choice was made to store all said parameters in the device's **constant memory**. Modern devices have a constant memory space of 64KB. This memory space is cached, meaning that a read from constant memory costs one memory read from device memory only on a cache miss; otherwise, it just costs one read from the constant cache. In general, there are two reasons why reading from the 64KB of constant memory can save bandwidth over standard reads of global memory: First, a single read from constant memory can be broadcast to other "nearby" threads, effectively saving up to 15 reads. Second, since constant memory is cached, consecutive reads of the same address will not incur any additional memory traffic. In the simulator's case, the first reason was important enough to attempt this approach. First, an array with the keyword *__constant __* is declared, and then all constant values are passed to the device array using the *cudaMemcpyToSymbol* function which is *cudaMemcpy*'s equivalent for data transfers between host and constant memory on the GPU. This modification did not cause performance decrease, though it didn't offer any kind of speedup either.
- Another technique used with the aim of gaining speedup was **loop unrolling**. A

common parallel design is to create a kernel where each loop iteration is mapped to one GPU thread, as happens in the simulator's code. The logic behind loop unrolling is to theoretically unroll the loop **X** times before writing the GPU code, and then map it to the GPU so that each thread is responsible for **X** more work and **X** times fewer threads are spawned, **X** usually being a power of 2. There are numerous published occasions on which such a modification resulted in a large speedup, but the STDP synaptic model proved to be an absolute obstacle to performance increase in this case. The reason the simulator actually slowed down in most tests is excessive branching on every loop, since unrolling a loop two times leads to having to branch twice as much on an already branch-heavy kernel, which is detrimental to GPU performance. A warp executes one common instruction at a time, so full efficiency is only realized when all threads of a warp agree on their execution path, otherwise each branch path is executed serially with a corresponding performance penalty. Adding such a technique to the simulator would hence actually result in a slowdown for every spike or connection configuration that doesn't match the warps layout. Loop unrolling was also tested at the same time with utilization of shared memory to enable fastest memory reads for each thread, though there were no positive results either.

- Modern GPUs running latest CUDA versions and adequate compute architectures offer the advantage of CUDA's **Unified Memory** model. In a typical PC or cluster node today, the memories of the CPU and GPU are physically distinct and separated by the PCI-Express bus. Data that is shared between the CPU and GPU must be allocated in both memories, and explicitly copied between them by the programmer. Since CUDA 6, Unified Memory creates a pool of managed memory that is shared between the CPU and GPU and requires a single pointer to be accessed by any of them. The system automatically migrates data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU. The only prerequisite code-wise is to allocate memory space using *cudaMallocManaged* (same parameters as *cudaMalloc*). This ability to automatically migrate data at the level of individual pages between host and device memory enables the elimination of "deep copies" mentioned in 5.1.5.4 , resulting in both significantly clearer code and minimal performance advantages. The reason this model was researched apart from development facilitation was the ability of devices of SM architecture 6.X or newer to oversubscribe device memory in order to virtually fit more data than allowed by the GPU's physical constraints. However, the GPUs used for experimentation for this thesis' purposes did not cover the prerequisites for memory oversubscription, meaning that memory allocation for the simulator's data structures was actually limited for both host and device by device constraints, eliminating experiment scaling support. Consequently, utilizing unified memory for the simulator was deemed needless.

- Finally, Host (CPU) data allocations are pageable by default. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or “pinned”, host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory. The cost of the transfer between pageable and pinned host arrays can be avoided by directly allocating host arrays in **Pinned host memory**. Therefore, *SpikeArray* was allocated in pinned memory by invoking the *cudaMallocHost* function with the same arguments as the classic *cudaMalloc*, as it is passed to device memory on every timestep of the simulation. This modification however also proved indifferent, as performance was identical to allocating the array in pageable memory even on the largest experiments.

5.2 OpenMP implementation

The models’ nature of loop-centric simulations characterized by absolute dependency between timesteps and minimal dependency between simulation elements on a single timestep greatly facilitates the basic implementation of another accelerated version of the simulator compatible with the C language. Useful for systems with no/weak GPU as well as a comparison tool to improve the CUDA version, the OpenMP API was utilized, providing adequate results for a secondary tool. In this implementation, data structures are left intact and there is no addition or modification to the code of the program’s *main* function, leaving it identical to the original C version. The only difference occurs with the addition of specific code lines in the three model functions:

```
void SolveNeurons(Neuron* neurons, int N, int *SpikeArray) {
    #pragma omp parallel for shared(neurons) private(_vm, _vt, _x)
    for(int i = 0; i < N; i++){
        /* update variables according to solution using _vm, _vt, _x */
        if(neurons[i].vm > neurons[i].vt){
            resetNeuron(&neurons[i]);
            SpikeArray[i] = 1;
        }
        else SpikeArray[i] = 0;
    }
}


void UpdateSynapses_pre(Synapse** Synapses, Neuron* neurons, int* SpikeArray)
 {
    #pragma omp parallel for collapse(2) shared(Synapses)
    for (int i = 0; i < N_S + N_Group_S; i++){
        if (SpikeArray[i+N_Group_T-N_Group_S] > 0){
            for (int j = 0; j < N_Group_T; j++){
                if (Synapses[i][j].conn) update_variables();
            }
        }
    }
    #pragma omp parallel for collapse(2) shared(Synapses,neurons)
    for (j = 0; j < N_Group_T; j++){
        for (i = 0; i < N_S + N_Group_S; i++){
            if (Synapses[i][j].conn && SpikeArray[i+N_Group_T-N_Group_S]){
            neurons[j].I = Synapses[i][j].target_I;
            }
        }
    }
}


void UpdateSynapses_post(Synapse** Synapses, Neuron* neurons, int* SpikeArray
) {
    #pragma omp parallel for collapse(2) shared(Synapses)
    for (int i = 0; i < N_Group_T; i++){
        if (SpikeArray[i] > 0){
            for (int j = 0; j < N_S + N_Group_S; j++){
                /* update synaptic variables */
            }
        }
    }

    #pragma omp parallel for collapse(2) shared(Synapses) reduction(+:mean,
num)
    for (int k = 0; k< N_S + N_Group_S; k++){
        for (int l = 0; l< N_Group_T; l++){
            /* calculate mean and num values */
        }
    }
    if (num > 0) mean = mean / num;

    #pragma omp parallel for collapse(2) shared(Synapses)
    for (i = 0; i < N_Group_T; i++){
        if (SpikeArray[i] > 0){
            for (j = 0; j < N_S + N_Group_S; j++){
                /* update synaptic variables */
            }
        }
    }
}
```

The OpenMP instructions are distinguished by beginning with the hash symbol (#) followed by the keyword *pragma*. In C, the *pragma* directive is used to instruct the compiler to use pragmatic or implementation-dependent features:

- The *omp parallel* directive explicitly instructs the compiler to parallelize the chosen block of code.
- The *parallel for* directive simply divides loop iterations between the spawned threads.
- The *collapse* clause allows parallelization of multiple loops in a nest without introducing nested parallelism, given that the loops are perfectly nested (there is no intervening code) and form a rectangular iteration space and the bounds and stride of each loop is invariant over all the loops, as happens in this case.
- *shared* and *private* clauses define whether the enclosed variables are visible to and accessible by all threads running in associated parallel regions (*Synapses*, *neurons*), or each thread owns a private copy of the variables, keeping their modifications hidden (*_vm*, *_vt*, *_x*).
- Finally, *reduction(+:mean,num)* enables the spawned threads to calculate the *mean* and *num* variable that need different summations by avoiding possible race conditions. Adding this clause instructs OpenMP to make a copy of the reduction variable per thread, initialized to the identity of the reduction operator. Each thread then reduces into its local variable and at the end of the loop the local results are combined, again using the reduction operator, into the global variable.

Chapter 6

Performance Analysis

6.1 Simulation Environment

Performance measurements were collected by running simulation experiments on a dual-GPU node of the ARIS (Advanced Research Information System) supercomputer developed and operated by the Greek Research and Technology Network. ARIS' GPU nodes each contain 2 processing units of the **Haswell - Intel(R) Xeon(R) E5-2660v3** type, each possessing 10 cores, 64 GB of total memory and 2 **NVIDIA Tesla K40** accelerators with 12 GB accelerator memory per each. Accelerator information, obtained by running the CUDA *deviceQuery* sample on a node, is listed below:

```
Device 0: "Tesla K40m"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11520 MBytes (12079136768
    bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            876 MHz (0.88 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536,
    65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048
    layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 4 / 0

Device 1: "Tesla K40m"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11520 MBytes (12079136768
    bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            876 MHz (0.88 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536,
    65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048
    layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 131 / 0

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.5, CUDA Runtime
    Version = 7.5, NumDevs = 2,
```

The final version of the simulator used for performance analysis contains a synaptic struct of arrays allocated in both device and host memory, utilizes the device's shared memory per block when operating on the said struct and calculates the mean value needed by the last kernel with a reduction function of the Thrust CUDA library, as presented in sections 5.1.5.4, 5.1.5.5 and 5.1.5.3 respectively. Single-precision *float* datatype was chosen for the majority of the simulation variables, since maximum reported error was lower than 1.00E-13, as reported in 5.1.5.1, and was deemed negligible. Kernel code remained identical to the initial implementations shown in 5.1.3.1. Regarding kernel execution configuration, a block size of 256 proved to outperform any other configurations on all experiments, so that was the absolute block size used in all experiments. Any difference to this version shall be reported on the following experimental results listing.

The parameters differing between experiments are:

- Number of Input and AdEx Neurons: The purpose is discovering the differences in acceleration in relation to the size of the problem, keeping all other parameters stable.
- Overall Spike Frequency: Defining total spiking activity which directly affects thread workload per neuron and its connected synapses.
- Synapse Connectivity: Same purpose as spike frequency, as variables of unconnected synapses are never operated on, reducing overall workload.
- Timesteps: simulation time, each timestep corresponds to 1 millisecond on the following experiments.

6.2 Experimental Results

6.2.1 Comparisons based on network size

Initializations of both Neurons and Synapses were absolutely corresponding to the paper this thesis is based on. For these experiments, synaptic connectivity was set to 100%, meaning that every Input or AdEx Neuron is connected to every other neuron of the network. Moreover, spiking frequency was set to 1 kHz (spikes appearing on every timestep) and 100 percent spiking frequency means that every single neuron was set to fire constantly on both NxM and MxM cases. It shall be reminded that *N_S* refers to the number of the network's Input Neurons, *N_Group_S* corresponds to the number of the presynaptic AdEx Neurons and *N_Group_T* corresponds to the network's postsynaptic AdEx Neurons.

Table 6.1: Acceleration by Network Size, MxM

N_S	0	0	0	0	0
N_Group_S	100	500	1000	5000	10000
N_Group_T	100	500	1000	5000	10000
Timesteps	1000	1000	1000	1000	1000
Connectivity(%)	100	100	100	100	100
Brian time	6,88	305,81	2459,35	41052,4	99621,59
CPU time	4,25	112,08	563,23	14800,12	67553,38
OpenMP time	0,79	17,9	89,97	2488,77	10381,62
GPU time	0,2	0,71	2,3	53,98	215,18
Acceleration GPU vs Brian	34,4	430,71	1069,28	760,51	462,96
Acceleration GPU vs CPU	21,25	157,85	244,88	274,17	313,93
Acceleration GPU vs OpenMP	3,95	25,21	39,11	46,11	48,24

Table 6.2: Acceleration by Network Size, NxM

N_S	200	200	200	500	500	500	1000	1000	1000
N_Group_S	0	0	0	0	0	0	0	0	0
N_Group_T	5000	10000	50000	5000	10000	50000	5000	10000	50000
Timesteps	1000	1000	1000	1000	1000	1000	1000	1000	1000
Connectivity(%)	100	100	100	100	100	100	100	100	100
CPU time	547,5	1076,16	5426,05	1384,46	2689,74	13610,42	3102,15	5454,1	29920,12
OpenMP time	92,33	170,79	889,51	232,68	433,83	2126,63	534,85	894,11	4749,23
GPU time	1,68	3,17	15,45	3,99	7,76	38,41	8,11	15,51	76,98
Acceleration GPU vs CPU	325,89	339,48	351,20	346,98	346,61	354,34	382,50	351,65	388,67
Acceleration GPU vs OpenMP	54,95	53,87	57,57	58,31	55,91	55,36	65,94	57,64	61,69

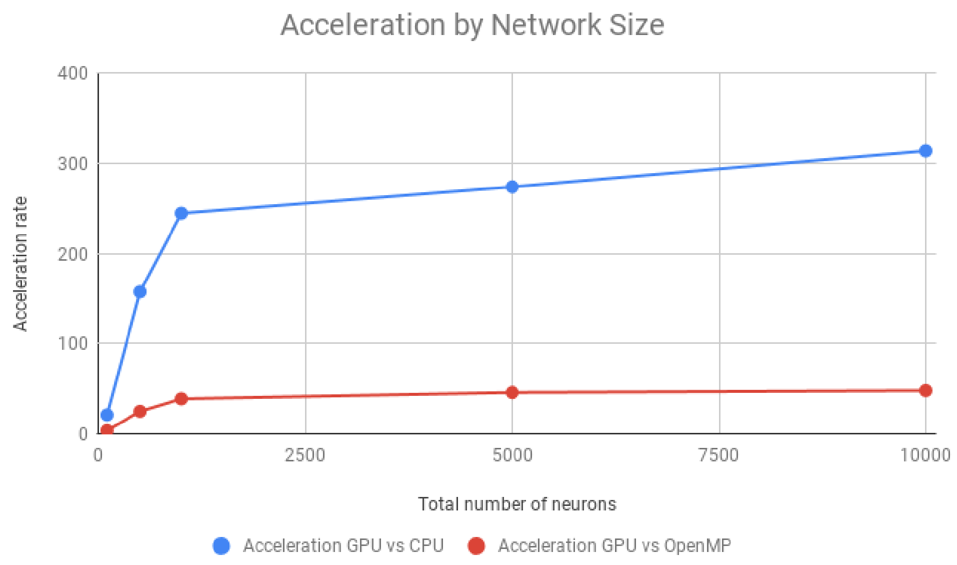


Figure 6.1: Acceleration GPU vs CPU vs OpenMP, MxM

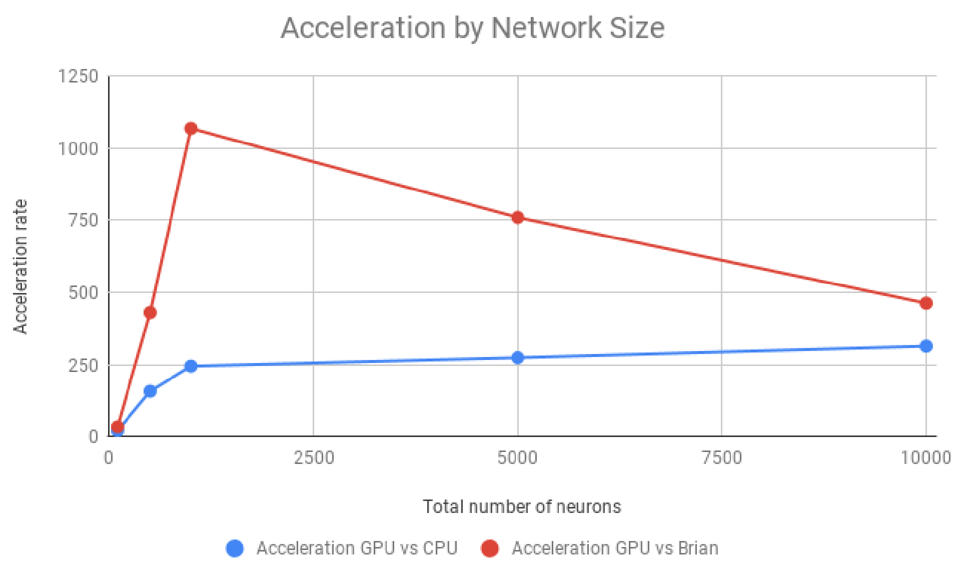


Figure 6.2: Acceleration GPU vs CPU vs Brian, MxM

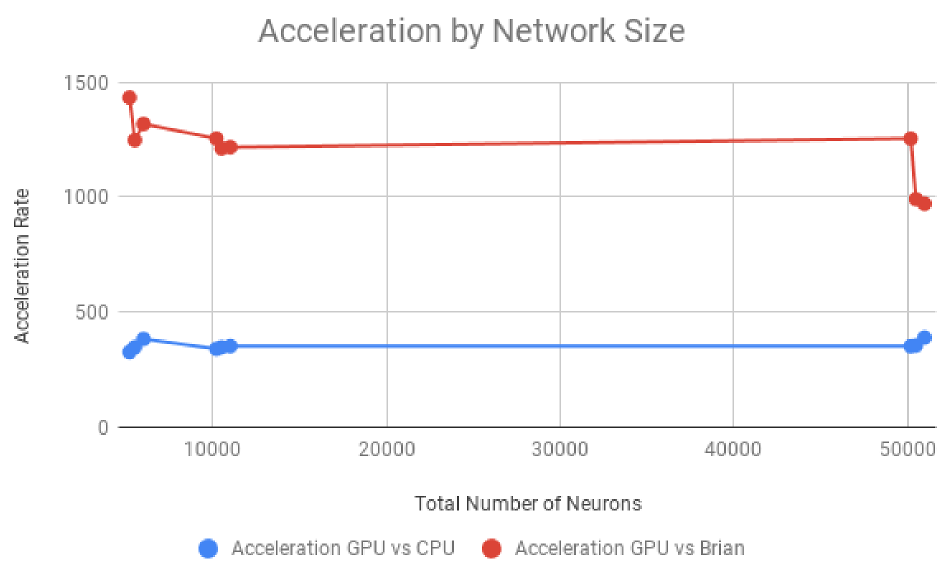


Figure 6.3: Acceleration GPU vs CPU vs Brian, NxM

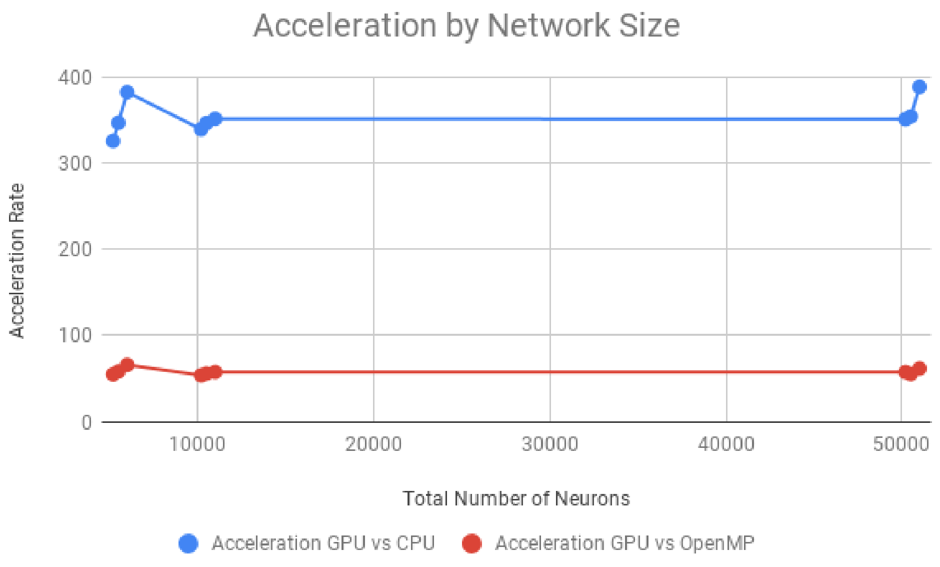


Figure 6.4: Acceleration GPU vs CPU vs OpenMP, NxM

First, it should be noted again that in all cases, the *SolveNeurons* kernel consumes a minimal percentage of total simulation time. This results in synaptic operations defining simulator performance. Moreover, as Brian's simulations perform far worse than the serial implementation of the thesis' simulator, it has been omitted from the majority of experiments as there was no gain in comparing acceleration rates against ones measured considering CPU time.

The OpenMP implementation offers a basic speedup that fluctuates around 6x original CPU time. 8 OpenMP threads were spawned at each execution and parallelism reached adequately close to optimal without extensive research. It was used as a baseline acceleration and is vastly outperformed by the CUDA implementation.

By observing GPU against CPU times of the MxM experiments, it is made clear that heavier computational load results in greater acceleration rates, as long as there are enough CUDA threads for the grid to cover the entire synaptic grid of the simulation, since each thread is assigned a single synapse and scaling such a layout greatly outworks a serial modification of the entirety of synapses. Maximum GPU acceleration on the MxM simulations reached almost 314x speedup against its serial counterpart.

By observing NxM simulation results with a similar scale to the MxM simulations, speedup rates are similar and even a little higher in the NxM case. The explanation lies in the fact that the simulation process differs in calculation load considering full spiking activity between the two neuron architectures. When there are Input Neurons, the synaptic variables of only the synapses that connect Input and AdEx Neurons are updated, as shown in 5.1.3.1, while in the case of solely AdEx Neurons the ensemble of the synaptic structure is updated regularly. Therefore, in the NxM case updating synapses based on postsynaptic STDP expressions is eliminated, as Input Neurons do not correspond to actual model

Neurons and require no modification whatsoever throughout the simulation, purely offering spikes to the network. As the CUDA version's heaviest kernels are the postsynaptic ones, NxM simulations are lighter for the GPU, slightly increasing performance. The greatest acceleration ratio achieved between GPU and CPU versions was 388x on the largest NxM experiment.

Large Scale Experimentation

In order to decrease execution time for a larger experiment, simulation time was reduced to **0.1 s** instead of regular 1 second simulation time, so experiments corresponded to 100 timesteps. The maximum number of Neurons simulated for this thesis' purposes is 20000. Connectivity and spiking percentages are set to 100 and both single and dual-GPU experiments were conducted.

Table 6.3: Acceleration by Network Size, Large Scale, MxM

N_S	0
N_Group_S	20000
N_Group_T	20000
Timesteps	100
Connectivity(%)	100
CPU time	27962,93
OpenMP time	3967,63
Single GPU time	890,17
Double GPU time	43,43
Acceleration single GPU vs CPU	31,41
Acceleration single GPU vs OpenMP	4,45
Acceleration dual GPU vs CPU	643,86
Acceleration dual GPU vs OpenMP	91,36
Acceleration dual GPU vs single GPU	20,5

As seen in the table, acceleration follows a similar scale and is even increased related to smaller experiments when compared to CPU performance using a node with two accelerator devices, since both devices operate concurrently and problem size is divided in half for the majority of the simulation's functions. The significant penalty in acceleration when using a single GPU is caused by the need to constantly copy portions of simulation data between device and host memory, which is worse when using a synaptic struct-of-arrays due to the overhead of separate *cudaMemcpyAsync* calls for each variable array. Acceleration reached the impressive level of 643x for the dual GPU version, while the single GPU version only achieved a 31x speedup and improved by just a limited factor of 4x compared to OpenMP, a performance significantly worse than on previous attempts.

6.2.2 Comparisons based on connectivity

A network of 50% connectivity was simulated to check the impact of sparse synaptic connections relative to absolute connectivity. Identical connection patterns were used in order to provide safe conclusions between MxM and NxM simulations, as only neurons with an even ID were interconnected, and neurons having odd IDs were basically inactive throughout the simulation. Spiking is once again set to network-wide, constant occurrence.

Table 6.4: Acceleration by Amount of Synaptic Connections, MxM

N_S	0	0	0	0	0	0	0	0
N_Group$_S$	500	500	1000	1000	5000	5000	10000	10000
N_Group_T	500	500	1000	1000	5000	5000	10000	10000
Timesteps	1000	1000	1000	1000	1000	1000	1000	1000
Connectivity(%)	100	50	100	50	100	50	100	50
CPU time	112,08	60,7	563,23	317,03	14800,12	8735,66	67553,38	40631,44
GPU time	0,71	0,7	2,3	2,31	53,98	54,27	215,18	215,91
Acceleration GPU vs CPU	157,85	86,71	244,88	137,24	274,17	160,96	313,93	188,18

Table 6.5: Acceleration by Amount of Synaptic Connections, NxM

N_S	200	200	200	500	500	500	1000	1000	1000
N_Group_S	0	0	0	0	0	0	0	0	0
N_Group_T	5000	10000	50000	5000	10000	50000	5000	10000	50000
Timesteps	1000	1000	1000	1000	1000	1000	1000	1000	1000
Connectivity(%)	50	50	50	50	50	50	50	50	50
CPU time	309,82	607,65	3078,36	786,43	1524,31	9093,08	1592,63	3161,05	17737,7
GPU time	1,69	3,19	15,5	4,03	7,82	38,53	8,17	15,61	77
Acceleration GPU vs CPU	183,32	190,48	198,61	195,14	194,92	236	194,93	202,50	230,35

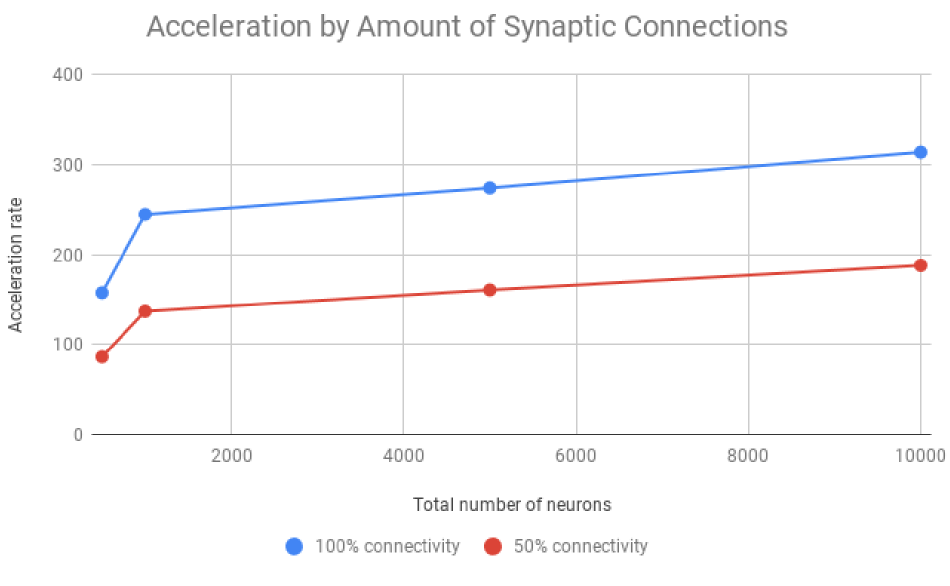


Figure 6.5: Acceleration GPU vs CPU, MxM

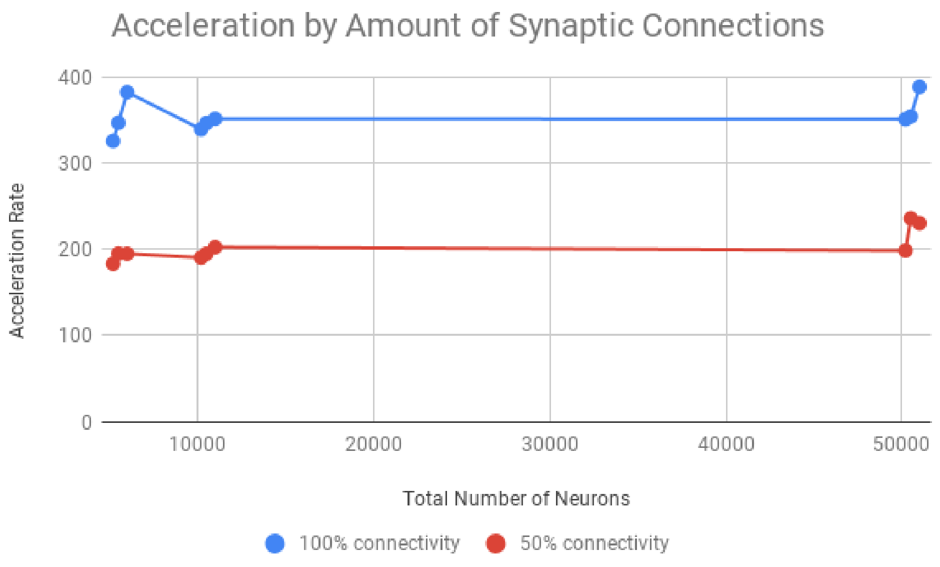


Figure 6.6: Acceleration GPU vs CPU, NxM

Results from this experimentation were especially interesting. In the entirety of conducted experiments, simulation times were equal to, or even minimally worse than, those of full connectivity. Since the CPU version was sped up significantly as a result of lower computational workload, acceleration rates were fairly lower. The reason for these results is most probably the way CUDA threads are executed in parallel. As each warp of 32 threads executes the same instruction simultaneously, the connectivity layout forces half of every warp's threads to be idle while the other half executes all the synaptic operations. In CUDA, thread divergence leads to serialization of execution inside a warp, so any possible speedup is eliminated as active threads undergo similar execution intensity in both connectivity cases and inactive threads do not aid in speeding up the simulation. As the OpenMP implementation followed identical acceleration patterns to those of the first presented experiments, OpenMP times were omitted from hereafter. Speedup patterns did not change regarding simulation scaling for the MxM experiments, as greatest acceleration was achieved on the largest experiment, reaching a 188x speedup. A similar pattern can be noticed on the NxM experiments, even though largest - 236x - speedup was reached on the second largest experiment.

6.2.3 Comparisons based on firing percentage

A similar approach to the previous subsection was conducting experiments with a difference in the number of neurons that are producing spikes on every timestep. Testing lower spiking frequency while keeping the number of firing neurons intact was pointless as that would lead to total program inactivity on the timesteps that no neuron has fired. Parameters were initialized as usual and only neurons with an even ID number were set to fire constantly throughout the simulation.

Table 6.6: Acceleration by Amount of Firing Neurons, MxM

N_S	0	0	0	0	0	0	0	0
N_Group_S	500	500	1000	1000	5000	5000	10000	10000
N_Group_T	500	500	1000	1000	5000	5000	10000	10000
Timesteps	1000	1000	1000	1000	1000	1000	1000	1000
Connectivity(%)	100	100	100	100	100	100	100	100
Spiking (%)	100	50	100	50	100	50	100	50
CPU time	112,08	64,07	563,23	338,8	14800,12	8991,75	67553,38	45335
GPU time	0,71	0,59	2,3	1,86	53,98	42,75	215,18	172,95
Acceleration GPU vs CPU	157,85	108,59	244,88	182,15	274,17	210,33	313,93	262,12

Table 6.7: Acceleration by Amount of Firing Neurons, NxM

N_S	200	200	200	500	500	500	1000	1000	1000
N_Group_S	0	0	0	0	0	0	0	0	0
N_Group_T	5000	10000	50000	5000	10000	50000	5000	10000	50000
Timesteps	1000	1000	1000	1000	1000	1000	1000	1000	1000
Connectivity(%)	100	100	100	100	100	100	100	100	100
Spiking (%)	50	50	50	50	50	50	50	50	50
CPU time	332,58	665,45	3335,21	847,13	1657,04	8351,59	1692,31	4527,19	16831,73
GPU time	1,24	2,33	11,12	2,89	5,63	27,59	5,9	11,29	55,13
Acceleration GPU vs CPU	268,21	285,61	299,92	293,12	294,32	302,70	286,83	400,99	305,31

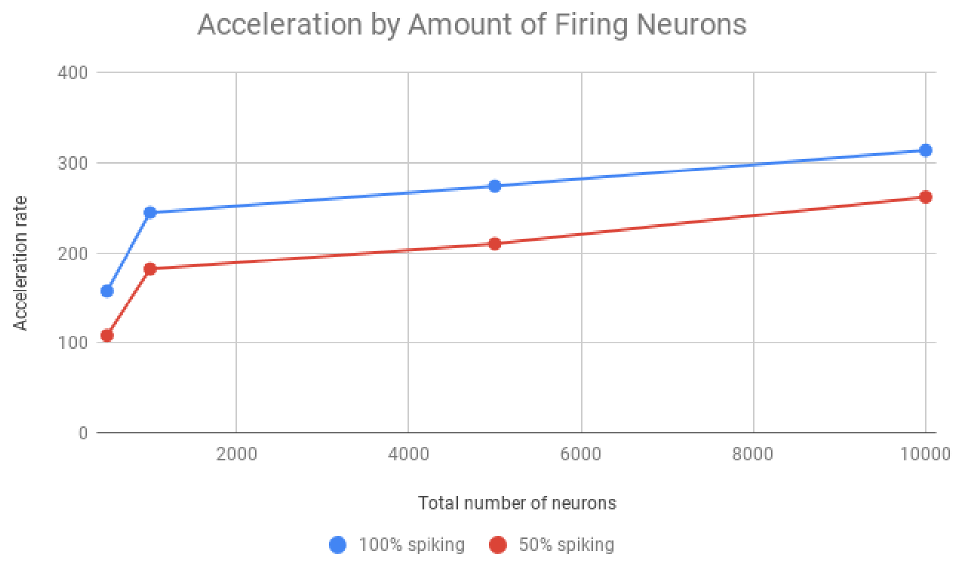


Figure 6.7: Acceleration GPU vs CPU, MxM

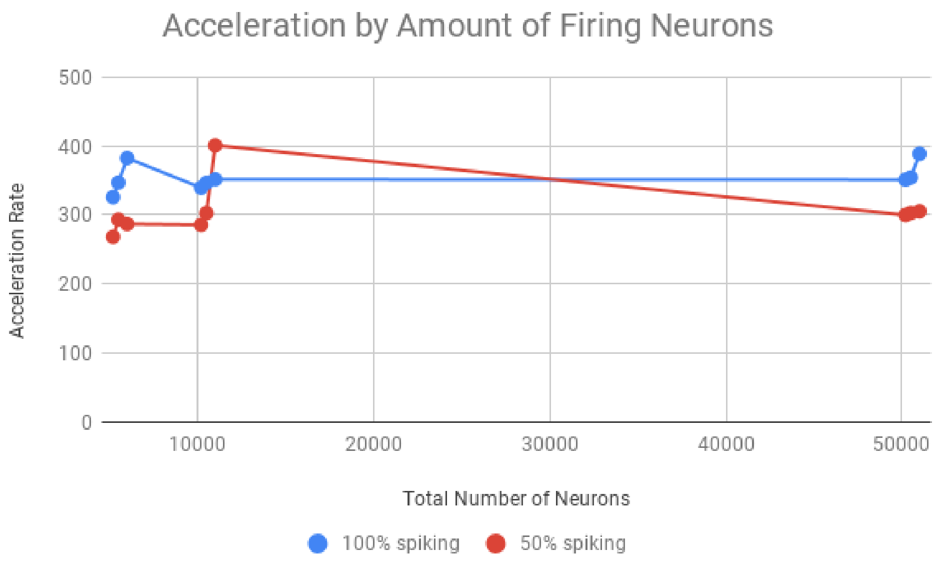


Figure 6.8: Acceleration GPU vs CPU, NxM

In the case of sparser firing occurence of these particular experiments, the entirety of spawned CUDA threads follow the same rules from an activity standpoint. This means that while every second timestep all threads execute the entirety of the kernel operations, they also exit the kernels almost immediately after creation in the other half of the timesteps, inflicting much less penalty to the GPU compared to the CPU execution. Greatest acceleration noted in this case was 262x for the MxM version, once again increasing along with the amount of neurons present in the experiment, while NxM experiments showcased similar acceleration rates except for a high kick up to 401x speedup noticed in the 11000 neurons simulation. This isolated increase in speedup is actually attributed to the CPU underperforming in that particular serial experiment, as all other CPU times are similar to those of lower connectivity except for that one, so there is no significant induction regarding GPU performance.

Chapter 7

Conclusion

7.1 Remarks

As Neuroscience research follows the modern rhythms of scientific progress, it becomes significantly dependent on computer simulations. As proven through both a research and an implementation point of view throughout my occupation with this diploma thesis, neuron and synapse modeling almost definitively contains highly complex arithmetic operations (differential equations solutions by numerical methods etc.), proper experimentation that could allow experts to push Brain Research farther demands significant simulation speedup to the point that regular (serial) implementations of neuroscience simulations burden experts with extensive waiting time between simulations. The increased demand for simulation outputs that enable effective data processing calls for universal acceleration of the implemented corresponding algorithms.

This Diploma Thesis successfully attempted to contribute to this demand by suggesting a highly efficient solution to the problem of large network simulations concerning the AdEx neuron model and synapses characterized by STDP. Firstly, there was a basic acceleration of the simulation by porting the code used by the Brian Simulator for the models of interest into a new simulator that used the low-level C programming language, which was then available for experimentation using two different parallel programming interfaces, OpenMP and Nvidia's CUDA parallel platform. The OpenMP implementation utilized 8 OpenMP threads and consistently achieved performance increase equal to around 6 times the original C simulator runtime. The CUDA implementation was the main research point this thesis evolved around and showcased impressive results, as experiments of a considerable size were offered speedup levels of above 100 times the serial version, reaching as high as almost a 401x speedup rate versus the C simulator and a 1069x speedup rate against Brian, though Brian approached the C version's times as network size increased to vast levels.

Throughout my work on this thesis I have discovered that algorithms with high parallelism potential are complemented perfectly with a CUDA-enabled GPU accelerator, as the very high amount of modern GPUs' processing cores offer immense performance advantages when used correctly. The CUDA memory model provides various memory distinctions that hasten memory access from GPU threads, which is extremely important since the majority of neuroscientific experiments are characterized by grand memory requirements. In the STDP synaptic model's case, CUDA's shared memory proved most advantageous for the execution of the corresponding arithmetic operations. Performance penalties from memory transaction between host and accelerating device were minimized, since the greatest possible amount of necessary data is passed to the device at the beginning of the simulation and needs not be transferred back before final results are calculated.

In general, acceleration results were even better than expected for all but one parameter change. Due to the model's architecture, device kernels were implemented enclosing numerous branch instructions that force diverging adjacent CUDA threads to execute serially. This resulted to execution time remaining almost identical for the GPU implementation when decreasing the amount of synaptic connections while keeping other network parameters unmodified, while the serial C implementation would greatly benefit from this change due to lesser workload. Hence, optimal speedup could be achieved only while maintaining the densest possible network.

Concluding, the CUDA API in total offers firm performance advantages for the models investigated on this diploma thesis. It was proven that neuron models of similar architecture benefit greatly from GPU accelerators, which scale competently enough along with network size. This thesis showed that a GPU can outperform its CPU counterparts significantly, especially on the synaptic models. However, supposing that scientists do not have access to such a device, the OpenMP version offers an adequate backup for basic runtime acceleration, while also posing a much lighter challenge from a development standpoint.

7.2 Future work

An important aspect that could be further explored based on this implementation is the optimization of neuron simulations that utilize multiple GPUs as accelerating devices, as my diploma thesis reached as far as splitting a network's workload between two CUDA GPUs without any device intercommunication.

Moreover, GPU devices of Pascal architecture or newer offer the possibility of oversubscribing GPU memory, enabling out-of-core computations for any code that is using CUDA's Unified Memory for allocations. There may be room for improvement regarding scaling the acceleration when utilizing a single GPU, as explained briefly in 5.1.5.6.

Finally, the CUDA implementation of the AdEx model simulation could be integrated into the Brian simulator, making the model natively accelerated. As GPU acceleration of the entire Brian simulator is still in early stages, such an addition can accelerate neuroscientists' research even further, since there would be no need of learning a different framework other than Brian in order to conduct desired experiments. This could prove very important modification, as ease of usage is one of the most important factors when developing frameworks that are to be used by scientists of disciplines not closely related to computer science.

Bibliography

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

[Heb49] Donald O. Hebb. *The organization of behavior: A neuropsychological theory.* New York: Wiley, June 1949. ISBN: 0-8058-4300-0.

[Hod52] A. F. Hodgkin A.L. & Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *Journal of physiology* (1952).

[Ral62] Wilfrid Rall. "Theory of Physiological Properties of Dendrites". In: *Annals of the New York Academy of Sciences* 96.4 (1962), pp. 1071–1092. DOI: `10.1111/j.1749-6632.1962.tb54120.x`. eprint: `https://nyaspubs.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1749-6632.1962.tb54120.x`. URL: `https://nyaspubs.onlinelibrary.wiley.com/doi/abs/10.1111/j.1749-6632.1962.tb54120.x`.

[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`.

[HHD86] F. R. A. Hopgood, R. J. Hubbold, and D. A. Duce, eds. *Advances in Computer Graphics II.* Berlin, Heidelberg: Springer-Verlag, 1986. ISBN: 3-540-16910-5.

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

[SKC88] Terrence Sejnowski, Christof Koch, and Pat Churchland. "Computational Neuroscience". In: *Science (New York, N.Y.)* 241 (Oct. 1988), pp. 1299–306. DOI: `10.1126/science.3045969`.

[Ooy00] Arjen Ooyen. "Methods in Neuronal Modeling (2nd Edition), by C. Koch I. Segev (eds.)" In: *International Journal of Neural Systems* 10 (Jan. 2000), pp. 331–332.

[BLN01] E P Bauer, J E LeDoux, and K Nader. "Fear conditioning and LTP in the lateral amygdala are sensitive to the same stimulus contingencies." In: *Nature neuroscience* 4.7 (2001), pp. 687–8.

[HC01] M. L. Hines and N. T. Carnevale. “Neuron: A Tool for Neuroscientists”. In: *The Neuroscientist* 7.2 (2001). PMID: 11496923, pp. 123–135. DOI: 10.1177/107385840100700207.

[PD01] L.F. Abbott Peter Dayan. *”Theoritical Neuroscience”*. The MIT Press, 2001, 11–14. ISBN: 0-262-54185-8.

[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.

[Har04] Mark Harris. *Fast Fluid Dynamics Simulation on the GPU.* 2004. URL: https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch38.html.

[Lyn04] MA Lynch. “Long-term potentiation and memory”. In: *Physiological reviews* 84.1 (2004), pp. 87–136.

[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: 10.1002/0470018860.s00363.

[Atk07] Denny Atkin. *The Right GPU for You.* 2007. URL: https://web.archive.org/web/20070506033224/http://computershopper.com/feature/200704_the_right_gpu_for_you.

[GD07] M. Gewaltig and M. Diesmann. “NEST (NEural Simulation Tool)”. In: *Scholarpedia* 2.4 (2007). revision #130182, p. 1430. DOI: 10.4249/scholarpedia.1430.

[Owe+07] John D Owens et al. “A Survey of general-purpose computation on graphics hardware”. In: *Computer graphics forum.* Vol. 26. 1. 2007, pp. 80–113.

[Sch+07] Michael C Schatz et al. “High-throughput sequence alignment using Graphics Processing Units”. In: *BMC bioinformatics* 8.1 (2007), p. 474.

[Sil07] Dee Unglaub Silverthorn. *”Human Physiology: An Integrated Approach”*. Pearson/Benjamin Cummings, 2007, p. 271. ISBN: 978-0-8053-6851-2.

[ASAE08] E. Alerstam, T. Svensson, and S. Andersson-Engels. “Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration”. In: *Journal of Biomedical Optics* 13 (2008), p. 060504.

[BR08] Nicolas Brunel and Mark van Rossum. “Lapicque’s 1907 paper: From frogs to integrate-and-fire”. In: *Biological cybernetics* 97 (Jan. 2008), pp. 337–9. DOI: 10.1007/s00422-007-0190-0.

[CD08] Natalia Caporale and Yang Dan. “Spike Timing–Dependent Plasticity: A Hebbian Learning Rule”. In: *Annual review of neuroscience* 31 (Feb. 2008), pp. 25–46. DOI: 10.1146/annurev.neuro.31.060407.125639.

[GDB08] V. Garcia, E. Debreuve, and M. Barlaud. “Fast k-nearest neighbor search using GPU”. In: *CVPR Workshop on Computer Vision on GPU.* 2008.

[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.

[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.

[KF+09] Andreas K. Fidjeland et al. “NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs”. In: July 2009, pp. 137–144. DOI: 10.1109/ASAP.2009.24.

[Ler09] Larry Lerner. *Viewpoint: Mass GPUs, not CPUs for EDA simulations.* 2009. URL: https://www.eetimes.com/document.asp?doc_id=1170767.

[Nag+09] Jayram Moorkanikara Nageswaran et al. “A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors”. In: *Neural Networks* 22.5 (2009). Advances in Neural Networks Research: IJCNN2009, pp. 791 –800. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2009.06.028. URL: http://www.sciencedirect.com/science/article/pii/S0893608009001373.

[BPS10] Mohammad Bhuiyan, Vivek Pallipuram, and Melissa Smith. “Acceleration of spiking neural networks in emerging multi-core and GPU architectures”. In: Apr. 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470899.

[SGS10] John Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in science engineering* 12 (May 2010), pp. 66–72. DOI: 10.1109/MCSE.2010.69.

[Ben13] Lubica Benuskova. *Lecture notes in Computational Neuroscience.* 2013.

[Hoa+13] Roger V Hoang et al. “A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling.” In: *Frontiers in neuroinformatics* 7 (2013), p. 19. DOI: 10.3389/fninf.2013.00019.

[VIM13] Maxim V Ivannikov and Gregory Macleod. “Mitochondrial Free Ca2+ Levels and Their Effects on Energy Metabolism in Drosophila Motor Nerve Terminals”. In: *Biophysical journal* 104 (June 2013), pp. 2353–61. DOI: 10.1016/j.bpj.2013.03.064.

[VC13] Richard Vuduc and Jee Choi. “A brief history and introduction to GPGPU”. In: *Modern accelerator technologies for geographic information science* (Aug. 2013), pp. 9–23. DOI: 10.1007/978-1-4614-8745-6_2.

[Lli14] Rodolfo Llinás. “Intrinsic electrical properties of mammalian neurons and CNS function”. In: *Frontiers in cellular neuroscience* 8 (Nov. 2014), p. 320. DOI: 10.3389/fncel.2014.00320.

[OAMD14] Bogdan Oancea, Tudorel Andrei, and Raluca Mariana Dragoescu. “GPGPU Computing”. In: *Challenges of the Knowledge Society* 2 (Aug. 2014).

[Has+15] Khondker Hasan et al. “Performance Prediction Model and Analysis for Compute-Intensive Tasks on GPUs”. In: (Sept. 2015).

[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.

[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.

[Inv] *Why is neuroscience important?* 2017. URL: http://invigorate.royalsociety.org/ks5/learning-its-all-in-your-head/why-is-neuroscience-important.aspx.

[Bea18] Jonathan Beard. *A SHORT INTRO TO STREAM PROCESSING.* 2018. URL: http://www.jonathanbeard.io/blog/2015/09/19/streaming-and-dataflow.html.

[Wikb] *Movement Disorders.* 2018. URL: https://www.neuromodulation.com/movement-disorders.

[Ole18] Olena. *A Brief History of GPU.* 2018. URL: https://medium.com/altumea/a-brief-history-of-gpu-47d98d6a0f8a.

[Zun18] Peter Zunitch. *CUDA vs. OpenCL vs. OpenGL.* 2018. URL: https://www.videomaker.com/article/c15/19313-cuda-vs-opencl-vs-opengl.

[BB19] James Bower and D Beeman. “Exploring Realistic Neural Models with the GEneral NEural SImulation System”. In: (May 2019).

[Pcm] *Definition of: GPU.* 2019. URL: https://www.pcmag.com/encyclopedia/term/43886/gpu.

[Wika] *MOOSE Framework - Open Source Multiphysics.* Idaho National Laboratory. URL: https://mooseframework.inl.gov/.

[Wikc] *Nvidia CUDA Home Page.* URL: `https://developer.nvidia.com/cuda-zone`.

[Thr] *The API reference guide for Thrust, the CUDA C++ template library.* NVIDIA. URL: `https://docs.nvidia.com/cuda/thrust/index.html`.

[Wikd] *The GENESIS 2 simulator home page.* URL: `http://genesis-sim.org/GENESIS/`.