PARALLEL SIMULATION OF NEURAL NETWORKS ON SPINNAKER UNIVERSAL NEUROMORPHIC HARDWARE

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2010

By
Xin Jin
School of Computer Science

Contents

Abstract **12**

Declaration **13**

Copyright **14**

Acknowledgements **15**

1 Introduction **16**
1.1 Background 16
1.2 Motivation 18
1.3 Research aims 19
1.4 Contributions 21
1.5 Thesis overview 22
1.6 Publications 22

2 Neural network modeling **26**
2.1 Overview 26
2.2 Biological neural networks 27
2.2.1 Research methodology 27
2.2.2 The brain and cortex 28
2.2.3 Neurons 28
2.2.4 Synapses 30
2.3 Mathematical neural modeling 31
2.3.1 Neuron electrophysiology 31
2.3.2 The Hodgkin-Huxley model 33
2.3.3 The Integrate-and-fire model 35
2.3.4 The Izhikevich model 37

2.3.5 Non-spiking neural models . . . 39
2.3.6 Neural coding . . . 39
2.3.7 Connectivity patterns . . . 40
2.3.8 Axonal delay . . . 42
2.3.9 Synaptic plasticity . . . 42

3 Parallel engineering solutions 44
3.1 Overview . . . 44
3.2 Simulation clues: processing, communication and storage . . . 45
3.2.1 Processing . . . 45
3.2.2 Communication . . . 45
3.2.3 Storage . . . 46
3.3 Current engineering solutions . . . 46
3.3.1 Analogue systems . . . 47
3.3.2 Software-based systems . . . 47
3.3.3 Supercomputer-based systems . . . 48
3.3.4 FPGA-based systems . . . 49
3.3.5 Other solutions . . . 49
3.4 Parallelism . . . 50
3.5 SpiNNaker . . . 51
3.5.1 Why SpiNNaker? . . . 51
3.5.2 The SpiNNaker chip . . . 53
3.5.3 The SpiNNaker system . . . 57

4 Spiking neural network on SpiNNaker 58
4.1 Overview . . . 58
4.2 The choice of neuronal dynamical model . . . 58
4.3 Modeling the Izhikevich model . . . 59
4.3.1 Choice of scaling factors – generic optimization . . . 60
4.3.2 Equation transformation – ARM specific optimization . . . 62
4.3.3 Precision – fixed-point v.s. floating-point arithmetic . . . 64
4.3.4 Processing speed . . . 68
4.4 Modeling spikes and synapses . . . 68
4.4.1 Propagation of spikes . . . 69
4.4.2 Event-address mapping scheme . . . 70
4.5 Updating the input current . . . 76

4.6 System scheduling . . . 77
4.7 Simulation results on a single-processor system . . . 78
4.7.1 Processing speed . . . 79
4.7.2 Data memory requirement . . . 81
4.7.3 Communication analysis . . . 82
4.8 Optimization . . . 83
4.8.1 Input processing optimization . . . 83
4.8.2 An alternative scheduler . . . 84
4.9 A discussion of real-time . . . 86

5 Multi-processor simulation 88
5.1 Overview . . . 88
5.2 Converting neural networks into SpiNNaker format . . . 89
5.2.1 Neural network description files . . . 91
5.2.2 Mapping flow . . . 92
5.2.3 Connection data structures . . . 93
5.2.4 Synapse allocation . . . 96
5.2.5 Routing table generation . . . 98
5.3 The multi-chip SoC Designer platform . . . 102
5.4 Simulation results . . . 103
5.4.1 Initialization . . . 103
5.4.2 Downloading and dumping . . . 104
5.4.3 A 60-neuron network test . . . 104
5.4.4 A 4000-neuron network test . . . 108
5.5 A practical application . . . 108
5.5.1 The Doughnut Hunter application . . . 108
5.5.2 Simulation results . . . 112
5.5.3 The Doughnut Hunter on a physical SpiNNaker chip . . . 114
5.6 Discussion . . . 116
5.6.1 Neuron-processor mapping . . . 116
5.6.2 Code and data downloading . . . 118
5.6.3 Simulation results dumping . . . 118
5.6.4 Software architecture for SpiNNaker . . . 119

6 Synaptic plasticity on SpiNNaker 121
6.1 Overview . . . 121

6.2 Spike-timing-dependent plasticity . . . 122
6.3 Methodology . . . 123
6.3.1 The pre-post-sensitive scheme . . . 123
6.3.2 The pre-sensitive scheme . . . 124
6.3.3 The deferred event-driven model . . . 125
6.3.4 The STDP process . . . 127
6.3.5 Implementation . . . 128
6.4 Simulation results . . . 130
6.4.1 10-second 60-neuron test . . . 130
6.4.2 30-second 76-neuron test . . . 133
6.5 Discussion . . . 137
6.5.1 Firing rates and the length of timing records . . . 137
6.5.2 Approximation and optimization . . . 138

7 MLP modeling on SpiNNaker 140
7.1 Overview . . . 140
7.2 Introduction . . . 140
7.3 MLP model with BP algorithm . . . 142
7.3.1 Feedforward and recurrent neural networks . . . 142
7.3.2 BP algorithm . . . 144
7.3.3 The neural simulator – Lens . . . 146
7.4 Partitioning and mapping schemes . . . 147
7.4.1 Review of partitioning schemes . . . 147
7.4.2 The CBP scheme and the non-pipelined model . . . 148
7.4.3 Mapping onto SpiNNaker with pipeline . . . 152
7.4.4 Data storage . . . 154
7.4.5 Analytical comparison . . . 155
7.4.6 Memory requirements . . . 157
7.4.7 Performance estimation . . . 158
7.5 Partially-connected RNNs . . . 159
7.5.1 The data structure . . . 161
7.5.2 Analytical comparison . . . 162
7.5.3 Memory requirements . . . 164
7.5.4 Performance estimation . . . 167
7.6 SpiNNaker VS. PC . . . 167
7.6.1 The fully-connected network . . . 168

7.6.2 The partially-connected network 170
7.7 Discussion and conclusion 172

8 Conclusion **175**
8.1 Summary of thesis 175
8.2 Future work 178

Bibliography **180**

List of Tables

4.1 Tonic spiking and bursting spike counts 65
4.2 Different spiking pattern spike counts 66
4.3 Input current required for generating rebound spikes 67

7.1 Computation and communication cost 155
7.2 The memory requirement of a GroupA processor in fully connected RNNs 157
7.3 Compressed Column Order Storage 162
7.4 The computation and communication time of partially-connected RNNs 163
7.5 The memory requirements for a GroupA processor in partially-connected RNNs 164
7.6 The time required for one update of the fully-connected networks 169
7.7 The time required for one update of the partially-connected networks 171

List of Figures

2.1 The brain 27
2.2 Neuronal structure 29
2.3 The connection and synapse. 30
2.4 Diffusion of K^+ ions down the concentration gradient though the membrane. 32
2.5 Equivalent circuit representation of cell membrane by Izhikevich [Izh07]. 33
2.6 Action potential generation in the Hodgkin-Huxley model [Izh07]. 35
2.7 The schematic diagram of the leaky integrate-and-fire model by Gerstner [GK02]. 37
2.8 Dorsal and lateral views of the brain connectivity backbone from [HCG+08]. 41

3.1 The SpiNNaker chip organization 54
3.2 The ARM968 subsystem 55
3.3 The SpiNNaker system 57

4.1 Neuron processing 60
4.2 A comparison between floating-point and fixed-point arithmetic implementation from the Matlab simulation. 67
4.3 The routing key 70
4.4 Synaptic weight storage. 70
4.5 An example of synaptic weight storage. 72
4.6 The lookup table 73
4.7 Finding the synaptic Block. 74
4.8 The Synaptic Word. 75
4.9 The input array. 76
4.10 The scheduler for the event driven model. 77

4.11 Results from a single-processor simulation. 80
4.12 The flow chart of the input processing. 83
4.13 An alternative scheduler. 85

5.1 Loading neural networks onto a SpiNNaker chip. 89
5.2 The system description files for the random mode and the predefined mode. 90
5.3 The neural network description files and their format. 90
5.4 The InitLoad flow chart 92
5.5 The data structure used to load connections. 94
5.6 An alternative compressed data structure used to load connections. 94
5.7 Chip indexing. 95
5.8 The partitions of the weight matrix in the linear mapping algorithm, and their mappings onto the processors. 96
5.9 The flow of the AllocConnections() function. 97
5.10 The flow of the generateRoutingTable() function. 99
5.11 Routing planning 101
5.12 SpiNNaker chip components in SoC Designer 102
5.13 A four chip model in SoC Designer 103
5.14 The fascicle and monitor processors during initialization. 104
5.15 A four chip simulation running on the SoC Designer simulator . . 105
5.16 Spike raster generated from 60-neuron network simulations within 1,000 ms. 106
5.17 Neuron states and input currents from SpiNNaker simulation . . . 107
5.18 Spike raster of 4000 neurons on SpiNNaker. 109
5.19 Single neuron activity of 4000-neuron simulation on SpiNNaker. . 110
5.20 The Doughnut Hunter model by Arash Ahmadi and Francesco Galluppi. 111
5.21 The platform for the Doughnut Hunter application. 111
5.22 The Doughnut Hunter GUI during simulation. 113
5.23 Neuron firing timing in the Doughnut Hunter simulation. 113
5.24 The test chip diagram. 114
5.25 A photo of the SpiNNaker Test Chip on the PCB. 115
5.26 The bootup process of the Test Chip. 115
5.27 An ideal neuron-processor mapping. 117
5.28 The software architecture for SpiNNaker 119

6.1 The STDP modification function. 122
6.2 The pre-post-sensitive scheme. STDP is triggered when either a pre-synaptic or a post-synaptic neurons fires. 123
6.3 The pre-sensitive scheme. STDP is triggered only when a pre-synaptic neurons fires (a spike arrives). 123
6.4 The pre-synaptic time stamp. 126
6.5 The time stamp representation. 126
6.6 The post-synaptic time stamp. 126
6.7 Updating the pre-synaptic time stamp. 126
6.8 STDP implementation flow chart. 128
6.9 Calculate the time window and time word. 129
6.10 STDP results. 131
6.11 Weight modification caused by the correlation of the pre and post spike times. 132
6.12 Comparison between the simulation with and without STDP on SpiNNaker during 1st second and 30th second. 134
6.13 The STDP result from the Matlab simulation. 135
6.14 The behavior of an individual neuron (ID 1) during the 1st second and the 30th second. 135
6.15 Weight Distribution. 136
6.16 Group of neurons with converging delays. 137
6.17 Relationship between the firing rate and the timing records length. 137

7.1 MLP models and the weight matrix. 143
7.2 Computational phases of a MLP network with BP learning. . . . 145
7.3 The sigmoid function. 147
7.4 Checkerboarding partitioning. 149
7.5 Communication patterns. 149
7.6 Network topologies and SpiNNaker 151
7.7 Mapping on SpiNNaker using the pipelined model 153
7.8 The six-stage pipeline. 154
7.9 A performance comparison between the non-pipelined. 159
7.10 Efficiency comparison. 160
7.11 Performance comparison. 160
7.12 Efficiency comparison. 161

7.13 Performance comparison between non-pipelined model and pipelined model on 500 SpiNNaker chips. 165
7.14 Efficiency comparison. 165
7.15 Performance comparison 166
7.16 Efficiency comparison. 166
7.17 A recurrent neural network 167
7.18 The speed comparison of the fully-Connected networks on SpiNNaker and on the PC. 169
7.19 The speed comparison of the partially-connected networks on SpiNNaker and on the PC. 171

Abstract

Artificial neural networks have shown great potential and have attracted much research interest. One problem faced when simulating such networks is speed. As the number of neurons increases, the time to simulate and train a network increases dramatically. This makes it difficult to simulate and train a large-scale network system without the support of a high-performance computer system. The solution we present is a "real" parallel system – using a parallel machine to simulate neural networks which are intrinsically parallel applications.

SpiNNaker is a scalable massively-parallel computing system under development with the aim of building a general-purpose platform for the parallel simulation of large-scale neural systems. This research investigates how to model large-scale neural networks efficiently on such a parallel machine. While providing increased overall computational power, a parallel architecture introduces a new problem – the increased communication reduces the speedup gains. Modeling schemes, which take into account communication, processing, and storage requirements, are investigated to solve this problem. Since modeling schemes are application-dependent, two different types of neural network are examined – spiking neural networks with spike-time dependent plasticity, and the parallel distributed processing model with the backpropagation learning rule. Different modeling schemes are developed and evaluated for the two types of neural network. The research shows the feasibility of the approach as well as the performance of SpiNNaker as a general-purpose platform for the simulation of neural networks. The linear scalability shown in this architecture provides a path to the further development of parallel solutions for the simulation of extremely large-scale neural networks.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First and foremost, I would like to express my thanks to my supervisor, Steve Furber, for his time and help throughout my research. His advice always guided my research towards the correct direction.

I would like to thank everybody in the APT group, especially those involved in the SpiNNaker project. I have spent a wonderful time working and study together with them. Steve Temple helped me on problems related to the SpiNNaker hardware. He provided detailed technical and tools supports for my research. Luis Plana was so generous in giving advice on the algorithms that came into my mind. Mukaram Khan was the person whom I worked most closely with, and provided a great SpiNNaker model on SoC Designer allowing me to develop and verify the software model prior to the release of real SpiNNaker chip. Many thanks to Francesco Galluppi for the nice chat about neural networks and the hard work on the Doughnut Hunter application which helped me to improve my software. Alex Rast contributed a lot to my paper list. Cameron Patterson, Eustace Painkras, Zhenyu Liu, and Shoujie Li have kindly helped on many occasions in a variety of issues. Mikel Lujan and Viv Woods were so patient in reviewing and commenting on my writing, from the first publication to the thesis. I also wish to thank Jim Garside for his help with ARM programming as well as reviewing my final thesis. Tom Sharp and Sergio Davies have helped in the documenting and further development of the software. Thanks to Martin Grymel for introducing me so many new lunch places.

Special thanks go to Shufan Yang, Wu Jian, Yebin Shi, Wei Shao, Zhen Zhang and all my other friends, who have helped me in both research and daily life during my study.

I dedicate my thesis to my wife Wenjun, who is always with me to provide support, my son Shenxi for bringing me so much happiness, and my parents for their constant support and encouragement.

Chapter 1

Introduction

1.1 Background

Biological neural networks such as the human brain have shown a great capability in tasks such as face recognition, speech processing and logic reasoning which cannot be handled efficiently by conventional computers. Although it is still not yet clear how the brain works, the great potential of neural systems has aroused enthusiasm of many scientists since the last century. Based on knowledge about neural systems acquired to date, a variety of neuronal models have been developed to mimic neuron behaviors.

Nevertheless, what makes the biological neural system so special is the large number of neurons and the high level of interactions among them. The special functionality of a neural system is a result of interaction among all neurons in each certain region. A human being has billions of neurons and may lose about 20 percent of them during a lifetime while maintaining similar functionality. In addition to looking for more accurate neuronal models, neuroscientists also focus on the their inter-connections and topology. Many neural network models have been developed, for instance, the multilayer perceptron [Ros58], Hopfield neural networks [Hop82], Kohonen self-organizing networks [Koh95]; each of these models comprises a number of inter-connected neurons/units. Larger neural networks based on these models are being created to simulate more complex behaviors. Some phenomena of interest only emerge when running a long simulation using a large-scale neural network [IE08].

The operation speed of a biological neural network is almost independent of its scale. It is always operating in "real-time" irrespective of how big the systems

is. This is not the case with artificial neural networks however; when the scale of the artificial neural network increases, the simulation speed drops, and the performance varies from one solution to another. Even with the most computationally efficient neuronal model, the performance of the neural network is still not comparable to its biological alternative. One second of activity of the neural network can take minutes or hours of simulation on a computer depending on the scale of the network and the processing power of the machine. The situation is even worse when training such a neural network system using learning algorithms; the time required increases dramatically with the number of neurons. This limitation has been one of the bottlenecks of research in the field of artificial neural networks since its very beginning [MP69].

Many methods investigated to speed up simulation basically go in two directions:

1. Developing more computationally efficient neuronal models and training rules. This has been explored mostly by psychologists; the approach is a trade-off between simulation performance and precision. The Hodgkin-Huxley model introduced in 1952 [HH52], is probably the best known model for neurons. It is biologically plausible, yet the most sophisticated and computationally expensive model. It takes about 1200 floating-point operations to simulate 1ms of activity [Izh04]. The simplest model – the integrate-and-fire model – takes less time to compute (5 floating-point operations per 1ms), but it is less precise and can reproduce fewer firing patterns than the Hodgkin-Huxley model. This indicates that neuronal dynamics involves trade-off between speed and accuracy.

2. Using more computationally powerful hardware. Computer scientists usually follow this path. A lot of hardware architectures, from FPGAs to the IBM Blue Gene supercomputer, have been investigated both theoretically and experimentally to support the simulation of neural networks.

General-purpose supercomputer systems such as Blue Gene or Beowulf clusters are extremely computationally powerful and also easier to program than dedicated hardwired devices such as FPGAs. However, firstly, their standard communication systems are usually not efficient enough to meet the high communication demands of neural networks; and secondly, their large physical size and power consumption make them almost impossible to use in embedded neural

network applications such as robots. On the other hand, dedicated hardware such as FPGAs and VLSI implementations lack scalability and flexibility. Different applications have variable network sizes. For example, some simple applications such as a control system requires fewer neurons while others such as brain simulation require a larger population of neurons and more connections. Thus linear scalability can help to maintain constant simulation speed by expanding the size of the hardware when the scale of the neural network increases. As computational neuroscience is still developing, the best neuronal model has not yet been discovered. Different models, connection patterns and learning rules are investigated. Therefore, the neuromorphic hardware needs to be reconfigurable and general-purpose to support different neural applications. In this context, what is expected is dedicated neuromorphic hardware which is not hardwired and offers programmability in order to support a variety of neural network applications. Such a system should be efficient in computation and communication, yet low power, compact and scalable.

Neural networks (both biological and artificial) are naturally parallel. Neurons in such a system operate concurrently. Information is stored in a distributed way among synaptic connections. Most traditional computer systems are sequential. This fundamental difference in the structure is one of the most important reasons why it is so inefficient to simulate neural networks on traditional computers. This problem is not only with the speed but also with the memory requirements. As computer science research moves in the direction of multi-core systems, parallel architectures have become a very hot research topic. Our solution is to produce a "real" parallel system, that uses a parallel machine to simulate parallel neural networks.

1.2 Motivation

The SpiNNaker project is motivated by this background. The aim of the SpiNNaker project is to provide a scalable, massively-parallel, computing system as a general-purpose platform for the parallel simulation of large-scale neural systems.

The specification of SpiNNaker is an attempt to capture the possible features that ideal parallel machines should have for neural network simulation, as discussed above. Each SpiNNaker chip is a chip multiprocessor (CMP) containing

up to 20 ARM968 processors and other components such as a router, communication controllers, etc. There are two different types of memory system associated with each chip, and each processor has an internal RAM block called the tight-couple memory (TCM) which is fast but small, and a block of external SDRAM which is large in capacity but much slower. The TCMs provide instant access to application code and variables, while the SDRAM stores large data sets with comparatively low access rates, for example, the synaptic connectivity data. Each SpiNNaker chip also has 6 self-timed external links by which multiple chips can be linked together to expand the scale of the system and a multicast mechanism is provided for efficient one to many communications. Small chip area and low power consumption are also taken into account in the design.

This thesis focuses on the software design for the SpiNNaker system. There are many neural simulators, such as Brian [GB09], NEURON [NEU] and PCSIM [PNS09] that run on desktop computers to make neural simulation straightforward to users. With a dedicated parallel machine such as SpiNNaker, software support is necessary to enable users to run previous experiments on the SpiNNaker system as easily and efficiently as on a desktop computer. To develop such software, the most important problem to be solved is to map large-scale neural networks efficiently onto SpiNNaker. Parallel machines are much more complex in terms of their structure than their sequential alternatives and it is not straightforward to map an application onto such hardware. Workloads and information have to be distributed across multiple processors, and different types of neural networks, different distribution approaches may be required. To achieve high performance, the design of the software and its modeling algorithms must emphasize efficiency.

1.3 Research aims

In this thesis, we aim to investigate the development of modeling algorithms to run neural networks efficiently on SpiNNaker. Three aspects of the modeling: communication, computation and storage have been taken into account.

- Communication. The parallel architecture provides more computational power, but also increases the number of exchanges of internal information. Processors need to send partial results or outputs to other processors where information is needed to carry out the computation of the next step. In a

sequential system, the information is normally locally available, while in a parallel system, the information is distributed across different processors. In this thesis, a number of approaches are used to reduce the communication overheads. For spiking neural networks, we keep synapses information in the memory of the postsynaptic processors, and use event-address mapping to locate the synaptic block. When presynaptic neurons fire, they just send identical information to all the postsynaptic neurons; this is handled very efficiently by the multicast mechanism. For multi-layer perceptron networks, we developed a pipelined checkerboarding partitioning scheme to reduce the communication overheads by overlapping the computation and communication.

- Computation. To speed up the computation, assembly code is used for all critical processing. Fixed-point arithmetic is used during the implementation, and a dual-scaling factor scheme is developed to reduce the loss of precision.

- Storage. The methods used to store the information are developed. Neuronal and synaptic information are distributed to different processors or chips. The data structure is organized very carefully and a compressed storage scheme is used to save space.

In addition, other features such as scalability and power-efficiency are also taken into consideration during the investigation.

Being general-purpose neuromorphic hardware, SpiNNaker is designed to support multiple types of neural network. In addition to implementing the spiking neural networks that SpiNNaker is mostly focused towards, the modeling of another very different type of neural network is also investigated, the multilayer perceptron (MLP). These two different types of neural network require different modeling schemes. In the spiking neural network case, real-time performance for large-scale networks is required. In the MLP case, as there is no explicit concept of time, optimum performance is pursued and the efficiency of the modeling scheme is analyzed semi-experimentally.

This thesis describes investigations into an approach to building low-level software support for top-level neural network applications. By implementing a prototype software system and running a very simple neural network application, the design flow of developing software for SpiNNaker is learned, which leads to

the proposal of a layered software model to minimizes the differences between running a neural network on SpiNNaker and running it on simulators based on desktop computers. The research not only focuses on solving the problems faced during the implementation, but also discovers many potential issues leading to further studies.

1.4 Contributions

In this thesis, two different modeling schemes and several sub-models are developed and evaluated for two types of neural network respectively.

- Modeling schemes are developed for simulating spiking neural networks with spike-time dependent plasticity (STDP), including: a dual-scaling factor scheme for fixed-point arithmetic, event-address mapping, an event-driven scheduler, route planning, neural data to SpiNNaker data conversion and a deferred model for implementing STDP. The work also involves the practical implementation, verification and evaluation of these schemes. The software built on SpiNNaker for modeling spiking neural network is tested, and the outputs match the results from Matlab simulations. The functionality is further verified by successful running of a practical neural network application (the Doughnut Hunter) on the *physical* SpiNNaker Test Chip.

- A novel pipelined checkerboarding scheme is developed for modeling multilayer perceptron networks with the backpropagation rule. The performance is estimated for both fully- and partially-connected networks, and is compared with the performance on a single PC simulation.

Though the modeling schemes are application-specific, common principles can still be found which minimize communication, optimize processing and maintain efficient storage. The difficulties found during this research are mostly common problems faced when developing other parallel solutions, hence solutions provided can be generalized.

The research shows the feasibility as well as the performance of simulating neural networks on the SpiNNaker parallel machine. Equally important, the research illustrates many unsolved problems involved in the parallel modeling and initializes discussions about these issues. The work in this thesis provides a path

to the further development of parallel solutions for the simulation of extremely large-scale neural networks.

1.5 Thesis overview

The rest of the thesis is organized as follows:

1. Chapter 2 gives a brief introduction to the modeling theory of spiking neural networks.

2. Chapter 3 reviews the current engineering solutions for simulating large-scale neural networks, and introduces the SpiNNaker system we developed.

3. Chapter 4 presents the approaches for building a neural system for the Izhikevich model on a single ARM968 processor. This includes the approaches for modeling neuronal dynamics, neural representations, synaptic delays and system scheduling. The system is tested by running a small network, and the performance is evaluated.

4. Chapter 5 extends the previous single processor system to a multi-chip system. A four-chip SpiNNaker model based on the SoC Designer is implemented and tested. The Doughnut Hunter application is tested on a physical SpiNNaker Test Chip.

5. Chapter 6 implements the spike-timing-dependent plasticity (STDP) rule on SpiNNaker.

6. Chapter 7 investigates the feasibility and performance of modeling multilayer perceptron networks on SpiNNaker.

7. Chapter 8 concludes the thesis.

1.6 Publications

1. X. Jin, M. Lujan, M. Khan, L. Plana, A. Rast, S. Welbourne and S. Furber. "Algorithm for Mapping Multilayer BP Networks onto the SpiNNaker Neuromorphic Hardware". In Proc. 9th International Symposium on Parallel and Distributed Computing (ISPDC 2010) Istanbul, Turkey, July, 2010.

This paper presents algorithm and performance analysis for mapping multilayer BP networks onto SpiNNaker. This corresponds to Chapter 7.

2. X. Jin, A. Rast, F. Galluppi, S. Davies and S. Furber. "Implementing Spike-Timing-Dependent Plasticity on SpiNNaker Neuromorphic Hardware". In Proc. 2010 International Joint Conference on Neural Networks (IJCNN 2010) Barcelona, Spain, July, 2010.

   This paper presents very detailed STDP implementation on SpiNNaker. This corresponds to Chapter 6.

3. X. Jin, F. Galluppi, C. Patterson, A. Rast, S. Davies, S. Temple and S. Furber. "Algorithm and Software for Simulation of Spiking Neural Networks on the Multi-Chip SpiNNaker System". In Proc. 2010 International Joint Conference on Neural Networks (IJCNN 2010) Barcelona, Spain, July, 2010.

   This paper presents the multi-chip simulation of spiking neural networks. This corresponds to Chapter 5.

4. X. Jin, M. Lujan, M.M. Khan, L.A. Plana, A.D. Rast, S.R. Welbourne and S.B. Furber. "Efficient Parallel Implementation of Multilayer Backpropagation Network on Torus-connected CMPs". Proc. of the ACM International Conference on Computing Frontiers, Bertinoro, Italy, May, 2010.

   This is a poster which gives a brief introduction to the algorithm for mapping multilayer perceptron networks on SpiNNaker. This corresponds to Chapter 7.

5. X. Jin, S. Furber, J. Woods. "Efficient modelling of spiking neural networks on a scalable chip multiprocessor". In Proc. 2008 International Joint Conference on Neural Networks, Hong Kong, 2008.

   This paper presents the scheme of modeling spiking neural networks on the SpiNNaker as well as performance results. This corresponds to Chapter 4 of this thesis.

6. X. Jin, A. Rast, F. Galluppi, M. Khan, and S. Furber. "Implementing Learning on the SpiNNaker Universal Neural Chip Multiprocessor". In Proc. 16th Intl. Conf. on Neural Information Processing (ICONIP2009), December 1-5, 2009, Bangkok, Thailand.

This paper presents the approach of modeling STDP rule on SpiNNaker. This corresponds to Chapter 6 of this thesis.

7. M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E. Painkras, and S. Furber. "SpiNNaker: Mapping Neural Networks onto a Massively-parallel Chip-multiprocessor". In Proc. Intl. Joint Conf. on Neural Networks (IJCNN2008), 2008 June 1-6, Hong Kong.

   In this paper, my contribution is in "Section IV: mapping neural networks" (about route planning and lookup table generation for spiking neural networks, corresponding to Section 5.2 of this thesis) and "Section VI: Implementation" (about mapping MLP networks, corresponding to Section 7.4 of this thesis).

8. M. Khan, J. Navaridas, X. Jin, L. Plana, J. Woods, and S. Furber. "Real-time Application Support for a Novel SoC Architecture". In Proc. 4th UK Embedded Forum, Southampton, UK, September 2008.

   I contributed to the work described in "Section III: Application Model". This corresponds to Section 4.6 of this thesis.

9. M. Khan, X. Jin, S. Furber, and L. Plana. "System-level Model for a GALS Massively Parallel Multiprocessor". In Proc. 19th UK Asynchronous Forum, page 19-22, September 2007.

   I contributed to "Section IV: Running neural application on the Model", giving performance results on modeling MLP networks. This corresponds to Section 7.6.

10. M. Khan, J. Navaridas, A. Rast, X. Jin, L. Plana, M. Lujan, J. Woods, J. Miguel-Alonso and S. Furber. "Event-Driven Configuration of a Neural Network CMP System over a Homogeneous Interconnect Fabric". In Proc. of Intl. Symposium on Parallel and Distributed Computing (ISPDC2009) 1-3 July 2009, Lisbon, Portugal.

    I contributed to "C. The SpiNNaker Execution Model", corresponding to Section 4.6.

11. A. Rast, X. Jin, F. Galluppi, L. Plana, C. Patterson and S. Furber. "Scalable Event-Driven Native Parallel Processing: The SpiNNaker Neuromimetic

System" In Proc. of the ACM International Conference on Computing Frontiers, Bertinoro, Italy, May, 2010.

I contributed to the modeling and experimental work in this paper.

12. A. Rast, F. Galluppi, X. Jin and S. Furber. "The Leaky Integrate-and-Fire Neuron: A Platform for Synaptic Model Exploration on the SpiNNaker Chip". In Proc. 2010 International Joint Conference on Neural Networks (IJCNN 2010) Barcelona, Spain, July, 2010.

    I contributed to the modeling and experimental work in this paper.

13. A. Rast, X. Jin, M. Khan, S. Furber, "The Deferred Event Model for Hardware Oriented Spiking Neural Networks". In Proc. 15th Intl. Conf. on Neural Information Processing (ICONIP2008), 2008 Nov. 25-28, Auckland, New Zealand.

    I contributed to "3.2 Neuron" (Corresponding to Sections 4.4 and 4.5) and "4 Application to system modelling" (Corresponding to Section 4.7)

14. A. Rast, M. Khan, X. Jin, L. Plana and S. Furber, "A Universal Abstract-Time Platform for Real-Time Neural Networks". In Proc. of Intl. Joint Conf. on Neural Networks (IJCNN2009), 2009 June 14-19 Atlanta Georgia (USA).

    I contributed to "Performance and functionality testing". This corresponds to Section 4.7.

15. A. Rast, S. Furber, D. Lester, S. Temple, L. Plana, E. Painkras, M. Khan, J. Wu, Y. Shi, S. Yang and X. Jin, "Abstracting both Architecture and Time: The SpiNNaker Neuromimetic Modelling Platform". In Proc. ESS-DERC2008, 15-19 Sep, Edinburgh.

16. A. Rast, S. Welbourne, X. Jin and S. Furber. "Optimal Connectivity In Hardware-Targetted MLP Networks". In Proc. 2009 International Joint Conference on Neural Networks, IJCNN2009, pp.2619-2626 Atlanta. Georgia, 14-19 June, 2009.

    I contributed to "II. SPINNAKER HARDWARE MAPPING". This corresponds to Section 7.4.

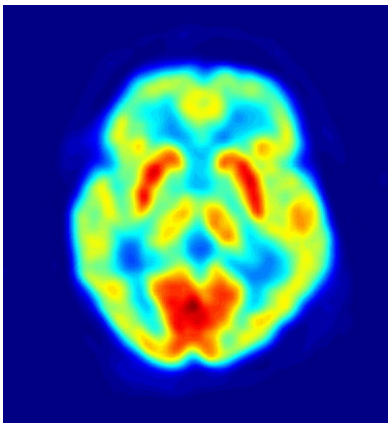
Chapter 2

Neural network modeling

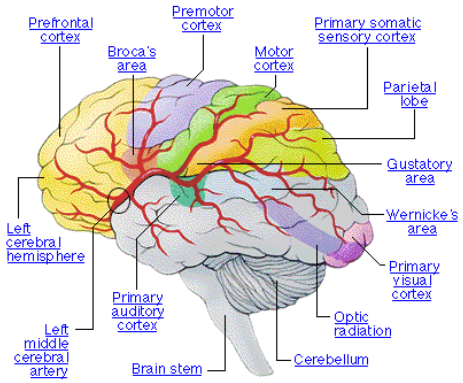
2.1 Overview

Computer programming, poetry and music composition, or even the Mass-Energy equation, are all products of the brain which is highly developed after millions of years of evolution. Now the brain is trying to mimic itself – to build an artificial version with same functionality. This looks more like a philosophy question – whether a creature can reproduce itself. Despite whatever answer a philosopher will give to this question, scientists are heading towards this objective and trying to find the answer themselves. Research into neural biology has been carried out for centuries. Although there are lots of mysteries remaining in this area, a lot of knowledge and understanding of the brain and biological neural networks have been developed.

Biological neuroscientists have discovered the electrochemical processes of neurons in the brain, their connectivity patterns and so on. This knowledge is used by computational neuroscientists to build artificial models. Mathematical approaches are used to create a variety of models that describe brain activities. In addition to the neuronal dynamic models, there are also modeling theories about neural coding, connectivity, and learning. This chapter gives a brief overview of the biological research as well as computational modeling of neural networks.



(a) An image taken from a typical PET acquisition by Jens Langner [Lan03].



(b) Division of brain function, by Leslie Laurien.

Figure 2.1: The brain

2.2 Biological neural networks

2.2.1 Research methodology

Since the last century, people have been seeking to understand the brain using a variety of approaches. The first approach was anatomical using stains and microscopes. With this technology neuroanatomists have been successful in revealing neural structures and connectivity.

Later electrophysiology methods were introduced which allowed scientists to observe the electrical activity of neurons. The two general approaches used are intracellular and extracellular recordings. As the names indicate, the intracellular technique is used to record electrical signals from the interior of a neuron with the help of glass electrodes. Tissues from brains of animal bodies are used for this research; this is invasive and could damage the tissue. The extracellular technique is used for capturing the electrical signals outside the cellular body, such as the action potential (spike) in the axon or synaptic connections. Compared to the intracellular technique, the extracellular technique is less invasive, thus can be used in living animals or even in humans.

More advanced techniques, such as Magnetic Resonance Imaging (MRI), Electroencephalography (EEG), Voltage-Sensitive dyes (VSDs), Positron Emission Tomography (PET) and Intrinsic Optical Imaging (IOI), are now in use for research. These techniques allow the observation of both functional processes and

the internal structure of the brain in two or three dimensional images with less impact on the objects which are being observed. The new techniques have helped to investigate phenomena previously unobserved, and hence moved neuroscience research to a new level. Figure 2.1(a) shows a picture taken using the PET technique.

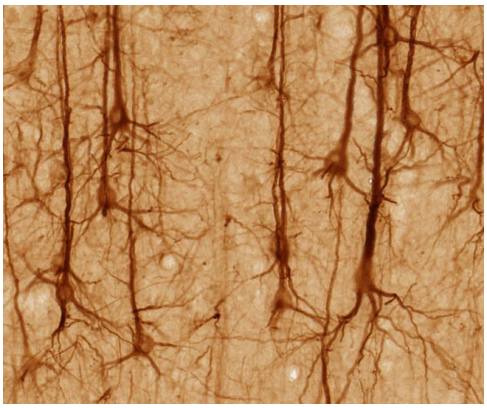
2.2.2 The brain and cortex

So what has been discovered by these observations? Although exactly how the brain works remains a mystery, neuroscientists have uncovered many interesting phenomena as well as brain structures. Basically, the most important three parts in the brain are: the brainstem, cerebellum, and cerebrum with both the left and right cerebral hemispheres [AC81, Ecc73, NMW92]. The brainstem is the lowest part of the brain and is connected to the spinal cord, which provides the main motor and sensory innervations to the face and neck. The cerebellum is the structure located behind the brain stem, responsible for the integration of sensory perception, coordination and motor control. The cerebral hemispheres of the cerebrum are the largest part of the brain, with an outer layer of the cerebral cortex and an inner layer of the white matter. The cortex, about two millimeters thick with a total surface area of about 1.5 square-meters, is the source of most of the high-level functions of the mind, and hence is the focus of most research in neuroscience.

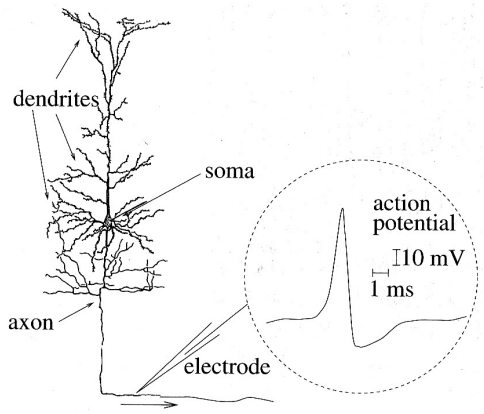
Functions in the cortex are virtually distributed in several specific regions, as shown in Figure 2.1(b). For instance, the motor cortex, comprised of several sub-regions, is responsible for planning and controlling muscle movements, while the main responsibility of the primary visual cortex is to process visual information about static and moving objects as well as pattern recognition. The frontal lobes respond to inputs from the other regions of the brain and are mainly responsible for making decisions and judgments.

2.2.3 Neurons

What is in the cortex that makes it versatile and so powerful? The answer was found quite a long time ago. The investigation of the microscopic structure of the brain started very early in around 1900 [Caj02], when scientists found that the key components in the cortex are neuron cells and their inter-connections.



(a) Neurons and connections found in neuronal morphologies of cerebral cortex. From http://brainmaps.org



(b) A single neuron drawn by Ramon y Cajal [Caj02]

Figure 2.2: Neuronal structure

Figure 2.2(a) shows a very small portion of the cortex comprising several neurons and connections. There are about 100 billion (10^{11}) neurons in total in the brain. They are connected with each other according to rules which we still do not yet fully understand. Each individual neuron is physically much like other cells in our body. However, it has its own unique character, which is essential to the functioning of the nervous system. More specifically, a neuron cell is different in that it has interaction with other neurons by receiving or sending electrical pulses (spikes).

A neuron comprises three parts: the dendrites, the soma (the cell body) and the axon, as shown in Figure 2.2(b).

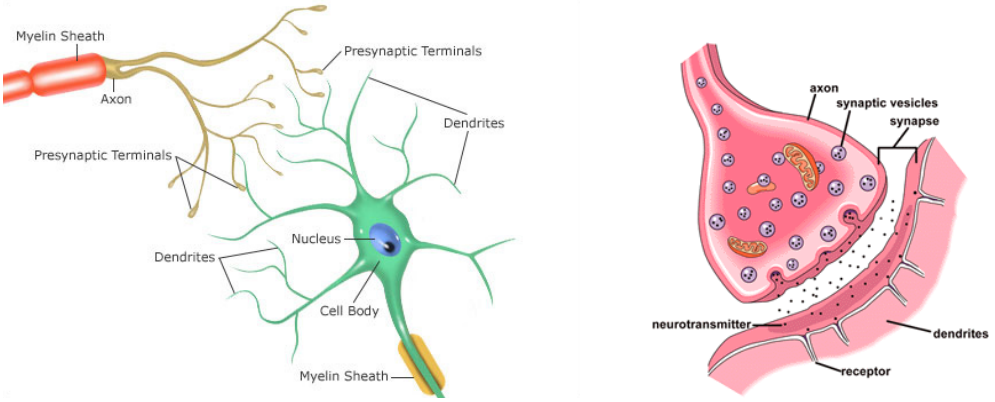
- The dendrites are short, branching fibres extending from the soma. They collect input electrical pulses from pre-synaptic neurons via synapses and transmit them to the soma. The dendritic branching of a neuron is changeable in the development of the nervous system, either growing or retracting, which we call "plasticity". The dendrites are the input devices of a neuron.

- The soma, containing the nucleus and other common cell tissues, is a key component of a neuron, whose size ranges from 0.005 mm to 0.1 mm, depending on its type. The soma is the place where the electrochemical progress occurs and electrical pulses are generated. It is the a central processing unit of a neuron.

- The axon is the output device of a neuron, which sends electrical pulses from the soma to other neurons. It is longer and thicker than the dendrites. Synapses may appear partway along an axon as it extends but most of them appear as terminals at the ends of axonal branches.

There are several different types of neuron, ranging both in size and shape. Different types of cortical neurons can generate different types of firing patterns which can be distinguished by several classes and sub-classes [CG90], [GBC99], [NASV+03].

2.2.4 Synapses



(a) (b)

Figure 2.3: The connection and synapse.

The synapse is the junction between an axon terminal of the pre-synaptic neuron and the dendrite of the post-synaptic neuron (shown in Figure 2.3(a)). The human brain is estimated to contain more than 10^{15} synapses, and on average, each neuron is connected to about 10,000 other neurons through synapses (The actual number varies greatly, depending on the type of the neuron).

The detail of a synapse is shown in Figure 2.3(b). The electrochemical process involved in its operation is complicated; when an electrical pulse arrives at the axon terminal, neurotransmitters stored in synaptic vesicles are activated and information is transmitted across the synaptic cleft to receptors at the post-synaptic dendrites. The receptors are then activated by the neurotransmitters, which results in the activation of certain ion channels. There are a number of different neurotransmitters and neuroreceptors, resulting in different types of synapses.

The two most common types of synapses are excitatory and inhibitory. The excitatory synapses have neuroreceptors with sodium channels [Hil01]. An incoming positive ion causes a depolarization on the postsynaptic membrane potential making an action potential more likely. The inhibitory synapse has neuroreceptors with chloride channels. The incoming negative ions cause a hyperpolarization on the postsynaptic membrane potential making an action potential less likely. The complicated electrochemical processes can be studied quantitatively using conductance-based equations.

Synapses are adaptive. The neural system learns using certain rules which alter the strengths of connections, and by growing new or deleting existing connections between neurons. Plasticity plays an important role in the development of the neural system, and it is one of the key ways in which a biological system differs from our engineered computing systems.

2.3 Mathematical neural modeling

2.3.1 Neuron electrophysiology

Neuronal activity is a result of ionic movement around the membrane of the cell body. There are basically four types of ions involved: sodium (Na^+), potassium (K^+), calcium (Ca^2+), and chloride (Cl^-). Their concentrations are different inside and outside of a cell, and their movements are driven by electrochemical gradients, see Figure 2.4.

There are two types of electrochemical gradient: concentration and electric potential gradients. These two forces drive ions in opposite directions, towards either the inside or the outside of the cell. When the ionic concentration and the electric potential gradient are equal and opposite, they counterbalance each other, and as a result, an equilibrium point is achieved and the net cross-membrane current is zero. The value of the equilibrium potential, which varies for the different ionic species, is given by the Nernst equation [Hil01]. The equilibrium potential is thus the electric potential caused by the concentration difference.

The cell membrane separates the interior of the cell from the extracellular space, and ions can flow through protein channels in the membrane according to their electrochemical gradients. The electric potential difference between the inside and outside of the membrane is called the membrane potential. Assuming

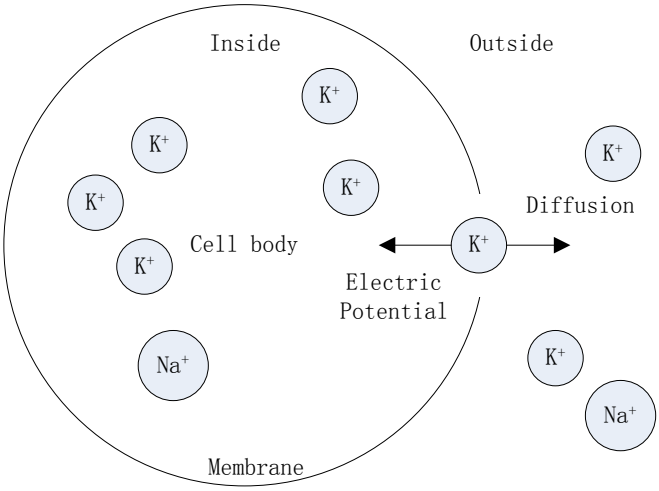


Figure 2.4: Diffusion of K^+ ions down the concentration gradient though the membrane.

the membrane potential is V, the equilibrium potential (Nernst potential) of K^+ is E_K and the net K^+ current is I_K ($\mu A/cm^2$), we have:

$$I_K = g_K(V - E_K) \tag{2.1}$$

where the positive parameter g_K is the K^+ conductance. As indicated in Equation 2.1, K^+ ions are driven by the difference between the membrane potential V and the equilibrium potential E_K; the same equation also applies to other ions.

The electrical properties of membranes can be represented by the equivalent circuits shown in Figure 2.5. According to Kirchhoff's law, we have:

$$I = C\,\dot{V} + I_{Na} + I_{Ca} + I_K + I_{Cl} \tag{2.2}$$

or in the standard dynamical system form:

$$C\,\dot{V} = I - I_{Na} - I_{Ca} - I_K - I_{Cl} \tag{2.3}$$

where I is the total current; C is the membrane capacitance ($C \approx 1.0\mu F/cm^2$), and $\dot{V} = dV/dt$ is the derivative of the voltage variable V with respect to time t. If there are no additional current sources such as synaptic current or current injections via an electrode, then $I = 0$.

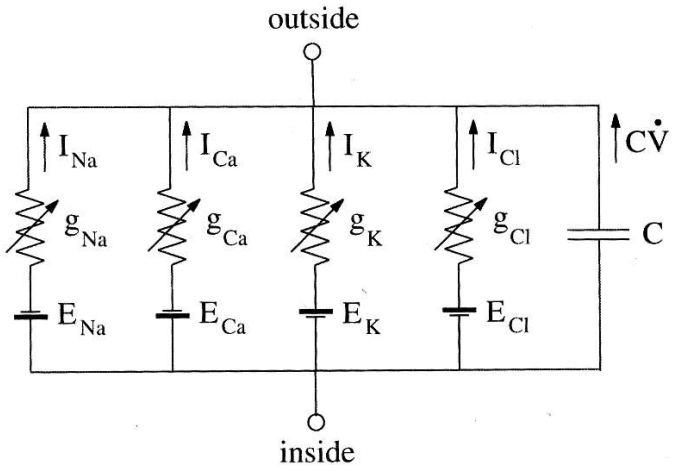


Figure 2.5: Equivalent circuit representation of cell membrane by Izhikevich [Izh07].

2.3.2 The Hodgkin-Huxley model

Hodgkin and Huxley performed a series of experiments on the giant axon of the squid and succeeded in measuring ion currents and described their dynamics by a set of nonlinear differential equations [HH52]. This is one of the most important qualitative models in computational neuroscience. Three types of current are taken into consideration [Izh07]: The K^+ current with four activation gates (resulting in the term n^4), the Na^+ current with three activation gates and one inactivation gate (resulting in the term m^3h), and the Cl^- Ohmic leak current (note that most neurons in the central nervous system have additional currents). The complete Hodgkin-Huxley equations are:

$$\begin{aligned} C\dot{V} &= I - g_K n^4 (V - E_k) - g_{Na} m^3 h (V - E_{Na}) - g_L (V - E_L) \\ \dot{n} &= \alpha_n(V)(1-n) - \beta_n(V) n \\ \dot{m} &= \alpha_m(V)(1-m) - \beta_m(V) m \\ \dot{h} &= \alpha_h(V)(1-h) - \beta_h(V) h \end{aligned} \tag{2.4}$$

where

$$\begin{aligned}
\alpha_n(V) &= 0.01\frac{10-V}{\exp(\frac{10-V}{10})-1} \\
\beta_n(V) &= 0.125\exp(\frac{-V}{80}) \\
\alpha_m(V) &= 0.1\frac{25-V}{\exp(\frac{25-V}{10})-1} \\
\beta_m(V) &= 4\exp(\frac{-V}{18}) \\
\alpha_h(V) &= 0.07\exp(\frac{-V}{20}) \\
\beta_h(V) &= \frac{1}{\exp(\frac{30-V}{10})+1}
\end{aligned} \tag{2.5}$$

The three variables n, m, and h are activation gates (or gating variables) for K^+ , Na^+, and Cl^-, respectively [GK02], [Izh07]. The g_K, g_{Na}, and g_L are the conductance variables. The membrane capacitance is $C = 1.0\mu F/cm^2$ and the applied current is $I = 0\mu A/cm^2$. The parameters that Hodgkin and Huxley used were based on a voltage scale that was shifted by approximately 65 mV, making the resting potential zero for convenience. The shifted Nernst equilibrium potentials are:

$$E_k = 12mv, \quad E_{Na} = 120mv, \quad E_L = 10.6mv \tag{2.6}$$

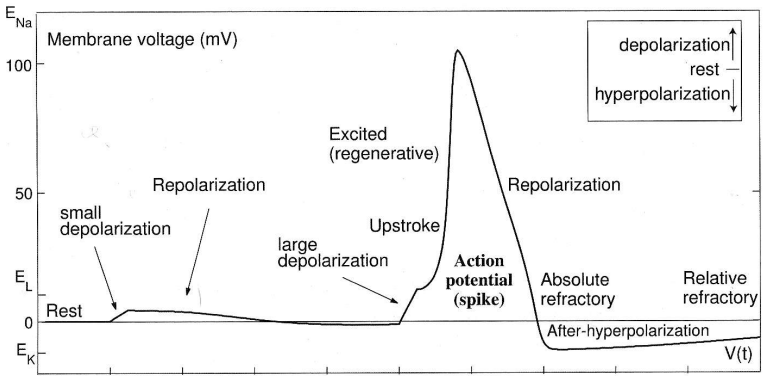
and the typical values of the conductances are:

$$g_K = 36mS/cm^2, \quad g_{Na} = 120mS/cm^2, \quad g_L = 0.3mS/cm^2 \tag{2.7}$$

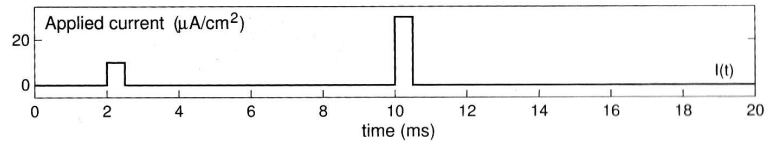
Now we look into the dynamics the Hodgkin-Huxley model to see how an activation potential (spike) is generated. When the membrane potential V equals its rest value V_{rest} ($0mV$ in the Hodgkin-Huxley model and about -65mV in reality). All types of currents balance each other and the rest state is stable.

Dynamics

When a small pulse of current I is applied as shown in Figure 2.6(b), the membrane potential V rises. This causes the variable m to be increased, hence increasing the conductance of the sodium (Na^+) channels. The influx of positive sodium currents to the cell body then pushes the membrane potential even higher. The effect of the input current I is thus amplified significantly, causing rapid increase of V. If the membrane potential is not big enough to generate a spike, only a positive perturbation of the membrane potential (a small depolarization) is produced as shown in Figure 2.6(a). This small depolarization is immediately pulled back



(a) Membrane potential.



(b) Applied current.

Figure 2.6: Action potential generation in the Hodgkin-Huxley model [Izh07].

to the resting value by a small net current. If the amplitude of input current I is much larger, a *spike* is generated. The sodium conductance is shut off due to the effect of h, when the membrane potential is high. The outflow of potassium (K^+) currents then pulls down the membrane potential V. In this case, the ongoing outflow of K^+ currents causes V to go below its rest value V_{reset}, which is called the *after-hyperpolarization* progress. This is followed by an *absolute refractory period*, which prevents the system from producing another spike, because the Na^+ currents are still depressed and take time to recover. After a long *relative refractory period*, the membrane potential V goes back to its rest value and the system reaches a new stable state. Generally speaking, the duration of a spike is about 1 ms and the amplitude is about 100 mv (based on a rest value of 0 mv). More detailed analysis of the model dynamics can be found elsewhere in [GK02, Izh07].

2.3.3 The Integrate-and-fire model

The detailed high-dimensional Hodgkin-Huxley model is biological plausible, but is complex to analyze and difficult to implement in hardware. As a result, simplified models are desired. As a first step, the objective is to reduce the four-dimensional Hodgkin-Huxley model to a two-dimensional model.

The key idea of the reduction is to eliminate two of the four variables in the

Hodgkin-Huxley model. This is based on two qualitative observations [GK02]. Firstly, in the Hodgkin-Huxley model, the time scale of the dynamics of the activation gate m is much faster than others variables n, h, and V. As a result, m can be treated as an instantaneous variable and can therefore be replaced by its steady-state value m_0; this is called a *quasi steady-state approximation*. Secondly, the n and h in the Hodgkin-Huxley model can be replaced by a single effective variable, since their time scales are roughly the same [GK02]. Based on these assumptions, several two-dimensional models have been proposed, such as the Morris-Lecar model and the FitzHugh-Nagumo model.

These models are conductance-based models, in which the variables and parameters have well-defined biological meanings, and can be measured experimentally. However, the conductance-based models are still complex to analyze. The simple phenomenal models, on the other hand, are not biological meaningful, but address most key properties of neurons and are less computationally intensive. The three key properties of a neuron that a phenomenological model usually addresses are:

- The ability to generate spikes when the membrane potential crosses a well-defined threshold.
- A reset value to initialize the membrane potential after firing.
- A certain refractory period to depress the neuron from generating another spike immediately.

Phenomenal models, which capture these features, are easier to implement and analyze, hence they are more popular in computational neuroscience. Among them, the leaky integrate-and-fire (LIF) model [Ste67, Tuc88] as well as its generalized versions (such as the nonlinear integrate-and fire model) are probably the best know spiking neuronal models. A schematic diagram of the LIF model is shown in Figure 2.7. It is an integrate-and-fire model with a "leak" term added to the membrane potential to solve the memory problem. The basic circuit of the LIF model is comprised of a capacitor C in parallel with a resistor R driven by a current I. Based on the circuit, we have:

$$I = \frac{V}{R} + C\frac{dV}{dt} \tag{2.8}$$

If we introduce a time constant $\tau_m = RC$ of the "leaky integrator", we get a

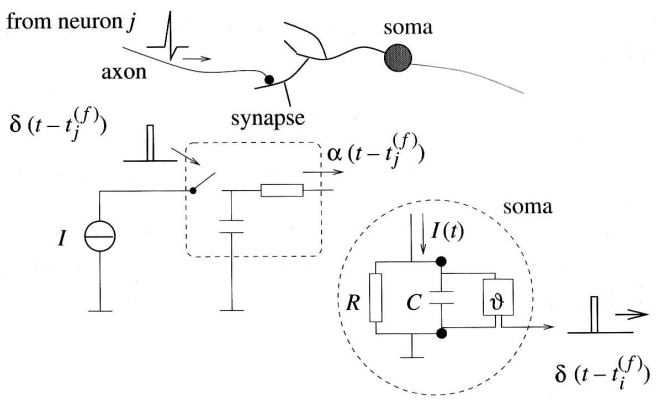


Figure 2.7: The schematic diagram of the leaky integrate-and-fire model by Gerstner [GK02].

standard form of the LIF model:

$$\tau_m \frac{dV}{dt} = -V + RI \tag{2.9}$$

where V is the membrane potential and τ_m is the membrane time constant. In this model, if the membrane potential V reaches the threshold value V_{thresh}, the neuron fires and then V is reset to a certain value V_{reset}. In the general version the LIF model also incorporates an absolute refractory period t_{abs}. If the neuron fired at time t, we stop the neuron dynamics for a period of t_{abs} and start the dynamics again at time $t + t_{abs}$ with $V = V_{reset}$. The LIF model is simple enough to implement and easy to analyze. However, it has a severe drawback - it is too simple to reproduce the versatile firing patterns of real neurons [Izh04].

2.3.4 The Izhikevich model

Another important phenomenal model is the Izhikevich model [Izh03]. This uses the *bifurcation theory* to reduce the high-dimensional conductance-based model to a two dimensional system with a fast membrane potential variable v and a slow membrane recovery variable u [Izh07]. The Izhikevich model is based on a

pair of coupled differential equations:

$$\begin{aligned} &\dot{v} = 0.04v^2 + 5v + 140 - u + I \\ &\dot{u} = a(bv - u) \\ &if \quad v \geq 30mV, \quad then \quad v = c, u = u + d \end{aligned} \tag{2.10}$$

where $\dot{v} = dv/dt$, t is time in ms, I is the synaptic current (in mV), v represents the membrane potential (in mV). u represents a membrane recovery variable (also in mV), which reflects the negative effects on the membrane potential caused by some factors such as the active of K^+ and the inactive of Na^+ ionic current. a, b, c , and d are adjustable parameters:

- a is the time scale of the recovery variable u. Smaller values results in slower recovery. A typical value is $a = 0.02$.
- b is the sensitivity of the recovery variable u to the membrane potential v. Greater values couple v and u more strongly. A typical value is $b = 0.2$.
- c is the after-spike reset value of the membrane potential v. A typical value is $c = -65$ mV.
- d is the after-spike offset of the recovery variable u. A typical value is $d = 2$.

It should be noted that the threshold value of this model is typically between -70 mV and -50 mV and is dynamic. In this model, when the membrane potential v exceeds the threshold value, the neuron spikes with a 30 mV apex of membrane potential v. The membrane potential v is limited to 30 mV. If the membrane potential v goes above the limitation, it is firstly reset to 30 mV. Then the membrane potential v and the recovery variable u are both reset according to equation 2.10.

There are two important features that make this model ideal for the real-time simulation of a large-scale network. Firstly, the Izhikevich model is computationally simple compared to the Hodgkin-Huxley model in that it takes only 13 floating-point operations to simulate 1 ms of modeling (with 1 ms resolution), but can reproduce firing patterns of all known types of cortical neuron. In comparison, the Hodgkin-Huxley model takes 1200 floating-point operations for 1 ms of modeling. Secondly, one of the most important advantages of the Izhikevich

model over the leaky integrate-and-fire model is that the former is capable of reproducing rich firing patterns. With the choice of neuron parameters a, b, c, and d, the Izhikevich model can generate all six known classes of firing pattern [Izh04].

2.3.5 Non-spiking neural models

The Hodgkin-Huxley, leaky integrate-and-fire and Izhikevich models are all spiking neural models. The output of a spiking model is a series of spikes called a spike train. The amplitude of spikes is not important. What really carries information is the time when a spike is generated. In addition to spiking neuronal models, there are also non-spiking neuronal models, which do not emphasize the importance of timing. These models adopt a continuous non-linear transfer function (usually a sigmoid function) to convey the input activation value to a real-valued output usually ranging from 0 to 1 depending on the strength of the activation value. An example of a non-spiking neural model is the well-known multi-layer perceptron network, but there are also simpler non-spiking neural models such as the Mcculloch-Pitts neuron model [MP43] which gives only a binary output of either 0 or 1 to indicate whether the neuron is active or inactive. Unlike the pulse a spiking neural model generated when a neuron fires, the output from a Mcculloch-Pitts neuron model is more like a step signal which switches between "on" and "off" states.

The non-spiking neural models have been well-studied and widely used as a traditional neural model. The outputs are seen as firing rates. It is widely agreed that spiking neural models are more biologically realistic than non-spiking neural models. Spiking models can easily be encoded as rate-based models, while non-spiking models, obviously, are unable to capture timing features. As a result, more research is now directed towards spiking models.

2.3.6 Neural coding

The question directly following the discussion of spiking and non-spiking neural models, is how information is actually coded in the brain or nervous system? This is fundamentally important for neural modeling, but as yet it is still not fully answered.

The most explored coding scheme is *rate coding* which measures the mean

firing rate of neurons based on a temporal average. It is not surprising that rate coding has been used so frequently, as it is easy to measure experimentally. However, rate coding is criticized because it neglects the timing information contained in the spike trains. Evidence for the importance of spike time is the very fast human response (about 400 ms [TFM96]) from receiving visual inputs to giving reaction outputs. The whole response procedure relies on several processing steps in the nervous system. The time for this reaction is not enough for the statistical results of firing rate, which rely on estimating temporal averages, to emerge. This discovery indicates that rate coding is not the only way information is coded in the brain.

Meanwhile, other research reveals that precise timing does play an important role in neuronal activities [BRSW91, Sin95, Les95]. These works point towards the so-called *spike coding* scheme which takes into consideration the precise timing of spikes. *Rank-order* codes, for example, use the order of firing of the neurons for information processing. They have been successfully used to recover images from sensory inputs. Another example is the synchrony behavior – the phenomenon that several neurons fire synchronously in timing-locked patterns. It has been observed experimentally [Sin95, KS92, GKES89] and implemented on computational neural models [GFvH94, RS97, Izh06]. A group of synchronously firing neurons with a specific temporal relationship may indicate a certain stimulus condition. Another group with a different temporal relationship may indicate a different stimulus condition. In this way, the input information can be encoded.

2.3.7 Connectivity patterns

The versatile functionalities of the brain are direct consequences of its circuitry, and how neurons are connected is a crucial issue of neural modeling. The connectivity issue is also closely related to the neural coding problem. Information is coded and propagated within the neural network through structural links such as synapses and fiber pathways. Accurate brain modeling requires not only the neuronal dynamics but also a comprehensive map of structural connection patterns in the human brain – the connectome [STK05].

Anatomically, the grey matter areas of the brain, comprising nerve cells, are connected and communicate through action potentials by axons. Axons are grouped into bundles and located in the white matter of the brain. The total length of axons decays with age. Males have a total axon length of 176,000 km

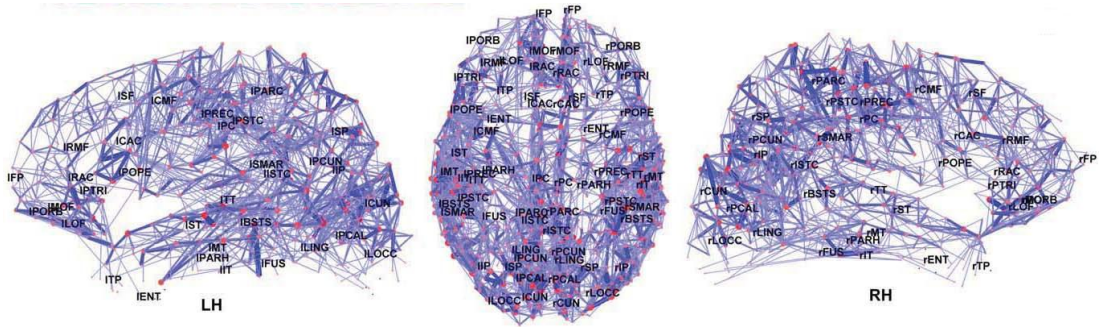


Figure 2.8: Dorsal and lateral views of the brain connectivity backbone from [HCG+08].

at the age of 20 and 97,200 km at the age of 80 [MNTP03]. The structure of the connections is very complicated: different regions of the cortex have different types of connection pattern; each region has a number of afferent inputs from other cortical regions and in return delivers a large number of output projections. The large scale and complexity of the networks make it extremely difficult for researchers to understand their organization.

Brain activity is measured using technologies such as functional magnetic resonance imaging (fMRI) and diffusion spectrum imaging (DSI). Quite a lot of knowledge about neural connectivity has been learned through experiments. Basically, there are several elemental connectivity patterns which recur frequently throughout the brain. These include convergent, divergent, reciprocal, local inhibitory, and topographic connections [TM07]. The anatomical connectivity information of several mammal species has been extensively explored [ST02]; this includes cortico-cortical and cortico-thalamic systems of the rat [BY00], cat [SBY95], and primate [FE91]. There is also much research exploring the topology of connectivity patterns of the human brain (see Figure 2.8) [AB07, HCG+08]. Based on connectivity patterns discovered in anatomy, several mathematical models of connectivity have been built [STE00, You92, BDM04, GW89].

The knowledge of neural connectivity has already been used to create large-scale neural network models. Izhikevich built large-scale neural network computer models to simulate brain activities [IE08, IGE04]. Statistical data from anatomical studies have been used to create neuronal connection models. Of course, simplification is required to capture only the important features of the connectivity patterns, due to the high complexity of the anatomical data.

2.3.8 Axonal delay

Axonal delays, also referred to as synaptic delays, result from the conduction time required for a spike to travel from a pre-synaptic neuron to a post-synaptic neuron. The neural system operates at a very low speed, on the scale of milliseconds. Each axon conveys tens to hundreds of spikes per second.

Depending on the type and location of the neurons, the speed of spike propagations along axons varies a lot. For mammalian motor neurons, the speed is 10-120 m/s, while for sensory neurons it is about 5-25 m/s. As a result, axon conduction delays in the mammalian cortex vary from 0.1 ms to about 44 ms [Swa88, Swa85, Izh06].

Axonal delays are involved in the spacio-temporal coding of the nervous system. For instance, there is evidence to show that the afferent axon delays of the owl's cochlear nuclei account for inter-aural time differences in the nucleus laminaris which is an important cue for sound localization [CK88]. Computational neuroscientists proved that the axonal delay plays an important role in the synchronization of networks of coupled cortical oscillators [CEVB97] or in polychronization behavior [Izh06].

2.3.9 Synaptic plasticity

One of the key features of biological neural networks is the ability to adjust itself to adapt to different environments or to improve itself to solve more complex tasks. This kind of behavior is usually referred to as *learning*.

Biological experiments have shown that both the efficacy of an individual synapse and the number of synapses in the brain change over time. In computational neuroscience, plasticity is modelled mainly by adjusting synaptic weights. In neuroscience, long-term plasticity found in slices of the hippocampus is an important mechanism for learning and for memories that last for a long time. It comprises long-term potentiation (LTP) and long-term depression (LTD). LTP is a persistent enhancement of synaptic weight following the synchronous firings of a pair of pre- and post-synaptic neurons [Mor03], while LTD is the long-lasting weakening of a synaptic weight.

The procedure used to adjust weights is called a *learning rule*. There are a lot of different learning rules. Correlation-based rules in the class of "Hebbian learning" are the most popular for spiking neural networks. Hebbian theory is a basic

mechanism for synaptic plasticity, with the property that synaptic changes are controlled by the timing correlations of pre- and postsynaptic activities. Hebbian theory [Heb49] states that:

- "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."

Though this theory was proposed on purely theoretical grounds, its correctness has been demonstrated regularly in experimental studies. Based on this rule, a class of learning rules has been developed to address the two questions of this rule: when are two neurons considered as being active together; and what is the amount of modification?

Spike-timing-dependent plasticity (STDP) is such a rule which quantifies both the timing and the amount of weight modification. STDP comprises two learning windows - LTP and LTD. If the pre-synaptic spike precedes the post-synaptic spike, the result is LTP. If the spike order is reversed, the result is LTD. The detailed description of STDP and its implementation on SpiNNaker will be discussed later in Chapter 6.

Chapter 3

Parallel engineering solutions

3.1 Overview

Some of the basic concepts involved in both biological and computational neuroscience have been reviewed in the last chapter. Here we move on to the area of neural systems engineering and address the question of how the brain or large-scale neural networks can be simulated very efficiently. The digital computer and the brain are both information processing systems, but differ greatly from each other in their flexibility, determinism, precision, fault-tolerance and so on. The differences arise from the fundamental variance in their materials and structure, which result in a great difference in behaviors. However, if all of the electrochemical processes and behaviors of the brain can be described mathematically, there should be no reason why the digital computer cannot be used to reproduce brain function. Based on the limited understanding of biological systems and the mathematical models developed so far, we are able to create artificial neural networks running on digital computers.

In this chapter, we firstly look into general issues related in neural engineering. A literature review of current implementations of neural networks is then presented, and a critical assessment is attempted. Parallelism is emphasized as a key feature of an implementation of a neural network. Some issues involved in parallel implementations are addressed, especially the trade-off between communication and processing, which requires finely-developed modeling algorithms. At the end of this chapter, we present our solution for dedicated parallel neuromorphic hardware - SpiNNaker.

3.2 Simulation clues: processing, communication and storage

From the modeling point of view, the structure of a biological system indicates that modeling work should focus on the three aspects of computation: the processing, communication and storage of information.

3.2.1 Processing

As the key components of biological neural systems, neurons exhibit a variety of dynamical features. Mathematical neuronal dynamic models, such as that of Hodgkin-Huxley model, have been built to mimic neuronal behavior. Processing is required to simulate these models.

In this aspect, the continuous time involved in the mathematical model needs to be replaced by discrete time steps. The biological neural process is quite slow in electronic terms; computational neuroscientists usually use 1 ms resolution for the modeling. In terms of precision, 32-bit floating-point arithmetic is most commonly used on the Intel x86 architecture. Sometimes 16-bit fixed-point arithmetic is also used because of hardware limitations or in order to achieve better performance.

The processing work varies according to which dynamical model is chosen. For spiking neural models, outputs are usually a series of timed pulses indicating the firing of the neurons. For non-spiking neural models, outputs are real-valued numbers without timing information. In the case of modeling large-scale systems, the importance of efficiency usually outweighs the need for high precision. Some simplified mathematical models have been built based on the phenomenal behaviors of neurons., and as long as they capture the key features of the neuronal dynamics, precision may be a secondary issue. On the other hand, for precise neural models, precision outweighs performance, and detailed conductance-based mathematical models are usually preferred.

3.2.2 Communication

The great processing power of biological neural systems arises from their efficient high connectivity and the resulting interactions between the neurons. Information is passed through the propagation of spikes from one neuron to another, this

communication process is another problem faced when modeling neural networks. Although an individual electronic link is much faster than a neural fiber, the population of links in the electronic system is much lower than in the biological system, which results in a low overall connectivity. From the modeling point of view, the problem of communication is usually more severe than the problem of processing.

3.2.3 Storage

There is no single place in the biological system where information is stored. The information is kept in different tissues or through their interconnections. Memory is a result of the intrinsic properties of the ions and molecules in the nerve cell. Storage behavior has to be abstracted during modeling, and information explicitly maintained in our system includes:

- Constants and variables used for neuronal dynamical models.
- Synaptic weights representing the efficacy of connections.
- History records required for synaptic plasticity.

Of course, depending on the implementation, more information may be required to be stored in the memory, for instance the axonal delays. The largest information requirement is for the synaptic weights, due to the extremely large population of connections. In a sequential system with a single processing node, the storage is simple in that all information has to go into the system memory. In a parallel system with multiple processing nodes however, optimization is required to fit in all information and to avoid inefficiency if information is kept remote from the processing nodes. A common solution in the parallel case is to distribute the information so that each component is placed where it is locally available to the appropriate processing node.

3.3 Current engineering solutions

Attempts have been made to build artificial neural systems since the last century, and systems have been built based on platforms ranging from generic desktop computers to neuromorphic hardware [FT07].

3.3.1 Analogue systems

An analogue based approach to the design of a system is obviously possible, because of the existence of electronic circuits with functionality very similar to the neural circuit [Mea89]. An analogue circuit is much cheaper and demands less power than a digital circuit in implementing some functions such as multiplication. For large-scale network simulation, the area of the circuit should be small. Hence, the simple Integrate and Fire (IF) model is the most popular model to be implemented on analogue VLSI [Ind03, GHS09, SJ95, IF07]. Typically, an IF neuron costs about 20 transistors. To achieve better precision, detailed conductance-based models such as the Hodgkin-Huxley model have also been implemented in analogue circuits [SD99, MD91]. The implementation of Izhikevich model in analogue circuits can be found in [WD07].

However, the drawback of an analogue system is that it lacks flexibility; as a kind of hardwired system, it is difficult to adjust parameters or to change the dynamical model. For the time being, a general-purpose neural platform is preferable because of the experimental nature of the neural modeling. This requirement for general-purpose flexibility makes the analogue implementation less interesting at present. Furthermore, the inflexibility of an analogue implementation potentially increases the cost of the system, because the implementation is dedicated only to a specific model.

3.3.2 Software-based systems

Software neural simulators based on desktop computers or clusters are the most conventional facilities used for neural network modeling. Standard software approaches include: NEURON [NEU], GENESIS [GEN], Brian [Bri, GB09, GB08], and so on. With the low level support provided by software, users can easily build a neural model and run it on most conventional computers, using a friendly user interface providing access to information such as synaptic weights and neuron states. Such systems are also flexible, allowing the user to choose from different dynamical models and network configurations. They are usually written in the C or Python programming languages and came with detailed documentation. Most are extensible – users can define their own models and add to the tool library.

These software systems are ideal for rapidly developing new models, testing new algorithms and teaching computational neuroscience. However, they are not

built for the efficient simulation of large-scale neural networks, and more efficient hardware solutions are sill sought.

3.3.3 Supercomputer-based systems

To overcome the performance limitations of software-based systems, some researchers simulate neural models using supercomputers instead of conventional desktop computers.

In this category, the Blue Brain project is probably the best-known. It was founded in 2005 by Henry Markram at the Brain and Mind Institute at EPFL in Switzerland [Mar06], with the aim of creating a digital 3D replica of the mammalian brain detailed down to the molecular level. This system is built on the IBM Blue Gene/L architecture with a 4-rack machine; it comprises 4,096 nodes or 8,192 700 MHz PowerPC CPUs in total, delivering a peak performance of 22.4 TFLOPS. A new MPI version of NEURON is used as the simulation software. The Blue Brain project emphasizes biological accuracy, hence the detailed compartmental neuron model - the Hodgkin-Huxley model, is chosen for the neuron dynamics.

The first goal of the Blue Brain project was to recreate the neocortical column (NCC) at the cellular level. The NCC, a group of neurons with similar properties, is known as a basic structure of the brain, repeated millions of times across the neocortex. Each NCC has a cylindrical shape 0.5 mm wide and 2 mm high, containing about 10,000 inter-connected neurons. In the Blue Brain project, neuron types and their connection patterns are carefully tuned based on data collected from neuron morphological research. This phase of the project is now coming to a close, and it is now heading in two directions: achieving molecular level resolution and increasing the scale of the system towards modeling the whole brain.

In another example of the use of powerful computing resources [IE08], a model of the mammalian thalamo-cortical system, with one million neurons and almost half a billion synapses, is simulated on a Beowulf cluster with 60 processors each running at 3GHz. One second of biological activity of the neural network takes one minute to simulate.

Supercomputer-based systems are powerful and flexible, and hence are ideal for large-scale neural network simulation, providing means for neuroscientists

to verify hypotheses with complex biologically-realistic models. It also validates the testing in a large and sophisticated environment of neuronal dynamic models and connectivity topologies that have been developed. The downside of supercomputer-based systems is obvious: their huge size makes it impossible for them to be applied in size-sensitive applications, for instance robots.

3.3.4 FPGA-based systems

The Field Programmable Gate Array (FPGA) is a platform used frequently for customized hardware neural simulation [PGG+05, HK04, MMG+07]. The advantages of using an FPGA-based system are that it is quick and easy to test the hardware implementation of a prototype neural network circuit. The reconfiguration ability that an FPGA provides is ideal for exploring neural modeling on hardware. The disadvantage of FPGA implementation comes from its inefficiency, in terms of both area and power consumption, when modeling synaptic connectivity and in handling inter-neuron communications. As a result, an FPGA-based system is suitable for building and testing prototype systems and investigating the modeling algorithms, but it is not suitable for large-scale systems simulation. The situation of an FPGA-based system is similar to that of a computer-based software system discussed in Section 3.3.2. Both are easy to use and are reconfigurable at the cost of efficiency.

3.3.5 Other solutions

Many other neuromorphic solutions have also been attempted. The Neurogrid project in Stanford University aims to produce a brain-like computer, using hybrid analog-digital VLSI technology, to model one million neurons and six billion synaptic connections [LMAB06, MASB07]. Neuronal activities (with two subcellular compartments per neuron) are computed in analog chips, each consists of a 2-D array of circuits. Communications are performed by a digital system, providing reconfigurability to softwire synaptic representations, using address-event representation (AER).

Graphics Processing Units (GPUs), a programmable and high performance computing platform, are also used for simulating large-scale spiking neural networks [NDK+09]. A single GPU (NVIDIA GTX-280 with 1GB of memory) simulation has been demonstrated to model 100K Izhikevich neurons with 50 million

synaptic connections, firing at an average rate of 7 Hz.

Technical University of Berlin has attempted a series of hardware systems in their long-lasting related projects: BIONIC, NESPINN, MASPINN, SP2INN and SPINN Emulation Engine (SEE) [HGG+05, SMJK98, SS98] – a number of different architecture and techniques have been investigated in modeling spiking neural networks ranging from tens of to a million neurons.

3.4 Parallelism

As can be seen from Section 3.3, there are many existing engineering solutions, each with advantages and disadvantages; each design is a trade-off between reconfiguration ability, size, energy and efficiency. Due to the experimental nature of the neural modeling caused by the "unknowns" in neuroscience, there is no "one-size-fits-all" solution in the short term. The choice of solution relies on the requirements of the user.

Though the solutions differ a lot from each other, they have been developed based on solving the three problems mentioned in Section 2.3.9: processing, communication and storage. Efficiency is achieved by accelerating the processing, using faster or more processing units. The performance of a single processing unit is limited by present technology. Even using the fastest available processing unit, the performance is still far from satisfying the demands of simulating the brain - an extremely computationally intensive task.

The most popular hardware architecture in computational neuroscience is the parallel machine. The IBM Blue Gene is a good example of a parallel system providing a massive computational power with extensibility. Parallelism is also the nature of biological neural networks. Biological systems are slow in comparison with modern computers which operate at several GHz. The massive population of neurons, high density of connectivity, and high frequency of interactions compensate however for their individual slowness.

The parallel implementation approach captures the nature of the biological system. It seems more promising than a sequential system. Before making such an assertion, however, a fundamental side-effect of the parallel realization must be addressed - the communication overhead. Problems of communication and workload allocation have accompanied parallel architecture since the concept of parallelism was born in the last century.

In the context of spiking neural network simulation, the situation is however somewhat simplified, here each individual neuron is a self-contained unit, interacting with other neurons through simple pulse-like spikes. Hence neurons can easily be decoupled and mapped onto different processing units, making the workload allocation task much easier than distributing generic software applications. However, investigation of modeling approaches will still be required to address issues such as storage, locality of communication, and optimization of performance.

In the case of non-spiking neural networks – the multilayer perceptron (MLP) network for example – efficient modeling algorithm is even more essential because neurons communicate by real-valued numbers, making the communication pattern more complex. In an MLP network, a well-defined modeling algorithm is therefore required.

Communication in spiking neural network models has its own characteristic as the information that needs to be transmitted is *when* and *which* neuron has fired. Furthermore, information is conveyed in a high fan-out (one-to-many) pattern. The same packet may be sent to several receivers through a multicast-like system. Generic parallel platforms, such as Blue Gene or the Beowulf cluster, usually use a standard communication mechanisms – for example the MPI – which is not customized for the purpose of neural network simulation. The communication system is so essential in terms of the performance of a neural network simulation that it indeed warrants special design and construction.

3.5 SpiNNaker

3.5.1 Why SpiNNaker?

Based on the present status of neural modeling research, a dedicated general-purpose system that is not only designed for a particular neural model, but is also capable of delivering high efficiency, is much favored. Efficiency improvements should be provided without sacrificing too much flexibility. A flexible and efficient design should have a relatively long life-time and better reusability.

Efficiency and flexibility

Universal programmable processing units are good candidates for providing central processing power in such a system. By using these processing units, switching between different neuronal dynamical models is just a programming issue. Compared to hardwired implementations, parameter adjustments will also be much easier. A bespoke high-performance communication system is also desirable to handle information exchange between neurons. Such a communication system should provide a powerful multicast capability to match the high fan-out communication of neural networks. The communication system should not only be efficient, but also be reconfigurable to accommodate variable neural connections. A distributed memory system is also required for storing information. Each processing unit needs to access memory blocks to get neuronal information and store updated neuronal states. The number of neuron parameters and variables is small compared to the amount of synapse information, but they require instant and frequent accessing. The number of synaptic weights is larger, but they are accessed at a comparatively low frequency. As a result, a combination of a small fast memory and a large slow memory can be used for tackling different scenarios.

Scalability

Neural networks not only differ in their dynamical models and conductivities, but also in terms of scale. Scalability is consequently another feature that has to be taken into consider while designing a neural chip. An initial system can be built and tested based on a very small machine. When the functionality of the system has been proved, the machine then can be expanded to simulate larger-scale systems. One way to achieve scalability is to build a system by connecting replicated nodes where each node is self-contained and fully functional. In this approach, an arbitrary scale of system can be built just by incorporating the required population of nodes.

Power efficiency and fault-tolerance

Biological neural systems are known to be power-efficient and fault-tolerant. These are properties that are pursued in computer science. Neuromorphic hardware inspired by biological systems is expected to explore power-efficiency and fault-tolerance of the biology, in addition to modeling the functionality.

Power-efficiency can be achieved in both hardware and software. For instance, in terms of the hardware, low-power processors can be chosen as the processing units; in terms of software, an event-driven mechanism can be used so that a processor is woken up by an event, otherwise, remaining in a low-power sleep mode.

Fault-tolerance can be achieved by the use of redundancy and information distribution. Redundancy guarantees that there are spare resources to take over from faulty units. The nature of information distribution in neural networks results in no individual neuron or connection being critical. Losing a small number of neurons or connections does not affect the functionality of the system.

3.5.2 The SpiNNaker chip

SpiNNaker overview

The SpiNNaker project aims to create general-purpose neuromorphic hardware to meet the challenges of processing, communication, storage, scalability, power-efficiency (as estimated in [PFT+07], each SpiNNaker chip consume 250 mW to 500 mW in 130-nm process technology) and fault-tolerance. It is an EPSRC (Engineering and Physical Sciences Research Council) funded project involving the Advanced Processor Technologies Group at the University of Manchester and the School of Electronics and Computer Science at the University of Southampton, in collaboration with industrial partners ARM Limited and Silistix Limited, using state-of-the-art technologies and tools.

SpiNNaker is mostly focused on simulating large-scale spiking neural networks in real-time. But it has been shown able to simulate traditional models such as the multilayer perceptron.

A block diagram of the SpiNNaker chip is shown in Figure 3.1.

Processing subsystem

Each SpiNNaker chip will contain up to 20 (in the full chip) identical ARM968 processing subsystems (processors) each running at 200MHz (denoted as Proc0, Proc1, ..., ProcN in the Figure 3.1). One of the processors on each chip will be selected as the Monitor Processor and thereafter performs system management tasks. Each of the other processors is called a Fascicle Processor and is responsible for modeling a fascicle (group) of neurons with associated inputs and outputs.

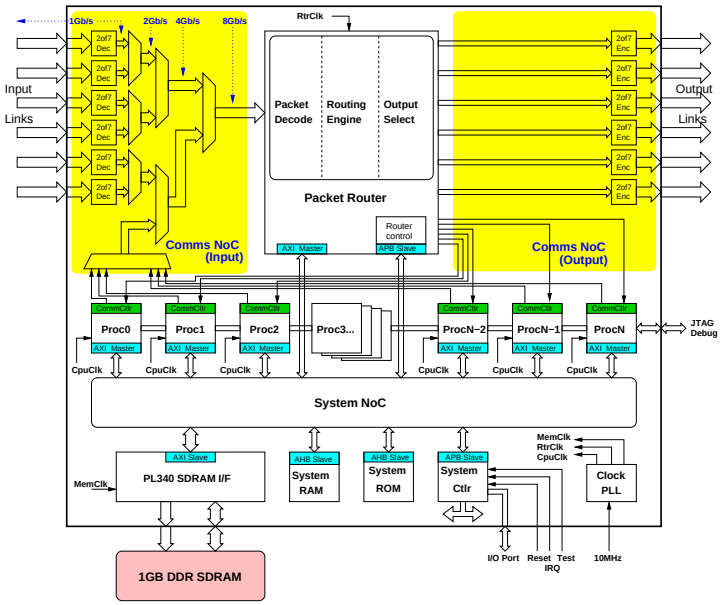


Figure 3.1: The SpiNNaker chip organization

The selection of the Monitor Processor is competitive, making sure a healthy processor is always selected [Kha09].

The detailed organization of a processing subsystem is illustrated in Figure 3.2 where an ARM968 processor provides the basic processing power. Each processor is able to simulate up to about 1,000 Izhikevich neurons in real-time. The processor is directly connected to two memory blocks called the Instruction Tightly Coupled Memory (ITCM) and the Data Tightly Coupled Memory (DTCM), respectively. The TCMs are small but extremely fast (running at processor clock speed), ideal for storing frequently accessed instructions or data. The ITCM is 32kB starting from address 0x00000000 and is for storing instructions. The DTCM is 64kB starting from address 0x00400000 and is for storing data, especially neural data such as neuron variables and parameters.

Other components connected to the ARM968 module through the local Advanced High performance Bus (AHB), includes a Communication Controller, a Timer, a Vector Interrupt Controller, and a DMA Controller. The processor communicates with other processors and chips through the Communication Network On-chip (Comms. NoC) via the Communication Controller which either sends to or receives packets from the Comms. NoC.

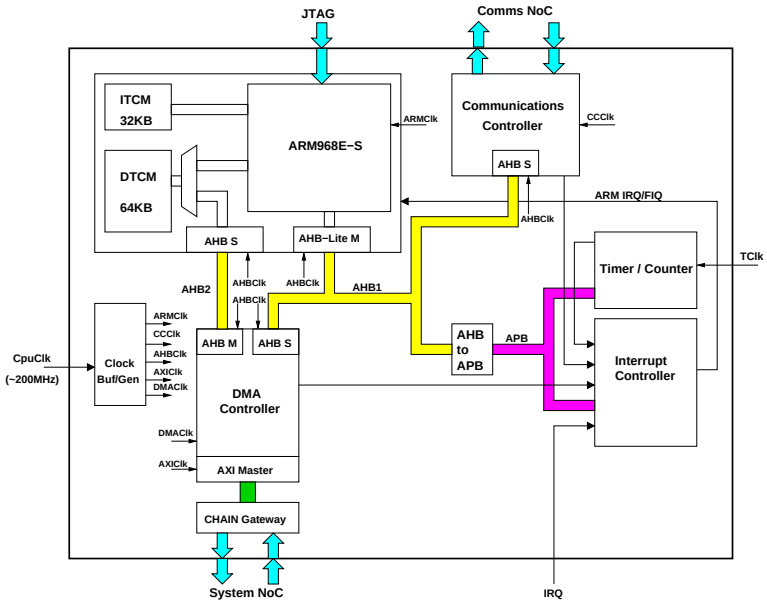


Figure 3.2: The ARM968 subsystem

Access to all other on-chip resources on the System NoC is via the DMA Controller which is mainly used for reading (when a packet arrives) or writing (when updating synaptic information during learning) neural synaptic information stored in the external SDRAM. The DMA Controller can also be used in a 'Bridge' mode, enabling direct read and write access to components outside the processing subsystem.

Router

There is one router on each chip used for routing input and output packets, enabling on- and off-chip communication. At the heart of the router is an associative multicast router subsystem, which is able to handle the high fan-out multicast neural event packets efficiently. There are 1024 programmable associative multicast (MC) routing entries on the full chip. Each entry contains a key value, a mask and an output vector. The routing rule is detailed in equation 3.1.

$$Output = \begin{cases} output\,vector & key == routing\,key\,AND\,mask \\ default\,routing & key \neq routing\,key\,AND\,mask. \end{cases} \tag{3.1}$$

When a packet arrives, the routing key encoded in the packet is compared with the key in each entry of the MC table, after being ANDed with the mask. If it matches, the packet is sent to the ports contained in the output vector of this entry, otherwise, the packet is sent across the router by default routing, normally via the port opposite the input. The routing operation is performed using parallel associative memory for matching efficiency.

In addition to the multicast mechanism, the router can also be used for routing point-to-point packets using a look-up table, for nearest-neighbor routing using a simple algorithmic process, or for emergency routing in the case when an output link is blocked for some reason.

Comms. NoC

At both the input and output of the router, there is a Comms. NoC which carries input and output neural event packets from/to different processors and the inter-chip links. The Arbiter in the input Comms. NoC merges arriving packets into a single serial stream, making sure that packets are presented at the router one by one. In the output Comms. NoC, 6 links from the router send packets directly off the chip . Other links go directly to the Comms. controllers on the same chip.

The Comms. NoC operation is based on address-event signaling with a self-timed packet-switched fabric [FTB06], which decouples the different clock domains of the processors, and hence makes scalability possible. In more detail, inter-chip communication goes through 8-wire inter-chip links using a self-timed 2-of-7 non-return-to-zero (NRZ) code [BPF03]. At the input-end, the input protocol converters translate the off-chip 2-of-7 NRZ codes to the on-chip CHAIN codes, and at the output-end, output protocol converters perform the inverse translation.

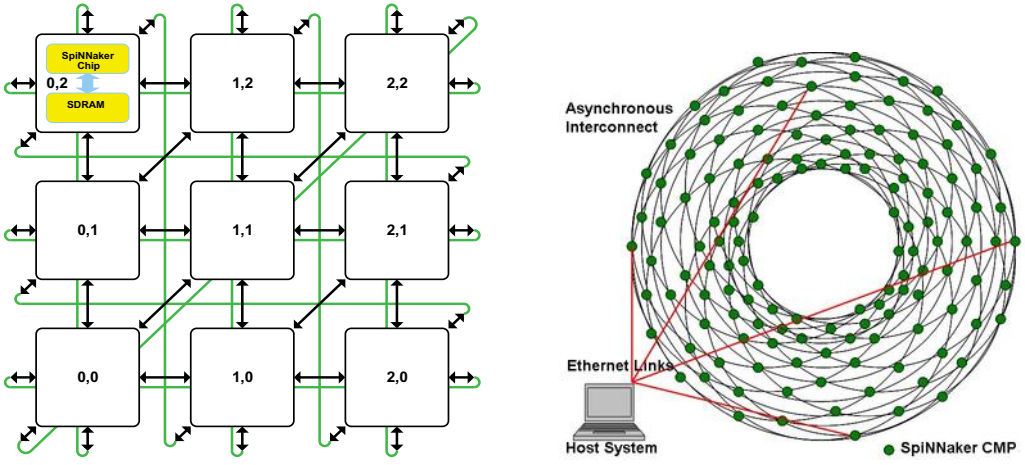
System NoC

The System NoC is responsible for connecting processors to the off-chip SDRAM as well as to a variety of on-chip components, such as the System RAM, the System ROM, and the System Controller. The System NoC is generated by the Silistix CHAINworks tool.

SDRAM

Each SpiNNaker chip is associated with a 1GB off-chip SDRAM device via the ARM PL340 SDRAM Controller, providing significant storage space for storing neural synaptic connection information. Each synaptic entry is 4 bytes, including a synaptic delay (the axon delay), a post-synaptic neuron id, and a real-valued synaptic weight. Each neuron receives a number of inputs, hence the storage of the synaptic connection information for each processor requires significant memory space beyond the capacity of the DTCM. The synaptic information is therefore stored in the off-chip SDRAM and retrieved by a DMA operation when a neural event packet arrives.

3.5.3 The SpiNNaker system



(a) (b)

Figure 3.3: The SpiNNaker system

A SpiNNaker system can be created by connecting a number of SpiNNaker chips through a two dimensional toroidal triangular mesh. Each SpiNNaker chip has 6 external links and thereby is able to connect to 6 neighbors as shown in Figure 3.3(a). A three dimensional view of the SpiNNaker system is provided in Figure 3.3(b) where a Host Computer is attached to the SpiNNaker system via an Ethernet link. The host computer is used for a variety of management and debugging purposes, for instance, sending the code image file and receiving output information. The SpiNNaker system is scalable. An arbitrary number of chips can be used to build a system.

Chapter 4

Spiking neural network on SpiNNaker

4.1 Overview

In this chapter, the algorithm developed for modeling spiking neural networks on SpiNNaker is described. The main focus is on how to model the Izhikevich model on the ARM968 processor of SpiNNaker, using 16-bit fixed-point arithmetic; how to model the neural representations using the Event-address mapping (EAM); the design of the event-driven scheduler. Finally a single processor simulation is run to verify the correctness of the functionality and evaluate the system performance. The work has been published in [JFW08]. Other issues involved in neural modeling on SpiNNaker, such as the neuron-processor allocation, the setup of the routing tables, application downloading, and so on, will be described in later chapters.

4.2 The choice of neuronal dynamical model

SpiNNaker has been developed to be a generic neural simulation platform. It ought to support multiple neuronal dynamical models. The neuronal dynamical models describe the neuron behavior in response to input stimuli and the process of producing output spikes. The model itself is usually independent from other parts of the neural network, for instance the connectivity, coding scheme and learning. Since neurons are only simulated on processors, it decouples the

implementation of neuronal dynamical model from the implementation of the network. This allows us to switch easily between different neuronal models without changing the modeling scheme.

For this system, the Izhikevich model is chosen as an example. As previously introduced in section 2.3.4, the Izhikevich model has a good balance between complexity and accuracy. It is able to reproduce a diversity of neural firing patterns while not sacrificing too much performance. The Izhikevich model has been shown to be suitable for modeling a realistic large-scale neural network system [IE08]. The implementation of LIF model on SpiNNaker can be found in [RGJF10].

4.3 Modeling the Izhikevich model

In a real-time simulation with 1 ms resolution, neuron states need to be updated once every 1 ms, which involves modifying neuron variables according to the input current (stimulus) I. The input current I is the summed amplitude of all pre-synaptic injections and changes every 1 ms. The generation of input current I will be explained in section 4.5. Neuron variables in the Izhikevich model include the membrane potential variable v and the membrane recovery variable u. The Izhikevich equations are the rule for updating the neuron variables, and have been implemented in the Supercomputer [IE08], FPGA [RCF+05], VLSI [CKWR09], and CUDA (Compute Unified Device Architecture) Graphics Processors [NDK+09].

Here we present the approach to efficient implementation of the Izhikevich equations on an ARM968 processor as shown in Figure 4.1. Floating-point numbers are used in the original Izhikevich equation. However, they are converted to 16-bit fixed-point numbers for two reasons:

1. The ARM968 processor does not provide hardware support for floating-point operations.

2. For the sake of performance.

Not only 16-bit fixed-point arithmetic is used here, but through all the work described in this thesis to achieve better performance and save storage space. In this implementation, all neuron information, including neuron state, variables, and the stimulus array, are kept in the DTCM for efficient accessing.

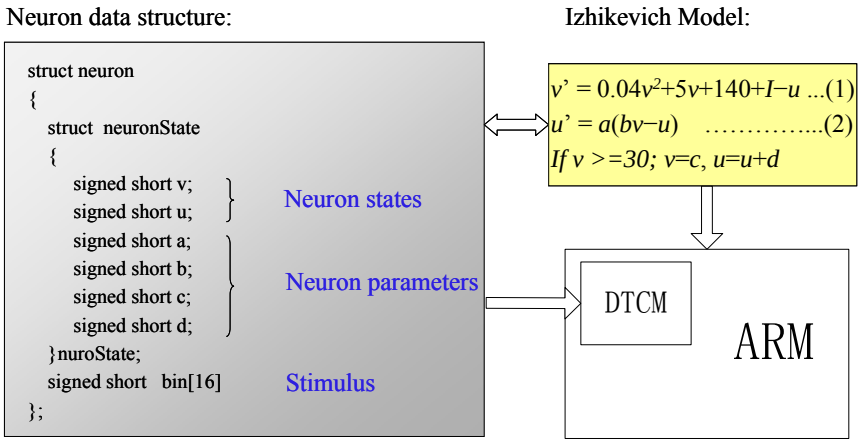


Figure 4.1: Neuron processing

4.3.1 Choice of scaling factors – generic optimization

A review of the original Izhikevich equation:

$$\begin{aligned} \dot{v} &= 0.04v^2 + 5v + 140 - u + I \\ \dot{u} &= a(bv - u) \\ if \quad v \geq 30mV, \quad then \quad & v = c, u = u + d \end{aligned} \tag{4.1}$$

To approximate the floating-point arithmetic using fixed-point arithmetic, scaling factors must be applied. The choice of scaling factors is essential in converting floating-point arithmetic to fixed-point arithmetic. A new dual-scaling factor scheme is presented to reduce the precision lost during the conversion. This is a generic optimization and can be used by other platforms.

To choose a proper scaling factor, the ranges of the variables and parameters relevant to the conversion must firstly be investigated. We run the Matlab file provided by Izhikevich in [Izh03], and did statistic analysis on the value range of the membrane potential v. According to experimental results, v is in the range of -80 to 380, where 380 is the value before reset (It is reset to 30 immediately after it reaches about 380 and then reset to a pre-defined constant c). A 16-bit half-word can represent a signed integer number in the range -32768 to 32768. Hence we get

$$-32768 \leq vp \leq 32767 \, (-80 \leq v \leq 380) \tag{4.2}$$

where, p is the scaling factor. According to equation 4.2, we have $p \leq 86$. In this case, only p as power of 2 is considered so that p can be implemented

simply by shifting. Since a larger p leads to a better precision (see experimental results below), $p = 64$ is chosen. If any p larger than 64 is selected, the membrane potential variable v is in a danger of overflow during processing.

ARM968 is a 32-bit processor; this allows expansion of some operations from 16 to 32-bit during processing to gain better numerical precision without losing performance, while keeping variables in the data structure in 16-bit format. In this way, a larger p can be applied to produce better precision without increasing computation time and storage space. Although the value of the membrane potential variable v is always in the range -80 to 380 during processing, the final value of v held in the data structure should be in the range -80 to 30. So we have

$$-32768 \leq vp \leq 32767\,(-80 \leq v \leq 30) \tag{4.3}$$

According to equation 4.3, $p = 256$ is the largest value we can use.

So far, only the variable which has the largest numerical value has been considered . Some parameters in equation 4.1 with very small floating-point values also need to be considered, since they may cause a loss of precision if the value of the scaling factor is too small. There are two parameters a, b, and one constant 0.04 that need to be examined in equation 4.1. a and b range roughly from 0.02 to 0.1 and from 0.2 to 0.25, respectively, when modeling different types of neurons. The scaling factor $p = 256$ is probably just large enough for these values.

To improve performance, some changes to the presentation of equation 4.1 are required. Parameters a and b are integrated into one parameter ab, in the range 0.004 to 0.025, which results in poor precision when the equations are converted to fixed-point arithmetic using the scaling factor $p = 256$.

The solution here is to use two scaling factors, p_1 and p_2, with a small and large value, respectively. The smaller scaling factor p_1 is applied to parameters, variables and constants with values greater than 0.5 and the larger scaling factor p_2 is applied to those with values less than 0.5. $p_2 = 65536$ is selected because it is both large and efficient enough to implement using multiply-accumulate operations. Thus we get:

$$p_1 = 256,\ p_2 = 65536 \tag{4.4}$$

4.3.2 Equation transformation – ARM specific optimization

This optimization is only specific to ARM processors. To increase the processing speed, a few changes are made to the presentation of equation 4.1, based on two principles:

- Pre-compute as much as possible.
- Reduce the number of operations as much as possible.

Continuous-time differential equations 4.1 can be implemented in discrete-time by the following equations:

$$\begin{aligned} v &= v + \tau(0.04v^2 + 5v + 140 + I - u) \\ u &= u + \tau a(bv - u) \end{aligned} \tag{4.5}$$

where τ is the value of the time step used to control the numerical precision of the discrete-time equation. We use $\tau = 1$ in this implementation for 1 ms resolution. In the ARMv5TE architecture, there is a signed multiply-accumulate instruction *SMLAWB* which does the operation "32bit = 32bit x 16bit + 32bit"[1] in one instruction. In the instruction "SMLAWB", "B" indicates use of the bottom half of the 32-bit register (bits [15:0]). Using this instruction, any operation with the form $(ax \cdot b)/x + c$ can be implemented in one CPU cycle in ARM968, when $x = 2^{16}$ and b is a 16-bit value.

To use the "SMLAWB" instruction, equation 4.5 is transformed to the following equations:

$$\begin{aligned} v &= v(0.04v + 6) + 140 + I - u \\ u &= -au + u + abv \end{aligned} \tag{4.6}$$

If the scaling factors p_1 and p_2 are applied, the implementation of equation 4.6 becomes:

$$\begin{aligned} vp_1 &= \{vp_1[(0.04p_2 \cdot vp_1)/p_2 + 6p_1]\}/p_1 + 140p_1 + Ip_1 - up_1 \\ up_1 &= [(-ap_2) \cdot up_1]/p_2 + up_1 + [(abp_2) \cdot vp_1]/p_2 \end{aligned} \tag{4.7}$$

Meanwhile, a new data structure for the Izhikevich neurons is created:

[1] The SMLAWB instruction multiplies a 32-bit value by a 16-bit value and then adds another 32-bit value, the result is kept in a 32-bit register

```
struct NeuronState
{
    signed short Param_v;    // vp1
    signed short Param_u;    // up1
    signed short Param_a;    // abp2
    signed short Param_b;    // -ap2
    signed short Param_c;    // cp1
    signed short Param_d;    // dp1
}NeuronStates;
```

In this data structure, scaling factors p_1 and p_2 have been applied to the variables to convert them into fixed-point numbers. Some operations are pre-computed such as $a \cdot b$, and stored as a new variable. This re-defined the parameters of Izhikevich neurons, for example, neuron parameters a and b are replaced by ab and $-a$, respectively. Constants $6p_1$ and $140p1$ in equation 4.7 are also pre-computed. In ARM assembly code, when $p_2 = 2^{16}$, one iteration of Izhikevich equations 4.7 consists of the following steps:

1. $A = (0.04p_2 \cdot vp_1)/p_2 + 6p_1$, one "SMLAWB" operation.
2. $A = A << (16 - \log_2 p_1)$, one shift operation.
3. $A = \{A \cdot vp_1\}/p_1 + 140p_1$, one "SMLAW" operation.
4. $A = A + Ip_1$, one "ADD" operation.
5. $vp_1 = A - up_1$, one "SUB" operation.
6. $A = [(-ap_2) \cdot up_1]/p_2 + up_1$, one "SMLATT" operation, which is a signed multiply-accumulate operation "32bit = 16bit x 16bit + 32bit", "T" indicated use of the top half of the register (bits [31:16]).
7. $A = A >> \log_2 p_2$, one shift operation.
8. $up_1 = A + [(abp_2) \cdot vp_1]/p_2$, one "SMLAWB" operation.

where A represents the partial result of a step. In step 1, vp_1 is stored in the bottom 16 bits of a register. When the "SMLAWB" instruction is obeyed, it multiplies $0.04p_2$ (32 bits) with vp_1 (16 bits) in the bottom 16 bits of a register, and only the top 32 bits of the multiplication result are preserved. If $p_2 =$

2^{16}, the division operation $/p2$ can be done automatically as the bottom 16 bits are discarded during the multiplication. The result is finally added with $6p_1$. However, in step 3, $p1 \neq 2^{16}$, so a shift operation is required in step 2 to obtain alignment. In step 6, $-ap_2$ and $-up_1$ are kept in the top 16 bits of two different registers. The computational result of step 6 is in the most significant 16 bits; as a result, a shift operation is required in step 7.

As a result, one iteration of Izhikevich equations takes 6 fixed-point mathematical operations plus 2 shift operations, which is more efficient than the original implementation which takes 13 floating-point operations [Izh04]. In a practical implementation, the whole subroutine for Izhikevich equations computation can be performed by as few as about 20 instructions if the neuron does not fire. If the neuron fires, it takes about 10 more instructions (depends on the implementation) to reset the value and send a spike event.

4.3.3 Precision – fixed-point v.s. floating-point arithmetic

Spike counting

Different choices of scaling factors p_1 and p_2 lead to different levels of precision in comparing with the original floating-point implementation. Table 4.1 illustrates a comparison of the number of spikes generated in a certain period of time by different combinations of p_1 and p_2. Results generated from floating-point implementations of equation 4.5 with $\tau = 1$ are also given in Table 4.1 as benchmarks. The top sub-table in Table 4.1 comprises results from "tonic spiking", while the bottom sub-table comprises results from "tonic bursting". As we can see, larger scaling parameters lead to better precision. Results from the simulation with $p_1 = 256, p_2 = 65536$, which we have chosen, are very close to the floating-point implementation benchmarks.

The proposed approach meets the requirement to reproduce all firing patterns of the original equation. Table 4.2 shows a comparison of the spike counts of four chosen patterns generated in a certain period of time by the 16-bit fixed-point arithmetic and the floating-point arithmetic. Other patterns (not illustrated in the Table) can also be modeled by this implementation with good precision. According to the results shown in Table 4.2, the numbers of spikes generated by the fixed-point simulation are exactly the same as those generated by the floating-point simulation in 1,000 ms.

Table 4.1: Tonic spiking and bursting spike counts

Tonic Spiking. Simulated for 20000 ms, 1ms resolution $a = 0.02, b = 0.2c = -65, d = 6$; $v(0) = -70, u(0) = 0.2v(0)$, $I = 14$ after 0 ms Number of spikes (floating-point): 642			
P_2	Number of spikes, 16-bit fixed-point		
	$P_1 = 64$	$P_1 = 256$	$P_1 = 8192$
256	449	437	482
2048	542	542	566
8192	596	620	611
65536	631	**654**	651
Tonic Bursting. Simulated for 5000 ms, 1ms resolution $a = 0.02, b = 0.2c = -50, d = 2; threshold = 3$; $v(0) = -70, u(0) = 0.2v(0)$, $I = 15$ after 22 ms Number of spikes (floating-point): 502			
P_2	Number of spikes, 16-bit fixed-point		
	$P_1 = 64$	$P_1 = 256$	$P_1 = 8192$
256	364	375	393
2048	424	443	454
8192	449	444	495
65536	462	**501**	502

The table shows a comparison of the number of spikes generated in a certain period of time by different choices of p_1 and p_2 in fixed-point arithmetic simulation. Results from the floating-point arithmetic implementation are also given as benchmarks. Results in the top sub-table are from a tonic spiking pattern while results in the bottom sub-table are from a tonic bursting pattern.

Table 4.2: Different spiking pattern spike counts

Simulated for 1000 ms, 1ms resolution				
$P_1 = 256, P_2 = 65536$				
Spikes	TS	TB	RS	IIS
Fixed-Point	34	102	1	6
Floating-Point	34	102	1	6

This table shows a comparison of the number of spikes generated in a certain period of time by the fixed-point arithmetic simulation with the number of spikes generated by floating-point arithmetic simulation. Results from 4 different firing patterns are listed. TS stands for tonic spiking, TB stands for tonic bursting, RS stands for rebound spiking and IIS stands for inhibition induced spiking.

Input demands

In addition to the number of spikes, the level of precision is also evaluated by other schemes. However, in some cases, the precision is not ideal. Table 4.3 shows the input current required to reproduce the pattern of rebound spiking with different choices of scaling factors. The result from the floating-point simulation is given as the benchmark. If $p_1 = 256, p_2 = 65536$ is chosen, the rebound spike can only be reproduced when the input current $I = -50$; in the floating-point arithmetic simulation, the required input current I is -21. If $p_1 = 8192, p_2 = 65536$ or even larger values are chosen, then the result is comparable to the result from the floating-point simulation. However, this is not achievable in 16-bit fixed-point arithmetic.

According to the results, although floating-point to fixed-point conversion does have drawbacks in respect of precision, generally it is still able to reproduce rich firing patterns.

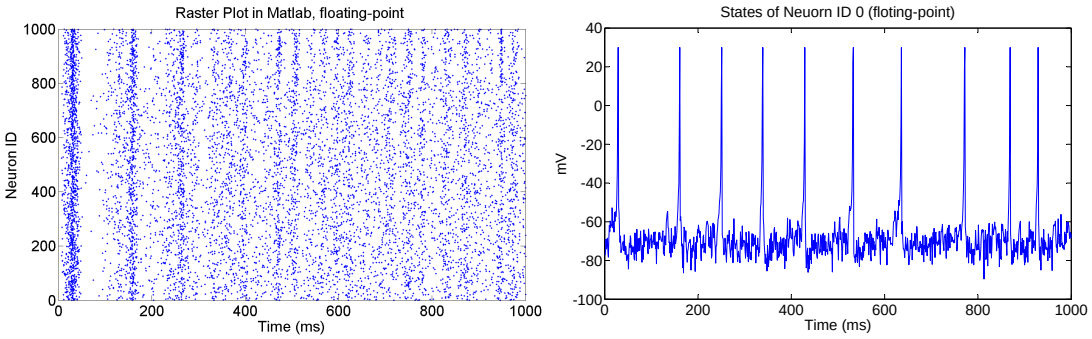
Matlab simulation

The fixed-point arithmetic is also compared to the floating-point arithmetic by running Matlab simulations with a random 1,000-neuron network. Neuron spike timings from the floating-point (using the same Matlab code provided in [Izh03]) and fixed-point arithmetic Matlab simulation (using the code provided in [Izh03], but converting floating-point to fixed-point arithmetic using dual-scaling factor scheme) are shown in Figure 4.2(a) and Figure 4.2(c), respectively. The two results are similar, taking into consideration that the network is randomly (with

Table 4.3: Input current required for generating rebound spikes

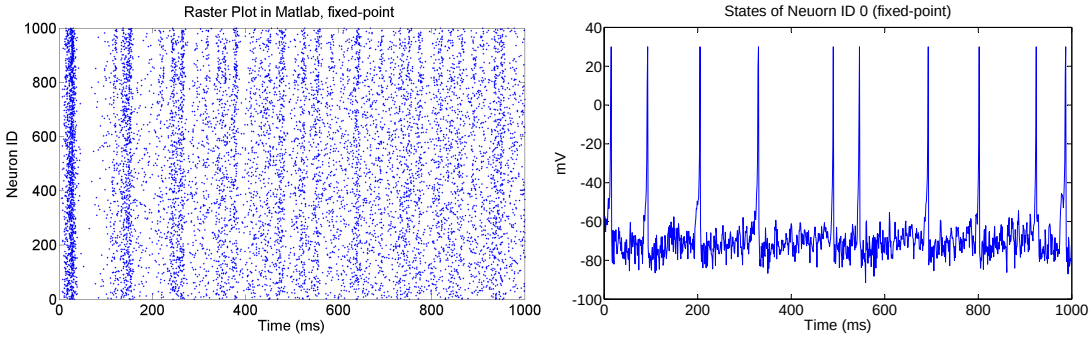
Rebound Spike. Simulated for 200 ms, 1ms resolution $a = 0.03, b = 0.25, c = -60, d = 4$; $v(0) = -64, u(0) = 0.2v(0)$, I lasts for 5 ms Floating-point: $I = -21$				
P2	I required in 16-bit fixed-point			
	$P_1 = 64$	P1 = 256	$P_1 = 8192$	$P_1 = 65536$
32768	-48	-50	-50	-50
65536	-49	**-50**	-23	-23

The input current I required to reproduce the rebound spiking with different choices of scaling factors. The result from the floating-point arithmetic simulation is given as the benchmark.



(a) Spike raster from floating-point arithmetic simulation

(b) Typical spiking activity of an excitatory neuron (floating-point)



(c) Spike raster from fixed-point arithmetic simulation

(d) Typical spiking activity of an excitatory neuron (fixed-point)

Figure 4.2: A comparison between floating-point and fixed-point arithmetic implementation from the Matlab simulation.

certain restrictions) generated each time when the simulation starts. The activities of an excitatory neuron in the two simulation are shown in Figure 4.2(b) and Figure 4.2(d).

4.3.4 Processing speed

The code is developed using the RealView Development Suite (RVDS) which provides an integrated development environment (IDE) for embedded system applications running on the ARM family of processors [Ltdb]. RVDS consists of a series of tools which enable users to write, build and debug the applications, either on target hardware or on software simulators.

The code is run on a virtual environment based on the RealView ARMulator Instruction Set Simulator (RVISS) supplied with RVDS. RVISS simulates the instruction sets and architecture of ARM processors together with a memory system and peripherals, useful for software development and for benchmarking ARM architecture targeted software.

The performance of the 16-bit fixed-point arithmetic implementation, using the dual-scaling factor scheme, is evaluated based on a virtual system with a 200 MHz ARM968 core, a 100 MHz AHB bus, a 32 KB ITCM, a 64 KB DTCM and a 100 MHz SDRAM, in RVISS 1.4.

If neuron data structures are stored in the DTCM and the input current I is reset to a constant number after the update, 1 ms simulation of processing of equation 4.7 (i.e. updating the states of one neuron in one millisecond), takes 240 nanoseconds (ns). If I is reset to a random number (to mimic input noise), then it takes 330 ns. When the data structures is stored in the SDRAM and input current I is reset to a constant value, the processing takes 660 ns.

4.4 Modeling spikes and synapses

The SpiNNaker chip has efficient on-chip and inter-chip connections and a multicast mechanism for high-performance communication. Each fascicle processor in the system models a bunch of neurons. Neurons communicate with each other through connections. To map neural networks onto SpiNNaker, the key problem that needs to be solved is to distribute processing workloads while keeping communication overheads low. This is a very common issue in the parallel computing domain.

In this section, a *event-address mapping* (EAM) scheme is proposed as a method for synaptic weight storage. The EAM scheme sets up a relationship between the spike event and the address of synaptic weights in the memory. As a result, the associated synaptic weights can easily be found from the incoming spike event.

A similar scheme termed *address-event representation* has been used in [Mah92] and [Siv91], where each pre-synaptic neuron in a system is given an unique address, which will be broadcast along a bus to all post-synaptic neurons when the neuron fires. A problem with the address-event representation is the potential of causing a timing error, when broadcasting two events with closed timing through the same bus, in which case either one event must be dropped or they must be serialized [FT07].

The EAM scheme proposed for SpiNNaker uses the asynchronous arbitration to provide scalability and to serialize the events with low timing error [Boa00]. A new multicast routing mechanism with internal lookup tables is developed for the event-address communication. The EAM scheme also employs two memory systems (DTCM and SDRAM) for efficient storing synaptic connections, and uses DMA operations to transfer data. The EAM scheme is explained in detail below.

4.4.1 Propagation of spikes

There is a high density of one-to-many (high fan-out) transmissions of packets due to the high connectivity of neural networks. This pattern of traffic leads to inefficient communication on conventional parallel hardware.

The heart of the communication system of the SpiNNaker chip is the Multicast Router mechanism. Using this mechanism, one-to-many communication with identical packets is efficient. In a neural network, each connection is associated with a synaptic weight which indicates the strength of the effect that the pre-synaptic neuron has on the post-synaptic neuron. In the EAM scheme we proposed, synaptic weights are kept at the post-synaptic end, hence, no synaptic weight information needs to be carried in a spike event. When a neuron fires, a number of *identical* packets are sent to post-synaptic neurons, this can be handled by the multicast mechanism efficiently. Thus the EAM scheme helps reduce the communication workload.

Each spike is represented by a spike event packet which contains only the ID information of the neuron that fired. The ID information, also called a *routing key*,

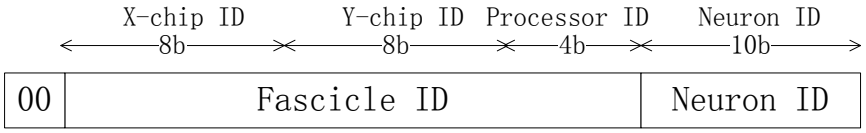


Figure 4.3: The routing key

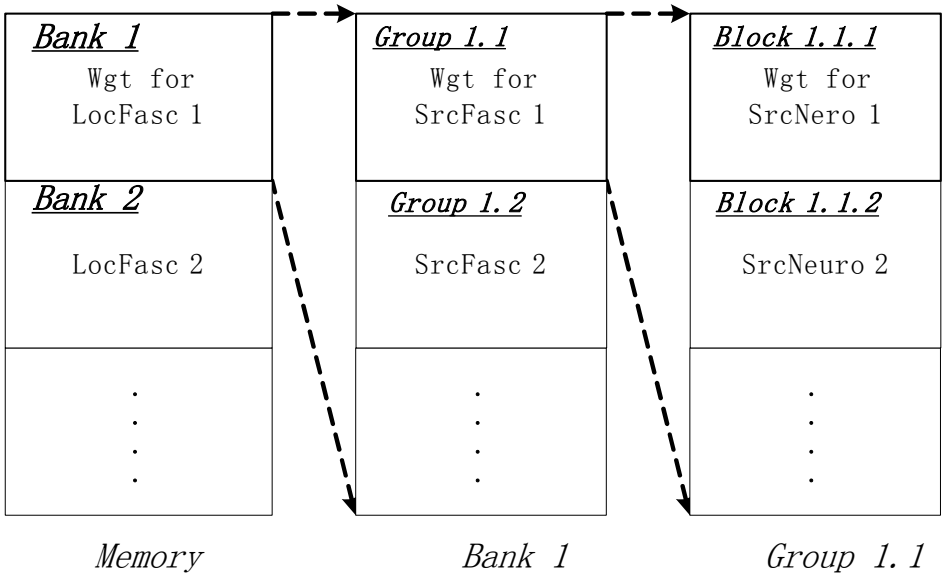


Figure 4.4: Synaptic weight storage. Synaptic weights are organized in a hierarchical structure. Each Bank comprises a number of Groups and each Group again comprises a number of Blocks.

is a combination of an 8-bit x- and an 8-bit y-coordinates of the chip ID, a 4-bit processor ID and 10-bit neuron ID (as shown in Figure 4.3). Other configurations can also be used, since this is purely software-defined. The packet propagates through a series of multicast routers according to the pre-loaded routing table in each router, and finally arrives at the destination fascicles (processors). Detailed information regarding the routing key and the routing algorithm of SpiNNaker can be found elsewhere [KLP+08, RJG+10].

4.4.2 Event-address mapping scheme

The event-address mapping (EAM) scheme is designed to find the right Synaptic Block in the SDRAM associated to the received routing key.

Memory organization

The SDRAM used for synaptic weight storage has the hierarchical structure as shown in Figure 4.4:

1. Synaptic Banks. The memory comprises a number of Banks, labeled "Wgt for LocFasc" in Figure 4.4. A Bank is associated with one of the processors on the chip, which is allocated a memory space in the SDRAM for storing the synaptic weights of all connections to a post-synaptic fascicle processor.

2. Synaptic Groups. Each Bank of memory comprises a number of Groups, labeled "Wgt for SrcFasc" in Figure 4.4. Each Group contains the synaptic weights of all connections from neurons on a source fascicle (processor) to neurons on this local fascicle processor.

3. Synaptic Blocks. Each Group comprises a number of Blocks, labeled "Wgt for SrcNero" in Figure 4.4. Each Block contains the synaptic weights for all connections from one pre-synaptic neuron (on the corresponding source fascicle) to post-synaptic neurons on this fascicle. This is a one-to-many connection scenario: one neuron on the source fascicle connects to many neurons on the local fascicle.

Figure 4.5 shows an example of synaptic weight storage in the SDRAM. There are two pre-synaptic processors (fascicles) and two post-synaptic processors (fascicles). The two pre-synaptic processors can be on the same chip or on different chips, while the two post-synaptic processors are on a same chip. Each processor is modeling several neurons denoted by grey circles. The synaptic weights are indexed from W1 to W8. The right part of the figure shows the synaptic weight organization in the SDRAM. In this example, each Block has a fixed size of 2 (Each pre-synaptic neuron is connected to two post-synaptic neurons). Note that in each Group, the size of a Block is fixed, if the number of connections is smaller than the size of the Block, the Block will be padded with zeroes.

The lookup table

When a post-synaptic neuron receives a spike event packet, this indicates that one of its pre-synaptic neurons has fired. The post-synaptic neuron must then find the weight associated with this input; this involves a search in the off-chip SDRAM of the post-synaptic processor. The EAM scheme is designed to locate

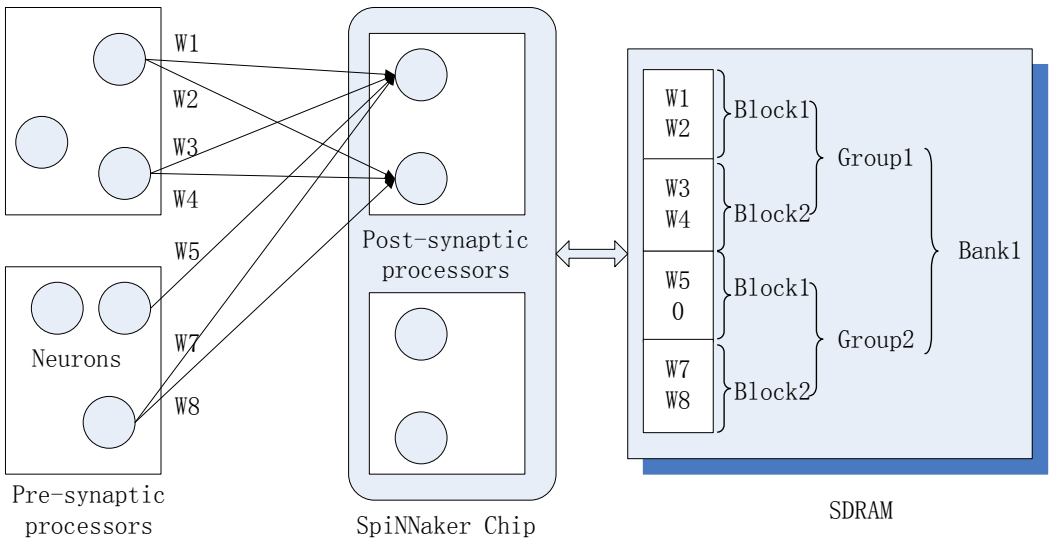


Figure 4.5: An example of synaptic weight storage. Each circle denotes a neuron. Synaptic weights are indexed by W1, W2, ..., W8. Synaptic weights are organized in a hierarchical structure.

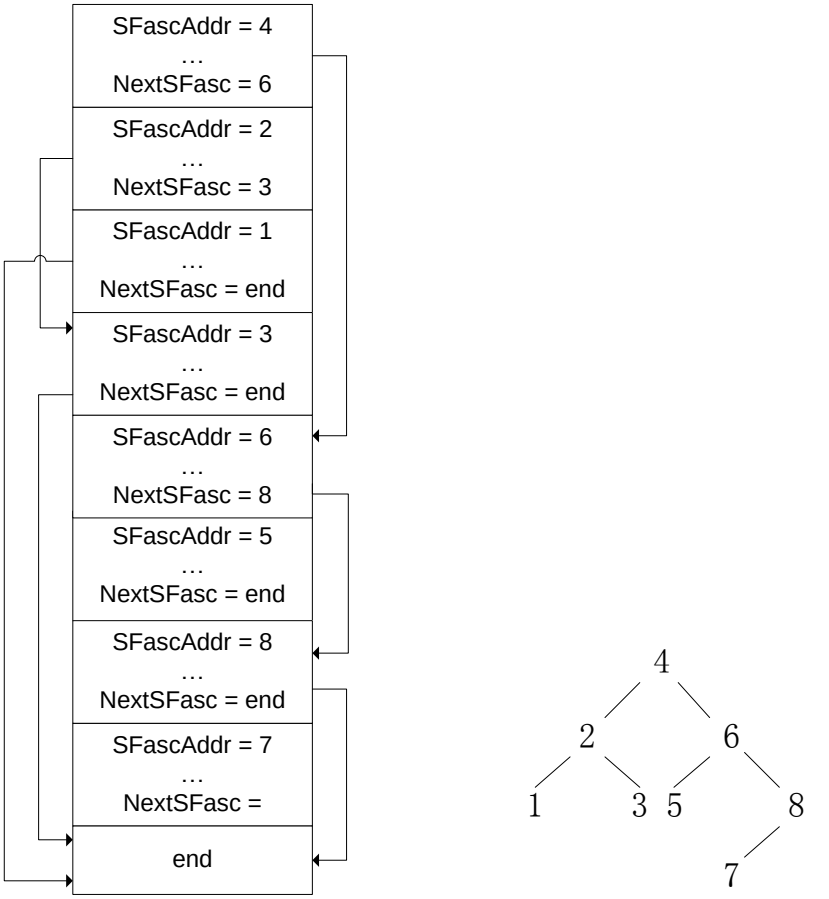
the associated weights for the received inputs using a lookup table for indexing Synaptic Blocks. There are four elements in each entry of the lookup table: a source fascicle address, a MASK, an address pointer to the SDRAM, and a pointer to the next entry:

```
struct SFascicle
{
    int SFascAddr;                  //address to match
    int SFascMask;                  //bits to ignore
    int *SFascPtr;                  //Sdram address of fascicle start
    struct SFascicle *NextSFasc;    //go here if >
}
```

Entries in the lookup table occupy contiguous memory locations, as shown in Figure 4.6(a), organized as a binary tree as shown in Figure 4.6(b).

Finding the synaptic Block

The routing key of the packet is initially combined with the MASK bits, which are set to 0xfffffc00 to mask off the neuron bits (bit [9:0]) of the routing key. The processor performs a binary tree search for a match between the fascicle ID in the routing key and the source fascicle address in the lookup table: if the fascicle



(a) Lookup table organization

(b) The corresponding binary tree

Figure 4.6: The lookup table

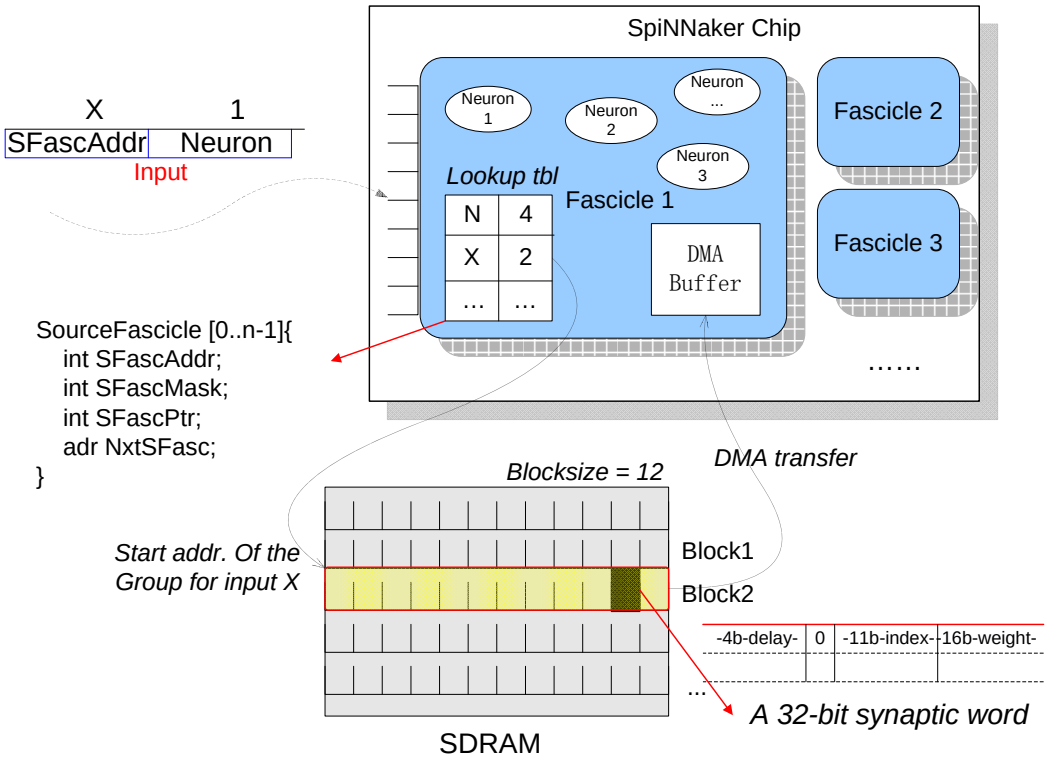


Figure 4.7: Finding the synaptic Block. The lookup table on the fascicle contains a list of source fascicle IDs each of which points to a base address of memory in the SDRAM. When a routing key arrives at the fascicle, the fascicle searches its lookup table based on the fascicle ID of the routing key. If a match is found, the base address to which the fascicle ID points (the Group address) will be fetched. The neuron ID in the routing key is then added to the base address to produce the synaptic Block starting address 0x100290. In this figure, the block size is 12 Words (48 bytes). The 12 words following the address 0x100290 are then transferred to the DTCM by a DMA operation.

ID in the routing key is larger than the fascicle address in the entry, it branches to the next entry pointed to by the "NextSFasc". If it is smaller, it goes to the next entry in the following memory location. If a match is found, the value of "SFascPtr" will be read. This is the start address of the Synaptic Group in the SDRAM associated with this input.

Figure 4.7 illustrates this scheme. An input packet arrives with a source fascicle address "X" and a neuron ID "1" at the post-synaptic processor, indicating that neuron "1" in source fascicle "X" has fired. The processor receives the packet and interrogates the lookup table in the DTCM for an entry matching the fascicle ID "X", and the start address 0x100260 of the Synaptic Group is

Figure 4.8: The Synaptic Word.

fetched. The Group start address is then increased by an offset address 0x30 (Hex(1*12*4)) to produce the Synaptic Block start address 0x100290. The block size is fixed to a constant "BlkSiz", 12 in this case. The BlkSize represents the number of post-synaptic receptors per input, and the 12 words following address 0x100290 contain the Synaptic Block required. The SDRAM is slow compared to the DTCM. So a DMA operation is used to transfer the Synaptic Block to a buffer in the DTCM and prepare for the next operation – "updating the input current". Each 32-bit word of a Synaptic Block contains a Synaptic Word for one synaptic connection, containing a 16-bit weight (bit[15:0]), an 11-bit post-synaptic neuron ID (bit[26:11]), and a 4-bit synaptic (axonal) delay (bit[31:28]), as shown in Figure 4.8.

Synaptic delay

The synaptic delay (axonal delay) is the latency between a pair of connections. Biologically, a spike generated by a pre-synaptic neuron takes time measured in milliseconds to arrive at a post-synaptic neuron. In an electronic system, the communication delay is variable depending on the distance and the network workloads, which is uncertain. Electronic delays, however, the ranges are in nanoseconds or microseconds, and are much smaller than delays in the biological system. Hence the electronic delay can be ignored, and the arrival time of a packet can be taken as the firing time of a neuron.

As discussed in Section 2.3.8, however, delays play an important role in neural modeling. Hence it is necessary to put delays back into the system, but using another approach, to enforce an agreement between biological and electronic time. The four most significant bits are used in a Synaptic Word to represent the delay of the connection. These four bits allow us to simulate a delay up to 15 ms [2] with 1 ms resolution. The electronic packet arrival time must be increased by a delay value to generate the real "biological" arrival time of a spike, before applying the

[2] 0 ms delay is not biologically realistic, and it cannot be guaranteed by our model. As a result, the actual delay ranges from 1 to 15 ms

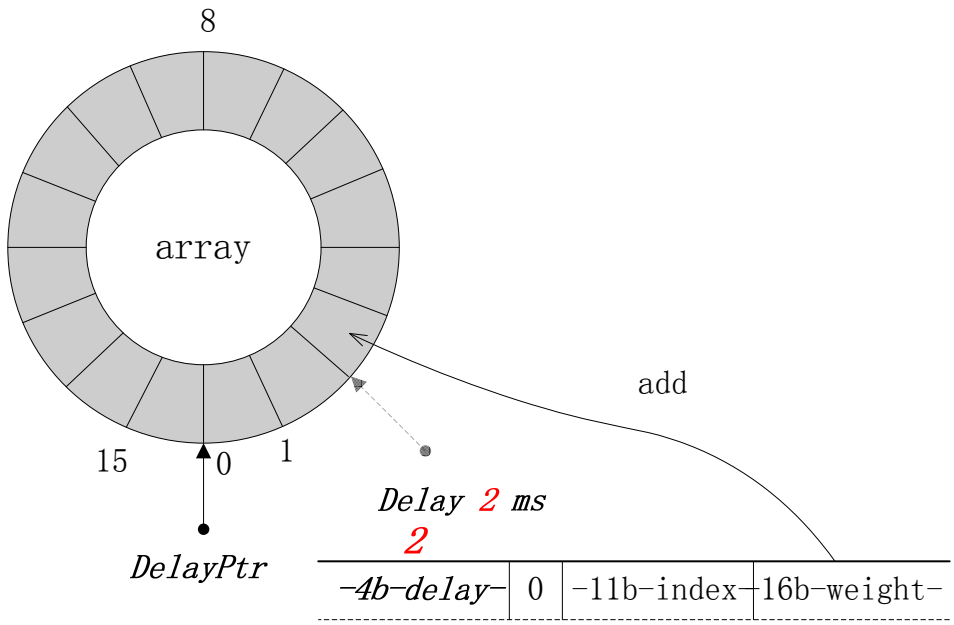


Figure 4.9: The input array. A spike arrives with a delay of 2 ms, so the Synaptic Weight is added to the position "DelayPtr + 2" in the array.

"update" process.

4.5 Updating the input current

Weights in the Synaptic Block transferred to the DTCM will be loaded into an 16 elements circular input array to implement the synaptic delay of 0 ms – 15 ms discussed in section 4.4.2. The 16 elements with time 0 ms to 15 ms (as shown in Figure 4.9), each is a 16-bit half-word integer number, representing the amplitude of all inputs applied to this neuron at a certain time. A pointer denoted as "DelayPtr" points to the element at the current time (0 ms) in the array. In the real-time simulation, the "DelayPtr" moves forward one element per 1 ms with wraparound after 15 ms. The elements following the "DelayPtr" represent a future time of 1 ms, 2 ms, ..., 15 ms in order.

During the updating of the input array, the 4-bit delay in the Synaptic Word is fetched and used to determine to which element the weight will be updated into. In Figure 4.9, for instance, the delay in the Synaptic Word is 2 ms, so the 16-bit weight included in the Synaptic Word needs to be added into element No. 2.

The weight updating is reconfigurable and is performed according to the type

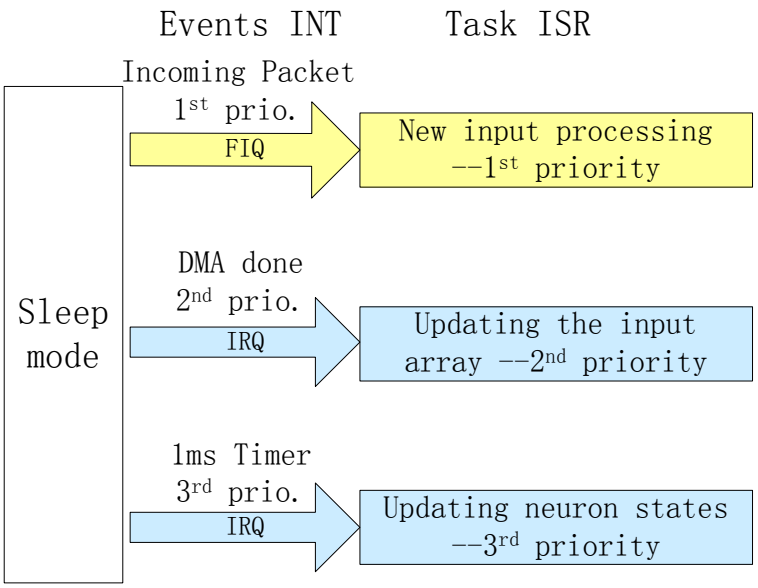


Figure 4.10: The scheduler for the event driven model. The processor normally stays in sleep mode to save power. When an event occurs, the processor wakes up and processes a certain task.

of synapse. In current system, a simple linear accumulation is used for weight updating, the same synapse model as used in most current neural network models. There are "BlkSiz" iterations of the weight updating to traverse all Synaptic Words in the Synaptic Block.

Elements in the input array are used for updating neuron states as previously discussed in section 4.3. In each 1 ms, the element pointed to by "DelayPtr" is fetched and its value is assigned to the variable I in equation 4.1.

4.6 System scheduling

The three tasks involved in modeling neural networks on SpiNNaker have been discussed. An event-driven model is proposed to schedule this multi-task system. As shown in Figure 4.10, the processor is normally in sleep mode to minimize power consumption, and it is woken up by one of the three event signals. Each signal corresponds to a different task:

1. "Updating neuron states". Neuron states are updated every 1 ms, with associated input current "I", according to the Izhikevich equations. This is driven by a 1 ms Timer signal as discussed in section 4.3. This task is the

least urgent and set to the third (lowest) priority.

2. "Updating the input array". The input array is updated when the Synaptic Block has been transferred by a DMA operation from SDRAM to DTCM. It is driven by the DMA operation done signal as discussed in section 4.5. This task is set to the second priority for a quick response to the DMA done signal and to clear the DMA buffer as soon as possible for the next DMA transfer.

3. "New input processing". The synaptic weights are fetched by DMA when a spike event packet arrives at the processor. This is driven by the packet arriving signal as discussed in section 4.4. This task has the first (highest) priority to make sure the packet is withdrawn from the communication controller as soon as possible to avoid congestion. A Fast Interrupt Request [3] (FIQ) is used for this task to reduce the latency of interrupt response.

To achieve real-time performance, all tasks should complete within one millisecond, so the workloads need to be adjusted to fit this requirement. If a processor cannot meet the real-time requirement, the workload of this processor needs to be reduced by reducing the number of either the neurons or the connections.

4.7 Simulation results on a single-processor system

The behavior of one fascicle processor is simulated based on the proposed model described in Section 4.3.4. As there is only one processor in the model, neurons on the same fascicle are wired together, without off-chip connections. Spikes propagate via a buffer instead of via the actual communication system. Neuron parameters are set in accordance with those used in [Izh03] to generate comparable results. The processor models 1,000 randomly connected neurons with about 10% connectivity.

Figure 4.11 shows firing patterns of an excitatory and an inhibitory neuron during one second simulation. For each type of neuron, the three curves from top to bottom are the membrane potential v, the membrane recovery variable u, and

[3] FIQ and IRQ are two different types of interrupt mechanism provided by the ARM architecture.

the input current I. The simulation verifies the functional correctness of modeling Izhikevich neuronal dynamics and neural representations on SpiNNaker.

4.7.1 Processing speed

The performance of the three tasks is evaluated:

1. "Updating neuron states". As already shown in Section 4.3.4, this takes 240 ns to update the state of one neuron.

2. "Updating the input array". This takes 280 ns in total to process an incoming packet and to start a DMA operation.

3. "New input processing". This takes 110 ns to process one Synaptic Word (4 bytes) and update the input array.

As a result, the processing time can be estimated by 240 ns/Neuron + 280 ns/Input + 110 ns/Connection. Where, in a certain period of time, Neuron is the number of neurons which have to be updated, Input is the number of incoming packets, Connection is the number of connections to be updated. According to this result, the performance of the system analytically can be estimated. Assuming:

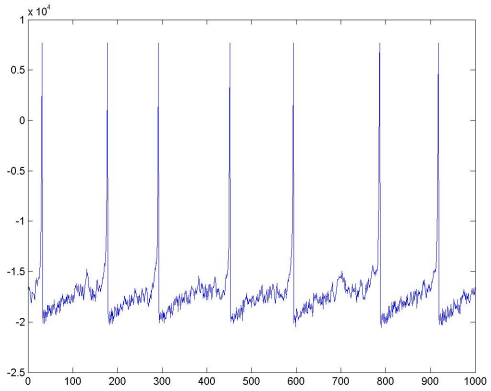
- Neurons fires at F Hz.
- Each fascicle contains N neurons.
- Each fascicle receives inputs from C_{nf} neurons.
- Each input connects to C_{nn} neurons within that fascicle (so $C_{nn} = BlkSize$).
- T (ns) is the required processing time for 1 ms simulation

we get

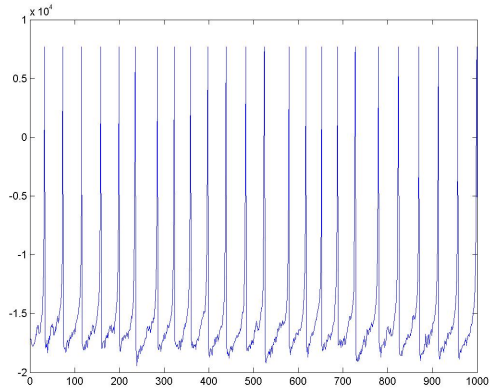
1. N neurons need to be updated per millisecond.

2. $FC_{nf}/1000$ inputs arrive per millisecond.

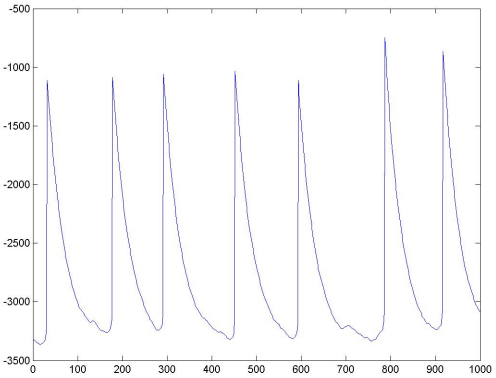
3. $FC_{nf}C_{nn}/1000$ connections need to be updated per millisecond.



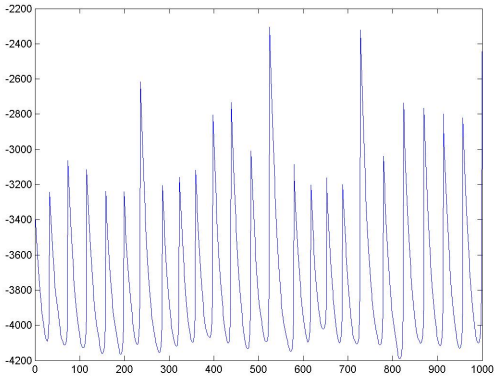
(a) Membrane potential v of an excitatory neuron



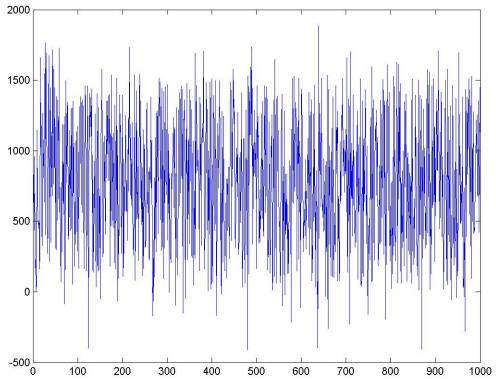
(b) Membrane potential v of an inhibitory neuron



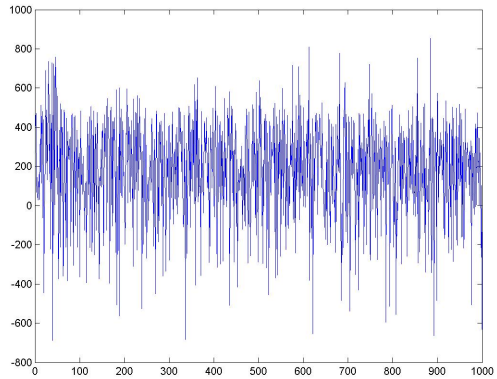
(c) Membrane recovery variable u of an excitatory neuron



(d) Membrane recovery variable u of an inhibitory neuron



(e) Input current I of an excitatory neuron



(f) Input current I of an inhibitory neuron

Figure 4.11: Results from a single-processor simulation.

For each fascicle, we get

$$T = 240N + 0.28FC_{nf} + 0.11FC_{nf}C_{nn} \tag{4.8}$$

According to Equation 4.8, the processing time T increases with N, F, C_{nf}, and C_{nn}. Since each connection occupies 4 bytes, the SDRAM bandwidth requirement is $4FC_{nf}C_{nn}$ Bytes/s for each processor and $80FC_{nf}C_{nn}$ for the whole chip. In a typical case of $C_{nf} = 1,000$, $C_{nn} = 1,000$, and $F = 10Hz$ the SDRAM bandwidth requirement is 800 MB/s, which is less than 1GB/s SDRAM bandwidth provided.

In the single processor simulation, a 1000-neuron network with each neuron connects to 100 others (around 10% connectivity) is configured. So $C_{nf} = 1000$, and $C_{nn} = 100$. Neurons are allowed to fire at a maximum of 67 Hz. If the network is configured to 1000 neurons with only 100 neurons pre-synaptic neurons, each connects to 1,000 post-synaptic neurons (the same 10% overall connectivity). Then $C_{nf} = 100$, and $C_{nn} = 1000$. Neurons are allowed to fire at a maximum of about 69 Hz. Thus different connectivity patterns with the same overall connectivity result in different performance. In a more severe case, if $C_{nf} = 1000$, and $C_{nn} = 1000$ (a fully connected network), it only allows a firing rate of up to 6.9 Hz.

4.7.2 Data memory requirement

DTCM usage

The data structure of a neuron is:

```
struct neuron
{
    struct  neuronState
    {
        signed short param_v;              //variable
        signed short param_u;              //variable
        signed short param_a;              //parameter
        signed short param_b;              //parameter
        signed short param_c;              //parameter
        signed short param_d;              //parameter
    }nuroState;
```

```
    signed short   bin[16];                  //input current array
};
```

The requirement of the DTCM is show below:

1. Neuron data structures. Each neuron data structure occupies 44 bytes of memory. As shown above, each structure includes two 16-bit variables, four 16-bit parameters, and an input array with 16 16-bit elements. N neurons take $44N$ bytes of the DTCM in total. In a typical case of $N = 1,000$, it uses 44 KB memory.

2. Lookup table. Each entry in the lookup table takes 16 bytes of memory. The total memory usage depends on the number of entries, which is determined by the number of pre-synaptic fascicles (source facicles). Let the number of source fascicles in the lookup table be e; this then requires $16e$ bytes.

3. DMA buffer. The size of DMA buffer is implementation dependent. Each block of DMA needs to be 4*BlkSize bytes ($4C_{nn}$ bytes). We use only two DMA buffers in the implementation. This takes $8C_{nn}$ bytes in total. In a typical case of $C_{nn} = 1,000$, it uses 8 KB memory.

SDRAM requirement

There are $C_{nf}C_{nn}$ connections from pre-synaptic neurons per fascicle. Each connection takes 4 bytes. So $4C_{nf}C_{nn}$ bytes are required in total. Each SDRAM is shared by 20 fascicles; therefore each SpiNNaker chip requires $80C_{nf}C_{nn}$ bytes if all 20 processors are used as fascicle processors (in fact one processor will be used as the Monitor Processor). In a typical case of $C_{nn} = 1,000$ and $C_{nn} = 1,000$, it uses 80MB of SDRAM.

4.7.3 Communication analysis

From a single chip point of view, the packets go through the router include input packets, output packets, and bypass packets:

Input packets Each packet is 32 bits, and there are 20 processors per chip. The throughput for input packets is $640FC_{nf}$ bit/s. For $F = 10Hz$ and $C_{nf} = 1000$, the throughput is 6.4 Mbit/s.

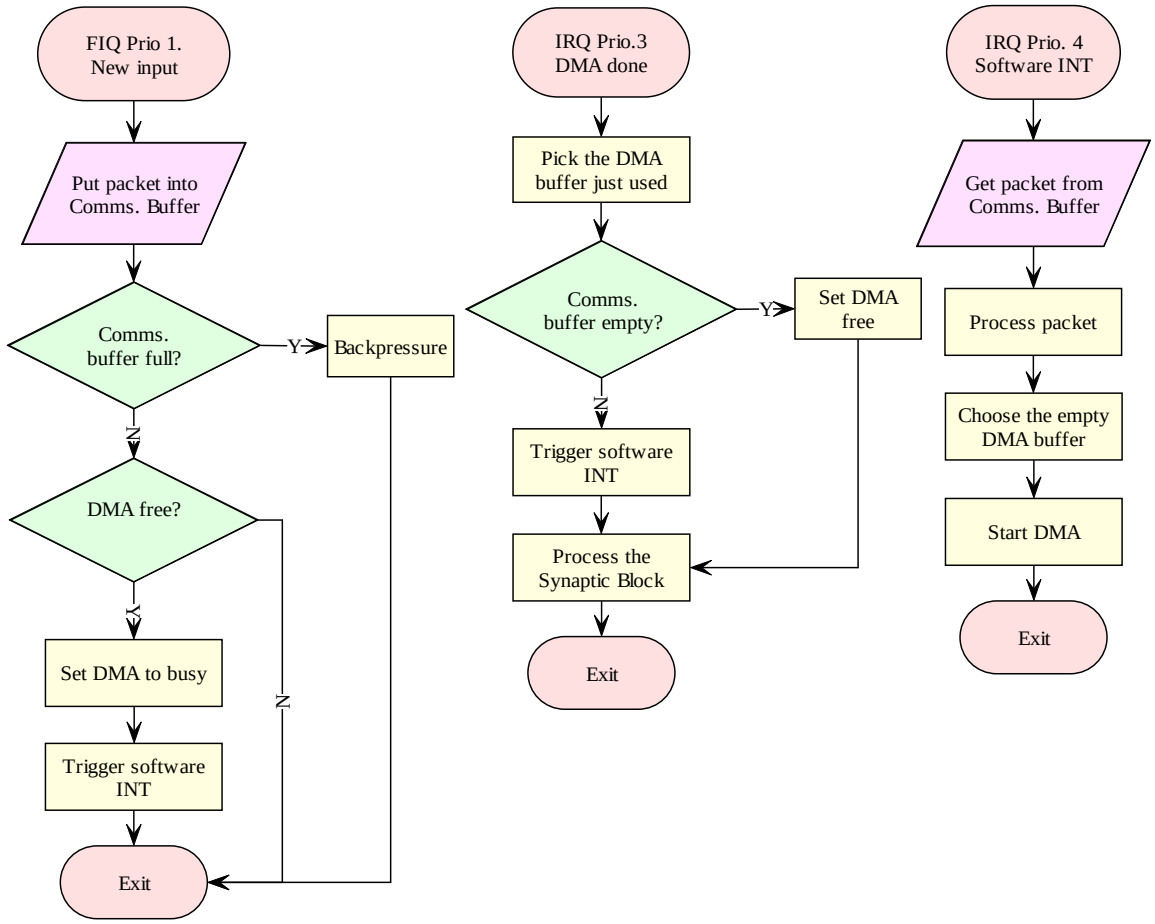


Figure 4.12: The flow chart of the input processing.

Output packets The throughput for output packets is $640FN$. For $F = 10Hz$ and $N = 1000$, the throughput is 6.4 Mbit/s.

Bypass packets The analysis of bypass packets throughput is complicated. It relies on the connectivity of the whole neural system. So we don't carry out an analysis here.

4.8 Optimization

4.8.1 Input processing optimization

The scheduler can be optimized by splitting the three events into four, with an additional software interrupt event. Since the communication controller needs to be cleared as quickly as possible to avoid congestion, the "New input processing"

task is obviously too large to fit this requirement. The "New input processing" task can be divided into two tasks:

1. The first task is still performed by the FIQ interrupt service routine (as shown on the left side of Figure 4.12). This task is only responsible for receiving the packet from the communication controller and triggering (when the DMA is free) the fourth event – the software interrupt event.

2. The second task is performed by the software interrupt service routine (ISR), as shown on the right side of Figure 4.12. In the Software ISR, a packet is retrieved from the communication buffer and is then processed to start the DMA operation. The priority of the Software interrupt is set lower than the DMA done interrupt. Although interrupt nesting is allowed, the Software interrupt will not be responded to before the DMA done ISR completes.

The flow chart in the middle of Figure 4.12 shows the process of the "Updating the input array" task performed by the DMA done ISR. The DMA buffer with the newly copied Synaptic Block is first located. The communication buffer is then checked to decide whether to trigger another software interrupt (when the comms. buffer is not empty) or to set the DMA flag free (when the comms. buffer is empty). Finally, the Synaptic Block is updated to the input array. Two DMA buffers are used in the system; each with a size of $BlkSiz * 4$ bytes, adequate to accommodate one Synaptic Block. The Synaptic Block is fetched from the SDRAM to the two DMA buffers in turn.

4.8.2 An alternative scheduler

To accommodate the new input processing approach described above, an alternative scheduler is designed with two main changes, as shown in Figure 4.13:

1. The software interrupt event and its ISR are included in the new scheduler.

2. The Timer ISR is simplified to assign values to several global variables: systemTimeStamp (indicating the number of millisecond that have passed) is increased by 1; DelayPtr (referred in section 4.5) is increased by 1; newTimeFlag is set to indicate that the system is moving into a new millisecond. The "updating neuron states" task is performed in the main loop instead

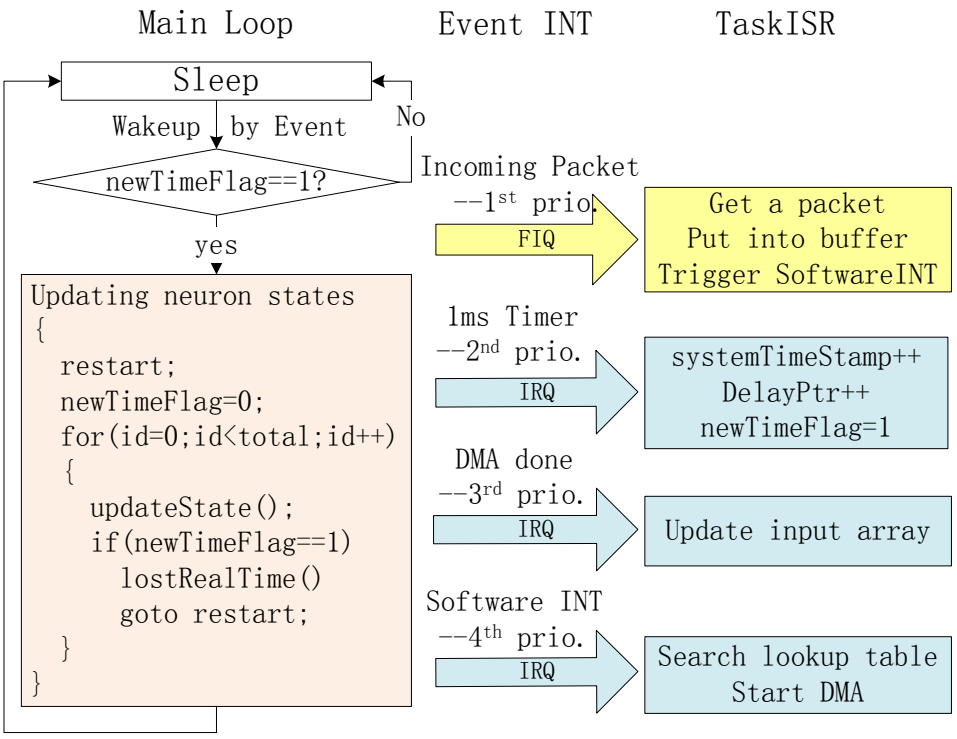


Figure 4.13: An alternative scheduler.

of an ISR. If all neuron state updating has completed, the processor goes to sleep. The sleep mode is interrupted if any event occurs. After executing the corresponding ISR, the processor checks the newTimeFlag status. If the newTimeFlag is set, which indicates a new Timer INT has been generated, "updating neuron states" needs to be performed for the new millisecond. Otherwise, the processor just goes back to sleep mode. In the "updating neuron states" task, firstly, the newTimeFlag is reset. Then the neuron states are updated in a loop, one neuron at a time. If newTimeFlag is detected as set during the updating, the updating will be interrupted compulsorily to skip the remaining neurons. The processor then branches to the beginning of the task to start a new iteration.

Both the schedulers have been implemented and tested to shown that they are able to produce exactly the same results in the real-time simulation.

4.9 A discussion of real-time

SpiNNaker aims to simulate spiking neural networks in real-time with 1ms resolution. As a result, all the tasks must complete before the deadline – a new millisecond Timer Interrupt. Our assumption is the system should not be fully loaded to avoid losing real-time performance. This can be achieved by tuning the number of neurons or connections simulated on each processor.

In the first scheduler proposed as shown in Figure 4.10, the Timer event is set to the lowest priority, and the task "updating neuron states" is executed in the Timer ISR. In this case, a new Timer event will not be responded to until all the tasks are finished. The pro is that it ensures accuracy – the states of all neurons will be updated, and all the inputs will be processed before moving on to the next millisecond. The con is that it results in a poor real-time performance – the response to the Timer event may be delayed. This is a kind of soft real-time system [4].

But how can we know if a processor loses real-time performance? In this case, the Timer Interrupt signal is cleared at the beginning of the ISR. By checking the Timer Interrupt request bit just before exiting the ISR, it is possible to identify whether the response to the Timer has been delayed – if the Timer Interrupt request bit has been set, the response is delayed; otherwise it has not. The delayed response to the Timer causes a disorder of the system timing, but it needs further study to establish how this will affect the functionality of the neural network.

In the second scheduler we proposed as shown in Figure 4.13, a hard real-time [5] scheme is used. In this scheduler, the Timer INT is set to the second priority (or even to the first priority); and the "updating neuron states" task is executed in the main loop instead of the Timer ISR, to keep the Timer ISR slim. This ensures that the Timer Interrupt request is responded to as soon as possible to guarantee the correctness of system timing. But, if an Timer Interrupt comes before the completion of all tasks, the processor is forced to move on to the new millisecond with the remaining tasks discarded. This leads to poor precision.

The type of the neural network is used to decide whether to use a soft real-time system or a hard real-time system. If the timing is more essential than the

[4] The completion of an operation after its deadline in a soft real-time system can be tolerated, but causes decreased service quality

[5] The completion of an operation after its deadline in a hard real-time system is considered useless, and may ultimately cause a critical failure of the complete system.

accuracy of the network, hard real-time is preferred. Otherwise, a soft real-time system offers a better precision.

The second scheduler provides better potential for improvement by using some algorithms to deal with the un-handled tasks. For instance:

1. Instead of simply clearing the packets in the communication buffer, they can be pushed into a history buffer associated with an old time stamp, and continue to be processed during the new millisecond with deferred timing.

2. Instead of skipping un-updated neurons, they can be updated with a reduced time resolution of 2 ms (so only 1 update is performed every 2ms), or with some simplified dynamics.

Chapter 5

Multi-processor simulation

5.1 Overview

In Chapter 4, the algorithm for modeling spiking neural networks on SpiNNaker was described. A single-processor simulation with results was presented to verify the system. In the single-processor simulation, the routing table and the lookup tables were very simple and therefore were setup manually. The neuron data and their synapses were generated randomly on-line. As a result, no external data file is required to start the simulation.

In this chapter, the technique involved in running a multi-processor or multi-chip simulation will be investigated. The question here is, for a given neural network, how to convert the neuronal data and its connection net-list into data sets, and to distribute them onto the SpiNNaker chips. The problem includes a series of sub-issues, such as neuron-processor allocation, synapse distribution, routing planning and routing table generation; these questions are investigated in this chapter. Mapping software called *InitLoad* is developed to help solve this problem more easily. The InitLoad software is not the final solution to these issues however, since there are, no doubt, many other possible solutions. But the development of this prototype software has lead to a better understanding of the issues involved. The simulation results from the four-chip SpiNNaker model, shown later in this chapter, match the results from fixed-point Matlab simulation perfectly, which is a strong evidence to prove the correctness of both approaches and implementations developed for spiking neural network modeling on SpiNNaker. The system is further verified by running a practical neural network application – Doughnut Hunter, on the *real* SpiNNaker Test Chip. The

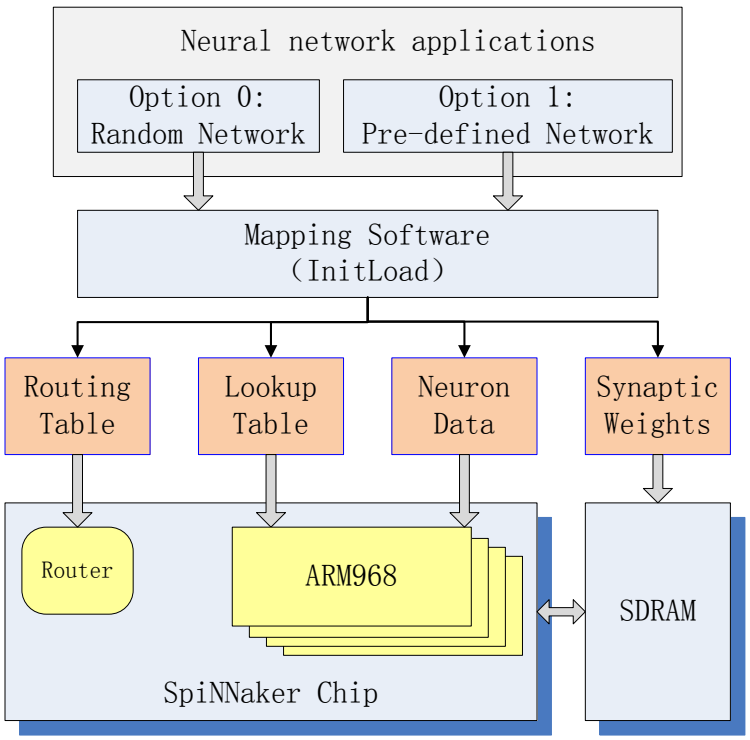


Figure 5.1: Loading neural networks onto a SpiNNaker chip.

investigation and discoveries finally lead to a discussion of the general software model for SpiNNaker at the end of this chapter. The work has been published in [JGP+10].

5.2 Converting neural networks into SpiNNaker format

A neural network is usually described by the neurons and their connections. This information needs to be downloaded and distributed to the processors and chips in SpiNNaker, before the simulation can start. As shown in Figure 5.1, processes of the specification of the neural network is required into a format suitable for loading into a SpiNNaker system, which can be achieved by mapping software called InitLoad.

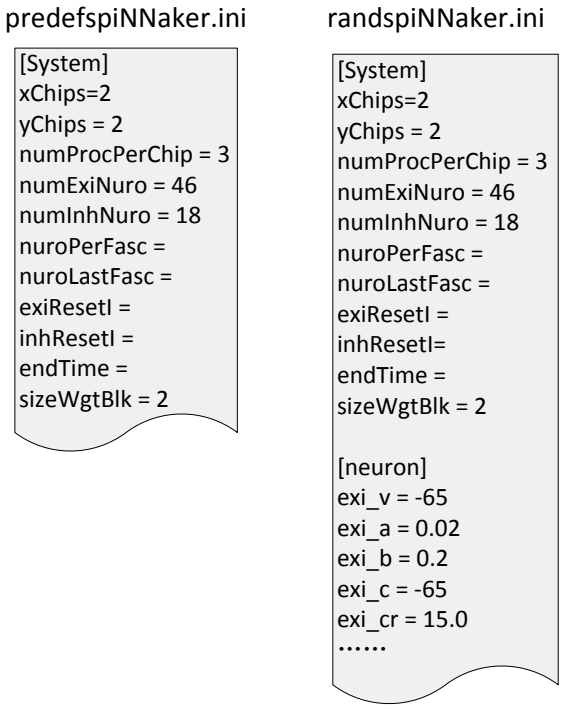


Figure 5.2: The system description files for the random mode and the pre-defined mode.

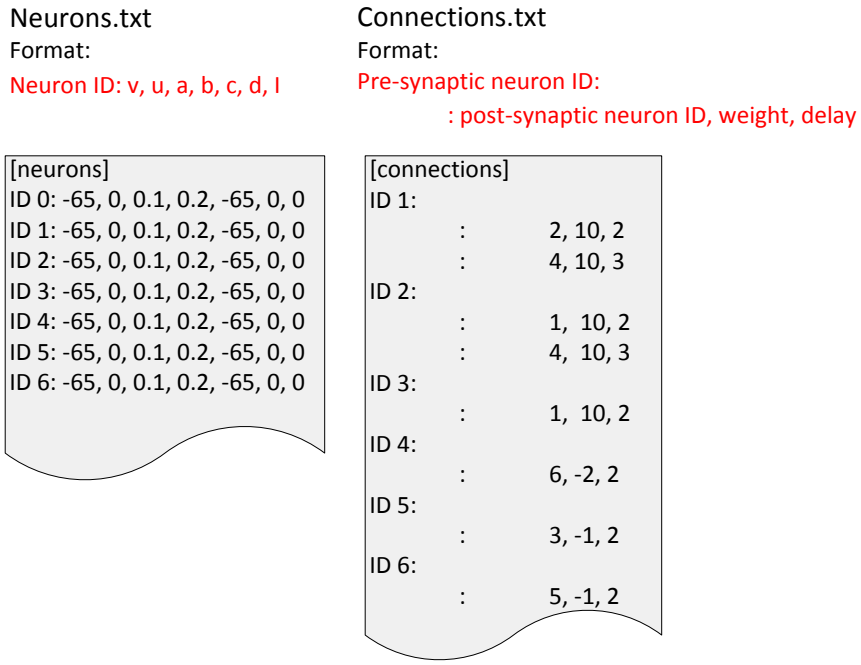


Figure 5.3: The neural network description files and their format.

5.2.1 Neural network description files

Three files are used to describe a system: "*SpiNNaker.ini*", "*Neurons. txt*", and "*Connections.txt*". The SpiNNaker.ini file describes a SpiNNaker system, and the basic information of the neural network. The random and pre-defined modes use different SpiNNaker.ini files, as shown in Figure 5.2, but have the same System part. In the System part, some basic system information is included as follows:

xChips The number of chips in the SpiNNaker system within x dimension.

yChips The number of chips within y dimension.

numProcPerChip The number of processors per chip.

numExiNuro The number of excitatory neurons in the neural network.

numInhNuro The number of inhibitory neurons.

nuroPerFasc Neurons allocated to each fascicle (processor).

nuroLastFasc Neurons allocated to the last fascicle (processor).

exiResetI The reset value of excitatory input current I after update.

inhResetI The reset value of inhibitory input current I after update.

endTime The simulation time in milliseconds.

sizeWgtBlk The size of the Synaptic Block.

The SpiNNaker.ini for the random mode has an additional part called Neuron. This information is used to control the random generation of Izhikevich neurons.

The formats of Neurons.txt and Connections.txt are shown in Figure 5.3. The parameters are set for the Izhikevich neurons.

The description file used here is simple. However, they can easily be extended by using some standard format such as XML for compatibility. Additional information can also be included, if necessary, to describe a more complicated system. Different types of neuronal models, for example the LIF model, will need different formats, because of different neuronal parameters.

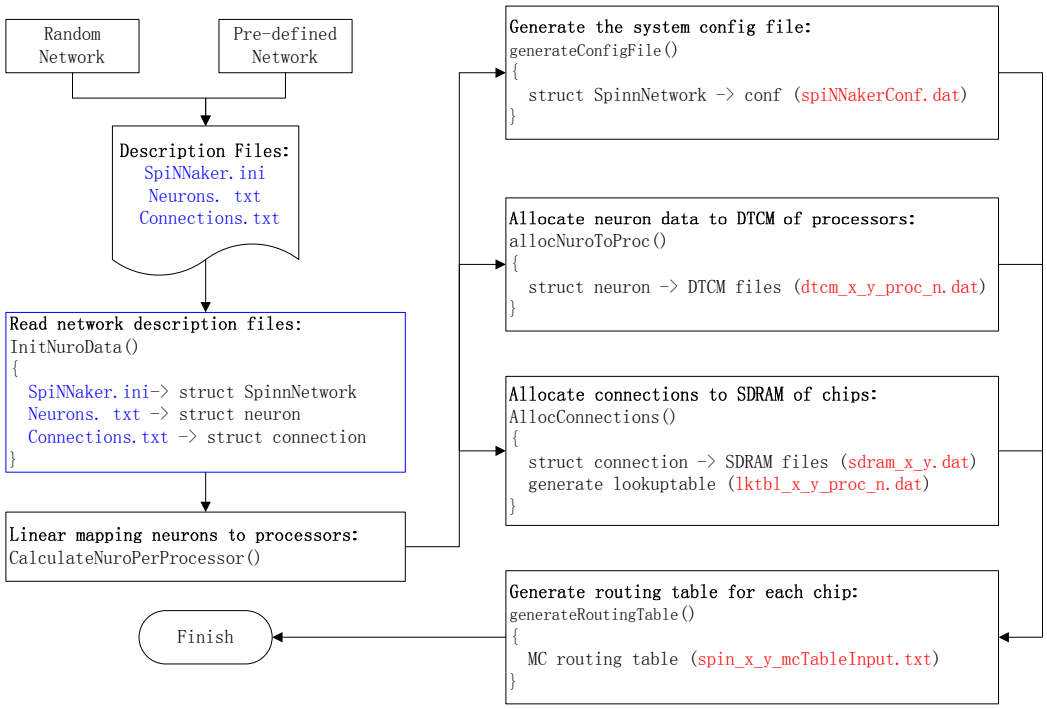


Figure 5.4: The InitLoad flow chart

5.2.2 Mapping flow

InitLoad receives inputs from the neural network description, and generates a set of data files according to the modeling scheme introduced in Chapter 4. These files include a set of neuron data files, a set of synaptic weight files, a set of lookup table files, and a set of routing table files.

Neuron data There is one neuron data file per processor, containing the variables and parameters of the neurons allocated to the specific processor. These files are stored in the DTCM of the processor. Each file is named as "*dtcm_x_y_proc_n.dat*", where x denotes the x-coordinate of the chip, y denotes the y-coordinate, and n denotes the processor ID.

Synaptic weights There is one synaptic weight file per chip, containing the connection (synapse) information allocated to the specific chip. These files are stored in the SDRAM of the chip. Each file is named as "*sdram_x_y.dat*".

Lookup table There is one lookup table file per processor, containing the lookup table used for the event-address mapping (EAM). These files are stored in the DTCM. Each is named as "*lktbl_x_y_proc_n.dat*".

Routing table There is one routing table per chip, containing the routing information for multi-cast packets. These files are stored in the router of the chip. Each is named as "*spin_x_y_mcTableInput.txt*".

Figure 5.4 shows the basic InitLoad flow chart. Two options for neural networks: a random and a pre-defined mode are available. In the random mode, a random neural network will be generated for quick testing purposes. In the pre-defined mode, a user defined neural network can be loaded for testing real applications. Both have three files to describe the system which are loaded into the structures by the InitNuroData() function. The neurons are then allocated to the processors by the CalculateNuroPerProcessor() function which does a linear neuron-processor mapping. According to the number of neurons and their IDs in each processor, the software produces the output files.

5.2.3 Connection data structures

The data structures used to store Connections.txt are shown in Figure 5.5. There are two data structures involved: *Csize[PreID]* and *Conn[PreID][n]*. Where the PreID is the ID of the pre-synaptic neuron. Each element in the Csize array denotes the number of post-synaptic neurons. Each element in the 2D array Conn has the following structure:

```
struct connection
{
    short postid;
    signed short weight;
    short delay;
};
```

where postid is the ID of the post-synaptic processor, weight is the Synaptic Weight of the connection, and delay is the synaptic delay.

The proposed structure shown in Figure 5.5 is not efficient in terms of memory usage. Differences in the number of post-synaptic neurons result in zero elements in the structure. To solve this problem, an alternative compressed data structure can be used, as shown in Figure 5.6. The compressed storage eliminates the zero elements and only stores a connection when it actually exists. The Csize in the compressed storage is used to denote the start of each pre-synaptic neuron in the Conn array.

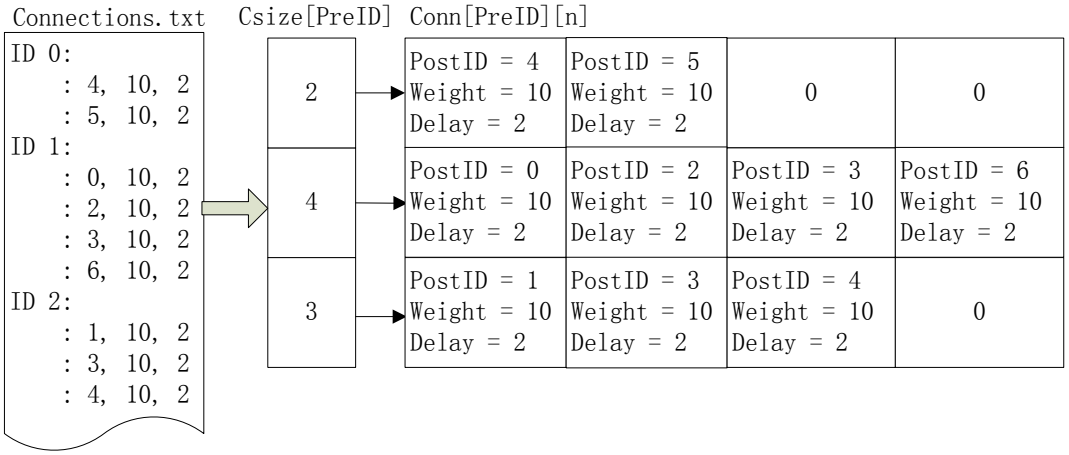


Figure 5.5: The data structure used to load connections.

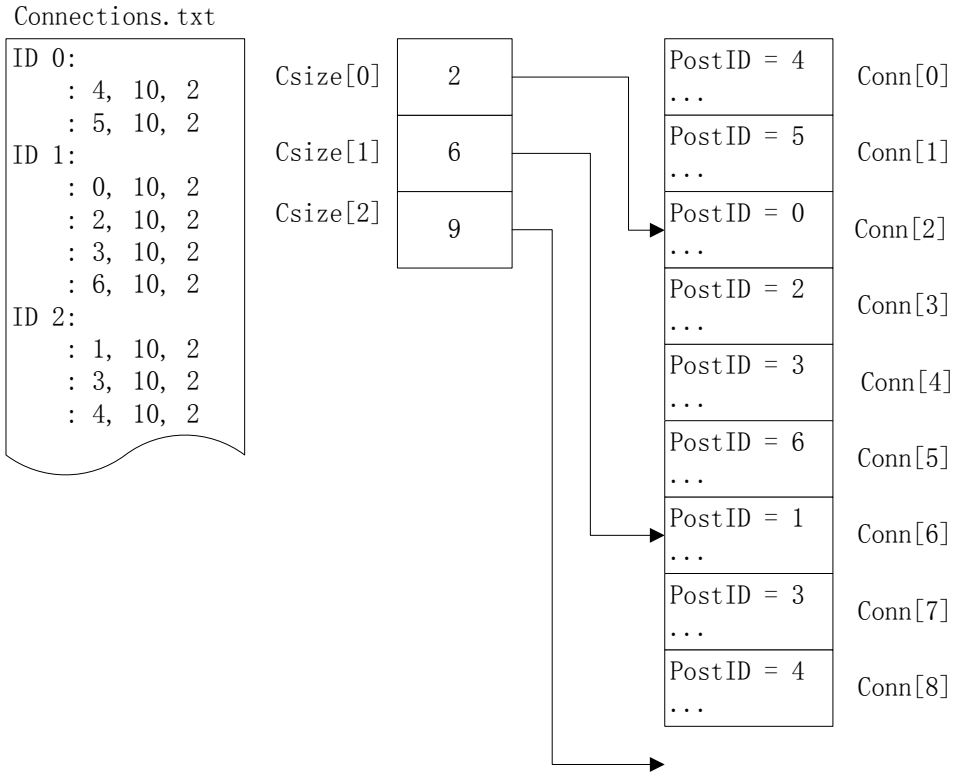


Figure 5.6: An alternative compressed data structure used to load connections.

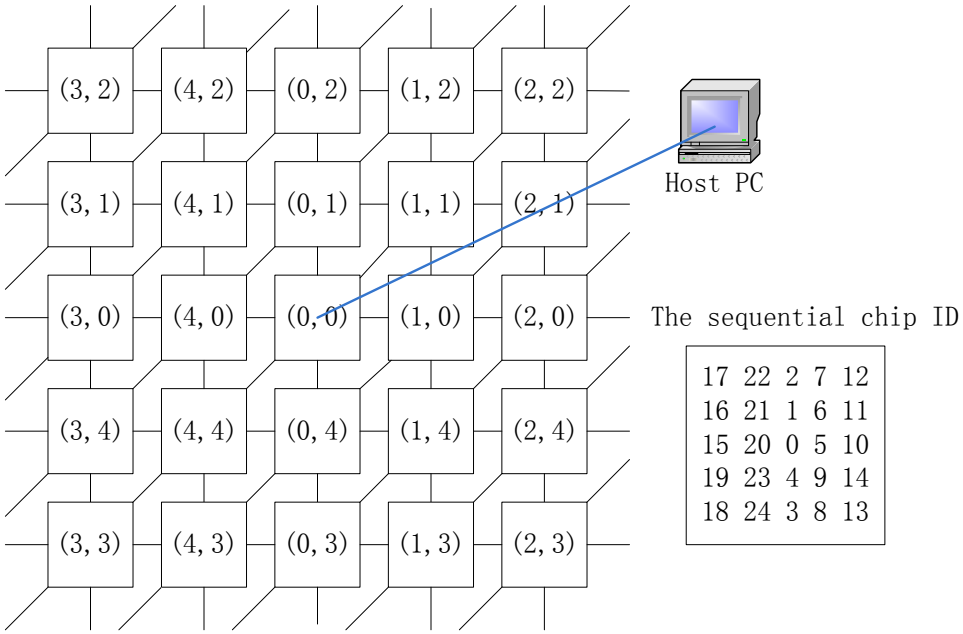


Figure 5.7: Chip indexing.

Chip IDs

The way chip IDs are allocated at present is shown in Figure 5.7. The chip attached to the Host PC is denoted as (0,0). SpiNNaker chips have wrap-around connections. As a result, it is always possible to take the host chip as the origin, and the rest of the chips sit in the first quadrant (I). Thus the rest of the chips can be indexed accordingly using their (x, y) coordinates. A table of the corresponding sequential ID is also shown at the right side of Figure 5.7.

Linear mapping

A linear mapping algorithm (more discussion about the neuron-processor mapping can be find in Section 5.6.1) is used that neurons are uniformly allocated to processors in order:

```
SpiNN.nuroPerFasc = SpiNN.numNuro/(numfasc-1)
SpiNN.nuroLastFasc = SpiNN.numNuro%(numfasc-1)
```

The last processor keeps the remaining neurons. In Figure 5.8, the direction of the connections are from x-coordinate (pre-synaptic) to y-coordinate (post-synaptic), and each processor simulates n neurons using a linear mapping scheme (Proc 0: 0–n, Proc 1: n–2n, ..., Proc 5: 5n–6n). As synaptic weights are stored in the

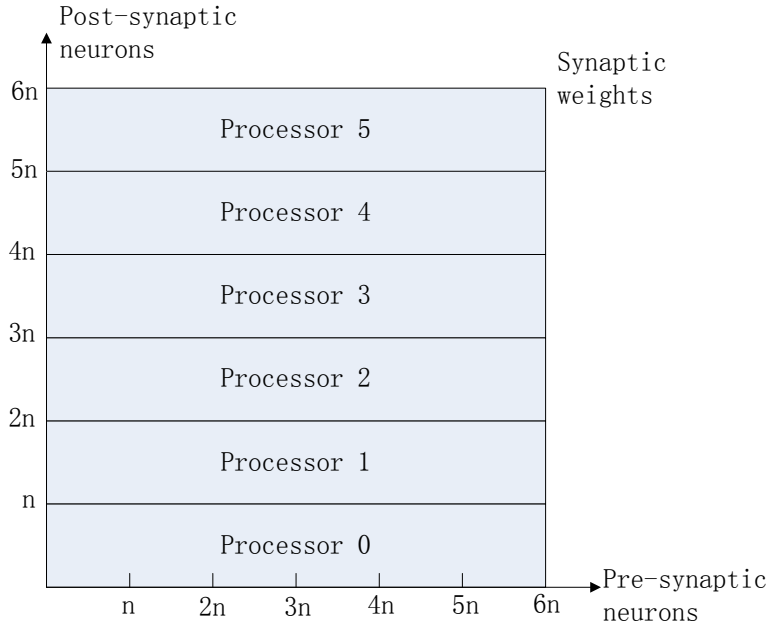


Figure 5.8: The partitions of the weight matrix in the linear mapping algorithm, and their mappings onto the processors.

receiving (post-synaptic) end, the synaptic weight matrix is partitioned through the y-coordinate. The partitions are mapped onto the processors in order with the linear mapping scheme shown in Figure 5.8.

Load balance and synchronization problems

Because the clock domain of each chip is decoupled and each chip is expected to run in real-time, the SpiNNaker system doesn't have a synchronization problem when modeling spiking neural networks. However, this requires a uniform distribution of workloads to guarantee that even the heaviest loaded processor meets the real-time requirement. Otherwise, either the correctness of the timing will be destroyed or the precision will be reduced, as previously discussed in Section 4.9.

5.2.4 Synapse allocation

Synapses (connections) are kept in the SDRAM. They need to be distributed to every chip according to the linear mapping, before simulation can start. The AllocConnections() function in InitLoad is designed to handle this task. In the AllocConnections() function, connections stored in the data structure Conn as shown in Figure 5.5 are gone though and distributed to different locations.

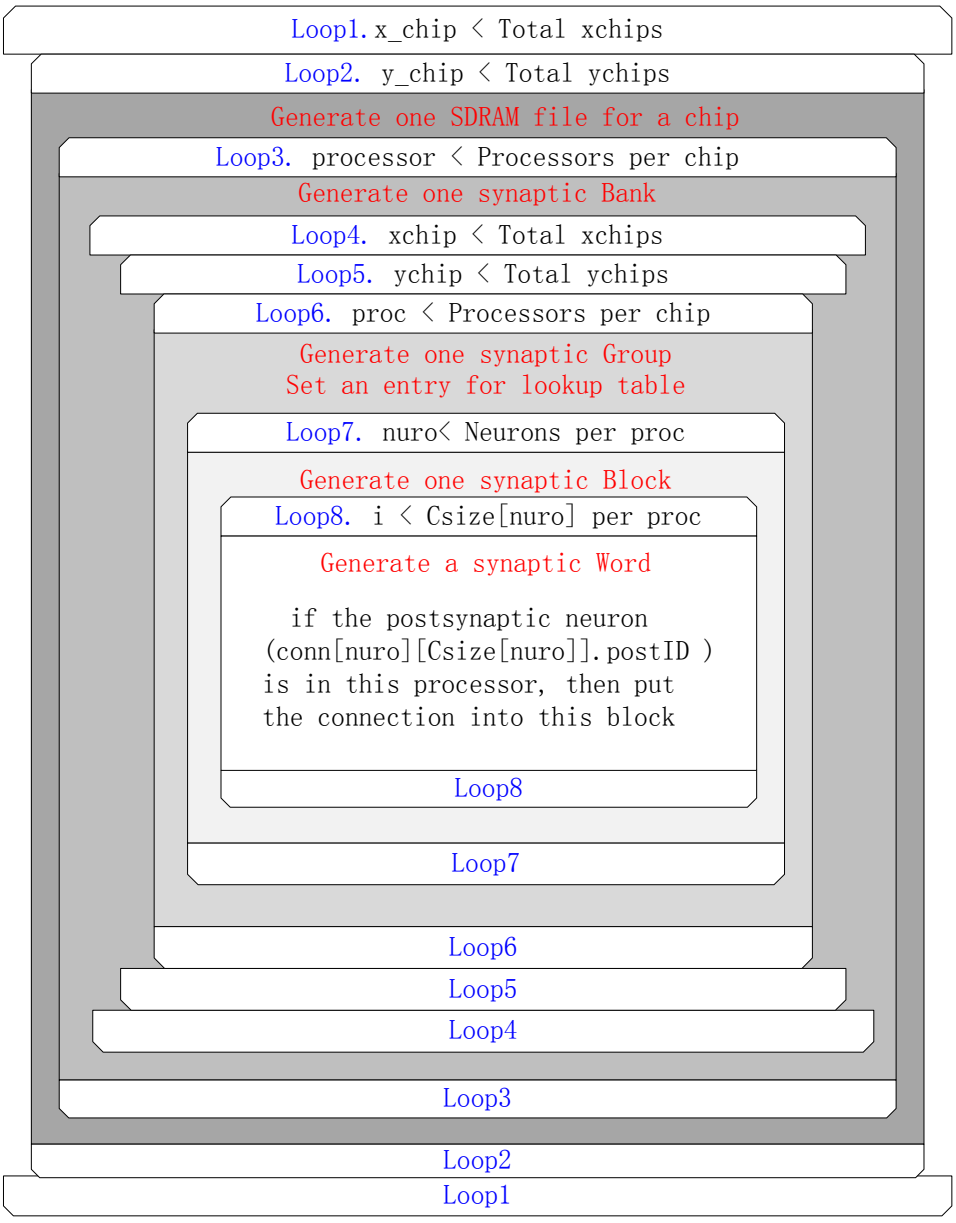


Figure 5.9: The flow of the AllocConnections() function.

There are eight nested loops in total for this task, as shown in Figure 5.9. This Figure is closely related to and can be better understood with reference Figure 4.5. In Loop1 and Loop2, the program generates an SDRAM file containing synapses information for every chip. In Loop3, a Synaptic Bank containing the input synapses received by this processor, is generated within each SDRAM file for each processor on a chip. In Loop4, Loop5, and Loop6, these synapses are checked and the pre-synaptic chips and processors are identified. Within each Synaptic Bank, a Synaptic Group is created, containing all synapses transmitted by the pre-synaptic processor. For each Synaptic Group, an entry is setup in the lookup table at the same time. Then in Loop7, the pre-synaptic neurons are identified. This creates a Synaptic Block for all synapses from the same pre-synaptic neuron. Finally, in Loop8, a Synaptic Word is created for each of the synapses.

5.2.5 Routing table generation

Routing tables are generated by the generateRoutingTable() function, according to the neuron connections, as shown in Figure 5.10. The routing is planned based on the order of the pre-synaptic neurons. For each pre-synaptic neuron, we get its fan-out connections. All the post-synaptic neurons are checked and the fascicle processors where these post-synaptic neurons are located will be identified. The IDs and the total number of these post-synaptic processors, will then be passed to the setupMcRoutingTbl() functions.

The setupMcRoutingTbl() function is responsible for generating routing tables for all synapses from one pre-synaptic neuron. It first calculates the location of the source fascicle where the pre-synaptic neuron is located, and stores it in the Pre data structure. Both Pre and Post data structures have following form:

```
struct coordinate
{
    int x;        //xchip
    int y;        //ychip
    int fasc;     //fasc id
    int nuro;     //neuron id
};
```

Based on the location of the source fascicle, the “mckey” of an entry in the

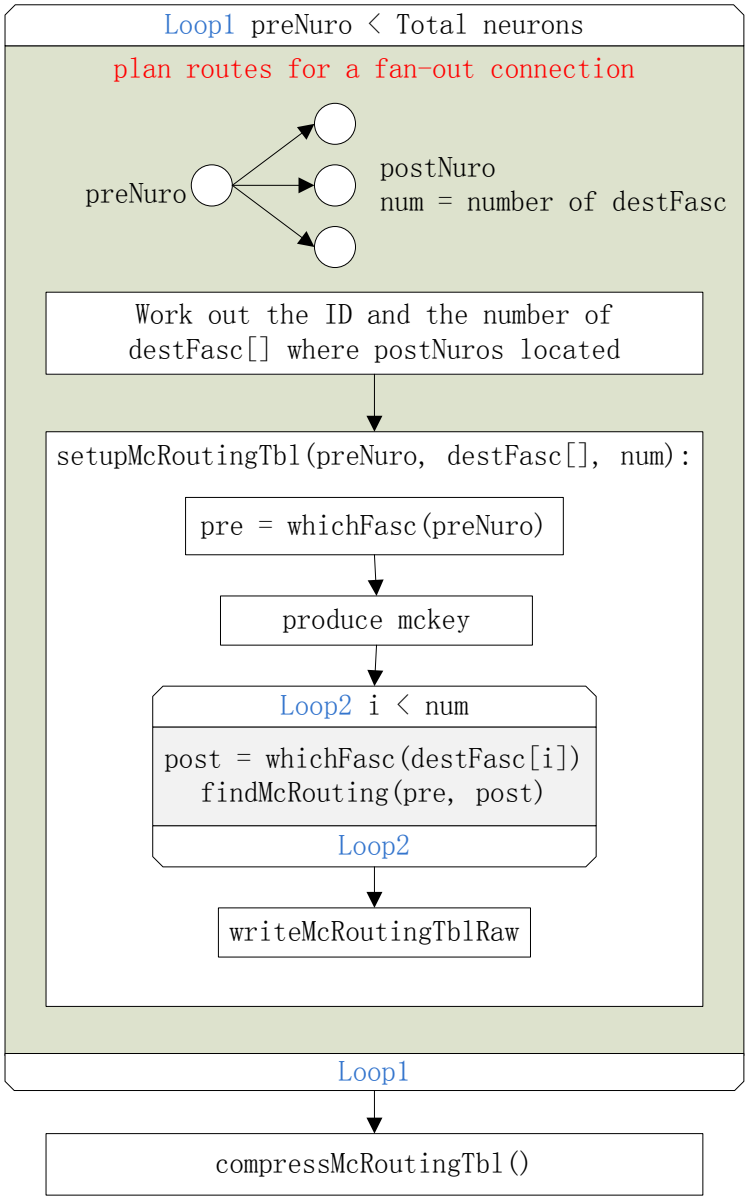


Figure 5.10: The flow of the generateRoutingTable() function.

routing table can be generated. The post-synaptic (destination) processors are then traversed, and their locations will be found. The findMcRouting() function will be used for planning the routing from the source fascicle to the destination fascicle. The routing information will be written to the raw multi-cast routing table.

When the raw routing tables have been generated, the compressMcRoutingTbl() function will be called to compress the routing tables.

Routing table compression

The raw routing table simply stores one route for each pre-synaptic neuron (one mckey), making the routing table large. In practice, most packets from neurons in the same pre-synaptic fascicle will be sent through the same routes. In this case, the MASK of the entry can be set properly to compress the routing table and reduce the number of entries. For example, in the following routing table:

```
mckey    mask        route
0b100    -1          64
0b101    -1          64
0b110    -1          64
0b111    -1          64
```

The four entries can be replaced by the following single entry:

```
mckey    mask        route
0b100    0xfffffffc  64
```

In the kernel of the compressMcRoutingTable() function is a third-party Logic Minimization Software – Espresso. Espresso was developed to help design large digital logic, particularly logic with Boolean functions that have more than 4 inputs or with multiple Boolean functions. It takes as input a two-level representation of a two-valued (or multiple-valued) Boolean function, and produces a minimal equivalent representation. The compressMcRoutingTable function uses an external call to Espresso to fulfill the task of compressing the routing table.

Route planning

The findMcRouting() function is used to plan the routing. There are a variety of possible algorithms for route planning on SpiNNaker. The one used for

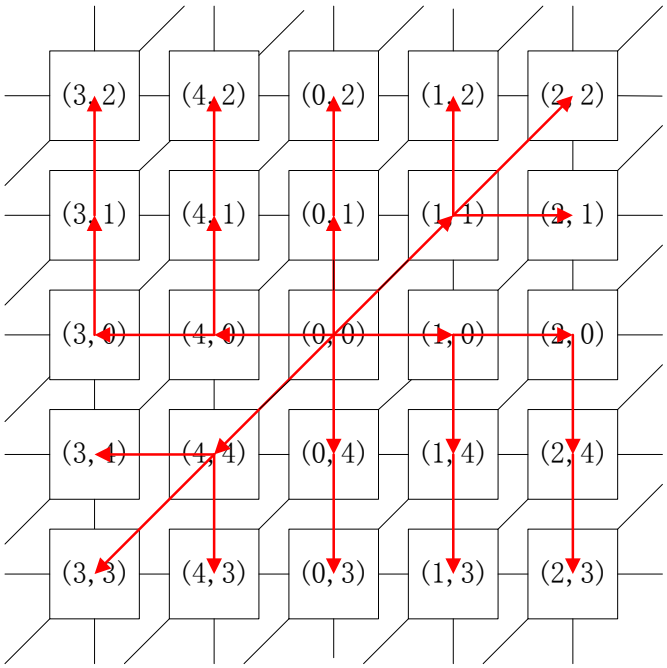


Figure 5.11: Routing planning

route planning in this system is shown in Figure 5.11. Assuming that a packet is transmitted by a neuron in chip (0,0), if the destination is in the first quadrant (I) or the third quadrant (III), the packet will firstly be sent in the diagonal direction. When it reaches the chip with the same x-coordinate or y-coordinate as the destination chip, it will then moves towards the destination following the x-/y-coordinate vertically/horizontally. If the destination is in the second quadrant (II) or fourth quadrant (IV), the packet firstly moves horizontally following the x-coordinate. When it reaches the chip with the same x-coordinate as the destination chip, it will then moves vertically towards the destination following the y-coordinate.

The route planning algorithm we used is simple and straightforward. It guarantees to find a very short path from one neuron to another. However, there are many possible routes between each pair of neurons. As a result, this is definitely not the only algorithm. If a more advanced route planning algorithm is developed, it can easily be integrated into the InitLoad software by replacing the existing findMcRouting() function.

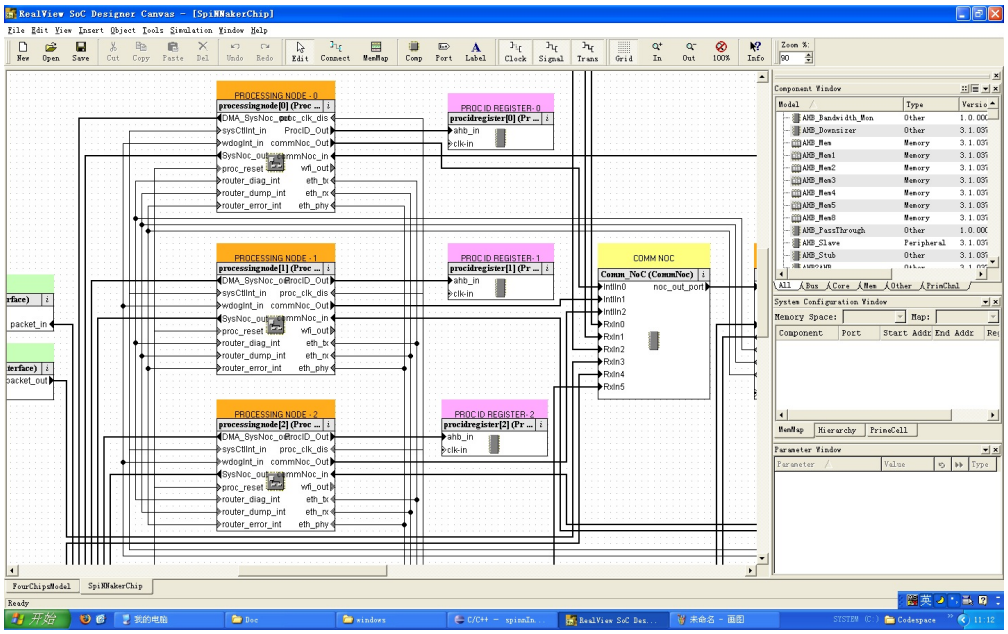


Figure 5.12: SpiNNaker chip components in SoC Designer

5.3 The multi-chip SoC Designer platform

Since the physical SpiNNaker hardware is not available at the time of this research, a virtual environment, SoC Designer, is used as the simulation platform. SoC Designer, provided by the Carbon Design Systems company (formerly maintained by ARM Ltd.), is a complete solution for the development, execution, and analysis of virtual system platforms. It offers an environment for easy modeling and fast simulation of integrated systems-on-chip with multiple cores, peripherals and memories using C++ (based on the SystemC language). The cycle-based scheduler and the transaction-based component interfaces enable high simulation speed while retaining full accuracy [Ltda]. Compared to the previously used RVISS model, the SoC Designer model is more accurate in terms of both the architecture and timing. The SpiNNaker model on SoC Designer was developed by M.M. Khan in the SpiNNaker group [KPJ+09], a four chip model is shown in Figure 5.13. The components of a SpiNNaker chip in SoC Designer are shown in Figure 5.12.

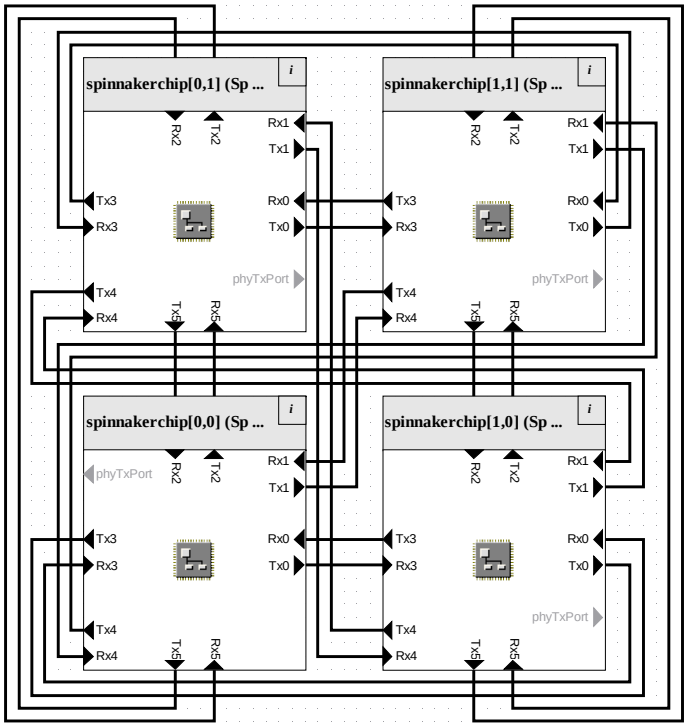


Figure 5.13: A four chip model in SoC Designer

5.4 Simulation results

5.4.1 Initialization

There are two processors (the same number of processors as in the SpiNNaker test chip), a Fascicle and a Monitor Processor, in each chip of the four-chip model. The Fascicle and the Monitor Processor load the same code at startup, as shown in Figure 5.14, each processor then performs a series of initialization processes. The Fascicle Processor mainly loads its private data set, while the Monitor Processor loads public data, such as synaptic weights and routing tables, which is shared by all the Fascicle Processors (if there are more than one Fascicle Processors) on the same chip.

The timer on the Fascicle Processor is started after all other initialization processes (in both Fascicle and Monitor processors) complete, since the timer interrupt triggers the neuron state update, which requires all the data to be loaded. To avoid conflict, a handshake procedure is introduced. When the Fascicle Processor finishes initialization, it sends a request to the Monitor Processor to check whether the initialization process on the Monitor processor is finished. When

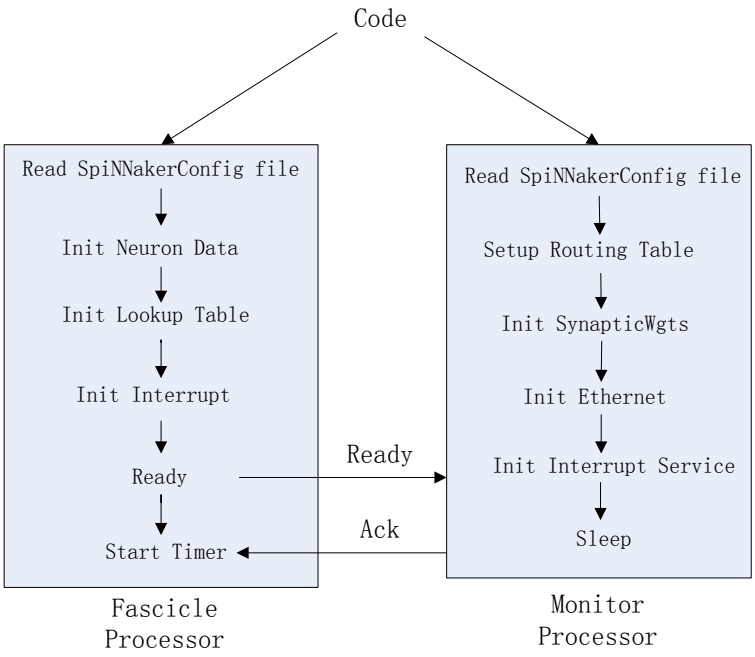


Figure 5.14: The fascicle and monitor processors during initialization.

an acknowledgement from the Monitor Processor is received, the timer will be started. The Monitor Processor functions is very simple in this test. Users are able to develop more comprehensive applications for the Monitor Processor if necessary.

5.4.2 Downloading and dumping

The application code and neural data need to be downloaded to SpiNNaker chips to start the simulation. The semihosting functions provides by SoC designer allow user to access files on the hard disk of the Host PC directly, making the code and data downloading easy. To dump simulation results (spike raster, single neuron activity, and so on) is is also easy during the simulation. The output data is written to files on the Host PC and plotted in the Matlab.

5.4.3 A 60-neuron network test

Due to the limited speed of running the SoC Designer on a desktop computer, the system is initially tested by a small scale neural networks with a total of 60 random Izhikevich neurons. The ratio of excitatory neurons to inhibitory neurons is 4:1 – 48 excitatory neurons and 12 inhibitory neurons. Each excitatory neuron

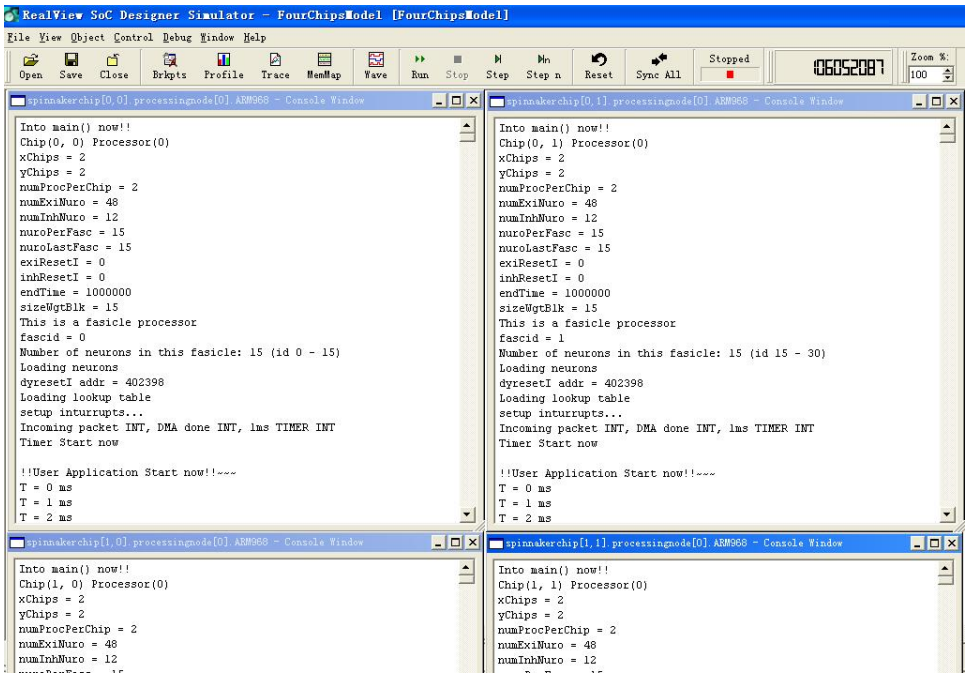


Figure 5.15: A four chip simulation running on the SoC Designer simulator

is randomly connected to 15 neurons, and each inhibitory neuron connects to 15 excitatory neurons, with a random delay of 0–15 ms. Among these neurons, 4 excitatory neurons and 1 inhibitory neuron are chosen as biased neurons, each receives a constant input current with an amplitude of 20 mV. The neural network data is converted by the InitLoad software with the "pre-defined network" option and downloaded to SpiNNaker chips with 15 neurons per fascicle processor.

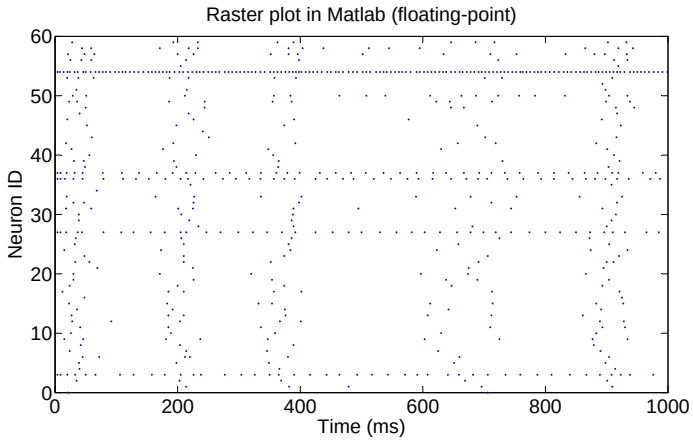
Spike raster

Figure 5.15 shows the neural network being simulated in the SoC Designer Simulator. Neuron states will be dumped to files on the hard disk of the host PC, using semihosting functions. The simulation runs for 1,000 ms.

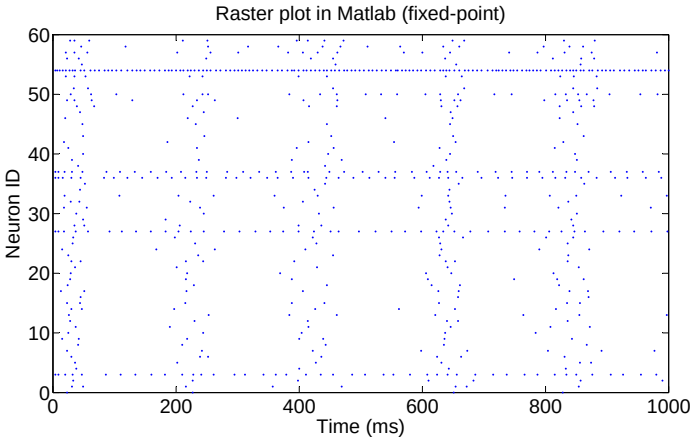
In Figure 5.16, spike raster results generated from three different simulations are listed: Matlab simulation with floating-point arithmetic[1] (Figure 5.16(a)), Matlab simulation with fixed-point arithmetic (Figure5.16(b)), and SpiNNaker simulation on SoC Designer (Figure 5.16(c)).

In Figure 5.16(c) neurons below the red line (at neuron ID 48) are excitatory and ones above the red line are inhibitory. The red lines at the bottom indicate the

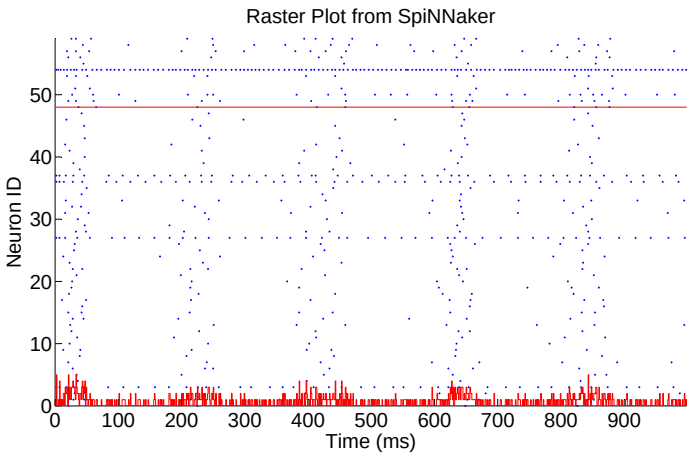
[1]The Matlab code used in the rest of the thesis, is largely based on the code provided in [Izh06]



(a) Spike raster of the 60-neuron simulation on Matlab using floating-point arithmetic.

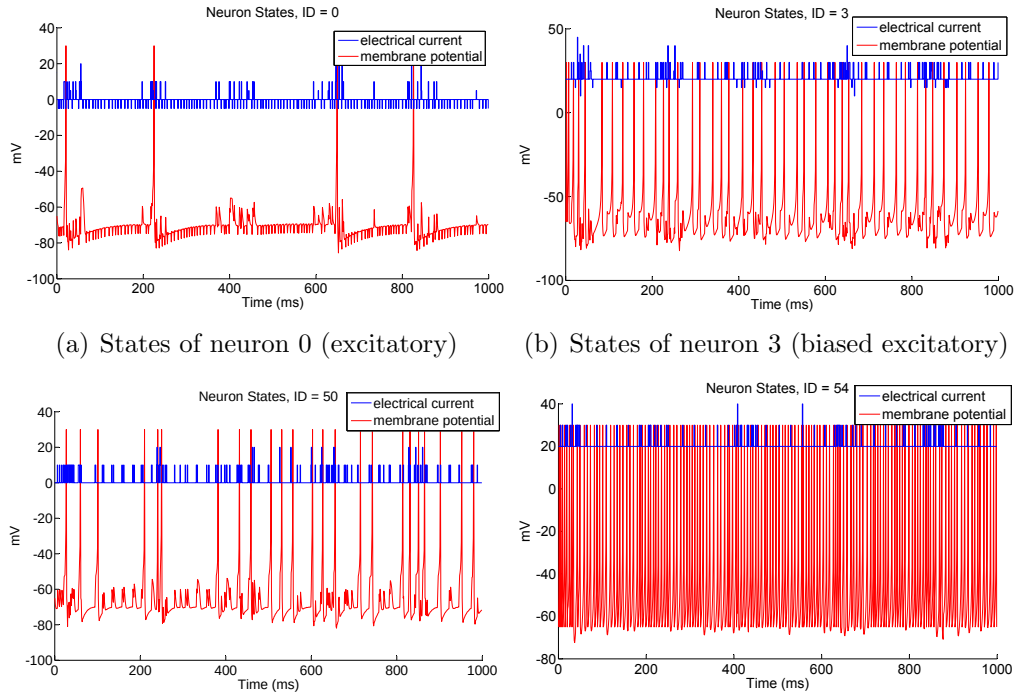


(b) Spike raster of the 60-neuron simulation on Matlab using fixed-point arithmetic.



(c) Spike raster of the 60-neuron simulation on SpiNNaker.

Figure 5.16: Spike raster generated from 60-neuron network simulations within 1,000 ms.



(a) States of neuron 0 (excitatory)

(b) States of neuron 3 (biased excitatory)

(c) States of neuron 50 (inhibitory)

(d) States of neuron 54 (biased inhibitory)

Figure 5.17: Neuron states and input currents from SpiNNaker simulation

neuron firing rates at specific times. The biased neurons chosen are ID 3, 27, 36, 37 and 54. These neurons keep firing at all times and thereby trigger other neurons to fire (burst). There are fewer inhibitory neurons than excitatory neurons, but they are tuned to have higher firing rates. Since inhibitory neurons connect only to excitatory neurons, their firings shut down the bursting periodically.

Each of the three Figures in 5.16 clearly shows five bursts within the simulation time, corresponding to a rhythmic activity with a frequency of 5 Hz. There are reasonable differences between Figure 5.16(a) and Figure 5.16(b), caused by the differences between floating-point and fixed-point arithmetics. The spike timing in Figure 5.16(b) and Figure 5.16(c) is exactly identical – a strong demonstration of the functional correctness of the SpiNNaker model.

Single neuron activity

Figure 5.17 shows the membrane potential (red) and input current (blue) of several neurons from the SpiNNaker simulation. Here sub-figure 5.17(a) shows a normal excitatory neuron (ID 0) without biased input. It receives both excitatory and inhibitory inputs. Sub-figure 5.17(b) shows a biased excitatory neuron (ID

3) with a base input current of 20. It receives both excitatory and inhibitory inputs. Neuron 3 fires much more frequently than neuron 0 because of the biased input. Subfigure 5.17(c) shows a normal inhibitory neuron (ID 50) without biased input, which receives only excitatory inputs. Subfigure 5.17(d) shows a biased inhibitory neuron (ID 54) with a base input current of 20, which receives only excitatory inputs. As a result, neuron 54 fires more frequently than neuron 50. It is obvious that in this test inhibitory neurons fire more than excitatory neurons.

5.4.4 A 4000-neuron network test

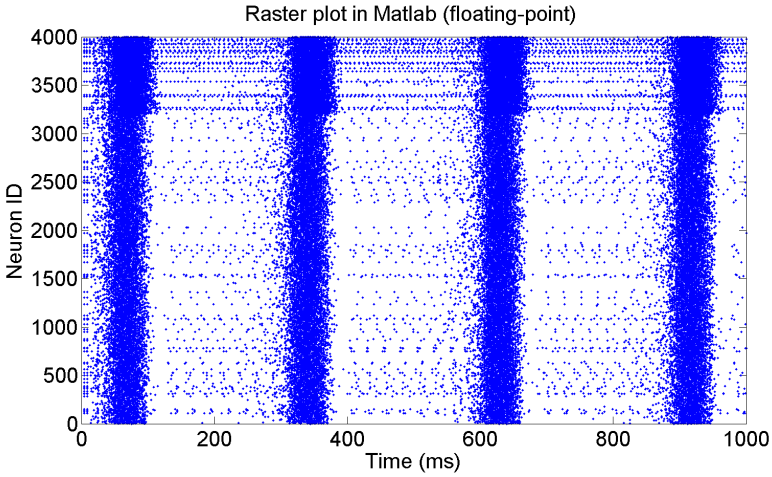
Another test involves a simulation of a 4,000-neuron network (1,000 neurons per fascicle) with an excitatory-inhibitory ratio at 4:1. Each neuron is randomly connected to 26 other neurons. 72 excitatory and 18 inhibitory neurons are randomly chosen as biased neurons, each receiving a constant input stimulus of 20 mV. The simulation results are compared between the floating-point arithmetic Matlab simulation (Figure 5.18(a)), fixed-point arithmetic Matlab simulation (Figure 5.18(b)), and 4-chip SoC Designer based SpiNNaker simulation (Figure 5.18(c)). The spike timings in the floating-point and fixed-point arithmetic Matlab simulations are different, however, they show the same rhythm of 4Hz. The 4-chip SpiNNaker simulation matches the fixed-point arithmetic Matlab simulation. Figure 5.19(a) shows the activity of an excitatory neuron (ID 0) and Figure 5.19(b) shows the activity of an inhibitory neuron (ID 3200), produced in the SpiNNaker simulation.

This test shows that the scale of the system can easily be increased when necessary. Each processor in a SpiNNaker chip is capable of modeling 1,000 neurons, indicating that to model a human brain with 100 billion (10^{11}) neurons, it will require 5 million full SpiNNaker chips with 20 processors per chip and consume 2.3 MW to 3.6 MW (based on power estimation provided in [PFT+07]).

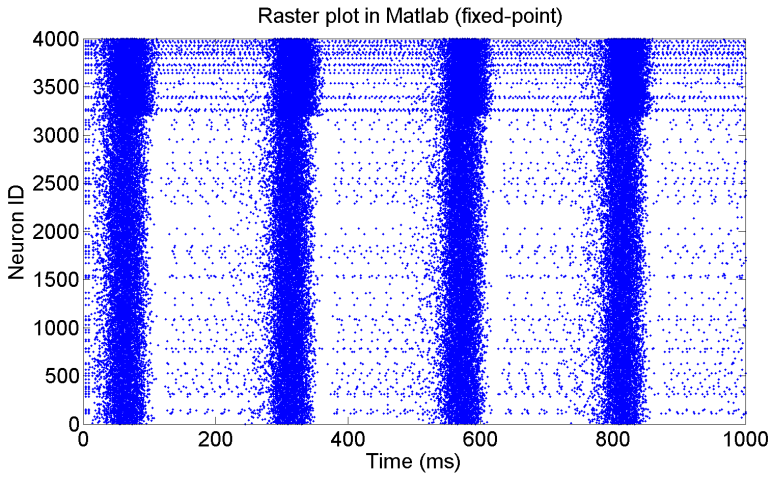
5.5 A practical application

5.5.1 The Doughnut Hunter application

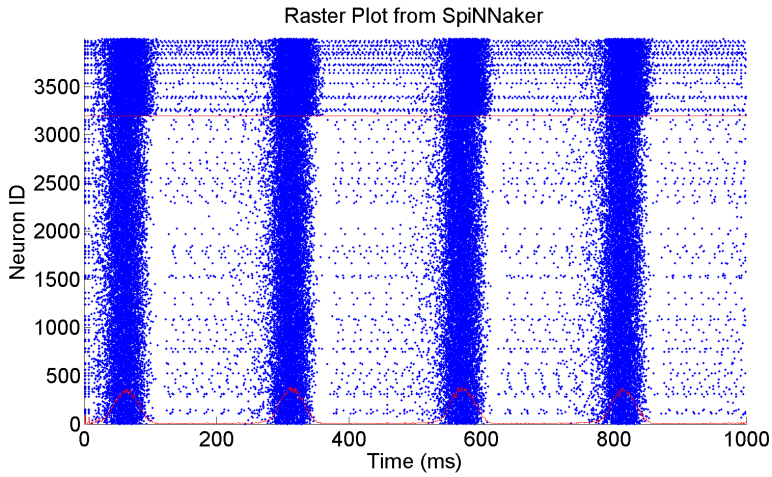
The Doughnut Hunter application was originally developed by Arash Ahmadi at Southampton and was ported onto SpiNNaker for testing. The model requires two application components: a server on the Host PC and a client on the SpiNNaker



(a) 4000-neuron floating-point Matlab simulation.

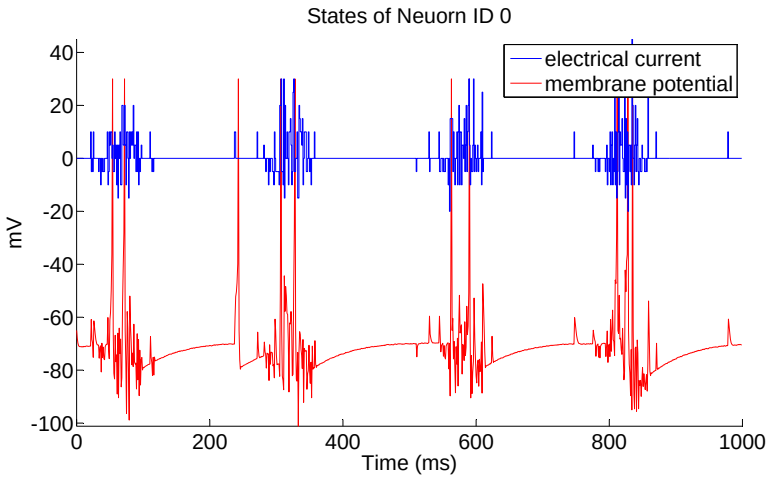


(b) 4000-neuron fixed-point arithmetic Matlab simulation.

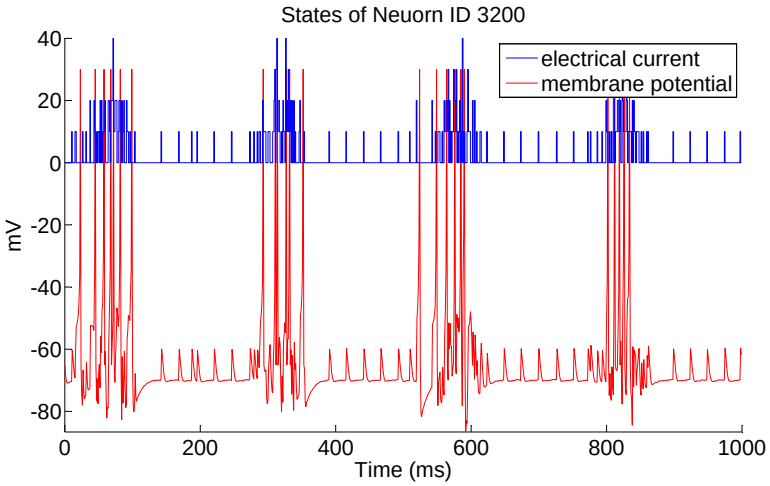


(c) 4000-neuron SpiNNaker simulation.

Figure 5.18: Spike raster of 4000 neurons on SpiNNaker.



(a) States of neuron 0 (excitatory)



(b) States of neuron 3200 (inhibitory)

Figure 5.19: Single neuron activity of 4000-neuron simulation on SpiNNaker.

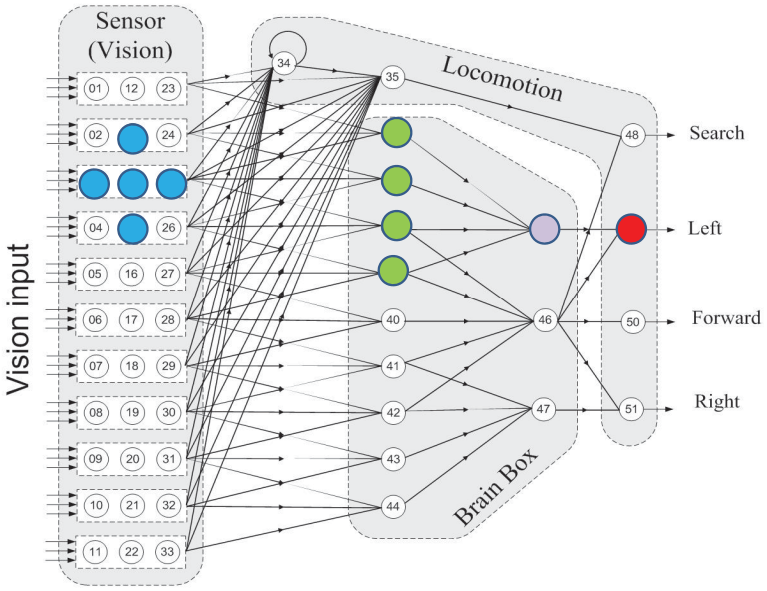


Figure 5.20: The Doughnut Hunter model by Arash Ahmadi and Francesco Galluppi.

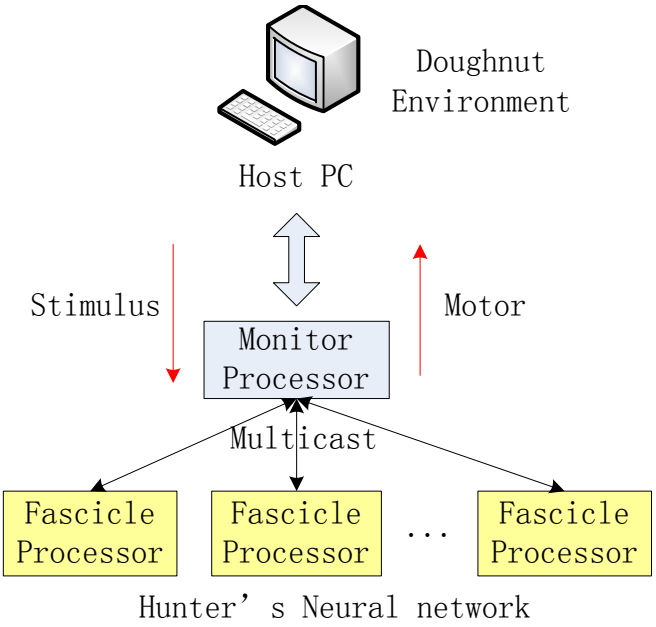


Figure 5.21: The platform for the Doughnut Hunter application.

as shown in Figure 5.21[2]:

1. The server runs a GUI application modeling the environment with a doughnut and a hunter. The doughnut appears at a random location on the screen, and the hunter moves on the screen to chase the doughnut and eat it. The server detects the location of the doughnut and the vision of the hunter at a certain frequency. These signals will be converted and sent to the client as input stimuli.

2. The client runs the neural network of the hunter. It receives inputs from the server, and propagates them to the motor neurons via a simple neural network (as shown in Figure 5.20). The motor signal will be sent back to the server and used to control the hunter movement. The connection between the server and client is via the Ethernet interface [Kha09].

Several colleagues who involved in this work were: Francesco Galluppi, Cameron Patterson, Mukaram Khan, and Xin Jin. Francesco Galluppi created and tuned the neural network model, Cameron Patterson and Mukaram Khan built the Ethernet connection between the Host PC and the SpiNNaker chip, and Xin Jin provided software support for SpiNNaker to run the model.

Originally, the LIF model was used as the neuronal dynamics of the Hunter's neural network and it was replaced by the Izhikevich model in this test. There are 51 Fast Spiking Izhikevich Neurons in the network. Their connections are shown in Figure 5.20. The details of the network are listed below:

1. Neurons are sorted into three layers: 33 in the visual layer, 12 in the brain layer, and 6 in the motion layer.

2. The delay is set to 1 millisecond between each pair of neighboring layers.

3. The connection weights are well tuned without plasticity.

4. Neurons 34 and 35 trigger the search pattern.

5.5.2 Simulation results

The simulation runs on the SoC Designer model and Figure 5.22 shows the environment displayed on the Host PC during the simulation. This shows a 3-D view

[2]Only one fascicle processor is used in this test

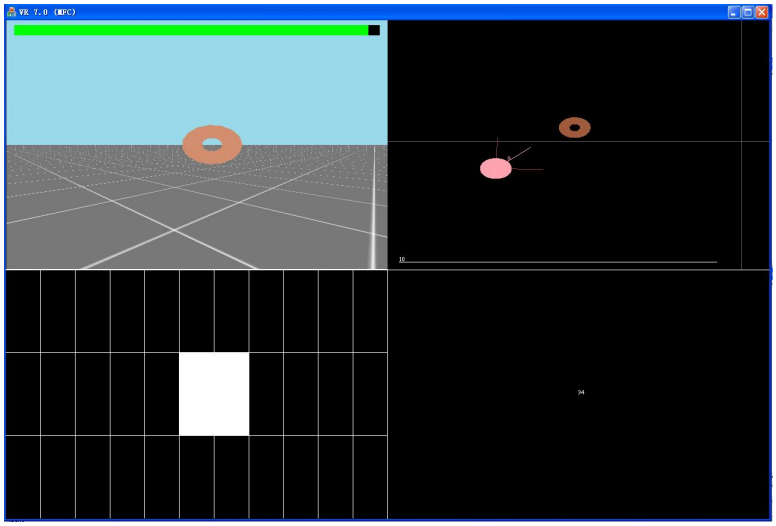


Figure 5.22: The Doughnut Hunter GUI during simulation.

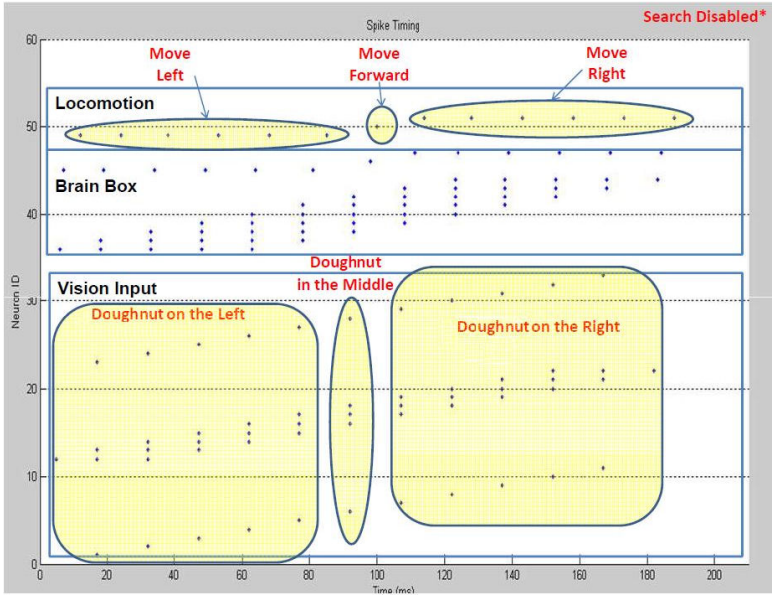


Figure 5.23: Neuron firing timing in the Doughnut Hunter simulation.

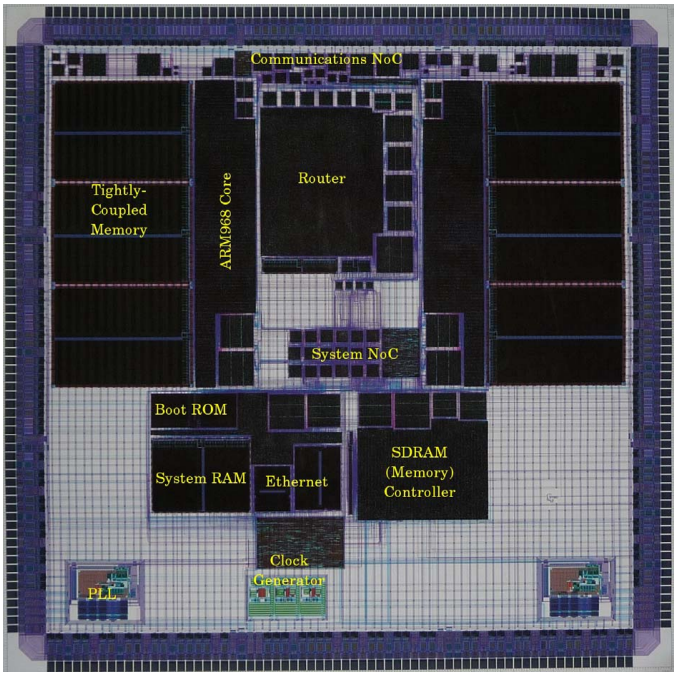


Figure 5.24: The test chip diagram.

of the doughnut, the movement and the vision of the hunter. The neuron activities are shown in Figure 5.23 where the corresponding activity and vision of the hunter are shown in yellow squares. Although the neural network used is small, the successful simulation of the Doughnut Hunter application is a verification of the SpiNNaker system.

5.5.3 The Doughnut Hunter on a physical SpiNNaker chip

The Doughnut Hunter model was also tested on the physical hardware – the SpiNNaker Test Chip (as shown in Figure 5.24). The Test Chip was released in December 2009. There are two processors on each chip. Figure 5.25 shows a SpiNNaker Test Chip plugged in to the PCB. A debug board is attached to the GPIO port of the PCB to provide the serial port connection to the Host PC. The development environment for the Test Chip, including bootup process, programme downloading, and so on, was created by Steve Temple in the SpiNNaker group. The process to run the Test Chip is shown in Figure 5.26:

1. An alternative boot programme (Bootup2) is downloaded from the Host PC to the serial ROM.

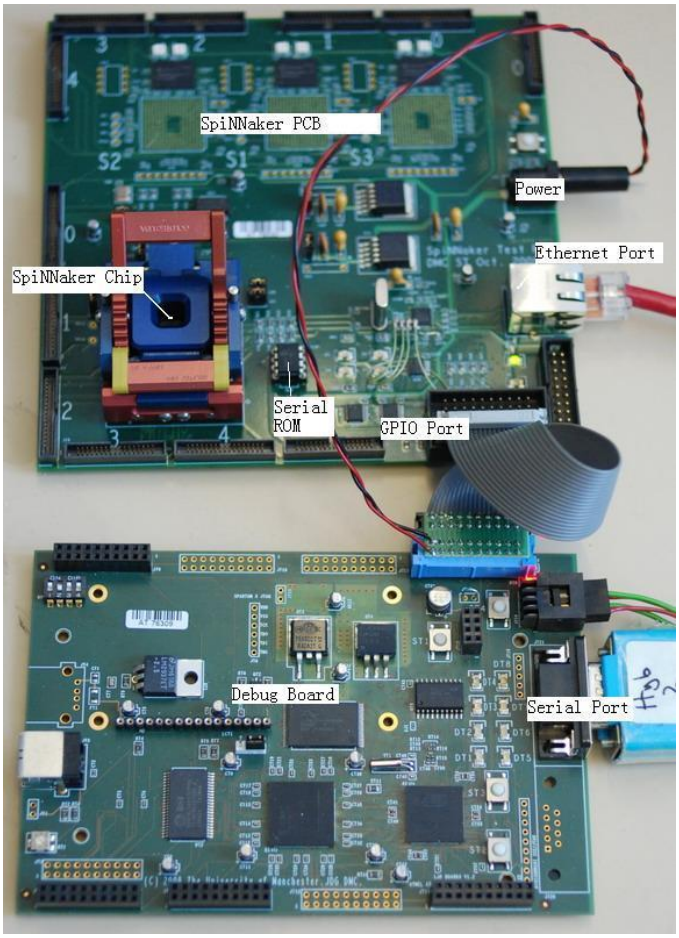


Figure 5.25: A photo of the SpiNNaker Test Chip on the PCB.

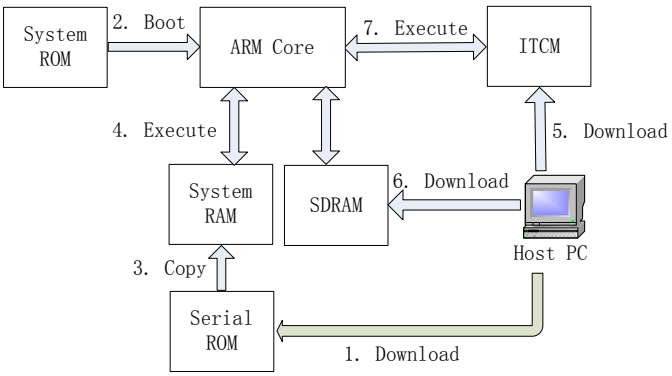


Figure 5.26: The bootup process of the Test Chip.

2. The processor boots from the Boot ROM (Bootup1).
3. The Bootup1 code copies Bootup2 code from the serial ROM to the System RAM.
4. The processor branches to the System RAM, and starts executing Bootup2 code.
5. The Bootup2 code enables the serial port, and allows the Host PC to download the application code to the ITCM.
6. The Host PC downloads neural data files to the SDRAM.
7. The processor branches to the ITCM, and starts executing the application code.

During the Doughnut Hunter test, the neural data files are downloaded to the SDRAM, and the neural code previously developed for the SoC Designer model is downloaded to the ITCM. When the neural code starts executing, the processor reads data files from the SDRAM, and configures the routing table, the lookup table, and DTCM accordingly. The communication between the client (the SpiNNaker chip) and the server (the Host PC) goes through the Ethernet connection. The simulation runs successfully in real-time performance.

5.6 Discussion

5.6.1 Neuron-processor mapping

The allocation of neurons to processors is an essential problem which affects the efficiency of both the communication and the synaptic weight storage. A simple linear neuron-processor mapping scheme is used in this implementation, as presented in Section 5.2.3. This problem, however, deserves further investigation.

In principle, locality issues have to be taken into consideration to allocate relatively highly-connected nearby neurons onto the same or nearby processors as much as possible. Figure 5.27 shows an example of an ideal neuron-processor mapping. Neurons in neighboring processors have a high density of connection, while the ones in remote processors have a low density of connections.

There are three aspects that need to be taken into consideration for the neuron-processor mapping:

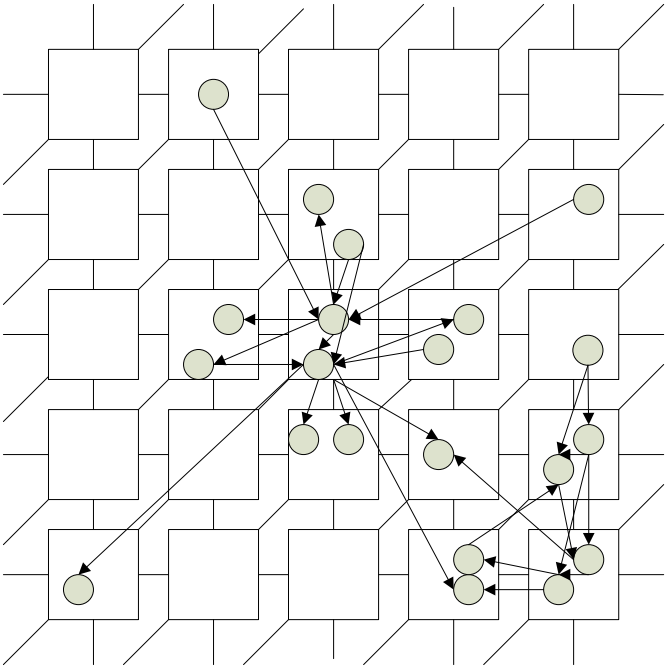


Figure 5.27: An ideal neuron-processor mapping.

- Chip index. SpiNNaker chip IDs are purely software defined. The physical distance information needs to be included in the index for a well-defined mapping model. Each SpiNNaker chip contains up to 20 processors. The index of processors in the same chip does not matter since it takes the same time for them to communication to other chips.

- Neuron index. Neuron IDs are usually allocated randomly when a neural network is built, ignoring their "distances". Here the "distance" denotes the biological distance – how far a pair of neurons are apart from each other. A long axonal delay usually implies a long distance. For a well-defined neuron-processor mapping model, the "connectivity" and "distance" need to be considered when neurons are indexed.

- Building relations between the chip IDs and the neuron IDs. The principle is that neurons nearby (at a short distance) or with high connectivity need to be mapped on to the same or neighboring SpiNNaker chips.

The development of the neuron-processor mapping model for spiking neural networks relies on knowledge of a variety of spiking neural networks, and a well-defined mathematic model of mapping.

5.6.2 Code and data downloading

On a physical SpiNNaker machine, the downloading process will become a major issue involved in the Boot-up Process [Kha09]. One difficulty is caused by the limited bandwidth between the SpiNNaker system and the Host PC, as Chip (0,0) is the only chip connected to the Host PC via Ethernet. Other chips will rely on their neighboring chips to receive data. To address this problem, a flood-fill mechanism is proposed where during the inter-chip flood-fill process, each chip sends one 32-bit word at a time by Nearest Neighbor packets to its 6 neighboring chips. The receiving chips then forward the data to their neighbors, so in this way, the data is propagated to all the chips in the system.

The application code is identical for every chip and processor, and therefore can be handled efficiently by the flood-fill mechanism. The neural data, on the other hand, is different for each chip or processor. The size of the data also increases when the system is scaled up. For a small system the flood-fill mechanism is capable to download neural data produced by the InitLoad software. For a large system, however, the huge amount of neural data results in an inefficient process. An alternative on-line data generation scheme is therefore required to send only statistical information of a neural network by the flood-fill mechanism. Each processor generates the real neural data as well as routing tables on-line according to the statistical information. This involves another potential research topic requiring a number of related issues to be solved, including how to produce the statistical information for a given neural network, how to modify routing tables dynamically, how to generate the synaptic weights for the event-address mapping scheme, and so on.

5.6.3 Simulation results dumping

It is probably not feasible to dump the state of every neuron from the human brain. Usually, the state of a single neuron is not of direct interest, but the global output (for example the human language or behaviors) of the brain is of concern. If neuronal behaviors in a certain brain region are of interest during neuronal study, they can be observed using techniques such as glass electrodes, MRI or PET.

The SpiNNaker system is quite like the human brain in this respect. It is difficult to observe the activity of all neurons in the system because of the limited

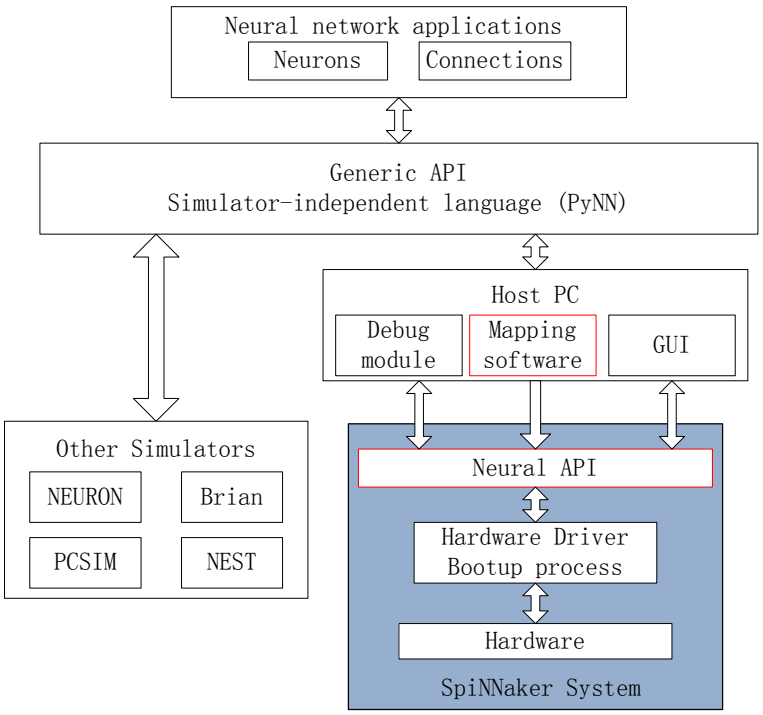


Figure 5.28: The software architecture for SpiNNaker

bandwidth between the Host PC and the SpiNNaker connection and the huge data flow. Users are still allowed to view the states of certain number of neurons in a certain time span however. One possible way to do this is for a user to send an observation request from the Host PC, which will be passed to the destination chips via the Host Chip (0,0). When the request is received by the Monitor Processor on the destination chip, it gathers the required information from local Fascicle Processors, and transmits the information to the Host PC, again via the Host Chip.

5.6.4 Software architecture for SpiNNaker

A good system requires not only high performance hardware but also an easy-to-use software system. Target users of the SpiNNaker system may not have sufficient time or skill to program such a complicated system. It will be more efficient to provide a system which is ready to use, without requiring the neural model to be changed.

Based on previous study, a software model is proposed as shown in Figure 5.28. the neural network model lies at the top-level where it is parsed by a generic

API, PyNN for instance, which is a simulator-independent language for the neural network model. The generic API supports a variety of neural simulators such as NEURON, Brain, PCSIM, NEST, and SpiNNaker. This allows a user to write code for a model once, and run it on different models.

The Host PC provides mid-layer support to the generic API obtaining output information from the generic API, and converting the information to the format which can be loaded by the SpiNNaker system. The Host PC is also responsible for providing debugging facilities and GUI support to help users monitor system behavior. The SpiNNaker system lies at the bottom layer, shown in the blue square. From the top down, the SpiNNaker system comprises a neural API layer, a hardware driver layer, a bootup process, and hardware. The neural API layer includes libraries used to maintain the neural network model on SpiNNaker, such as updating neuron states, processing spike events, modeling synaptic plasticities and so on. The neural API layer requires low level support from hardware drivers. The bootup process is responsible for booting the system, checking the status of components, downloading application codes and so on.

By using this or similar software architecture, the hardware details of the SpiNNaker system can be hidden in a black box for end users, which makes it easier for users to run their code on a SpiNNaker system and to observe the results. Of course, there is much work required to fulfill this objective. The work presented so far mainly concerns the mapping software development, and the neural API implementation. Aspects of the hardware drivers and the bootup process development were also involved. The work made up a minimum system for testing and verifying the SpiNNaker system.

Chapter 6

Synaptic plasticity on SpiNNaker

6.1 Overview

Synaptic plasticity is one of the most important features of a neural network, and many different plasticity mechanisms have been developed since the last century to mimic the biological process of learning. Hebbian theory is one mechanism that has been widely accepted and spike-timing-dependent plasticity (STDP) based on Hebbian theory has received much attention in recent years.

In this chapter, the approach to developing STDP rule on SpiNNaker will be demonstrated. The STDP rule modifies synaptic weights according to the difference between pre- and post-synaptic spike timing. Normally, STDP is triggered whenever a pre-synaptic spike arrives, or a post-synaptic neuron fires. The difficulty in implementing this scheme on SpiNNaker's event-driven model is that when a post-synaptic neuron fires, relevant synaptic weights are still located in the external memory. A Synaptic Block will show up in the local memory only when a pre-synaptic spike arrives. This problem is solved by applying a novel pre-synaptic sensitive scheme with an associated deferred event-driven model. The pre-sensitive scheme only triggers the STDP when a pre-synaptic spike arrives, and hence reduces the processing and the memory bandwidth requirements. The methods shown in this chapter validate the practicality of universal on-chip learning, and illustrate ways to translate theoretical learning rules into actual implementations. The work has been published in [JRG+09] and [JRG+10].

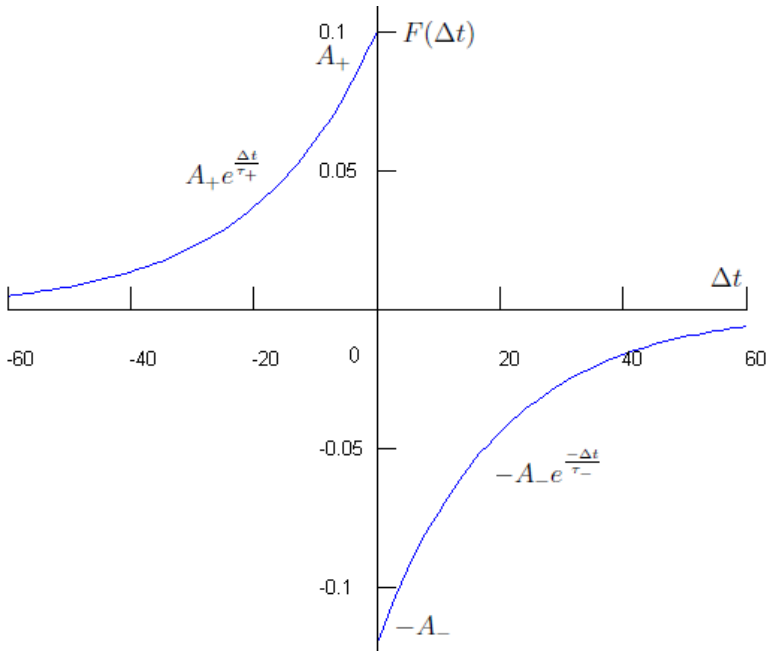


Figure 6.1: The STDP modification function. Parameter used: $\tau_+ = \tau_- = 20ms$, $A_+ = 0.1$, and $A_- = 0.12$.

6.2 Spike-timing-dependent plasticity

The learning rule implemented on SpiNNaker is the well-known spike-timing-dependent plasticity (STDP) [SMA00, ZTH+98, BP98, Izh06, SA01], where the amount of weight modification is decided by the function shown below:

$$F(\Delta t) = \begin{cases} A_+ e^{\frac{\Delta t}{\tau_+}} & \Delta t < 0, \\ -A_- e^{\frac{-\Delta t}{\tau_-}} & \Delta t \geq 0. \end{cases} \tag{6.1}$$

Where Δt is the time difference between the pre- and post-synaptic spike timing ($\Delta t = T_{pre} - T_{post}$, being T_{pre} the pre-synaptic spike time stamp and T_{post} the post-synaptic spike time stamp), A_+ and A_- are the maximum synaptic modifications, and τ_+ and τ_- are the time windows determining the range of spike interval over which the STDP occurs. If the pre-synaptic spike arrives before the post-synaptic neuron fires (i.e. $\Delta t < 0$), it causes long-term potentiation (LTP) and the synaptic weight is strengthened according to $A_+ e^{\frac{\Delta t}{\tau_+}}$. If the pre-synaptic spike arrives after the post-synaptic neuron fires (i.e. $\Delta t \geq 0$), it causes long-term depression (LTD) and the synaptic weight is weakened according to $A_- e^{\frac{-\Delta t}{\tau_-}}$. The corresponding function curve is shown in Figure 6.1 with parameters $\tau_+ = \tau_- =$

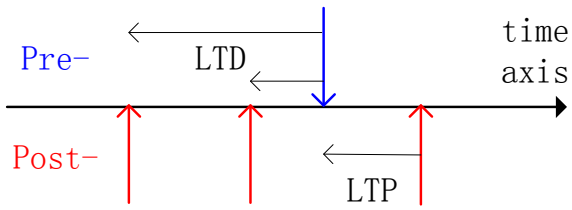


Figure 6.2: The pre-post-sensitive scheme. STDP is triggered when either a pre-synaptic or a post-synaptic neurons fires.

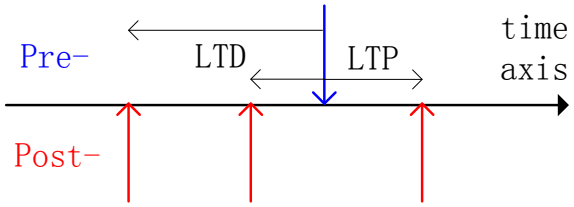


Figure 6.3: The pre-sensitive scheme. STDP is triggered only when a pre-synaptic neurons fires (a spike arrives).

$20ms$, $A_+ = 0.1$, and $A_- = 0.12$. More about the choice of STDP parameters can be found elsewhere [SMA00]. The value of A_- is usually larger than the value of A_+, so that the depression is stronger than the potentiation to ensure that the weights of uncorrelated pre- and post-synaptic connections go slowly to zero, while the weights of strongly correlated connections are strengthened.

6.3 Methodology

6.3.1 The pre-post-sensitive scheme

In most desktop computer simulations, the implementation of STDP is quite straightforward. Because all synaptic weights are locally accessible, the STDP can be triggered whenever a spike is received or a neuron fires. In this approach, calculating Δt is simply a matter of comparing the history records of spike timings. This corresponds to examining the past spike history (at least within the STDP sensitivity window), as shown in Figure 6.2 where pre-synaptic spikes are shown in blue and post-synaptic spikes are shown in red. We call the STDP triggered by the receiving spike as a *pre-synaptic STDP*, since it is caused by a pre-synaptic spike; and the STDP triggered by a neuron firing hence as a *post-synaptic STDP*, since it is caused by a post-synaptic spike. The pre-synaptic

STDP causes LTD which depresses the connection, whereas the post-synaptic STDP causes LTP which potentiates the connection. The scheme used by most desktop computer simulations is termed a *pre-post-sensitive scheme*, since both pre-synaptic and post-synaptic STDPs are involved.

Problems on SpiNNaker

As described previously, SpiNNaker uses a distributed memory system where each chip is associated with one SDRAM shared by 20 processors, and each processor has a fast internal DTCM. The Event Address Mapping (EAM) scheme stores synaptic weights in the SDRAM, and are transmitted to the DTCM by a DMA operation only when a spike arrives. This memory organization guarantees a good balance between the memory space and accessing speed.

Two problems arises however if STDP is implemented using the conventional pre-post-sensitive scheme:

1. The required synaptic weights are NOT in the DTCM when a local neuron fires which disables post-synaptic STDP. It is inefficient to use a second DMA operation to move synaptic weights from the SDRAM to the DTCM when a neuron fires, as it will double the memory bandwidth requirement.

2. Since the synapse block is a neuron-associative memory array, it can only be indexed either by the pre- or post-synaptic neuron. If synapses are stored in pre-synaptic order, the pre-synaptic STDP will be very efficient while the post-synaptic STDP will be inefficient, and vice versa - because one or the other lookup would require a scattered traverse of discontiguous areas of the synaptic block.

As a result, an alternative scheme is required for STDP implementation on SpiNNaker.

6.3.2 The pre-sensitive scheme

We propose a new scheme for STDP implementation on SpiNNaker – a the *pre-sensitive scheme* as shown in Figure 6.3. The pre-sensitive scheme triggers both pre-synaptic STDP (LTD, left headed arrow) and post-synaptic STDP (LTP, right headed arrow), when a pre-synaptic spike arrives. This ensures the synaptic weights are always in the internal DTCM when STDP is triggered, and makes

accessing individual synapses possible by efficient iteration through the array elements when the synapse block is in pre-synaptic order.

The difficulties

The implementation of the pre-sensitive scheme is not, however, as easy as the pre-post-sensitive scheme; two difficulties are involved:

1. This scheme requires the examination of not only the past spike history records, but also of future records. Naturally, future spike timing information is not available at the time the pre-synaptic spike arrives since it has not yet happened.

2. SpiNNaker supports a range of synaptic delays from 0 ms to 15 ms for each connection [JFW08] to compensate for the time differences between electronic and neural timings. The spike arrives at the electronic time rather than the neural time, while the effect of the input is deferred until its neural timing due to the delay. The STDP should be started at the neural time.

6.3.3 The deferred event-driven model

Both of the above difficulties can be overcome by deferring the STDP operation by introducing another model termed *deferred event-driven* (DED) model. In the DED model, no STDP is triggered immediately on receiving a pre-synaptic spike. Instead, the spike timing is recorded as a time stamp and STDP is triggered after waiting a certain amount of time (the current time plus the maximum delay and the time window). The DED model ensures that information on future spike timings is obtained.

Timing records

STDP requires information on both pre-synaptic and post-synaptic spike timings. A pre-synaptic time stamp at 2 ms resolution [1] is kept in the SDRAM along with each synapse block as shown in Figure 6.4 (the global ID of the pre-synaptic neuron is added in front of the time stamp for debugging purposes), and is updated when pre-synaptic spikes arrive. The time stamp comprises two parts, a coarse

[1] The resolution is a tradeoff between the precision and performance. 2 ms is less accurate than 1 ms, but it is much more efficient.

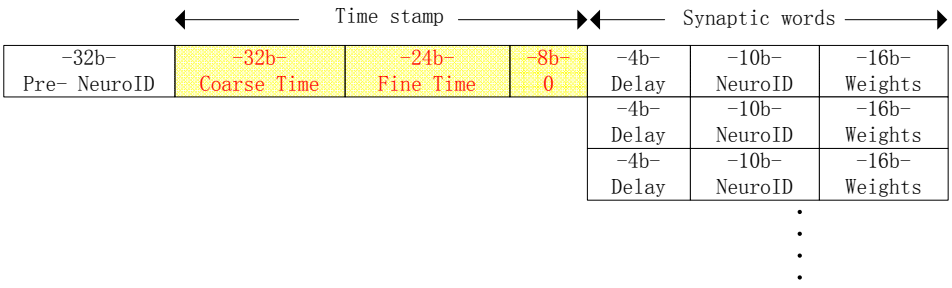


Figure 6.4: The pre-synaptic time stamp.

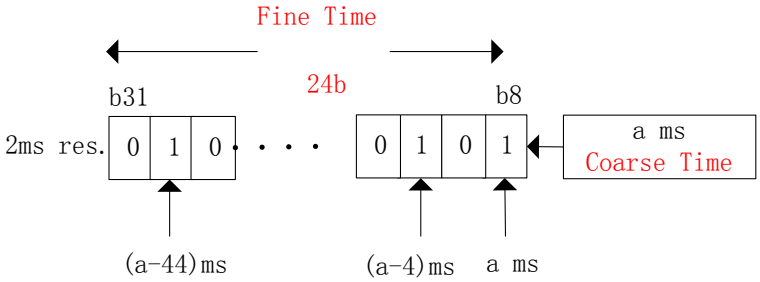


Figure 6.5: The time stamp representation.

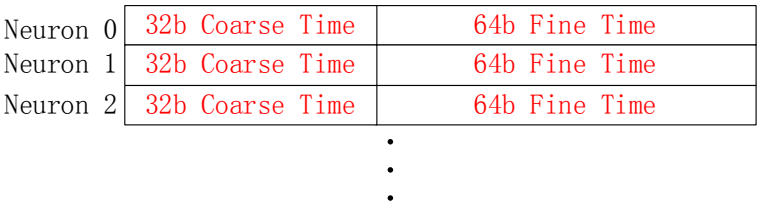


Figure 6.6: The post-synaptic time stamp.

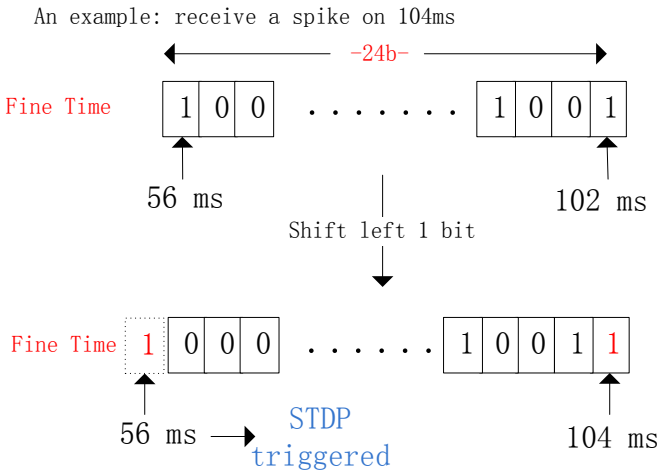


Figure 6.7: Updating the pre-synaptic time stamp.

and a fine time. The coarse time is a 32-bit digital value representing the last time the neuron fired. The fine time is a bit-mapped field of 24 bits (bit [31:8]) representing spike history in the last 48 ms. The coarse time always points to the least significant bit of the fine time (bit 8). As a result, the least significant bit (bit 8) of the fine time is always set.

Figure 6.5 shows how time history is represented by the time stamps. Assuming the coarse time is a ms, bit 8 in the fine time represents the last spike arriving at a ms. Each higher bit represents a spike arrival time which is 2 ms earlier. In Figure 6.5 for instance, it is able to calculate that the pre-synaptic spikes arrive at a, $(a-4)$ and $(a-44)$ ms respectively.

Post-synaptic time stamps reside in local DTCM (Figure 6.6) and have a similar format to pre-synaptic time stamps except that they are 64 bits long (bit [63:0], representing 128ms), allowing longer history records.

Updating timing records

A pre-synaptic time stamp is updated when a packet is received. During the update, firstly, the coarse time is subtracted from the new time to produce a time difference t_{dif}, as shown in Figure 6.7. The time difference is divided by the time resolution, to get the number of bits to be shifted (2ms in this case, so the shift is by $t_{dif}/2$ bits). Then the fine bit is shifted left by $t_{dif}/2$ bits. If any "1" is shifted out of the most significant bit, STDP will be triggered. Bit 32 represents the pre-synaptic spike time which triggers STDP.

The updating of the post-synaptic time stamp is similar to that for the pre-synaptic, except:

1. The post-synaptic time stamp is updated when a neuron fires.

2. The update of the post-synaptic time stamp will NOT trigger STDP.

6.3.4 The STDP process

If STDP is triggered by a "1" going to bit 32 in the pre-synaptic fine time, its post-synaptic connections in the Synaptic Block are firstly traversed word by word. For each Synaptic Word (one connection), the pre-synaptic spike time (the time of bit 32) is added to the synaptic delay to convert the electronic timing to the neural timing T; the processor then calculates the LTD $[T-\tau_-, T]$ and the

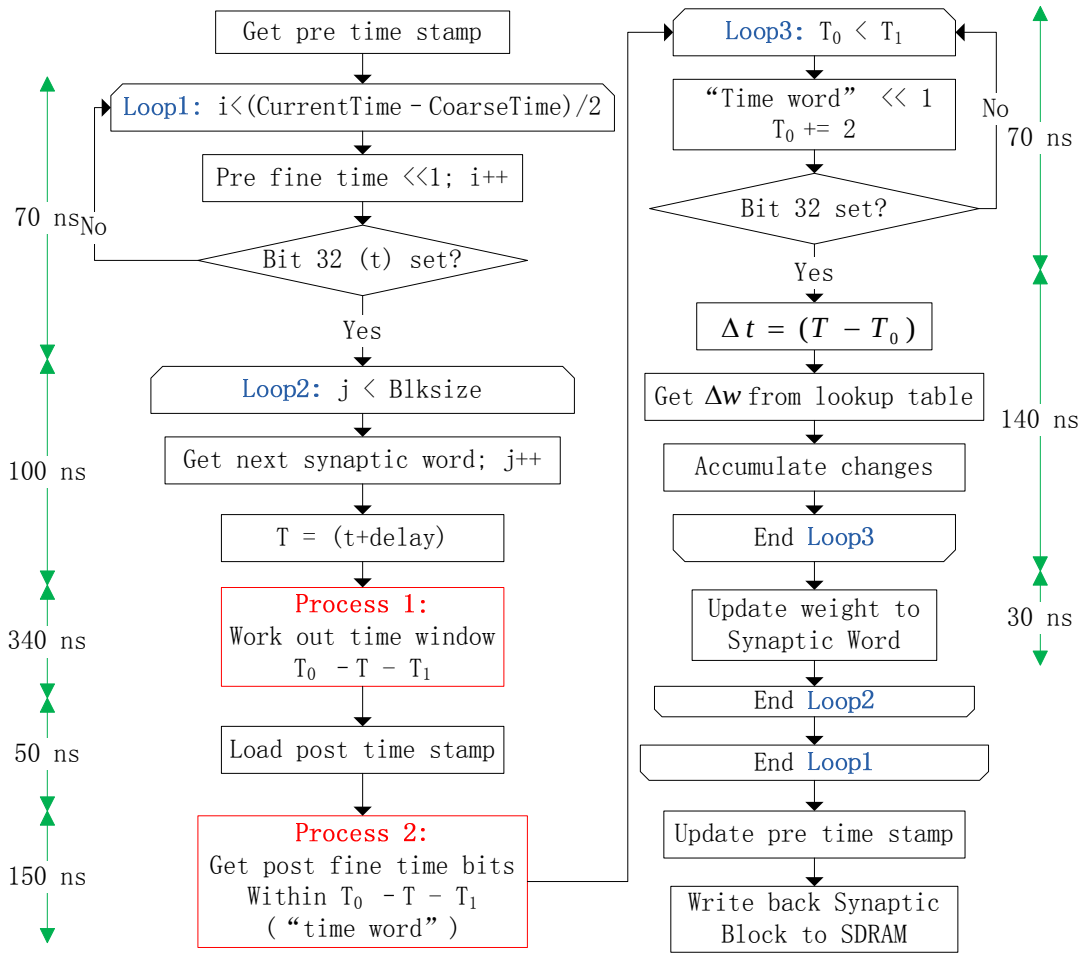


Figure 6.8: STDP implementation flow chart.

LTP $[T, T+\tau_+]$ windows. If any bit in the post-synaptic time stamp is set within the LTD or LTP window, the synaptic weight is either weakened or strengthened according to the STDP rule.

The post-synaptic time stamp can be retrieved from the DTCM as determined by the neuron ID in the Synaptic Word. The processor then scans the post-synaptic time stamp looking for any "1" located within the learning window, and updates the weight accordingly.

6.3.5 Implementation

The flow chart of the STDP implementation is shown in Figure 6.8 which comprises three nested loops in the programme to handle a new spike and to do STDP. Each of the three loops may run through several iterations:

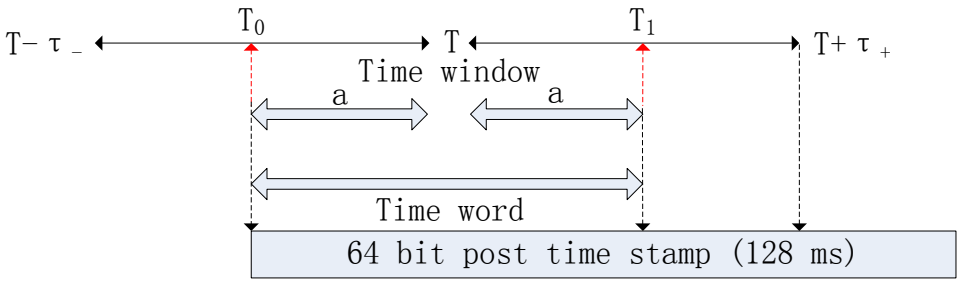


Figure 6.9: Calculate the time window and time word.

- Loop1: Update the pre-synaptic time stamp. This loops $t_{dif}/2$ times. If there are n "1"s shifted to bit 32, the STDP will be triggered n times.
- Loop2: Traverse the post-synaptic connections. This loops $m = Blksize$ times, the number of words in a Synaptic Block, i.e. the number of post-synaptic connections in this fascicle from the pre-synaptic neuron that fired.
- Loop3: Scan the post-synaptic time stamp. This loops $x = (T_1 - T_0)/2$ times (T_0 and T_1 will be explained later), and each time 1 bit will be detected. If there are y bits found within the time window, the weight updating will be executed y times.

The computational complexity of the bit-detection in Loop3 is $o(nmx)$ and the computational complexity of weight updating in Loop3 is $O(nmy)$. As a result the shifting and the weight updating in Loop3 needs to be as efficient as possible.

Process 1 - the time window

Process 1 in flow chart 6.8 is responsible for calculating the time window, in this implementation a dynamic window, from T_0 to T_1, which differs from the window defined in the STDP rule by τ_- and τ_+. Three restrictions are applied when calculating the time window (shown in Figure 6.9):

1. The time window must be in the range of $[\tau_-, \tau_+]$.
2. There are history records in the post-synaptic time stamp in the time window. In Figure 6.9, the time window becomes $[T_0, T + \tau_+]$.
3. The left window and the right window are the same length. In Figure 6.9, the time window becomes $[T_0, T_1]$, as $T - T_0 = T_1 - T = a$.

Process 2 - the time word

The post-synaptic fine time stamp field is 64 bits and a 64-bit shifting operation in ARM takes 8-9 CPU cycles, while a 32-bit one takes only 1. The meaningful bits of the fine time stamp are those within the time window $[T_0, T_1]$, which is smaller than 32 bits if $\tau_- = \tau_+ \leq 32ms$. These bits are referred to as the "time word", which represents the bits of the post-synaptic fine time stamp within the time window $[T_0, T_1]$, after bit-time conversion. If any of the bits is set in the "time word", the weight needs to be updated accordingly.

Bit detection

There are two bit detection operations in the STDP implementation. The "LSLS" instruction provided by the ARM instruction set is efficient in detecting if there is any "1" moved into the carry bits (bit 32), and allows the processor to do a conditional branch. To use the "LSLS" instruction, bits [31:8] (instead of bit [23:0]) of a word is used for the pre-synaptic fine time stamp.

Lookup table

Since the parameters of the STDP are determined before the simulation starts, the Δw can be pre-computed based on different values of Δt and loaded into a lookup table. When the Δt is obtained, Δw can easily be fetched from the lookup table. Compared to the real-time computation of Δw, using a lookup table is obviously more efficient.

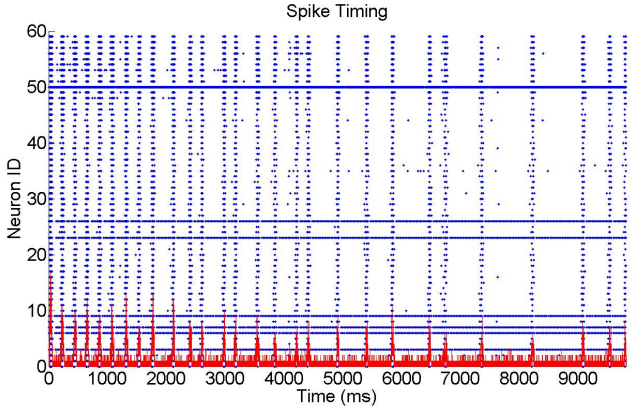
Performance

The processor time usage for each step of processing is shown in Figure 6.8 where process 1 and 2 in Loop 2 are the most time consuming operations. The calculation of Δw in Loop 3 takes only 140 ns, with the help of a lookup table.

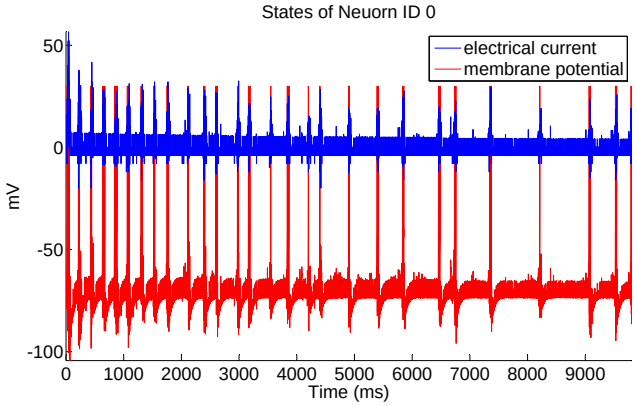
6.4 Simulation results

6.4.1 10-second 60-neuron test

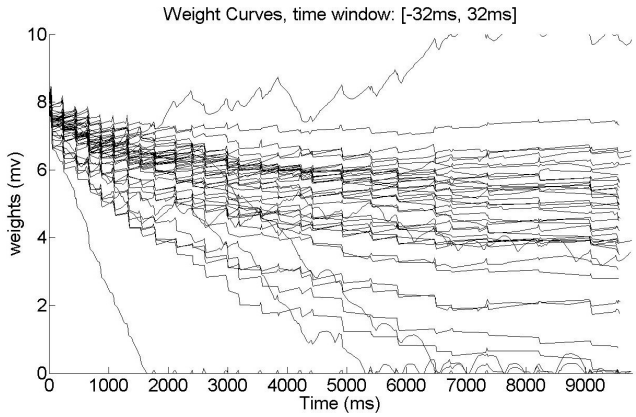
A 60-neuron network is simulated on the four-chip SpiNNaker SOC Designer model. The network is largely based on the code published in [Izh06] (but in a



(a) Spike raster plot.

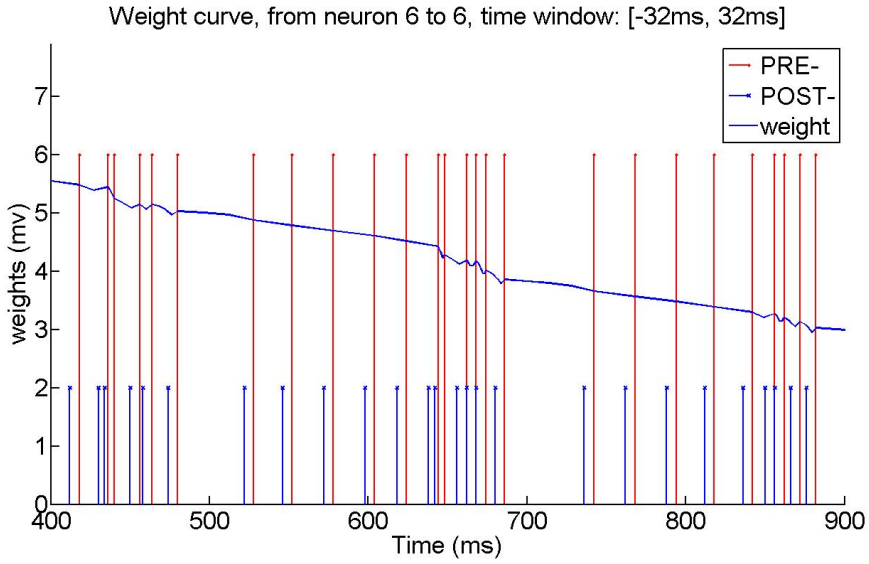


(b) The spike train and input current of neuron 0.

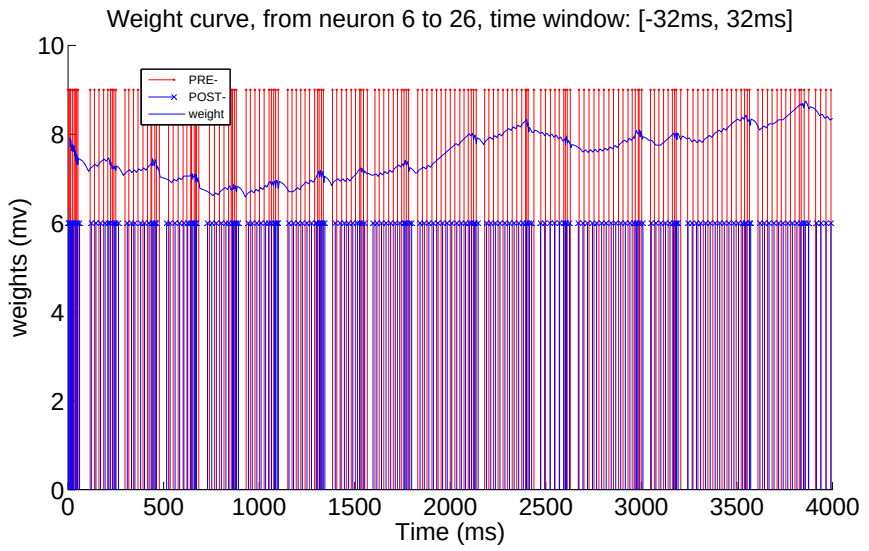


(c) Weight curves of connections from pre-synaptic neuron 6. The synaptic weight going rapidly to 0 is a self-connection.

Figure 6.10: STDP results.



(a) Synapse of neuron 6 to 6 (weaken) within 400 – 900 ms.



(b) Connection from neuron 6 to 26 (strengthen) within 0 – 4000 ms.

Figure 6.11: Weight modification caused by the correlation of the pre and post spike times.

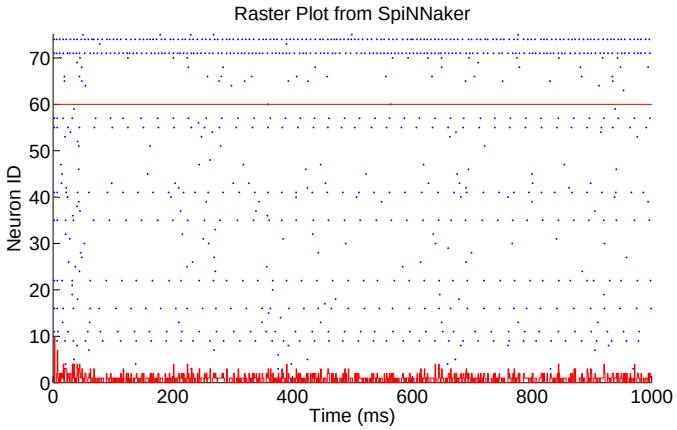
smaller scale), which was also used to test the consistency of our results. There are 48 Regular Spiking excitatory neurons $a = 0.02, b = 0.2, c = -65, d = 8$ and 12 Fast Spiking inhibitory neurons $a = 0.1, b = 0.2, c = -65, d = 2$. Each neuron connects randomly to 40 neurons (self-connections are possible) with random 1-15 ms delays; inhibitory neurons only connect to excitatory neurons. Initial weights are 8 and -4 for excitatory and inhibitory connections respectively. Parameters $\tau_+ = \tau_- = 32ms$, $A_+ = A_- = 0.1$ are used for STDP. Inhibitory connections are not plastic [BP98]. Following learning the weights of excitatory neurons are clipped to [0, 20] (in accordance with [Izh06]). There are 6 excitatory and 1 inhibitory input neurons, receiving constant input current $I = 20$ to maintain a high firing rate. Weights are updated in real-time (every 1 ms).

The simulation is run for 10 sec (biological time) and Figure 6.10(a) shows the spike raster, Figure 6.10(b) shows the spike train of neuron ID 0, and Figure 6.10(c) shows the evolution of synaptic weights of connections from pre-synaptic neuron ID 6 (an input neuron). At the beginning of the simulation input neurons fire synchronously, exciting the network which exhibits high-amplitude synchronized rhythmic activity around 5 to 6 Hz. As synaptic connections evolve according to STDP, uncorrelated synapses are depressed while correlated synapses are potentiated. Since the network is small and the firing rate is low, most synapses will be depressed, leading to a lower firing rate. The synaptic weight going rapidly to zero is the self-connection of neuron ID 6: since each pre-synaptic spike arrives shortly after the post-synaptic spike the synapse is quickly depressed.

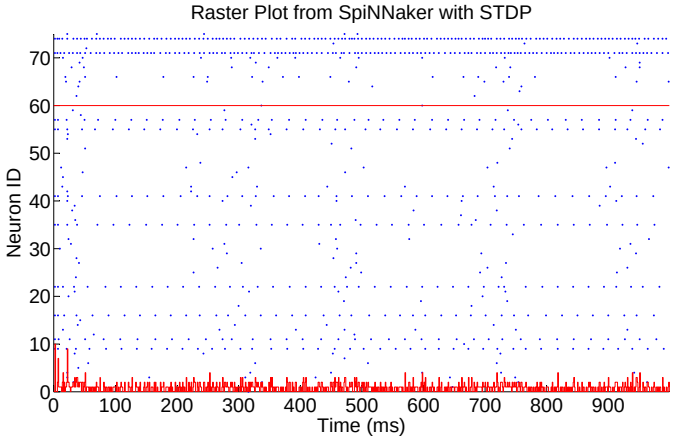
Detailed modifications of the self-connection weight (the blue curve) is shown in Figure 6.11(a) along with pre- (red vertical lines) and post-synaptic timing (blue vertical lines), from 400 ms to 900 ms. Modification is triggered by pre-synaptic spikes. The weight curve between two pre-synaptic spikes is firstly depressed because of the LTD window and then potentiated because of the LTP window. The detailed modification of the strengthened synapse (from neuron 6 to 26) from 0 ms to 4000 ms is shown in Figure 6.11(b).

6.4.2 30-second 76-neuron test

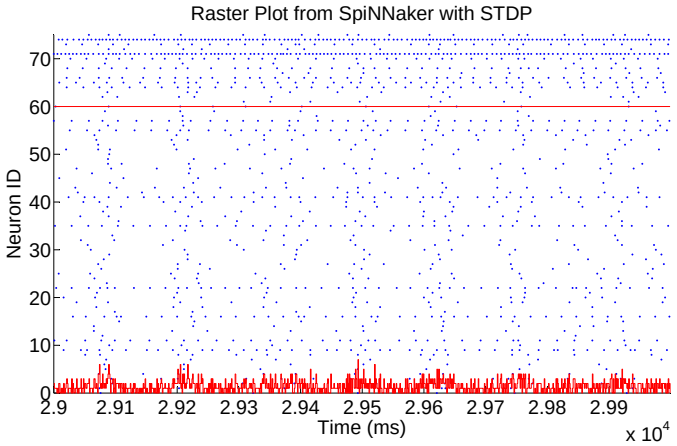
The system is also tested by a 30-second simulation of a 76-neuron network (60 excitatory and 16 inhibitory neurons) with each excitatory neuron randomly connects to 10 other neurons and each inhibitory neurons randomly connects to 10 excitatory neurons. A random 10 neurons receive a constant biased input of 20



(a) Simulation with STDP disabled during the 1st second.



(b) Simulation with STDP enabled during the 1st second.



(c) Simulation with STDP enabled during the 30th second.

Figure 6.12: Comparison between the simulation with and without STDP on SpiNNaker during 1st second and 30th second.

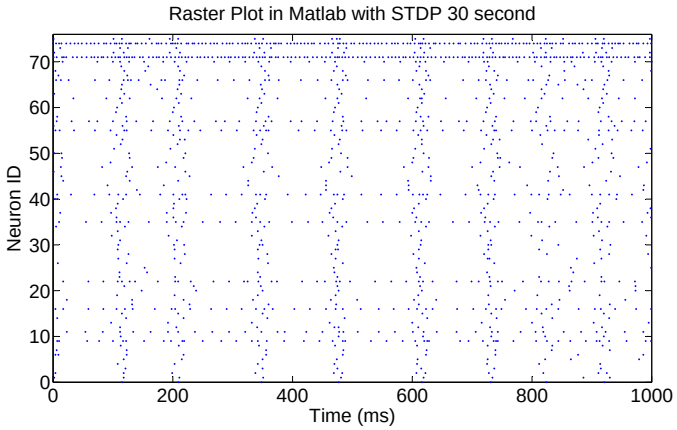
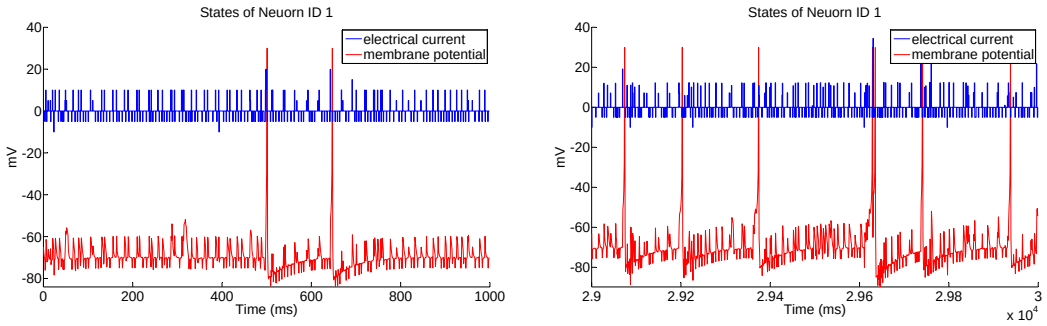


Figure 6.13: The STDP result from the Matlab simulation (fixed-point) with parameters: $\tau_+ = \tau_- = 15ms$, $A_+ = 0.1$, and $A_- = 0.12$.



(a) Neuron 1 during the 1st second.

(b) Neuron 1 during the 30th second.

Figure 6.14: The behavior of an individual neuron (ID 1) during the 1st second and the 30th second.

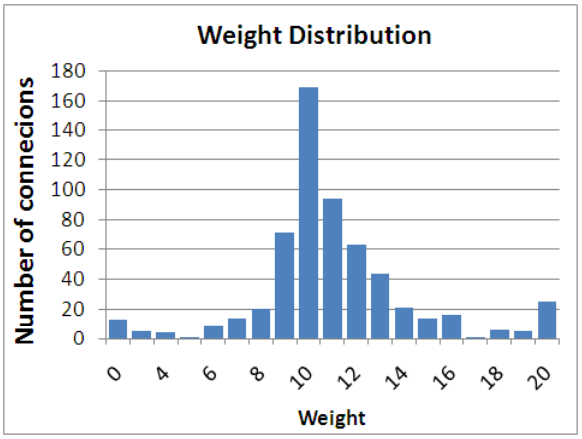


Figure 6.15: Weight distribution at the end of the simulation. Weights for excitatory neurons are clipped to [0, 20]

mV. A comparison between a simulation without STDP (Figure 6.12(a)) and with STDP (Figure 6.12(b)) after the first 1,000 ms, is shown. The simulation with STDP shows a more synchronized firing pattern than the simulation without.

The result from the 30th second is shown in Figure 6.12(c), and the corresponding Matlab (fixed-point) results[2] is shown in Figure 6.13. The firing pattern during the 30th second shows a more obviously synchronized behavior. Figure 6.14(a) shows the behavior of neuron ID 1 during the 1st second. Figure 6.14(b) shows the behavior of neuron ID 1 during the 30th second.

The weight distribution at the end of the simulation can be observed in Figure 6.15: most of the excitatory weights are slightly de/potentiated around their initial value of 10. Some connections are potentiated up to their maximum (20) because they are systematically reinforced, due to converging delays. For example we found a group of 7 neurons strongly interconnected at the end of the simulation. Examining their connections and delays (as shown in Figure 6.16 two circuits with converging delays could be found: the first one starts from neuron 57 and propagates through neuron 8, 47 and 43, ending at neuron 19; the second one starts from neuron 55 exciting neuron 57, propagates through neurons 47 and 19 and ends at neuron 42. All the weights are strongly potentiated, near or up to their maximum at the end.

[2]The Matlab simulation is based on the code provided in [Izh06]. The STDP parameter setting is slight different: $\tau_+ = \tau_- = 15ms$, $A_+ = 0.1$, and $A_- = 0.12$; weights are updated every 1 second instead of 1 millisecond.

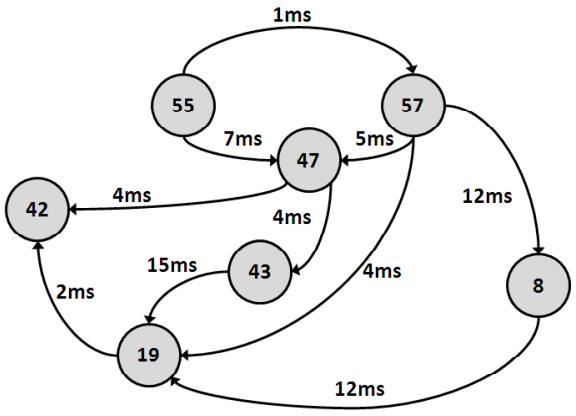


Figure 6.16: Group of neurons with converging delays. It is possible to track down two circuits with converging delay (neurons 55, 57, 47, 19 ending at neuron 42 and neuron 57, 47, 43 and 8 ending at neuron 19.

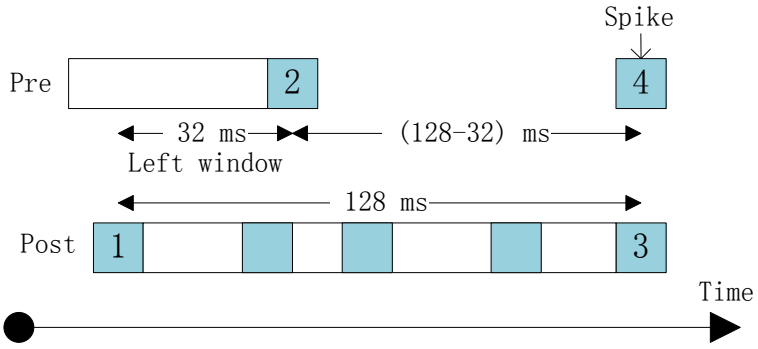


Figure 6.17: Relationship between the firing rate and the timing records length.

6.5 Discussion

6.5.1 Firing rates and the length of timing records

The length of the time stamp affects both the performance and precision of the STDP rule. Longer history records yield better precision at the cost of significantly increased computation time. The optimal history length is therefore dependent upon the required precision and performance. A 24-bit pre-synaptic time stamp with 2 ms resolution and a maximum of 15 ms delay guarantees a $24 * 2 - 15 > 32ms$ right (LTP) window for any delay.

The pre-sensitive scheme and the deferred event-driven model require the new input to push the old input record into the carry bit to trigger STDP. What happens, however, if the new input does not arrive or it arrives at a very low rate?

The post-synaptic time stamp is pushed forward when new post-synaptic spikes are generated, and the history records will be lost. If the pre-synaptic firing rate is too low, there will be no post-synaptic time records within the time window at the time STDP is triggered. As a result, there are certain restriction in terms of the firing rate of pre-synaptic neurons to ensure that STDP will be triggered in time. As shown in Figure 6.17, at the time a new pre-synaptic spike (4) arrives, the time difference between pre-synaptic spike 2 and post-synaptic spike 1 is 32 ms – the same size of left window (LTD) as the size of right window. Let the average interval of two pre-synaptic spikes to be T_{pre}, and the average interval of two post-synaptic spikes to be T_{post}.

1. When $T_{post} \leq T_{pre} + 32$ (post-synaptic neurons fire more frequently), to guarantee a 32 ms left window, the interval between two pre-synaptic spikes must be no more than $128 - 32 = 96ms$; this, in turn, requires a firing rate of more than $1000/96 = 10.4Hz$.

2. When $T_{post} > T_{pre} + 32$ (post-synaptic neurons fire less frequently), a 32 ms left window can be guaranteed with any pre-synaptic firing rate.

6.5.2 Approximation and optimization

Since the Matlab and SpiNNaker simulations employ different implementation schemes, exactly the same results are not achievable. Actually, the level of biophysical realism necessary to achieve useful behavior or model actual brain dynamics is unclear. For instance, in the case of the well-known STDP plasticity rule, while many models exist describing the behavior [GKvHW96, SBC02, HTT06], the actual biological data regarding STDP is noisy and of low accuracy. Observed STDP synaptic modifications exhibit a broad distribution for which the nominal functional form of STDP models usually constitute an envelope or upper bound to the maximum modification [MT96, BP98]. This suggests that high repeatability or precision in STDP models is not particularly important. In this context, the STDP implementation on SpiNNaker focuses on efficiency instead of accuracy.

The SpiNNaker STDP implementation can be further simplified by using "nearest spike approximation" [MGT08] which limits LTD/LTP to the first/last presynaptic spike before/after the postsynaptic one. The implementation of the STDP rule involves a series of processing steps. Most of the processing steps are

in nested loops and will be executed for a number of iterations during the STDP process. Thus the performance will decrease significantly with STDP enabled, a common problem of using STDP. The use of the "nearest spike approximation" potentially reduces the number of iterations, and will therefore significantly reduce the overhead.

The length and resolution of the time stamp are reconfigurable to meet different requirements; if a larger time window is required, the length of the presynaptic time stamp can be increased or the resolution can be reduced to 4 ms per bit. The dynamic adjustment of the time stamp length is also a possible optimization, by which users will be able to modify the length of the time stamp or the time resolution at run-time, to meet the accuracy-performance requirement during different simulation periods.

Chapter 7

MLP modeling on SpiNNaker

7.1 Overview

Previous chapters have presented approaches to the modeling of spiking neural networks with STDP learning on SpiNNaker. Although the spiking neural model has received much attention in recent years, the well-known traditional multi-layer perceptron (MLP) model is still widely used in various applications. Having been developed for tens of years, the MLP model has been proved to be a good model for solving practical problems, especially in speech or pattern recognition. This chapter investigates, by analysis and simulation, whether SpiNNaker can and how it should operate when dealing with MLP networks. The work has been published in [JLK+10a] and [JLK+10b].

7.2 Introduction

Rather than generating spikes when the membrane potential reaches a certain value, as neurons do in spiking neural networks, neurons of MLP networks use nonlinear activation functions to produce continuous output values at each propagation cycle. The output of an MLP unit is often interpreted as the mean spiking rate of a group of biological neurons. MLP networks are normally trained by the back-propagation (BP) learning rule developed by David E. Rumelhart et al. in 1986 [RHW86, RMG86].

Compared to their sequential alternatives, parallel implementations of MLP networks can partition and distribute the computational tasks but usually incur severe communication overheads, leading to low actual speedups. Speedups

gained from multiple processors rely on a well tuned parallel scheme. A good partitioning scheme needs to maximise the distribution of the computation tasks while keeping communication overheads at low levels. There are a lot of existing schemes to parallelize a MLP neural network with BP learning rule such as networking partitioning, pattern partitioning, hybrid partitioning and so on [9]. These partitioning schemes have been mapped onto a variety of parallel hardware with different topologies such as the hypercube machine, mesh connected multiprocessors, transputers, networks of workstations as well as dedicated neural hardware [AB03], [SOM05], [FSS97].

SpiNNaker, as dedicated parallel neural hardware, has its own special hybrid topology – SpiNNaker CMPs are interconnected through a two dimensional torus mesh with diagonal connections. To achieve a high efficiency (the ratio of speedup to number of processors), we present a new mapping scheme for the parallel implementing of MLP networks with the BP learning rule on SpiNNaker. The new mapping scheme relies on a checker-board partitioning (CBP) scheme [KSA94], but the key advantage comes from introducing a pipelined mode.

The CBP scheme cuts the whole weight matrix into small sub-matrices enabling communication to be localized and thus reducing the number of communication packets. In the context of neural networks, the CBP scheme was used in [KSA94], [YI93] and others, but none addressed the parallelism *within* each individual submatrix. Based on the CBP scheme, we have developed a pipelined checker-boarding partitioning (PCBP) scheme. In the PCBP scheme, in addition to the traditional group of cores which perform vector-matrix computation, an extra two groups of cores are employed to compute partial sums and outputs. The three groups of cores are able to work in parallel as a six-stage pipeline, allowing overlap of computation and communication. Previous work has considered pipelined implementations, but was either based on pipelining the work in each neural network layer [PDG93] or on pipelining between patterns [Pet94]. These can not be applied to recurrent neural networks (RNNs), since all layers in RNNs are updated concurrently. Our algorithm, on the other hand, overcomes this barrier by considering pipelining within each partition.

Although the pipelined checker-boarding partitioning (PCBP) scheme proposed works for both feedforward and recurrent networks, we focused mostly on recurrent networks in this study, based on three reasons:

1. More and more neural models are built based on RNNs to simulate more

complex situation.

2. RNNs are much more computationally demanding to train than FFNNs.

3. Less research has been undertaken on the parallel implementation of RNNs than of FFNNs.

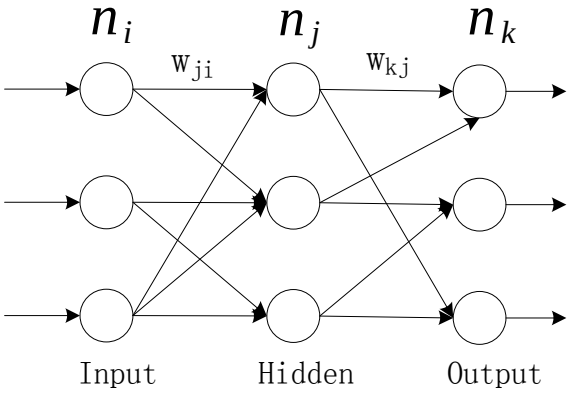
An analysis of parallel execution of the algorithm on the SpiNNaker architecture is presented. The performance evaluation of such a mapping of both fully-connected and partially-connected networks is also an important objective. The evaluation is carried out in a semi-experimental and semi-analytical way. Simulation results from the SOC Designer model of SpiNNaker based on the PCBP scheme are shown at the end of this chapter and are compared with results based on the CBP scheme. The performance curves produced show that with the PCBP scheme, a better speedup can be achieved than with the traditional non-pipelined CBP scheme. The work proves that SpiNNaker can also deal with traditional MLP networks very efficiently. The mapping scheme and results of the performance estimation were used in the proposal for the PDP-squared project (September 2008 - August 2013) to the EPSRC Cognitive Systems call, which was granted in February 2008.

7.3 MLP model with BP algorithm

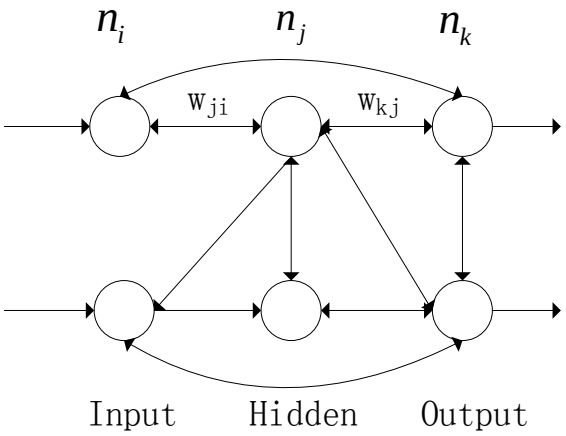
7.3.1 Feedforward and recurrent neural networks

A feedforward network (FFNN), as a standard form of multi-layer perceptron, consists of a number of simple neurons or units, organized in layers (Figure 7.1(a)). Neurons in one layer are connected only to neurons in the next layer. There is no self-connection or connection within the same layer. In such networks, groups are updated in the order in which they appear in the network's group array (sequential updates) and only one layer needs be computed at a time.

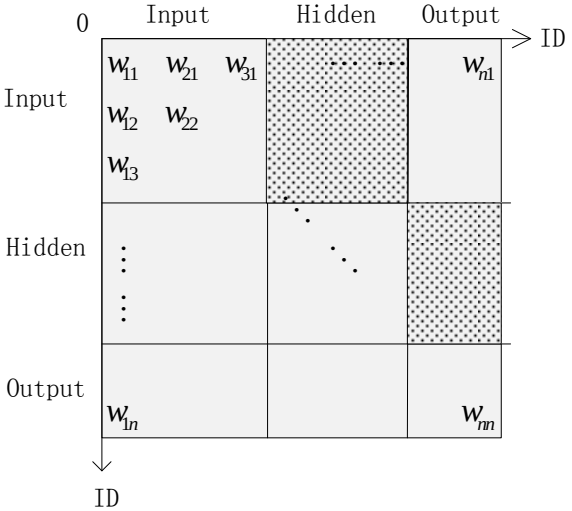
Recurrent Neural Networks (RNNs), as shown in Figure 7.1(b), not only have feedforward connections but also have feedback connections. A neuron in an RNN may connect to any other neurons including itself. Furthermore, RNNs use concurrent updates and propagate error derivatives backwards through time. That is, in each time tick, all layers first update their inputs and then update their outputs. Backpropagation will not start until the forward phase has run for



(a) A random feedforward network.



(b) A random recurrent network.



(c) The weight matrix.

Figure 7.1: MLP models and the weight matrix.

a given number of time ticks. Backpropagation then loops for the same number of ticks. In this case, all layers need to be computed at each time. The way neurons are connected and the way they are updated make RNNs more computationally intensive than FFNNs.

Suppose there is an RNN with P_{pt} patterns per batch, I_{tv} time intervals per pattern and T_{tk} ticks per time interval. A training iteration for this RNN includes P_{pt} BP iterations. In each BP iteration, the system carries out $I_{tv}T_{tk}$ forward propagation ticks, and then $I_{tv}T_{tk}$ backward propagation ticks. Each backward propagation tick computes a weight change value which will be accumulated, and updated later as required (usually every batch of training).

The weights can be seen as a matrix, as shown in Figure 7.1(c). Depending on the connectivity, the weight matrix may be dense or sparse. In FFNNs, elements are spread only in the areas indicated by the dot-filled blocks in Figure 7.1(c), while in RNNs, weights are spread over the whole matrix. In this study, the partitioning scheme will be discussed based on the weight matrix.

7.3.2 BP algorithm

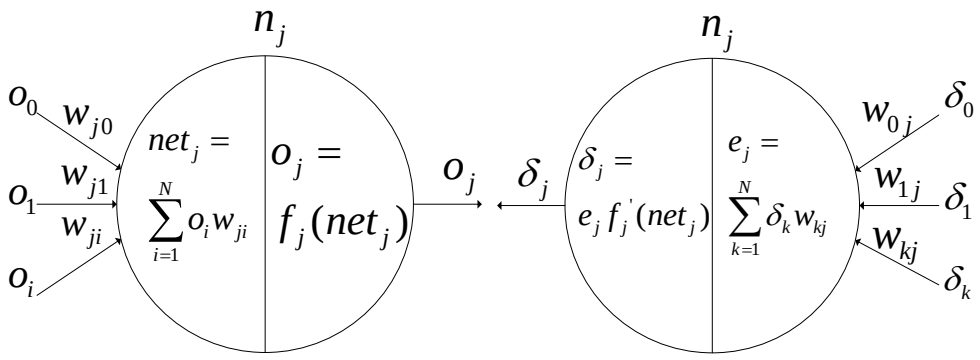
The BP algorithm was introduced in 1986 [RHW86][RHM86], and is the most commonly used supervised learning algorithm for training MLP networks. At the core of the BP algorithm is a delta rule which implements a gradient descent method in the error space looking for the optimal weight set which minimizes the error. The BP algorithm requires both forward and backward phases. During the forward phase, the network gets inputs and produces outputs. During the backward phase, the delta error δ between the actual output vector and the target output vector (provided by the training pattern) is computed and δ is propagated back through the network and weight changes are generated accordingly.

To give a better description of the BP rule, neurons in the input, hidden and output layers are denoted as n_i, n_j, n_k respectively; w_{ji} and w_{kj} are the weights of connections from neuron n_i to n_j and n_j to n_k respectively.

Forward phase

During the forward phase, information propagates from the input to the output layer (Figure 7.2(a)).

In each step of the propagation, neuron n_j receives an output vector $o_0, \ldots, o_i$



(a) Forward phase. (b) Backward phase.

Figure 7.2: Computational phases of a MLP network with BP learning.

from the previous layer and produces a net input net_j according to

$$net_j = \sum_{i=1}^{N} o_i w_{ji}. \tag{7.1}$$

The generated net_j is then passed to a continuous non-linear activation function f_j to generate the output o_j according to

$$o_j = f_j(net_j). \tag{7.2}$$

The output o_j will be passed to neurons in the next layer until the propagation goes through the output layer.

Backward phase

In the backward phase, firstly, the output o_k produced in the output layer is compared with the target t_k to generate an error e_k which is $e_k = t_k - o_k$. This error e_k is then used to produce the delta error δ_k given by

$$\delta_k = e_k f'(net_k) = (t_k - o_k) f'(net_k). \tag{7.3}$$

The delta error δ_k can be used to produced weight updates of w_{kj} denoted by Δw_{kj} according to $\Delta w_{kj} = \eta \delta_k o_j$, where η is the learning rate. Then, δ_k is propagated back to the previous layer and is used to compute the error e_j according to

$$e_j = \sum_{k=1}^{N} \delta_k w_{kj}, \tag{7.4}$$

as indicated in Figure 7.2(b). The delta error of this layer δ_j is given by

$$\delta_j = e_j f'(net_j) = \sum_{k=1}^{N} \delta_k w_{kj} f'(net_j). \tag{7.5}$$

Weight changes Δw_{ji} for connections from n_i to n_j are given by

$$\Delta w_{ji} = \eta \delta_j o_i. \tag{7.6}$$

The BP rule is usually required to be run for a number of iteration to have a network fully trained. Normally, weights are updated once per training batch. A training batch contains a number of patterns. One pattern will apply for a number of intervals, each is comprised of a number of ticks. In a FFNN, one tick comprises a full BP process – one forward phase and one backward phase; while in an RNN, the backward phase starts only after the forward phase runs for the given number of ticks.

The activation function

The activation function converts a net input to an output. Different activation functions can be used, but they should have some common features: continuous, differentiable, nonlinear, and monotonically nondecreasing. For the sake of efficiency, both the activation function and its derivative should be easy to compute. A commonly used activation function is the sigmoid function defined in Equation 7.7:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{7.7}$$

with its derivative:

$$f'(x) = f(x)[1 - f(x)] \tag{7.8}$$

The sigmoid function is illustrated in Figure 7.3:

7.3.3 The neural simulator – Lens

Lens is a light, efficient network simulator written in the C and Tcl languages [Roh], which runs on a variety of platforms including both Unix and Windows. It is developed mainly to support MLP networks with backpropagation learning (both feedforward and recurrent networks), but it also supports a range of other

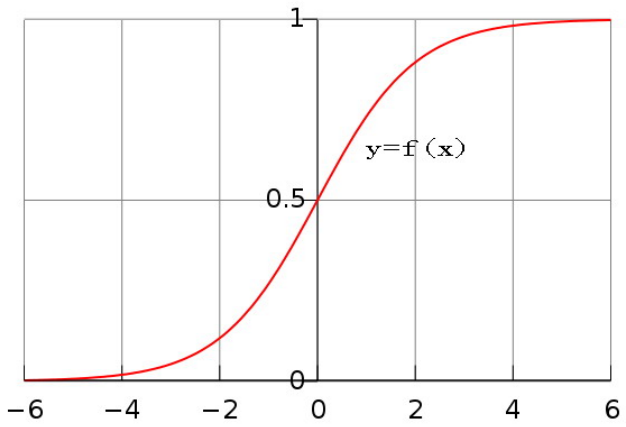


Figure 7.3: The sigmoid function.

neural network models such as deterministic Boltzmann machines and Kohonen networks.

The Lens implementation of backpropagation networks is used as a practical example during our study. At the end of this chapter, a comparison, based on the same network, will be shown, between the performance estimated on SpiNNaker simulation and the performance on a single PC using Lens.

7.4 Partitioning and mapping schemes

7.4.1 Review of partitioning schemes

In a parallel simulation, computation tasks are distributed over a set of processors to achieve better performance. Each processor in the parallel system produces only a partial result and information exchanges among processors are therefore required to produce the final result. This communication overhead usually results in a decrement in the overall performance and parallel schemes have been developed to solve this problem: Firstly, a partitioning scheme may be used to divide a neural network into small sub-networks; a mapping scheme may then be used to map the sub-networks onto different processors.

Depending on the level of parallelism, MLP networks with the BP algorithm can be partitioned either by training patterns, the network, or a hybrid of these two [KSA94, SM98, SS98]. Pattern parallelism is often used by general-purpose neural software, for instance – Lens, running parallel training on PC Clusters. It

runs different sets of training patterns concurrently on different processors. Each processor keeps a local copy of the complete network and accumulates weight changes for the given training patterns [JMM89, WC90, RB88]. Since most of the learning algorithms compute weight changes on a per-pattern basis, pattern parallelism can be applied on top of most network parallelism.

In this study, the focus is on network parallelism which captures the inherent parallelism in the neural networks. The multiplication-accumulation operations between the output vector and the weight matrix in equations 7.1 and 7.4 can be parallelized by partitioning the weight matrix. The weight matrix can be partitioned by element, row, column, or sub-matrix, which corresponding to the complete weight partitioning [Ble87], inset grouping [ZMMW90b, ZMMW90a], outset grouping and checkerboarding partitioning schemes [KH89, KSA94, YI93] respectively. Complete weight partitioning allocates one processor per weight for maximum concurrency but incurs too much communication. Inset grouping is efficient in the forward phase but not efficient in the backward phase. Outset grouping is efficient in the backward phase but not efficient in the forward phase. Some schemes allocate both inset weights and outset weights to the same processor by duplicating weights [SM98]. But this causes inefficiency during weight updating, because weights in different processors have to be synchronized. The CBP scheme partitions the weight matrix into sub-matrices and thereby optimizes both forward and backward phases[1].

7.4.2 The CBP scheme and the non-pipelined model

The CBP partitioning scheme is used to split the weight matrix as the starting point of the algorithm. The first step, for simplicity, is to analyze the mapping of RNNs onto a 2D torus topology with one processor per node. Figure 7.4 shows an example of a 6x6 weight matrix mapped onto 9 nodes (or processors) interconnected by a 2D torus. Each processor keeps a 2x2 sub-matrix of weights in its local memory. Processors $1-9$ are assigned to GroupA responsible for the vector-matrix multiplication. Among those 9 processors in GroupA, 3 processors in the main diagonal are selected and named $m1-m3$. Processors $m1-m3$ are assigned to GroupB, responsible for the partial result accumulation and output computation.

[1]It has been proved in [YI93] that the minimum communication steps are obtained only when the matrix is divided into square sub-matrices.

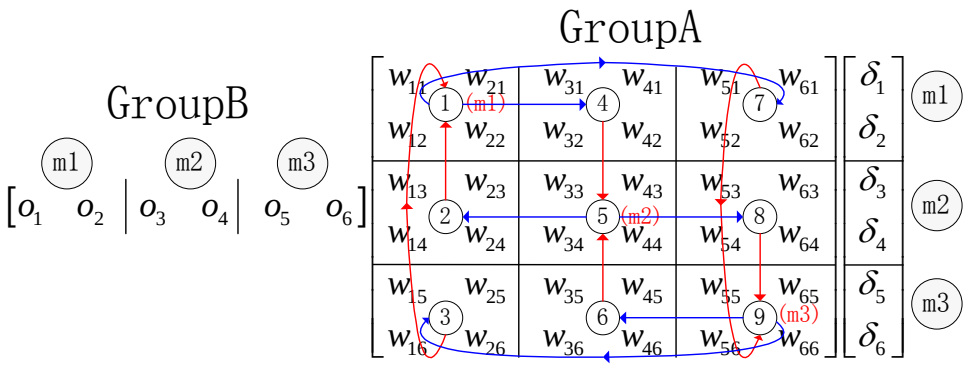


Figure 7.4: Checkerboarding partitioning. Each processor keeps a 2x2 sub-matrix of weights. Processors 1 .. 9 are assigned to GroupA and responsible for the vector-matrix multiplication. Among those 9 processors in GroupA, 3 processors in the main diagonal are selected and named m1..m3 (GroupB), responsible for the partial result accumulation and output computation. In step1 of the forward phase, processor m1 (the same one as processor 1) sends the output to processor 1, 4, 7 (shown as blue arrows). In step2, processors 1, 2, 3 do the computation and send packets to processor m1 (shown as red arrows). Processor m1 accumulates partial results and produces an output. Other processors behave in the same way. The backward phase is the same as the forward phase, but swapping the order of columns and rows.

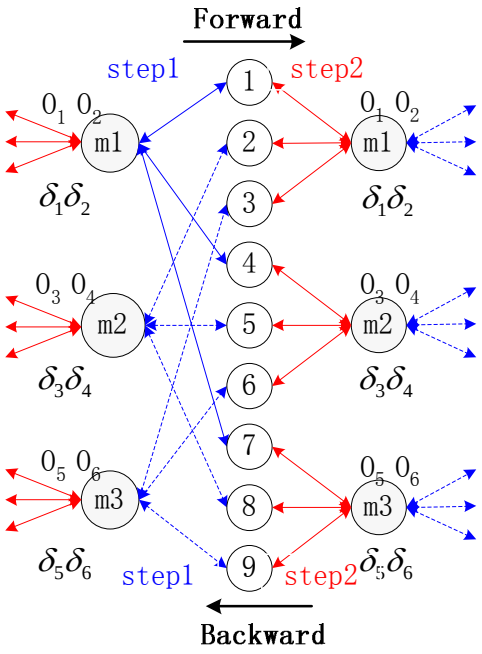


Figure 7.5: Communication patterns generated accordingly from Figure 7.4, showing both the forward phase (right-directed arrows) and the backward phase (left-directed arrows).

Communication pattern analysis

For a more general discussion, an n-neuron network and a torus with p processors (nodes) are now considered, while maintaining the same one processor per node relation. In fully connected networks, the size of the weight matrix is n by n and each processor stores and updates a $\frac{n}{\sqrt{p}}$ by $\frac{n}{\sqrt{p}}$ sub-matrix. Assuming the transmission of a packet along a link takes α time units, while sending or receiving a packet takes β time units, in each forward tick the operations can be divided into two discrete steps following the communication pattern illustrated in Figure 7.5.

In step1, for each column, the GroupB processors compute their outputs according to Equation 7.2 and send them to GroupA for a specific row (See blue arrows in Figures 7.4 and 7.5). The communication is localized within each row of processors. For this communication, the $\sqrt{p}$ processors in one row of the 2D torus can be seen as connected in a ring topology (see Figure 7.6(b)). The row-wise communication requires a broadcast[2] in the ring topology with $\sqrt{p}$ processors. The diameter[3] of the ring is $\frac{\sqrt{p}}{2}$. So the packets' traveling time is $\alpha\frac{\sqrt{p}}{2}$. There are also two other operations, one send and one receive, which take 2β time units. There are $\frac{n}{\sqrt{p}}$ columns of weights in each processor, so the communication time for step1 of a forward tick T_{c1} (for any $1 < p < n^2$) is

$$T_{c1} = (\alpha\frac{\sqrt{p}}{2} + 2\beta)\frac{n}{\sqrt{p}}. \tag{7.9}$$

In step2, for each column, GroupA processors in one column do the vector-matrix multiplication according to equation 7.1 (see the red arrows in Figures 7.4 and 7.5). Note that each processor in the column only produces a partial result which is sent to the GroupB processors in the same column. The communication is thus localized within each column of processors. The $\sqrt{p}$ processors in one column of the torus can again be seen as connected in a ring topology. The column-wise communication requires the accumulation in a single processor[4] in a ring with $\sqrt{p}$ processors. For each single node accumulation, the diameter of the ring is $\frac{\sqrt{p}}{2}$. It takes $\alpha\frac{\sqrt{p}}{2}$ time units for the packet transmission. However, in step2, the first packet will arrive very soon since there is only one link to travel.

[2]Sending the same packet from a single processor to every other processor [BT89].

[3]The maximum distance of the network, where the distance is the minimum number of links between any pair of processor [BT89].

[4]Sending a packet to a given processor from every other processor[BT89].

In most practical situations, β is larger than α. So a new packet usually arrives before the previous packet was processed. As a result, the communication time for packets, $\alpha\frac{\sqrt{p}}{2}$, is hidden. Only the time of one send, one link transfer and $\sqrt{p}$ receive operations are considered, and they take $\beta(\sqrt{p}+1)+\alpha$ time units. There are $\frac{n}{\sqrt{p}}$ columns of weights in each processor, so the communication time for step2 of a forward tick T_{c2} (for any $1 < p < n^2$) is

$$T_{c2} = [\beta(\sqrt{p}+1)+\alpha)]\frac{n}{\sqrt{p}}. \tag{7.10}$$

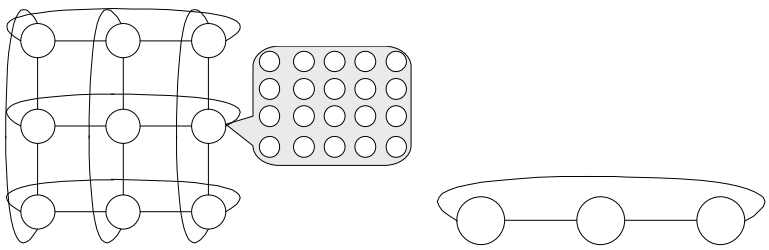
When $\sqrt{p}$ is large, $\beta(\sqrt{p}+1)+\alpha \approx \beta\sqrt{p}$. When $1 < p < n^2$, we obtain

$$T_{c1} = \frac{\alpha}{2}n + 2\beta\frac{n}{\sqrt{p}}, T_{c2} = \beta n. \tag{7.11}$$

In RNNs, BP starts when the forward phase has looped for a certain number of ticks. As shown in Figure 7.5, the communication pattern and time in the backward phase are exactly the same as in the forward phase.

In this model, only operations within the vector-matrix multiplication task are parallelized. But the three main tasks, vector-matrix multiplication, output computation and communications, are not parallelized and neither can they work in a pipelined mode, since the GroupB processors are selected from the processors in GroupA. This is also a common limitation found in some other published implementations [KSA94], [YI93]. Hereafter this model is referred to as the *non-pipelined model*.

Mapping onto SpiNNaker without pipeline



(a) Torus-connected CMPs (b) Ring

Figure 7.6: Network topologies and SpiNNaker

What if each node in Figure 7.4 is replaced by a chip multiprocessor (CMP),

as shown in Figure 7.6(a)? SpiNNaker is used as a paradigm for such an analysis. SpiNNaker is a system based on a torus-connected CMP topology (it also contains diagonal connections). In this study, the diagonal connections are not required, hence the system can be seen as a standard 2D torus topology (one SpiNNaker chip, or 20 cores, per node) as shown in Figure 7.6(a).

If the CBP is mapped onto SpiNNaker directly (without pipeline), all processors in a chip can be used for processing. Since there are 20 processors (in a 4x5 rectangle) in each chip, the diameter of the ring is reduced from $\frac{\sqrt{p}}{2}$ to $\frac{\sqrt{p}}{8}$, about 1/4 of the original value. The time required on SpiNNaker without pipeline becomes:

$$T_{c1} = \frac{\alpha}{8}n + 2\beta\frac{n}{\sqrt{p}}, T_{c2} = \beta n. \tag{7.12}$$

7.4.3 Mapping onto SpiNNaker with pipeline

To achieve better efficiency, a new pipelined model called PCBP is proposed. The mapping on SpiNNaker using the PCBP scheme is illustrated in Figure 7.7. Each rectangle (with rounded corners) represents one SpiNNaker chip. Each circle within a rectangle denotes a processing core. Up to 19 processors out of 20 are used in each chip. In this mapping scheme, rather than picking the GroupB processors from the GroupA processors, an independent set of processors (1 processor per chip) are employed as GroupB processors. In addition to the 16 GroupA processors and 1 GroupB processor, 2 GroupC processors are also employed. In step1, rather than sending packets from GroupB to GroupA directly, the packets now go through GroupC processors. GroupC processors get single node broadcast packets from GroupB processors, and then forward to GroupA processors. Similarly, in step2, the GroupC processors are responsible for receiving and accumulating partial results from the GroupA processors on the same chip and then forwarding the results to the GroupB processors. Each GroupC processor takes care of two columns/rows of GroupA processors (8 processors) in turn. In each chip, the three groups of processors are working in parallel and produce a six-stage pipeline as shown in Figure 7.8. Hereafter this mapping algorithm is referred to as the *pipelined model* (PCBP).

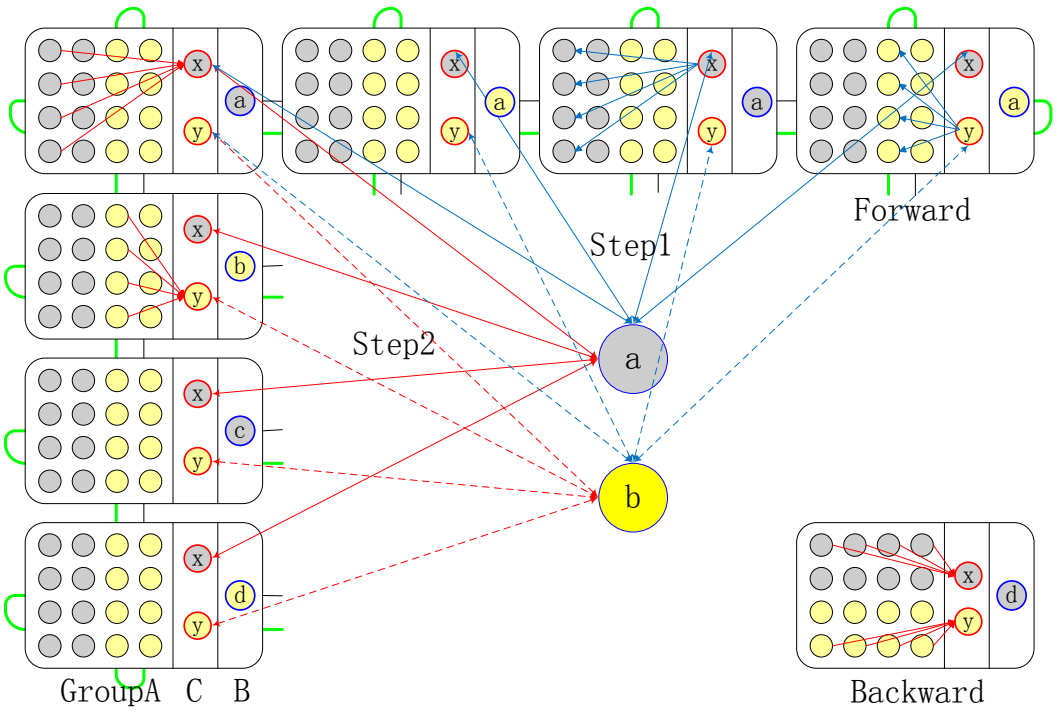


Figure 7.7: Mapping on SpiNNaker using the pipelined model (PCBP). Each rectangle (with rounded corners) represents one SpiNNaker chip. Each circle in a rectangle denotes a processing core. 19 processors out of 20 are used in each chip. Among them 16 (4 by 4) processors are allocated to GroupA, 1 (a/b/c/d) processor is allocated to GroupB and the other 2 (x and y) are allocated to GroupC. In step1, GroupB processors produce outputs and send to GroupC processors. GroupC processors get single node broadcast packets from GroupB processors, and then forward to GroupA processors. In step2, GroupA processors do the vector matrix computation and send results to GroupC processors with same color. Each GroupC processor receives packets from GroupA processors in two columns (2 by 4 processors) in turn and accumulates partial results, then forwards the results to GroupB processors. Notice that whatever the number of chips in one column, only four GroupB processors in total are required, each responsible for one column, since there are only four columns of GroupA processors in total. GroupC processors need to send packets to two GroupB processors in the same color in turn (for example processor x sends to processor a and c). The backward phase works exactly the same as the forward phase, but swaps the order of columns and rows. In each chip, the three groups of processors are working in parallel and produce a six-stage pipeline.

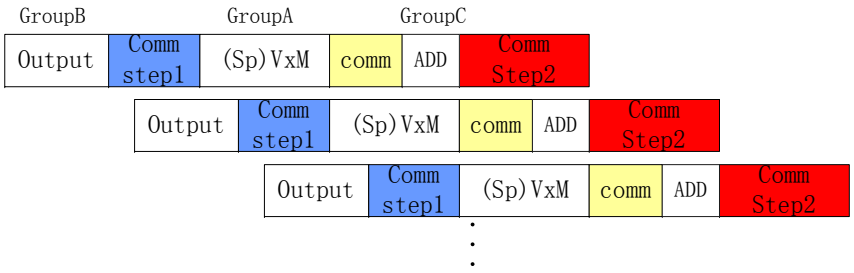


Figure 7.8: The six-stage pipeline. The pipeline is generated according to the mapping shown in Figure 7.7. The communication and computation are well overlapped with each other in this pipeline.

Communication analysis

Compared to the CBP model shown in Figures 7.4 and 7.5, in the PCBP model the on-chip communication between GroupA and GroupC processors is localized and small-scale (4 packets per column); the accumulation operation performed by GroupC processors requires only four processor cycles per column. Both of them are fast enough to be hidden by the off-chip communication or other computation.

The off-chip row-wise/column-wise communication can be seen as communication in a ring topology with a diameter of $\frac{\sqrt{p}}{8}$; and with the help of the GroupC processors, the number of packets that go to a GroupB processor is reduced to 1/4 of the original number. The communication time in step1 T_{pc1} and in step2 T_{pc2} when $1 < p < n^2$ is

$$T_{pc1} = \frac{\alpha}{8}n + 2\beta\frac{n}{\sqrt{p}}, T_{pc2} = \frac{\beta}{4}n. \tag{7.13}$$

7.4.4 Data storage

The way a sub-matrix is stored effects the performance of the computation. A matrix is stored either by row order storage (RS) or by column order storage (CS). RS is efficient for the backward computation, because elements in the same row are kept in a continuous space of the memory. But it is inefficient for the forward computation, because of the non-continuous weight storage. Similarly, CS is efficient for forward computation but inefficient for the backward computation. As a result, the time taken for the forward and backward computations is different. The CS scheme is used in this implementation, which causes the forward computation to be faster than the backward computation.

7.4.5 Analytical comparison

With the basic analytical models of the mapping algorithms in hand, they can be compared. For this, certain parameters need to be fixed. Assuming:

1. There are a number p of GroupA processors ($p = 20$ per chip in the non-pipelined model, $p = 16$ per chip in the pipelined model)
2. A multiply-accumulate operation of GroupA processors in a forward tick takes time θ_f(including the average time of memory access), while the same operation takes θ_b in a backward tick ($\theta_f \neq \theta_b$ due to the data presentation)
3. The output computation of GroupB processors in the forward phase takes o_f, and in the backward phase takes o_b

Thus we have:

- computing a sub-matrix in a forward and backward tick takes $\theta_f \frac{n^2}{p}$ and $\theta_b \frac{n^2}{p}$ respectively
- computing the outputs in a forward and backward tick takes $o_f \frac{n}{\sqrt{p}}$ and $o_b \frac{n}{\sqrt{p}}$ respectively

These are summarized in Table 7.1.

Table 7.1: Computation and communication cost

Operations	**Forward**		**Backward**	
	step1	step2	step1	step2
Comms. pipelined	$\frac{\alpha}{8}n + 2\beta\frac{n}{\sqrt{p}}$	$\frac{\beta}{4}n$	$\frac{\alpha}{8}n + 2\beta\frac{n}{\sqrt{p}}$	$\frac{\beta}{4}n$
Comms. non-pipelined	$\frac{\alpha}{8}n + 2\beta\frac{n}{\sqrt{p}}$	βn	$\frac{\alpha}{8}n + 2\beta\frac{n}{\sqrt{p}}$	βn
Comp. GroupA	$\theta_f \frac{n^2}{p}$		$\theta_b \frac{n^2}{p}$	
Comp. GroupB	$o_f \frac{n}{\sqrt{p}}$		$o_b \frac{n}{\sqrt{p}}$	

On a single processor

Let $p = 1$, then no communication is required. Time required T_{seri} for processing one forward tick plus one backward tick on a single processor is:

$$T_{seri} = (\theta_f + \theta_b)n^2 + (o_f + o_b)n. \tag{7.14}$$

The non-pipelined model

In the non-pipelined mode, communication and computation are carried out sequentially. The time required in the non-pipelined model for one forward tick plus one backward tick T^{np}_{para} is:

$$T^{np}_{para} = (\theta_f + \theta_b)\frac{n^2}{p} + (o_f + o_b)\frac{n}{\sqrt{p}} + (\frac{\alpha}{4} + 2\beta + \frac{4\beta}{\sqrt{p}})n. \tag{7.15}$$

The pipelined model

The pipelined model allows the overlap of communication and computation. In most practical problems, the link transmission time α is small compared to the packet send/receive time β, and the communication of step1 can be hidden behind the communication of step2 due to the pipeline (Figure 7.8). θ_b is obviously larger than θ_f when using column order storage (CS) of the weight matrix. The computation times of GroupB processors are smaller than either the communication time or the computation time of GroupA processors (shown later), therefore the computation of the GroupB processors is hidden behind either the communication or the computation time of GroupA processors. The time required in the pipelined model for one forward tick plus one backward tick T^{p}_{para} relies on the relationship between the computation and the communication because of the overlapping:

Situation1 Communication takes more time than either forward or backward computation when $p \geq 4\theta_b\frac{n}{\beta}$. In this situation, computation is fully hidden behind communication.

$$T^{p}_{para} = \frac{\beta}{2}n \tag{7.16}$$

Situation2 Communication takes longer than forward computation, but less time than backward computation, when $4\theta_f\frac{n}{\beta} \leq p < 4\theta_b\frac{n}{\beta}$. In this situation, communication is hidden partially.

$$T^{p}_{para} = \theta_b\frac{n^2}{p} + \frac{\beta}{4}n \tag{7.17}$$

Situation3 Communication takes less time than forward or backward computation when $p < 4\theta_f\frac{n}{\beta}$. In this situation, the communication is hidden behind the

computation.

$$T^{p}_{para} = (\theta_f + \theta_b)\frac{n^2}{p} \tag{7.18}$$

Speedup and efficiency

The speedup S and efficiency of an algorithm E are defined as:

$$S = \frac{T_{seri}}{T_{para}}, E = \frac{S}{p_{total}} \tag{7.19}$$

Where p_{total} is the total number of processors in the system, including GroupA, GroupB and GroupC processors (For the sake of fairness, $p_{total} = 20$ per chip are used in both non-pipelined model and pipelined model when calculating the efficiency E. This highlights the improvement provided by the new method).

7.4.6 Memory requirements

Here the memory requirement for GroupA processors (the most memory demanding processors among the three groups of processors) is assessed. Each GroupA processor needs to keep four types of data: weights, weight derivatives, input histories and delta errors, as shown in Table 7.2.

Table 7.2: The memory requirement of a GroupA processor in fully connected RNNs

Elements	**Symbol**	**Number**
Weights	w	$\frac{n^2}{p}$
Weight Derivatives	Δw	$\frac{n^2}{p}$
Input Histories	o	$\frac{n}{\sqrt{p}} T_{tk} I_{tv}$
Delta Errors	δ	$\frac{n}{\sqrt{p}}$

Each weight in the the sub-matrix is associated with a weight derivative to accumulate changes. A history buffer is required for keeping old inputs in a forward tick. The history inputs will be required in Equation 7.6 during a backward tick. The length of history is $T_{tk}I_{tv}$, since there are $T_{tk}I_{tv}$ forward propagations before backward propagations start. Each history comprises a vector of $n/\sqrt{p}$ inputs. Each processor also needs to keep a vector of $n/\sqrt{p}$ delta errors as inputs during a backward tick. Assuming each element takes s bytes and M_{full} is the

total memory requirement, we get:

$$M_{full} = [2\frac{n^2}{p} + (T_{tk}I_{tv} + 1)\frac{n}{\sqrt{p}}]s \tag{7.20}$$

7.4.7 Performance estimation

The performance of the system is evaluated analytically, since the real SpiNNaker chip is not available yet, and there is still quite a lot of software programming work remaining to be done to make the system fully functional. However, the analytical model presented helps us to determine the system scale and to predict the system performance.

A medium size system configuration up to 1000 chips is run to get first performance results on the SpiNNaker SoC Designer model. By using 16-bit fixed-point arithmetic we have (in nanoseconds)[5]:

$$\theta_f = 26, \theta_b = 42, \alpha = 10, \beta = 140, o_f = 880, o_b = 1400 \tag{7.21}$$

The first evaluation carried out is based on a scheme that runs a variety of scales of RNNs on a 500-chip SpiNNaker configuration. A performance comparison between the non-pipelined (CBP) and the pipelined (PCBP) model is shown in Figure 7.9. Initially the performance of the pipelined model is dominated by the communication, shown in red. When the scale of the network increases, the computation starts dominating the performance partially at the point of about 7,000 neurons, shown in green. When the number of neurons are more than about 11,000, the computation is fully dominating, shown in blue. The efficiency comparison between the pipelined and non-pipelined model on a 500-chip SpiNNaker, modeling up to 15,000 fully-connected neurons, is shown in Figure 7.10. The pipelined model is more efficient than the non-pipelined model in most of the cases and its efficiency can reach as high as 0.8 when computation dominates. The efficiency of the non-pipelined model increase when the ratio of neurons to processors increases, getting close to 1 when the ratio is large. However, in that case, the system behavior is closer to serial computing than to parallel computing.

[5]According to table 7.1, when $p \geq 1600$, $\frac{\beta}{4}n \geq o_b\frac{n}{\sqrt{p}}$; when $p < 1600$ and $n > 1360$, $\theta_f\frac{n^2}{p} > o_f\frac{n}{\sqrt{p}}$ and $\theta_b\frac{n^2}{p} > o_b\frac{n}{\sqrt{p}}$. So on any network with a population of 1360 neurons and above, the computation time of GroupB processors is either hidden by the communication or hidden by the computation time of the GroupA processors.

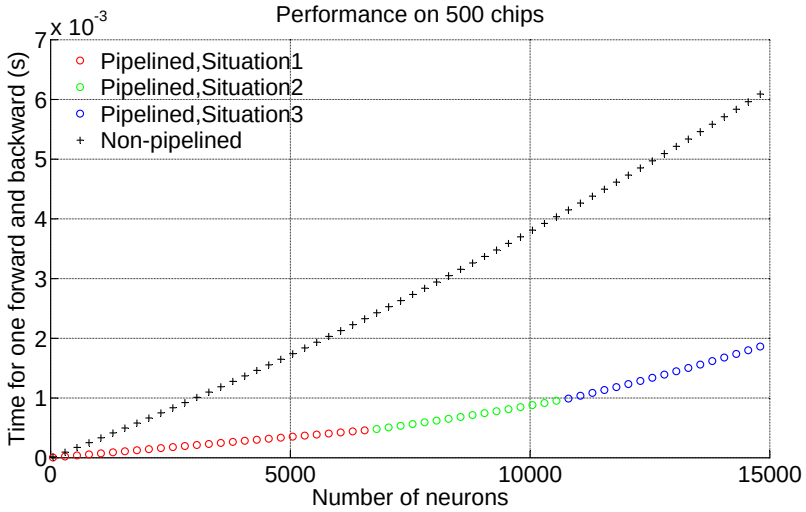


Figure 7.9: A performance comparison between the non-pipelined (CBP) and the pipelined (PCBP) models. Initially the performance of the pipelined model is dominated by the communication shown in red (situation 1). When the scale of the network increases, the computation starts dominating the performance partially at the point of about 7,000 neurons, shown in green (situation 2). When the number of neurons are more than about 11,000, the computation is fully dominating, shown in blue (situation 3).

In the second simulation, the effect of running a fixed number (5000) of neurons on a variant number of SpiNNaker chips is considered. As shown in Figure 7.11, it is very effective to use more chips initially, as the processing time drops dramatically. However, the gain in the performance becomes small at about 210 chips and above, where the communication becomes the bottleneck. The efficiency comparison is shown in Figure 7.12.

7.5 Partially-connected RNNs

This far, we have restricted our discussion to fully-connected neuronal networks. The proposed pipelined model works also for partially-connected RNNs. Connection weights in a partially-connected network form a sparse matrix. The efficiency of computation is effected by the distribution of elements in the matrix. If the elements are evenly distributed, each processor carries out a similar amount of computation and the backpropagation algorithm can be parallelized well. Otherwise, the workloads in the processors are not balanced, in which case, some of the processors may be busy while others are idle. A partially-connected RNN

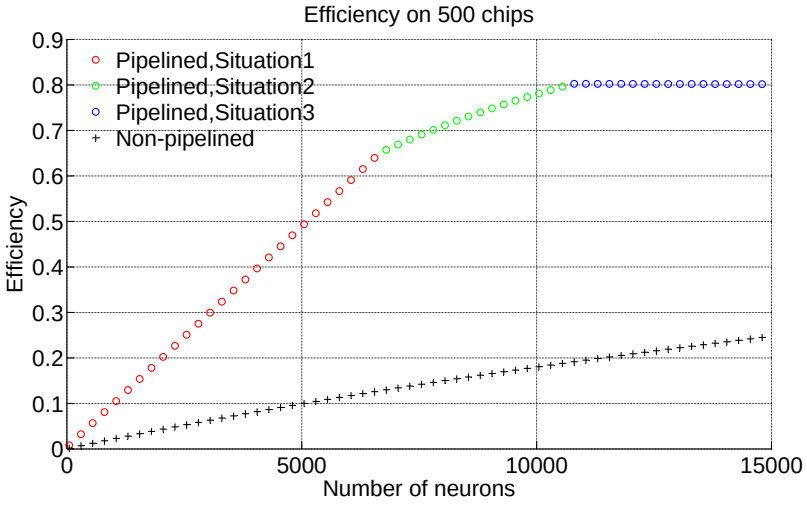


Figure 7.10: Efficiency comparison. Different situations of the pipelined model are illustrated by different colors. The pipelined model is more efficient than the non-pipelined model in most of the cases and its efficiency can reach as high as 0.8 when computation dominates. The efficiency of the non-pipelined model increases when the ratio of neurons to processors increases, getting close to 1 when the ratio is large. However, in that case, the system behavior is closer to serial computing than to parallel computing.

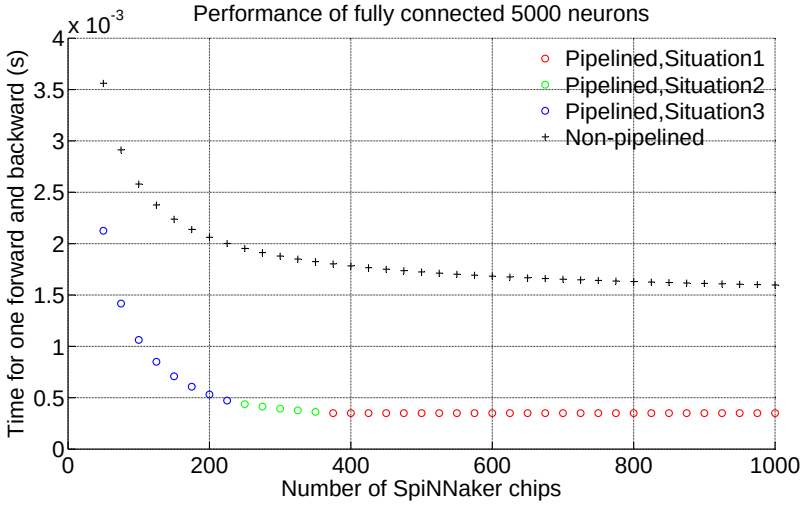


Figure 7.11: Performance comparison. It is very effective to use more chips initially (blue), as the computation is dominating. However, the processing time stops descending when using more than about 360 chips, in which case communication becomes the bottleneck (red).

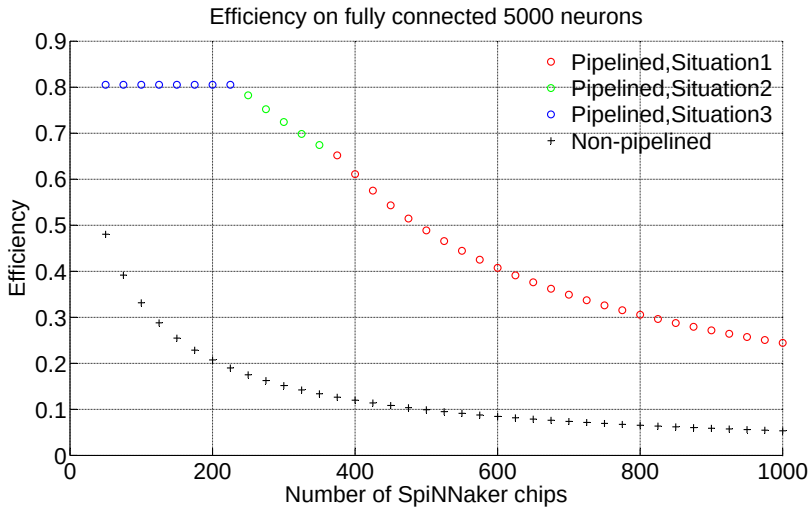


Figure 7.12: Efficiency comparison. Initially the efficiency of the pipelined model is high (blue), and then the efficiency drops because of the communication dominating (green and red), while in the non-pipelined case, the efficiency is always descending.

is less computationally intensive due to the sparsity. However, the gain in the communication time by the CBP partitioning is very little, as the communication can only be avoided when all of the elements in a column/row of the sub-matrix are non-zero (NZ) elements, which is a very rare case in a partial problem.

7.5.1 The data structure

The definition of the data structure is a problem of sparse matrix storage. Using sparsity saves a large amount of computation and memory space. In this case, the data structure has to be optimal for the sparse matrix-vector multiplication (SpMxV)). There are lots of proposed optimization techniques to improve the performance of SpMxV [KGK08].

A sparse data structure should be both compact and easy to access. The compressed column storage (CCS) [DGL89] is a widely used storage scheme for sparse matrices. The CCS scheme gets NZ elements in order from each column and keeps them in a continuous memory space. The representation of elements is given by three one-dimensional arrays. The first array keeps the numerical values of NZ elements. The second array keeps a list of row indices. There is one to one mapping between the first and the second array, and therefore both of them have a number of members equal to the total number of NZ elements. The third

array contains address pointers each of which points to an element in the first and the second arrays, indicating the starting address of a column. The length of the third array is equal to the number of columns. As an example, consider the matrix A below:

$$\begin{bmatrix} 1 & -2 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 9 & 0 & 0 & -1 \\ 5 & 7 & 0 & 8 \end{bmatrix}$$

The corresponding CCR representation is:

Table 7.3: Compressed Column Order Storage

ID	0	1	2	3	4	5	6	7
Val	1	9	5	-2	7	4	-1	8
Row	0	2	3	0	3	1	2	3
Col_start	0	3	5	6				

In Table 7.3 "Val", "Row" and "Col_start" denote the first, second and third array respectively.

The CCS scheme is quite efficient during the forward SpMxV computation but is inefficient during the backward SpMxV computation. The backward computation requires instant row-wise accessing of NZ elements. The compressed row storage (CRS) scheme, on the other hand, keeps the matrix row-wise and hence optimizes the backward computation. To achieve high computation speed in both phases, the weights for partially-connected RNNs are duplicated (not as in the fully-connected case), and CCS and CRS for forward and backward are used respectively. The drawback of duplicating weights is that the updating time is doubled, and it requires more memory space. In a practical problem, the duplicated updates will not decrease the performance too much, firstly because there is usually a long interval between two updates and secondly both the weight sets are located in the same processor.

7.5.2 Analytical comparison

In this study, assuming each neuron connects to a fixed population of m other neurons, the number of NZ elements in each row of the sparse matrix is m and

there are mn NZ elements in the sparse matrix in total. If the weights are uniformly distributed, there will be also about m NZ elements in each column. There are an average of mn/p NZ elements in each sub-matrix. Therefore, the computation times for the forward and backward phases is $\theta_{sf}mn/p$ and $\theta_{sb}mn/p$ respectively. The communication is the same as in Table 7.1. The results are summarized in Table 7.4

Table 7.4: The computation and communication time of partially-connected RNNs

Operations	**Forward**		**Backward**	
Comms. pipelined	step1	step2	step1	step2
	$\frac{\alpha}{8}n + 2\beta\frac{n}{\sqrt{p}}$	$\frac{\beta}{4}n$	$\frac{\alpha}{8}n + 2\beta\frac{n}{\sqrt{p}}$	$\frac{\beta}{4}n$
Comp. GroupA	$\theta_{sf}\frac{mn}{p}$		$\theta_{sb}\frac{mn}{p}$	
Comp. GroupB	$o_f\frac{n}{\sqrt{p}}$		$o_b\frac{n}{\sqrt{p}}$	

On a single processor

The time required T^s_{seri} for processing one forward tick plus one backward tick in partially-connected RNNs on a single processor:

$$T^s_{seri} = (\theta_{sf} + \theta_{sb})mn + (o_f + o_b)n \tag{7.22}$$

The non-pipelined model

The time required in the non-pipelined model for one forward tick plus one backward tick in partially-connected RNNs T^{snp}_{para} is:

$$T^{snp}_{para} = (\theta_{sf} + \theta_{sb})\frac{mn}{p} + (o_f + o_b)\frac{n}{\sqrt{p}} + (\frac{\alpha}{4} + 2\beta + \frac{4\beta}{\sqrt{p}})n \tag{7.23}$$

The pipelined mode

The time required for one forward tick plus one backward tick T^{sp}_{para} in the partially-connected RNNs is:

Situation1 Communication takes more time than either forward or backward computation when $p \geq 4\theta_{sb}\frac{m}{\beta}$. In this situation, computation is fully hidden behind communication.

$$T^{sp}_{para} = \frac{\beta}{2}n \tag{7.24}$$

Situation2 Communication takes more time than forward computation, but takes less time than backward computation, when $4\theta_{sf}\frac{m}{\beta} \leq p < 4\theta_{sb}\frac{m}{\beta}$. In this situation, communication is hidden partially.

$$T_{para}^{sp} = (\theta_{sb}\frac{m}{p} + \frac{\beta}{4})n \tag{7.25}$$

Situation3 Communication takes less time than forward or backward computation when $p < 4\theta_{sf}\frac{m}{\beta}$. In this situation, the communication is hidden behind the computation.

$$T_{para}^{sp} = (\theta_{sf} + \theta_{sb})\frac{mn}{p} \tag{7.26}$$

7.5.3 Memory requirements

Input histories and delta errors consume the same amount of memory space in a partially-connected RNN and a fully-connected RNN. However, a partial RNN requires an additional memory space for duplicated weights. CCS weights for the forward phase take $(2mn + n/\sqrt{p})$. CRS weights and weight derivatives for the backward phase take $(3mn + n/\sqrt{p})$. So we get Table 7.5:

Table 7.5: The memory requirements for a GroupA processor in partially-connected RNNs

Elements	**Symbol**	**Number**
CCS weights	w_{ccs}	$\frac{2mn}{p} + \frac{n}{\sqrt{p}}$
CRS Weights and Wight Derivatives	w_{crs},Δw	$\frac{3mn}{p} + \frac{n}{\sqrt{p}}$
Input Histories	o	$\frac{n}{\sqrt{p}}T_{tk}I_{tv}$
Delta Errors	δ	$\frac{n}{\sqrt{p}}$

Let each element be s bytes and total memory usage be M_{part}, we get:

$$M_{part} = [5\frac{mn}{p} + (T_{tk}I_{tv} + 3)\frac{n}{\sqrt{p}}]s \tag{7.27}$$

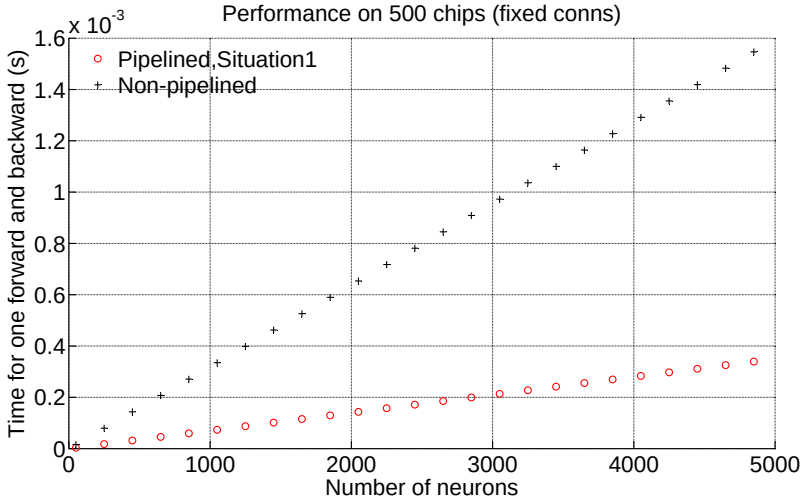


Figure 7.13: Performance comparison between non-pipelined model and pipelined model on 500 SpiNNaker chips, modeling up to 5000 neurons with a fixed number of 500 connections per neuron. In this case, the number of connections remains constant when the number of neurons increases, hence the time cost increases linearly. Since less communication is required as compared to the fully-connected case, the computation keeps dominating in this scenario.

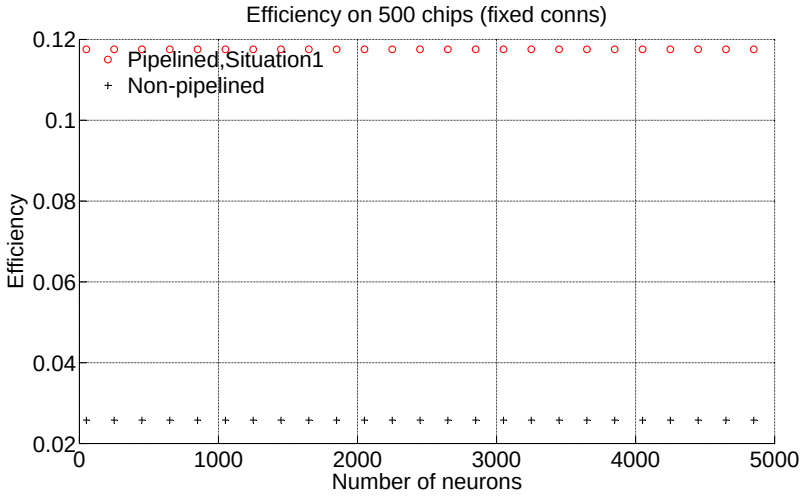


Figure 7.14: Efficiency comparison. The efficiency remains almost constant when the number of neurons increases, since the computation is always dominating in this scenario.

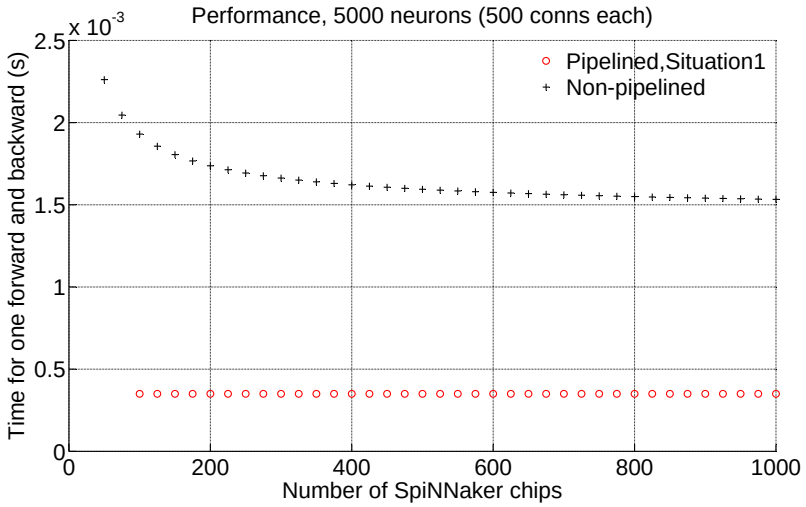


Figure 7.15: Performance comparison between non-pipelined model and pipelined model on modeling 5000 neurons with a fixed 500 connections per neuron, on SpiNNaker with up to 1000 chips.

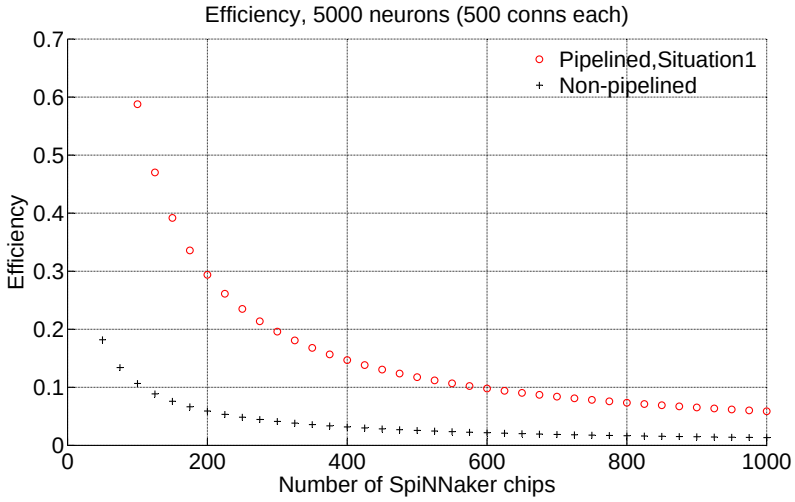


Figure 7.16: Efficiency comparison.

7.5.4 Performance estimation

According to the simulation results, we have[6]:

$$\theta_{sf} = 65, \theta_{sb} = 95 \tag{7.28}$$

Let $m = 500$. Similar evaluations as shown in the previous section are performed. Figure 7.13 shows the performance when modeling a range of numbers of neurons with a fixed number of connections on 500 SpiNNaker chips. Figure 7.14 shows the efficiency comparison of the non-pipelined model to the pipelined model. Figure 7.15 shows performance curves when modeling 5000 neurons on a number of SpiNNaker chips, and Figure 7.16 shows the efficiency comparison.

7.6 SpiNNaker VS. PC

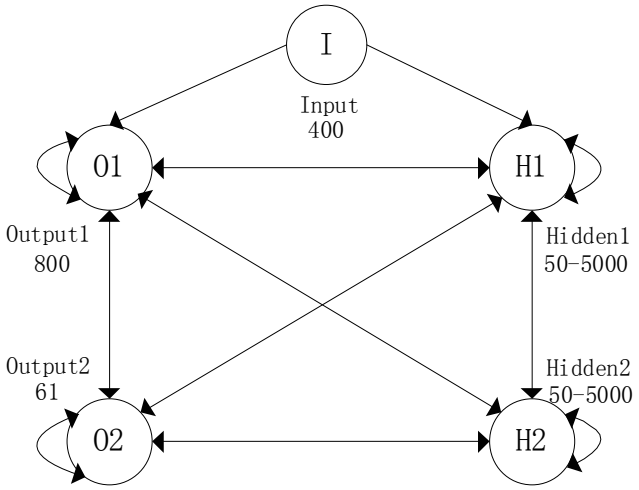


Figure 7.17: A recurrent neural network

Here, the performance on a 500-chip SpiNNaker machine to the performance on a Pentium 3.2GHz PC with 1GB RAM are compared. The network used for testing is a model of word reading, called the "primary system" [WR07, WCWRed]. The primary system model comprises one input layer, two hidden layers and two output layers as shown in Figure 7.17 (each circle denotes a layer of neurons). There are 400 neurons in the input layer I, 800 neurons in

[6]Because the weight derivatives are calculated in the backward phase according to equation 7.6, the backward computation takes longer than the forward computation

the output layer O1, 61 neurons in the output layer O2, and a variable number of 50 - 5000 neurons in each hidden layer. Weights are updated once after a batch. There are 8000 GroupA processors in the system (16 GroupA processors per chip). Assuming the number of hidden neurons in one hidden layer is h. For both fully-connected and partially-connected networks, we have:

$$T_{tk} = 5, I_{tv} = 1, P_{pt} = 3000, n = 1261 + 2h, p = 8000 \tag{7.29}$$

The performance on the 500-chip SpiNNaker machine is estimated analytically based on the equations developed in previous sections. The simulation time required for one update T^{SpiNN} is:

$$T^{SpiNN} = T_{para} T_{tk} I_{tv} P_{pt} \tag{7.30}$$

The performance on the Pentium PC is obtained from Lens simulations. Regression analysis on results from Lens simulations confirms that in the fully-connected case the processing time T^{pc}_{full} (in seconds) can be described by the quadratic equation:

$$T^{pc}_{full} = 0.000421h^2 + 0.308h + 180 \tag{7.31}$$

whereas in the partially-connected case (a fixed number of connections), the relationship between time T^{pc}_{part} and number of hidden units h is clearly linear, and can be described by the equation:

$$T^{pc}_{part} = 0.594h + 180 \tag{7.32}$$

7.6.1 The fully-connected network

The primary system model with full connections (each neuron connects to every other neurons) is firstly investigate.

Speed

The processing time required for one update is listed in Table 7.6.

In Table 7.6, "Hidden" is the number of neurons in one hidden layer; "Time on Spi." is the processing time for one weight update on the 500-chip SpiNNaker

Table 7.6: The time required for one update of the fully-connected networks

Hidden	Time on Spi. (s)	Sub-Matrix	Time on PC (s)
5000	16.1	125x125	12245
4500	13.6	114x114	10091
4000	11.6	103x103	8148
3500	9.71	92x92	6415
3000	7.96	81x81	4893
2000	5.52	59x59	2480
1000	3.42	36x36	909
500	2.37	25x25	439

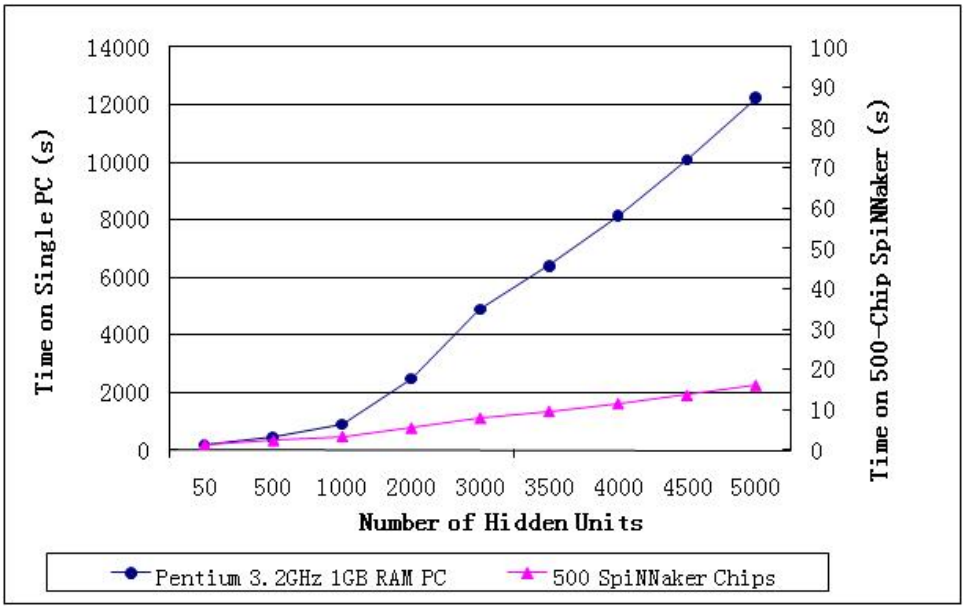


Figure 7.18: The speed comparison of the fully-Connected networks on SpiNNaker and on the PC.

machine, and "Sub-Matrix" is the size of the weight matrix each processor is modeling. "Time on PC" is the processing time on the Pentium PC simulation.

The speed comparison is shown in Figure 7.18. According to the results shown in Figure 7.18, the simulation speed on a 500-chip SpiNNaker (right scale) is approximately 400-700 times faster than the simulation speed on the Pentium PC (left scale). On both SpiNNaker and PC, the processing time increases exponentially when the size of the network increases.

Memory usage

Each element in this implementation is 16-bit (2 bytes), so $s = 2$. Assuming up to 60KB local memory in each processor is available to use, we have:

$$M_{full} < 60000 \tag{7.33}$$

By substituting Equation 7.20, and Equation 7.29 to Equation 7.33, we get:

$$h < 4870$$

As a result, with 500 SpiNNaker chips, it is possible to model a fully-connected RNN with up to about 4870 neurons per hidden layer.

7.6.2 The partially-connected network

In this simulation, the same "primary system" model, as the one used above, is used, but with only partial connections – each neuron connects to a fixed number of 500 other neurons ($m = 500$).

Speed

Results from the SpiNNaker simulation and from the PC simulation are listed in Table 7.7:

The speed comparison shown in Figure 7.19 indicates that the SpiNNaker machine (right scale) is about 200-300 times faster than the PC (left scale) in simulating the partially-connected networks. On both SpiNNaker and PC, the processing time increases linearly when the number of hidden units increases.

Table 7.7: The time required for one update of the partially-connected networks

Hidden	Time on Spi. (s)	Time on PC (s)
5000	11.82	3150
4000	9.72	2556
3000	7.62	1962
2000	5.52	1368
1000	3.42	774
500	2.37	447

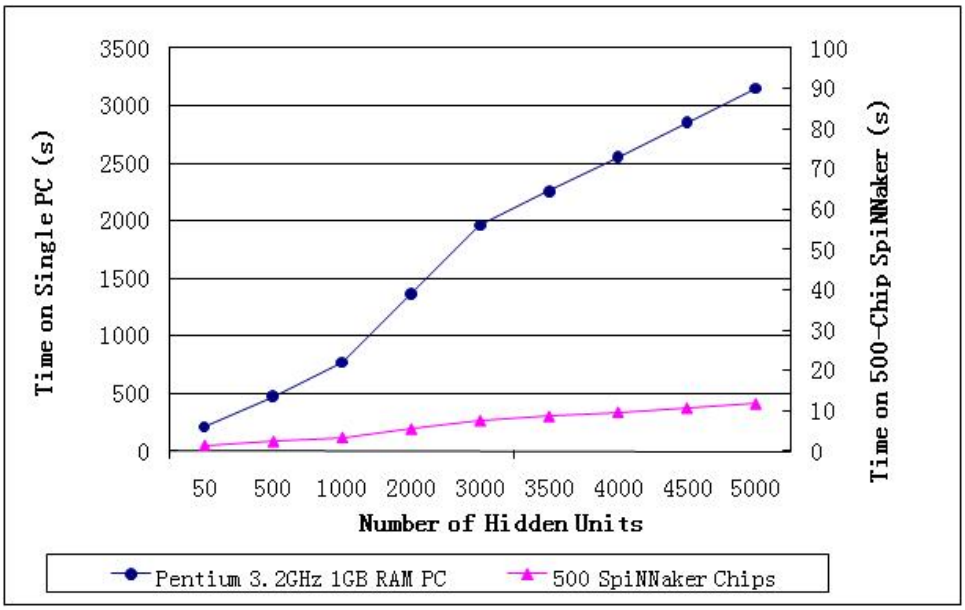


Figure 7.19: The speed comparison of the partially-connected networks on SpiNNaker and on the PC.

Memory usage

According to Equation 7.27, and Equation 7.29. Let $s = 2$ and $m = 500$. For $M_{part} < 60000$, we get:

$$h < 22011$$

As a result, with a 500-chip SpiNNaker machine, we are able to simulate a partially-connected RNN with up to about 22011 neurons per hidden layer using the duplicated compressed storage scheme.

7.7 Discussion and conclusion

This chapter presents an efficient implementation of MLP networks with the BP rule on a 2D torus-connected CMP topology. An efficient pipelined mapping scheme is proposed over the traditional non-pipelined scheme. By using the pipelined model, the communication can be localized and communication/computation overlap is allowed to avoid/reduce the overhead. Compared to the traditional non-pipelined mode, our scheme is more efficient in most practical cases. A detailed performance analysis of such an implementation is also presented. A mapping scheme is usually a trade-off between computation and communications. The performance curves shown are significant and can help to determine the ideal number of processors for a given scale of problem. We have shown analytically and through simulation that SpiNNaker is suited for computing MLP networks with the BP rule, providing a path to the further development of parallel solutions for the simulation of large-scale neural networks.

We have focused our analysis on recurrent networks. The pipelined (PCBP) model can also be applied to feedforward networks. To do this, each sub-matrix block (dot-filled blocks shown in Figure 7.1(c)) needs to be mapped on such a topology. Instead of using the same weight matrix in each tick as we do for recurrent networks, different sub-matrix blocks will be applied each time after update in the feedforward case, and it is still able to get benefits from the pipelined model.

There is no doubt that mapping schemes are mostly topology or architecture dependent. In this chapter, analysis is made on the torus-connected CMP topology and SpiNNaker is used as a platform for evaluation. The CBP mapping relies on a torus-like topology to partition the network into squares to achieve

good balance between the forward and backward phases of computation. The main contribution of this work is the new pipelined model which increases the efficiency of the original CBP mapping. The pipeline model may be applicable to other mapping schemes on other topologies.

The Drain time of the pipeline during analysis are not taken account. The large sub-matrix partition provides better consistency which leads to an improvement in the efficiency (load initialization and procedure call overheads). Therefore, an excessive number of processors with very small sub-matrix partitions are not recommended.

The communication bottleneck is that the receiving end during single node accumulation. It causes a blockage in the communication system, if an output processor (GroupB processor) receives too many packets at a time. This problem can be solved by a binary tree comprised of more GroupB processors. GroupA and GroupC processors are on the same CMP, this also brings a possibility to speedup the communication by using shared a memory scheme instead of using the standard multi-cast mechanism provided by the on-chip router.

There is a possible workload balance problem within different groups of processors: There are three groups of processors. In each column/row of processing, the workloads of Groups B and C are fixed, while the workload of Group A is variable, depending on the scale of the neural network. It has been shown that for any network with more than 1360 neurons on a system with fewer than 1600 processors, the processing time of Group A dominates. On a system with more than 1600 processors, the communication time dominates. As a result, the time cost of Groups B and C is always hidden in the pipeline when the system scale is large.

There is also a possible workload balance problem caused by neural network applications: The differences in neuron population in different layer of RNNs will NOT cause any workload balance problem, since RNNs use concurrent updating. But a non-uniform distribution of connections (which may happen in partially-connected networks) can cause balance problems. Workload balance problems are very common issues in parallel distributed processing, and we are not focused on solving these problems in this study.

We focus mostly on the communication between processors in the same column/row, while ignoring the concurrency of communication between processors in

different columns/rows. Since the communication is well localized to rows/-columns and the bottleneck is the packet receiving operation, the communication overhead brought in by the concurrency can be neglected.

One potential issue involved in the implementation is the synchronization between each group of processors. For instance, a GroupC processor needs to receive and accumulate the four packets generated from the first column of sub-matrix computation, by four GroupA processors in the same column, before moving on to process the next four packets generated from the second column of sub-matrix computation. Neither packets from different columns of the sub-matrix, nor the ones from different columns of processors should be mixed up. In a parallel system, the order of packet arrivals can not be guaranteed, which consequently requires a receiving processor to be able to identify the source of a packet (from which processor and which column/row of sub-matrix processing the packet was transmitted). A synchronization mechanism is required to solve this problem.

Chapter 8

Conclusion

8.1 Summary of thesis

This thesis explores algorithms as well as software implementation for efficient parallel simulation of neural networks on the SpiNNaker chip multiprocessor system. The main focus are on dealing with communication overhead incurred by the distributed processing, minimizing the processing time, and saving memory usage. Two typical neural networks with learning rules have been investigated: the spiking neural network model with STDP learning and the multi-layer perceptron model with BP learning. The modeling schemes are either fully implemented and tested (in the case of spiking neural network with STDP learning), or analytically studied and evaluated (in the case of multi-layer perceptron model with BP learning).

In this thesis, initially, a brief introduction to the modeling theory of spiking neural networks is presented in Chapter 2. Neural network applications are intrinsically parallel systems. The large population of processing units and complicated inter-connections are the source of the great potential of a neural network system, but also making the neural network modeling task extremely computationally demanding, which over-stretches the resources of the conventional desktop computers in simulating large-scale neural networks.

There have been many attempts to build parallel engineering systems for simulating large-scale neural networks. They were reviewed in Chapter 3. Most of the solutions have their particular benefits as well as downside. Based on the study of neural network models and existing engineering systems, we find that neuromorphic hardware ought not to be hardwired to a certain neural model,

since neural modeling research is still in the discovering stage. Neuromorphic hardware also needs to deliver considerable processing power for large-scale network simulation. The system reconfigurability with variant size is necessary to be applied to different fields. A new system called SpiNNaker was proposed accordingly to these requirements. SpiNNaker not only provides a general-purpose and high-performance platform for large-scale neural network simulation, but it is also flexible, scalable and power efficient. These features create the potential of using SpiNNaker for very different purposes. The proposed SpiNNaker architecture raises the research topic of understanding how to map neural models onto such a system – a topic investigated in the rest of the chapters.

Chapter 4 discusses building a neural system using the Izhikevich model on a single ARM968 processor. The first problem addressed was to determine how to simulate the Izhikevich equations efficiently on the ARM968 using 16-bit fixed-point arithmetic. A scheme called "dual scaling factor" was used to achieve an accurate resolution without sacrificing performance. By converting the presentation of the equations, and using ARM specific instructions, 1 ms simulation can be performed with 6 fixed-point mathematical operations plus 2 shift operations. The approach of modeling neural representations was then introduced – using an event-address mapping (EAM) scheme to store synaptic weights in the external memory at the receiving (post-synaptic) end. A lookup table maintains a mapping from the spike event to the address of the synaptic weight. Synaptic weights are fetched and transferred into the local memory when a spike arrives. Being an important feature of spiking neural networks, the synaptic delays are also implemented in this model. An event driven model is used for system scheduling. At the end of Chapter 4, the system was functionally tested by running a small network, and the performance was evaluated.

In Chapter 5, the previous single processor system is extended to a multi-processor system. Software running on the host PC called "InitLoad" was developed to do the automatic mapping. "InitLoad" loads the neural network description files, and converts them into data files which can be loaded onto SpiNNaker. The multi-processor simulation was tested using the four-chip SoC designer model of SpiNNaker to produce coincident results as Matlab simulations. The system is further tested by running a Doughnut Hunter neural application on both the SoC Designer model and the physical SpiNNaker Test Chip. Based on the experience gained from this study, software architecture for SpiNNaker was discussed at the

end of this chapter, leading to future study.

STDP was selected as the learning rule for neural network training. The implementation of STDP on SpiNNaker was described in Chapter 6. The distributed nature of parallel processing causes an inefficiency when implementing STDP on SpiNNaker using traditional pre-post-sensitive scheme. The problem is solved by an alternative pre-sensitive scheme, which triggering STDP only when a pre-synaptic spike arrives, because only at that time are the synaptic weights in the local memory of a processor and it only requires the weights to be kept in one order (indexed by the pre-synaptic neuron). Due to the difference between the electronic and biological time of packet arrival caused by the synaptic delay, a deferred event-driven model is used to postpone the STDP until there is enough information in the history record, which guarantees the size of time window for STDP. The result of the neural network with STDP simulation was shown at the end of the chapter.

As universal neuromorphic hardware, SpiNNaker is designed to support different neural models. The feasibility of modeling MLP networks onto SpiNNaker was investigated in Chapter 7. To achieve low communication overheads, the CBP scheme was used to partition the weight matrix. The performance of training MLPs on SpiNNaker was estimated analytically. To achieve better performance, a new scheme, based on the CBP scheme, called the PCBP scheme, is proposed to allow overlapping of communication and processing. The performance analysis shows that the PCBP scheme is more efficient than the CBP scheme in both full-connection and partial-connection cases. The memory requirements are also estimated for both types of network. The time of training the primary system model on SpiNNaker with the time on a single PC, is compared, to highlight the speedups achieved.

The research work described in this thesis demonstrates the feasibility of, and provides the implementation details for, modeling different types of neural network on a scalable chip multiprocessor system. During the study, a number of problems were solved by developing novel approaches which may also be applicable to other, larger, neural hardware models.

8.2 Future work

The research discovers several potential issues related to the real-time parallel simulation of neural networks, leading to further investigation:

1. Supporting more neural models. Spiking neural networks are the type of neural network that SpiNNaker was originally designed for. The Izhikevich neuronal model is used as an example during the study. There are also a range of other popular neuronal models such as the LIF model and the Hodgkin-Huxley model. They can also be implemented on SpiNNaker. Different models can be integrated into the system library, then users can choose which model to use in their simulation. The implementation of a neuronal model is dependent on other parts of the system, making it easy to extend the library of models. Other learning rules can also be investigated and implemented, in addition to STDP.

2. Monitor processor application. There is very little monitor processor function developed in this thesis. More monitor processor functions will be required for system management and fault-tolerance purposes..

3. Neuron to processor mapping. As previously discussed in Section 5.6.1, a well defined algorithm for neuron to processor mapping for the spiking neural network simulation is needed. This is a little bit complicated because it relates to how the neurons are indexed in an application network. The indices need to be distance-related to give extra information. This may require a rule to be built for indexing neurons based on their distances and communication patterns.

4. An easy-to-use software model. As previously discussed in Section 5.6.4, a well developed software model will be required to reduce the time for a new user to use such a parallel system. An interface between SpiNNaker and a general-purpose neural network description language is much preferred. It is ideal for users to run their existing application on SpiNNaker without changing their original code. A graphical user interface running on the Host PC will also be required for the easy downloading of neural codes, to check neuron states, and do the debugging when necessary.

5. More application tests. As soon as a more comprehensive software model is built on SpiNNaker, more applications can be run on SpiNNaker for the

further testing of the system. A more complicated application supported by a larger scale neural network is a good examination of the whole system.

6. New neural functions. The functionality of neurons implemented on SpiNNaker so far is very simple. There are a lot more neural dynamic effects, such as short-term plasticity, dopamine effects, conductance-based synapses and so on. These functions may also be required in some neural simulations. In addition, neural network theory is still developing, leading to discoveries of novel functions and models. The SpiNNaker neural library must be regularly updated to support the new theories.

Bibliography

[AB03] R.A. Ayoubi and M.A. Bayoumi. Efficient mapping algorithm of multilayer neural network on torus architecture. *Parallel and Distributed Systems, IEEE Transactions on*, 14(9):932–943, Sept. 2003.

[AB07] Sophie Achard and Ed Bullmore. Efficiency and cost of economical functional brain networks. *PLoS Comput Biol.*, 3:E17, 2007.

[AC81] J. Angevine and C. Cotman. *Principles of Neuroanatomy.* NY: Oxford University Press, New York, 1981.

[BDM04] Tom Binzegger, Rodney J. Douglas, and Kevan A. C. Martin. A quantitative map of the circuit of cat primary visual cortex. *The Journal of Neuroscience*, 24:8441–8453, 2004.

[Ble87] Guy Blelloch. Network learning on the connection machine. In *In Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 323–326, 1987.

[Boa00] K. A. Boahen. Point-to-point connectivity between neuromorphic chips using address events. *IEEE Trans. Circuits Syst.*, 47(5):416–434, 2000.

[BP98] Guoqiang Bi and Muming Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *The Journal of Neuroscience*, 18(24):10464–10472, 1998.

[BPF03] W.J. Bainbridge, L.A. Plana, and S.B. Furber. The design and test of a smartcard chip using a chain self-timed network-on-chip. In *Proc. DATE'04*, 2003.

[Bri] http://www.briansimulator.org/.

[BRSW91] W. Bialek, F. Rieke, Rob R. De Ruyter Van Steveninck, and D. Warland. Reading a neural code. *Science*, 252:1854–1857, 1991.

[BT89] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[BY00] G.A.P.C. Burns and M.P. Young. Analysis of the connectional organization of neural systems associated with the hippocampus in rats. *Philosophical Transactions of the Royal Society of London*, 355:55C70, 2000.

[Caj02] Santiago R.y Cajal. *Texture of the Nervous System of Man and the Vertebrates*. Springer, 2002.

[CEVB97] S.M. Crook, G.B. Ermentrout, M.C. Vanier, and J.M. Bower. The role of axonal delay in the synchronization of networks of coupled cortical oscillators. *Journal of Computational Neuroscience*, 4:161C172, 1997.

[CG90] B. W. Connors and M. J. Gutnick. Intrinsic firing patterns of diverse neocortical neurons. *Trends in Neurosci.*, 13:99–104, 1990. article.

[CK88] C. E. Carr and M. Konishi. Axonal delay lines for time measurement in the owl's brainstem. *PNAS*, 85:8311–8315, 1988.

[CKWR09] Gert Cauwenberghs, Moonjung Kyung, Eric Weiss, and Venkat Rangan. A vlsi implementation: Izhikevich's neuron model. Technical report, BENG/BGGN 260 Neurodynamics, University of California San Diego, 2009.

[DGL89] I. S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1):1–14, 1989.

[Ecc73] J. Eccles. *The Understanding of the Brain*. NY: McGraw-Hill Book Co., 1973.

[FE91] Daniel J. Felleman and David C. Van Essen. Distributed hierarchical processing in the primate cerebral cortex. *Cerebral Cortex*, 1:1–47, 1991.

[FSS97] Shou King Foo, P. Saratchandran, and N. Sundararajan. Parallel implementation of backpropagation neural networks on a heterogeneous array of transputers. *Systems, Man, and Cybernetics, Part B, IEEE Transactions on*, 27(1):118–126, Feb 1997.

[FT07] Steve Furber and Steve Temple. Neural systems engineering. *Journal of the Royal Society Interface*, 4(13):193–206, April 2007.

[FTB06] S. B. Furber, S. Temple, and A. D. Brown. On-chip and inter-chip networks for modelling large-scale neural systems, 2006.

[GB08] Dan Goodman and Romain Brette. Brian: a simulator for spiking neural networks in python. In *Front. Neuroinform.*, 2008.

[GB09] Dan F M. Goodman and Romain Brette. The brian simulator. In *Front. Neurosci.*, 2009.

[GBC99] Jay R. Gibson, Michael Beierlein, and Barry W. Connors. Two networks of electrically coupled inhibitory neurons in neocortex. *Nature*, 402:75–79, 1999.

[GEN] http://www.genesis-sim.org/genesis/.

[GFvH94] Raphael Ritz1and Wulfram Gerstner, Ursula Fuentes, and J. Leo van Hemmen. A biologically motivated and analytically soluble model of collective oscillations in the cortex. *Biological Cybernetics*, 71:349–358, 1994.

[GHS09] Mark Glover, Alister Hamilton, and Leslie S. Smith. An analog vlsi integrate-and-fire neural network for sound segmentation. In *Seventh International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems*, 2009.

[GK02] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[GKES89] C.M. Gray, P. Konig, A.K. Engel, and W. Singer. Oscillatory responses in cat visual cortex exhibit inter-columnar synchronization which reflects global stimulus properties. *Nature*, 338:334–337, 1989.

[GKvHW96] W. Gerstner, R. Kempter, J. L. van Hemmen, and H. Wagner. A neuronal learning rule for sub-millisecond temporal coding. *Nature*, 383:76–78, 1996.

[GW89] C.D. Gilbert and T.N. Wiesel. Columnar specificity of intrinsic horizontal and corticocortical connections in cat visual cortex. *Journal of Neuroscience*, 9:2432–2442, 1989.

[HCG+08] Patric Hagmann, Leila Cammoun, Xavier Gigandet, Reto Meuli, Christopher J. Honey, Van J. Wedeen, and Olaf Sporns. Mapping the structural core of human cerebral cortex. *PLoS Biol*, 6:e159, 2008.

[Heb49] D. O. Hebb. *The organization of Behavior*. Wiley-Interscience, New York, 1949.

[HGG+05] H. H. Hellmich, M. Geike, P. Griep, P. Mahr, M. Rafanelli, and H. Klar. Emulation engine for spiking neurons and adaptive synaptic weights. In *In Proc. IJCNN*, page 3261C3266, 2005.

[HH52] A. L. Hodgkin and A. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Physiol. London*, 117:500–544, 1952.

[Hil01] Bertil Hille. *Ion Channels of Excitable Membranes (3rd Edition)*. Sinauer Associates Inc, 2001.

[HK04] H. Hellmich and H. Klar. An FPGA based simulation acceleration platform for spiking neural networks. In *Circuits and Systems, 2004. MWSCAS '04. The 2004 47th Midwest Symposium on*, volume 2, pages II–389–II–392 vol.2, July 2004.

[Hop82] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences*, volume 79, pages 2554 – 2559, 1982.

[HTT06] M. Hartley, N. Taylor, and J. Taylor. Understanding spike-time-dependent plasticity: A biologically motivated computational model. *Neurocomputing*, 69(16):2005–2016, July 2006.

[IE08] Eugene M. Izhikevich and Gerald M. Edelman. Large-scale model of mammalian thalamocortical systems. *PNAS*, 105:3593–3598, 2008.

[IF07] G. Indiveri and S. Fusi. Spike-based learning in VLSI networks of integrate-and-fire neurons. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3371–3374, May 2007.

[IGE04] Eugene M. Izhikevich, Joe A. Gally, and Gerald M. Edelman. Spike-timing dynamics of neuronal groups. *Cerebral Cortex*, 14:933–944, 2004.

[Ind03] G. Indiveri. A low-power adaptive integrate-and-fire neuron circuit. In *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, volume 4, pages 820–823, May 2003.

[Izh03] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Trans. Neural Networks*, 14(6):1569– 1572, 2003.

[Izh04] E. M. Izhikevich. Which model to use for cortical spiking neurons. *IEEE Trans. Neural Networks*, 15(5):1063– 1070, 2004. article.

[Izh06] Eugene M. Izhikevich. Polychronization: Computation with spikes. *Neural Computation*, 18(2):245–282, February 2006.

[Izh07] Eugene M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. The MIT Press, 2007.

[JFW08] X. Jin, S. Furber, and J. Woods. Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In *Proc. 2008 International Joint Conference on Neural Networks*, Hong Kong, 2008.

[JGP+10] X. Jin, F. Galluppi, C. Patterson, A.D. Rast, S. Davies, S. Temple, and S.B. Furber. Algorithm and software for simulation of spiking neural networks on the multi-chip spinnaker system. In *Proc. 2010 International Joint Conference on Neural Networks*, 2010.

[JLK+10a] X. Jin, M. Lujan, M.M. Khan, L.A. Plana, A.D. Rast, S.R.Welbourne, and S.B. Furber. Algorithm for mapping multilayer bp networks onto the spinnaker neuromorphic hardware. In *Proc. International Symposium on Parallel and Distributed Computing (ISPDC'2010)*, 2010.

[JLK+10b] X. Jin, M. Lujan, M.M. Khan, L.A. Plana, A.D. Rast, S.R. Welbourne, and S.B. Furber. Efficient parallel implementation of multilayer backpropagation network on torus-connected cmps. In *Proc. of the ACM International Conference on Computing Frontiers*, 2010.

[JMM89] K. Joe, Y. Mori, and S. Miyake. Simulation of a large-scale neural network on a parallel computer. In *Proc. 1989 Conf. Hypercubes, Concurrent Computation Application*, pages 1111–1118, 1989.

[JRG+09] X. Jin, A. Rast, F. Galluppi, M. Khan, and S. Furber. Implementing learning on the spinnaker universal neural chip multiprocessor. In *Proc. 16th Intl. Conf. on Neural Information Processing (ICONIP2009)*, Bangkok, Thailand, 2009.

[JRG+10] X. Jin, A.D. Rast, F. Galluppi, S. Davies, and S.B. Furber. Implementing spike-timing-dependent plasticity on spinnaker neuromorphic hardware. In *Proc. 2010 International Joint Conference on Neural Networks*, 2010.

[KGK08] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 87–96, New York, NY, USA, 2008. ACM.

[KH89] S. Y. Kung and J. N. Hwang. A unified systolic architecture for artificial neural networks. *J. Parallel Distrib. Comput.*, 6(2):358–387, 1989.

[Kha09] Muhammad Mukaram Khan. *Configuring a Massively Parallel CMP System for Real-Time Neural Applications*. PhD thesis, Computer Science, University of Manchester, 2009.

[KLP+08] M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E.Painkras, and S. Furber. Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *In Proc. Intl. Joint Conf. on Neural Networks (IJCNN2008)*, Hong Kong, 2008.

[Koh95] Teuvo Kohonen. *Self-Organizing Maps*. Springer Series in Information Sciences, 1995.

[KPJ+09] M.M. Khan, E. Painkras, X. Jin, L.A. Plana, J.V. Woods, and S.B. Furber. System level modelling for spinnaker cmp system. In *Proc. 1st International Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'09)*, 2009.

[KS92] A. K. Kreiter and W. Singer. Oscillatory neuronal responses in the visual cortex of the awake macaque monkey. *European Journal of Neuroscience*, 4:369–375, 1992.

[KSA94] Vipin Kumar, Shashi Shekhar, and Minesh B. Amin. A scalable parallel formulation of the backpropagation algoritm for hypercubes and related architectures. In *IEEE Transactions on Parallel and Distributed Systems*, volume 5, 1994.

[Lan03] J. Langner. *Development of a Parallel Computing Optimized Head Movement Correction Method in Positron Emission Tomography*. PhD thesis, University of Applied Sciences Dresden and Research Center Dresden-Rossendorf, Germany,, 2003.

[Les95] Rmy Lestienne. Determination of the precision of spike timing in the visual cortex of anaesthetised cats. In *Biological Cybernetics*, 1995.

[LMAB06] J. Lin, P. Merolla, J. Arthur, and K Boahen. Programmable connections in neuromorphic grids. In *49th IEEE Midwest Symposium on Circuits and Sysmtems*, pages 80–84, 2006.

[Ltda] ARM Ltd. *ARM RealView SoC Designer v7.1 User Guide*.

[Ltdb] ARM Ltd. *RealView Development Suite v3.1*.

[Mah92] M. Mahowald. *VLSI analogs of neuronal visual processing: a synthesis of form and function*. PhD thesis, California Inst. Tech., Pasadena, CA, 1992.

[Mar06] H. Markram. The blue brain project. *Nat Rev Neurosci.*, 7:153–160, 2006.

[MASB07] Paul A. Merolla, John V. Arthur, Bertram E. Shi, and Kwabena A. Boahen. Expandable networks for neuromorphic chips. In *IEEE Transactions on Circuits and Systems*, February 2007.

[MD91] Misha Mahowald and Rodney Douglas. A silicon neuron. *Nature*, 354:515 – 518, 1991.

[Mea89] Carver Mead. *Analog VLSI and neural systems*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[MGT08] Timothe Masquelier, Rudy Guyonneau, and Simon J. Thorpe. Spike timing dependent plasticity finds the start of repeating patterns in continuous spike trains. *PLoS ONE*, 3(1):e1377, 01 2008.

[MMG+07] L.P. Maguire, T.M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin. Challenges for large-scale implementations of spiking neural networks on FPGAs. *Neurocomputing*, 71(1-3):13 – 29, 2007.

[MNTP03] L. Marner, J.R. Nyengaard, Y. Tang, and B. Pakkenberg. Marked loss of myelinated nerve fibers in the human brain with age. *J Comp Neurol.*, 462:144–52, 2003.

[Mor03] RGM. Morris. The discovery of long-term potentiation. *Philosophical Transactions of the Royal Society of London*, 358:617–620, 2003.

[MP43] W. Mcculloch and W. A. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943. incollection.

[MP69] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, 1969. book.

[MT96] H. Markram and M. Tsodyks. Redistribution of synaptic efficacy between neocortical pyramidal neurons. *Nature*, 382(382):807–810, 1996.

[NASV+03] Lionel G. Nowak, Rony Azouz, Maria V. Sanchez-Vives, Charles M. Gray, and David A. McCormick. Electrophysiological classes of cat primary visual cortical neurons in vivo as revealed by quantitative analyses. *J Neurophysiol*, 89:1541–1566, 2003.

[NDK+09] Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeffrey L. Krichmar, Alex Nicolau, and Alex Veidenbaum. Efficient simulation of large-scale spiking neural networks using cuda graphics processors. In *International Joint Conference on Neural Networks*, 2009.

[NEU] http://neuron.duke.edu/.

[NMW92] J. Nichols, A. Martin, and B. Wallace. *From Neuron to Brain*. MA: Sinauer Associates, Inc., Sunderland, 3rd edition edition, 1992.

[PDG93] A. Petrowski, G. Dreyfus, and C. Girault. Performance analysis of a pipelined backpropagation parallel algorithm. *Neural Networks, IEEE Transactions on*, 4(6):970–981, Nov 1993.

[Pet94] A. Petrowski. Choosing among several parallel implementations of the backpropagation algorithm. In *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*, volume 3, pages 1981–1986 vol.3, Jun-2 Jul 1994.

[PFT+07] L. Plana, S. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang. A gals infrastructure for a massively parallel multiprocessor. *IEEE Design & Test of Computers*, 24(5):454–463, Sep.-Oct. 2007.

[PGG+05] Martin Pearson, Ian Gilhespy, Kevin Gurney, Chris Melhuish, Benjamin Mitchinson, Mokhtar Nibouche, and Anthony Pipe. A real-time, fpga based, biologically plausible neural network processor. In *In proceeding of Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005*, 2005.

[PNS09] D. Pecevski, T. Natschlager, and K. Schuch. Pcsim: a parallel simulation environment for neural circuits fully integrated with python. In *Front. Neuroinform*, 2009.

[RB88] Charles R. Rosenberg and Guy Blelloch. An implementation of network learning on the connection machine. *Connectionist models and their implications: readings from cognitive science*, pages 329–340, 1988.

[RCF+05] M. LA ROSA, E. CARUSO, L. FORTUNA, M. FRASCA, L. OCCHIPINTI, and F. RIVOLI. Neuronal dynamics on fpga : Izhikevich's model. In *Proceedings of the International Society for Optical Engineering*, July 2005.

[RGJF10] A. D. Rast, F. Galluppi, X. Jin, and S.B. Furber. The leaky integrate-and-fire neuron: A platform for synaptic model exploration on the spinnaker chip. In *Proc. 2010 International Joint Conference on Neural Networks*, 2010.

[RHM86] D.E. Rumelhart, G.E. Hinton, and J.L. Mcclelland. *A General Framework for Parallel Distributed Processing*, chapter 2, pages 45–76. MIT Press, 1986.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*, chapter 8, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[RJG+10] A.D. Rast, X. Jin, F. Galluppi, L.A. Plana, C. Patterson, and S.B. Furber. Scalable event-driven native parallel processing: The spinnaker neuromimetic system. In *ACM International Conference on Computing Frontiers 2010*, 2010.

[RMG86] D. E. Rumelhart, J. L. Mcclelland, and The P. D. P. Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations.* Cambridge, MA: MIT Press, 1986.

[Roh] Douglas Rohde. Lens manual - the light, efficient network simulator.

[Ros58] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[RS97] Raphael RitzE and Terrence J Sejnowski. Synchronous oscillatory activity in sensory systems: new vistas on mechanisms. *Current Opinion in Neurobiology*, 7:536–546, 1997.

[SA01] Sen Song and L.F. Abbott. Cortical development and remapping through spike timing-dependent plasticity. *Neuron*, 32(2):339 – 350, October 2001.

[SBC02] H. Z. Shouval, M. F. Bear, and L. N. Cooper. A unified model of nmda receptor-dependent bidirectional synaptic plasticity. *PNAS*, 99(16):10831–10836, August 2002.

[SBY95] J.W. Scannell, C. Blakemore, and M.P. Young. Analysis of connectivity in the cat cerebral cortex. *Journal of Neuroscience*, 15:1463–1483, 1995.

[SD99] M.F. Simoni and S.P. DeWeerth. Adaptation in a vlsi model of a neuron. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 46(7):967–970, Jul 1999.

[Sin95] W. Singer. The role of synchrony in neocortical processing and synaptic plasticity. In *Physics of Neural Networks. Models of Neural Networks II*, pages 141–173, 1995.

[Siv91] M. Sivilotti. *Wiring considerations in analog VLSI systems, with application to field-programmable networks*. PhD thesis, California Inst. Tech., Pasadena, CA., 1991.

[SJ95] S.R. Schultz and M.A. Jabri. Analogue vlsi 'integrate-and-fire' neuron with frequency adaptation. *Electronics Letters*, 31(16):1357–1358, Aug 1995.

[SM98] V. Sudhakar and C. Siva Ram Murthy. Efficient mapping of backpropagation algorithm onto a network of workstations. *Systems, Man, and Cybernetics, Part B, IEEE Transactions on*, 28:841–848, 1998.

[SMA00] Sen Song, Kenneth D. Miller, and L. F. Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, 3:919 – 926, 2000.

[SMJK98] T. Schoenauer, N. Mehrtash, A. Jahnke, and H. Klar. Maspinn: Novel concepts for a neuroaccelerator for spiking neural networks. In *Proc. VIDYNN'98*, Stockholm, 1998.

[SOM05] S. Suresh, S.N. Omkar, and V. Mani. Parallel implementation of back-propagation algorithm in networks of workstations. *Parallel and Distributed Systems, IEEE Transactions on*, 16(1):24–34, Jan. 2005.

[SS98] N. Sundararajan and P. Saratchandran. *Parallel Architectures for Artificial Neural Networks : Paradigms and Implementations.* Wiley-IEEE Computer Society Press, December 1998.

[ST02] Olaf Sporns and Giulio Tononi. Classes of network connectivity and dynamics. *Complexity*, 7:28–38, 2002.

[Ste67] R. B. Stein. Some models of neuronal variability. *Biophys.*, 7:37–68, 1967.

[STE00] O. Sporns, G. Tononi, and G.M. Edelman. Theoretical neuroanatomy: Relating anatomical and functional connectivity in graphs and cortical connection matrices. *Cerebral Cortex*, 10:127–141, 2000.

[STK05] O. Sporns, G. Tononi, and R. Kotter. The human connectome: A structural description of the human brain. *PLoS Comput Biol.*, 1:245C251, 2005.

[Swa85] H. A. Swadlow. Physiological properties of individual cerebral axons studied in vivo for as long as one year. *Journal of Neurophysiology*, 54:1346–1362, 1985.

[Swa88] H. A. Swadlow. Efferent neurons and suspected interneurons in binocular visual cortex of the awake rabbit: receptive fields and binocular properties. *Journal of Neurophysiology*, 59:1162–1187, 1988.

[TFM96] Simon Thorpe, Denis Fize, and Catherine Marlot. Speed of processing in the human visual. *Nature*, 381:520–522, 1996.

[TM07] Jean-Philippe Thivierge and Gary F. Marcus. The topographic brain: from neural connectivity to cognition. *Trends in Neurosciences*, 30:251–259, 2007.

[Tuc88] Henry C. Tuckwell. *Introduction to Theoretical Neurobiology*. Cambridge University Press, 1988.

[WC90] B. W. Wah and L. Chu. Efficient mapping of neural networks on multicomputers. In *Int. Conf. Parallel Processing*, pages I234–1241, 1990.

[WCWRed] Stephen R. Welbourne, J. Crisp, A. Woollams, and Matthew A. Lambon Ralph. Phonological, deep and surface dyslexia from a single pdp model of reading: An implementation of the primary systems hypothesis. *Psychological Review*, submitted.

[WD07] J.H.B. Wijekoon and P. Dudek. Spiking and bursting firing patterns of a compact vlsi cortical neuron circuit. In *IJCNN*, 2007.

[WR07] Stephen R. Welbourne and Matthew A. Lambon Ralph. Using parallel distributed processing models to simulate phonological dyslexia: The key role of plasticity related recovery. *Journal of Cognitive Neuroscience*, 19:1125–1139, 2007.

[YI93] T. Yukawa and T. Ishikawa. Optimal parallel back-propagation schemes for mesh-connected and bus-connected multiprocessors. *Neural Networks, 1993., IEEE International Conference on*, 3:1748–1753, 1993.

[You92] M. P. Young. Objective analysis of the topological organization of the primate cortical visual system. *Nature*, 358:152–155, 1992.

[ZMMW90a] Xiru Zhang, Michael McKenna, Jill P. Mesirov, and David L. Waltz. The backpropagation algorithm on grid and hypercube architectures. *Parallel Computing*, 14(3):317–327, 1990.

[ZMMW90b] Xiru Zhang, Michael Mckenna, Jill P. Mesirov, and David L. Waltz. An efficient implementation of the back-propagation algorithm on the connection machine cm-2. *Advances in neural information processing systems*, 2:801–809, 1990.

[ZTH+98] Li I. Zhang, Huizhong W. Tao, Christine E. Holt, William A. Harris, and Muming Poo. A critical window for cooperation and competition among developing retinotectal synapses. *Nature*, 395:37–44, 1998.