PARALLELISATION OF NEURAL PROCESSING ON NEUROMORPHIC HARDWARE

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

2022

Luca Peres
Department of Computer Science

Contents

Word Count: 43361

List of Tables

List of Figures

Glossary

AER Address Event Representation. 46, 59

AI Artificial Intelligence. 33

ANN Artificial Neural Network. 32, 124

API Application Programming Interface. 41, 42, 54, 55, 68, 73, 76, 100, 153

ARM Advanced RISC Machine. 47, 48, 56, 196

ASIC Application Specific Integrated Circuit. 39, 43

CNN Convolutional Neural Network. 43

CPU Central Processing Unit. 40, 41, 43

DDR Double Data Rate. 151

DMA Direct Memory Access. 49, 66, 76, 82, 83, 88–90, 92, 102, 127, 128, 130, 153, 158

DNN Deep Neural Network. 43

DTCM Data Tightly-Coupled Memory. 49, 61, 65, 69, 74, 101, 128

FIQ Fast Interrupt Queue. 55

FPGA Field Programmable Gate Array. 42, 51, 111, 141

FPU Floating Point Unit. 117

GALS Globally Asynchronous Locally Synchronous. 47

GPU Graphic Processing Unit. 42, 43, 98, 99, 108, 110, 111, 188

HBP Human Brain Project. 28

HPC High-Performance Computing. 40, 41, 98, 99, 108, 110, 111

IoT Internet of Things. 197

IRQ Interrupt ReQuest. 55

ISR Interrupt Service Routine. 55

ITCM Instruction Tightly-Coupled Memory. 49, 54

LIF Leaky Integrate-and-Fire. 36, 95, 158, 169

LSB Least Significant Bit. 64, 104

LTD Long Term Depression. 37

LTP Long Term Potentiation. 37

ML Machine Learning. 32

MLP Multi Layer Perceptron. 32

MNIST Modified National Institute of Standards and Technology database. 124, 125, 128, 132, 134, 135, 141, 194

MPI Message Passing Interface. 41, 42

MSB Most Significant Bit. 104

NEST NEural Simulation Tool. 40, 42, 111

NoC Network-on-Chip. 47, 151, 196

RNN Recurrent Neural Network. 34

ROLLS Reconfigurable On-Line Learning Spiking Neuromorphic Processor. 45

SARK SpiNNaker Application Runtime Kernel. 54

SC&MP SpiNNaker Control Monitor Program. 54

SDRAM Synchronous Dynamic Random Access Memory. 48, 49, 53, 54, 61, 63, 64, 66, 69, 73, 82, 83, 90, 91, 102, 128, 130, 132, 147, 150, 151, 153, 154, 156–158, 160, 163, 166, 167, 183, 196

SNN Spiking Neural Network. 25, 27–30, 33, 34, 36, 39–41, 43, 47, 51, 54, 56, 59, 61, 64, 70–72, 74, 77, 85, 97, 98, 110, 111, 113, 117, 125, 127, 142, 145, 146, 148, 155, 157, 158, 160, 167, 169, 171, 174, 175, 189, 191, 193, 194, 196, 197

SpiNNaker Spiking Neural Network Architecture. 26, 29, 30, 32, 47, 48, 50, 51, 53–57, 59, 61, 64, 69–74, 76, 77, 79, 84, 85, 90, 91, 93, 98–102, 104, 107–111, 113, 114, 117, 118, 120, 124–126, 128, 131, 132, 134–137, 139, 141, 142, 146, 149–153, 157, 160, 162, 169–171, 176, 190, 193, 194, 196

STDP Spike-Timing Dependent Plasticity. 37, 42, 45–47, 118

UDP User Datagram Protocol. 53

US Urbanczik-Senn. 117, 119–122, 137

Abstract

Parallelisation of Neural Processing on Neuromorphic Hardware
Luca Peres
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2022

Learning and development in biological brains typically happen over long timescales, making experimental exploration at the level of individual neurons challenging. Computer simulations of Spiking Neural Network (SNN) models offer a potential route to investigate these phenomena. Accelerating large-scale brain simulations on conventional hardware however is a challenge. Researchers therefore need efficient simulation tools and platforms to complete these tasks in real- or sub real time, to enable exploration of features such as long-term learning and neural pathologies over meaningful periods. Neuromorphic engineering aims to provide suitable platforms for such tasks by building architectures whose structures emulate the mammalian brain and therefore to reduce the time and energy impact that Neural Networks simulations have on standard computing platforms. In order to perform real-time simulations of biologically representative Spiking Neural Networks however, digital Neuromorphic platforms need innovative programming paradigms to best exploit their hardware features. This research explores parallelisation strategies for neural applications to address real-time simulations of SNNs, including on-line learning strategies, with the aim of maximising the throughput of neural operations.

This work employs the many-core SpiNNaker digital Neuromorphic hardware as a research platform, and proposes strategies that enabled the world's first real-time simulation of the Cortical Microcircuit model, a benchmark SNN describing the behaviour of the mammalian cortex, achieving performance 20× better than previously published results. The parallelisation strategies are then extended and generalised to on-line learning applications, involving the use of multicompartmental neuron models for classification and regression tasks. Finally, new partitioning strategies affecting the placement of neural components on Neuromorphic hardware are presented. These strategies make more efficient use of the available hardware features, effectively reducing the required resources and providing additional flexibility in order to handle sparser SNNs simulations. Through this final development, up to 9× higher throughput of neural operations is demonstrated, together with improved handling of biologically-representative sparse connectivity patterns.

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

A PhD is a long journey which sets new challenges every day. In this journey I have been extremely lucky to find the best guidance possible. First of all I would like to thank Professor Steve Furber for his supervision throughout these three years and for believing in me even during the most challenging phases.

I am extremely thankful to my co-supervisor Dr. Oliver Rhodes for his invaluable guidance, not only for my professional development, but also for being the best mentor I could ask for on the personal side. I wouldn't have made it this far without him.

I would like to thank the EPSRC and The University of Manchester for funding my PhD and investing in my education.

A big acknowledgment goes to the SpiNNaker team for providing all the necessary support with the platforms and tools. Particularly I would like to thank Andrew Rowley, Luis Plana, Andrew Gait, Christian Brenninkmeijer and Donal Fellows. An important thank goes to my fellow PhD studens who helped me through this long journey always cheering me up when I needed it: Adam Perrett, Edward Jones, Sara Summerton and Mollie Ward. A special acknowledgment goes to Mark Kynigos for our very helpful coffee breaks during which we pushed each other to keep up writing.

Finally, I would like to thank my friends and family for all the patience and support 1000 miles away and for always being so available I felt they were here with me: Alessandro Zanotto, Marco Caporaso and Fulvia Barolo.

Chapter 1

Introduction

1.1 Motivation

Networks of neurons allow the brain to perform tasks such as learning, inference and motor control reliably and with limited energy consumption [HH11]. However, simulating such tasks is non-trivial, as long-range connectivity and sparse temporal signals, typical of biologically-representative Spiking Neural Networks (SNNs), make traditional communication mechanisms inefficient [FMCR19].

Conventional computer hardware is therefore not suited to perform such tasks, as the communication cost dominates performance, scaling nonlinearly with neural network size, slowing down simulations and increasing the energy consumption of the underlying simulator [vARS+18, IEPD17]. Alternatives have been proposed to address these challenges, including Neuromorphic engineering [TMC+18], an emerging field which aims at building machines by taking inspiration from the mammalian brain. Various platforms and technologies [CDLB+22] have been developed with the ultimate goal of performing real-time simulations of complex biologically-representative Spiking Neural Networks (SNNs). These aim at better understanding the biological mechanisms which regulate activities such as learning, memory and cognitive tasks on the one side and at finding alternative architectures in order to overcome the end of Moore's law [Moo06] on the other [SKP+22]. Neuromorphic systems approach SNN simulations from a different perspective, compared to conventional hardware [ILBH+11], by being power efficient and architecturally similar to neural networks. They furthermore provide great scalability capabilities, overcoming the communication bottleneck of conventional hardware. Although looking promising, these architectures still struggle to achieve the needed performance on more complex simulations. The reasons can often be found in sub-optimal exploitation of the hardware resources and in the lack of targeted programming models for these unique architectures [vARS$^+$18]. These machines are therefore far from their potential, as demonstrated by cases such as SpiNNaker, where no biologically-plausible network has been simulated employing the full one-million core machine to date.

This work aims to address these issues by researching programming paradigms based on parallelisation of neural processing on digital Neuromorphic hardware, in order to bridge the gap in communication between the neuroscientific community and computer architects. The target of this research is therefore to optimise neural network simulations, by performing a more efficient use of the available hardware resources, and highlight the weaknesses of such systems, in order to inform the design of the next generation of Neuromorphic hardware.

1.2 Research Questions

The main hypotheses constituting the foundation of this work are presented here in the form of research questions:

1. How can the process of mapping biologically-representative SNNs be optimised on Neuromorphic Hardware?

2. What are the challenges of implementing on-line learning algorithms in real time on Neuromorphic hardware?

3. How can the brain's sparse connectivity and activity be modelled efficiently on Neuromorphic hardware?

1.3 Contributions

The research presented in this manuscript addresses the questions formulated in Section 1.2. The contributions provided by this work are here summarised:

1. The Cortical Microcircuit network [PD12], commonly regarded as benchmark in the neuroscience field, was used as a vehicle to explore neural models and connectivity patterns, typical of biologically-representative SNNs. Detailed profiling and optimisations are here presented, together with the first

real-time Neuromorphic simulation of the SNN, constituting a 20 × speedup compared to previously published results. This contribution is addressed in detail in Chapter 3.

2. Challenges and limitations related to the implementation of on-line learning algorithms in real time on Neuromorphic hardware are explored, further improving the developed software framework to include multicompartmental neural modelling and rate-based densely-connected SNNs. This contribution is addressed in Chapter 4.

3. Brains' levels of connectivity, fan-in and neuron density, are explored, providing a new efficient SNN placement methodology for digital Neuromorphic hardware, which enables efficient simulations of biologically-representative sparsity levels. This approach achieved a reduction in required hardware resources and up to 9× neural operations throughput compared to previous methodologies. This contribution is addressed in detail in Chapter 5.

1.4 Publications

Most of the work presented in this thesis has been submitted for publication and is available in various forms:

- Journal Articles
  - Oliver Rhodes, **Luca Peres**, Andrew G. D. Rowley, Andrew Gait, Luis A. Plana, Christian Brenninkmeijer, and Steve B. Furber. Real-time cortical simulation on neuromorphic hardware. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 378, Dec 2019 - this article summarises the work presented in Chapter 3.
  - **Luca Peres** and Oliver Rhodes. Parallelization of Neural Processing on Neuromorphic Hardware. Frontiers in Neuroscience, 16, May 2022 - this article constitutes the work presented in Chapter 5.
- Posters

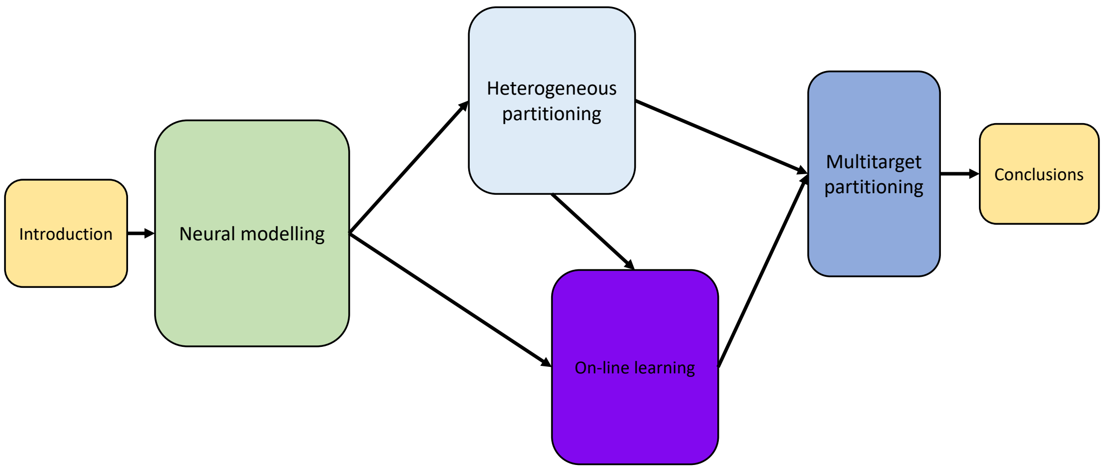


Figure 1.1: Thesis Structure. Each box corresponds to a chapter, while the arrows indicate the relations between chapters.

– **Luca Peres**, Oliver Rhodes (2020) "Real-time cortical simulation on SpiNNaker". figshare. Poster. `https://doi.org/10.6084/m9.figshare.17086667.v1` [PR20]

1.4.1 External Coverage

The published work described in Chapter 3 received coverage from the general press and was addressed in the 12th October 2019 issue of the New Scientist [Gen19] magazine. The same work was cited among the biggest achievements during the introductory keynote at the Human Brain Project (HBP) Summit in Athens in February 2020.

1.5 Thesis Structure

This thesis is composed of 6 chapters in total. Its structure is shown in Figure 1.1. Each box indicates a topic which maps to a specific chapter in this manuscript and the arrows indicate the relations between them.

This chapter describes the motivations and the research questions addressed in this work. Chapter 2 (Neural Modelling) provides an overview of the field and the necessary background to understand this work. Chapter 3 (Heterogeneous partitioning) describes a parallelisation approach to improve SNN simulations on Neuromorphic hardware. Chapter 4 (On-line learning) expands this approach to biologically-plausible learning strategies. Chapter 5 (Multi-target partitioning) combines the previous developments and presents an evolved parallelisation method, achieving optimal performance and including additional edge cases. Chapter 6 (Conclusions) summarises the research and discusses potential future work.

1.5.1 Chapter 2

Chapter 2 serves as a background on Spiking Neural Networks and neural modelling, providing the reader with the information necessary to understand the work. The initial focus is on the evolution of neural networks over the years, with emphasis on Spiking Neural Networks, spiking neuron models and plasticity rules. An overview of simulation platforms is also provided, showing the available options on the market. Neuromorphic hardware is addressed in more detail and chosen as target hardware for this research. Finally, a detailed descritpion of the SpiNNaker Neuromorphic platform is provided as the main platform used for this work.

1.5.2 Chapter 3

Chapter 3 addresses the first research question, and presents the development and application of a parallelisation strategy, namely the Heterogeneous Programming model, on digital Neuromorphic hardware. This programming model targets real-time simulations of biologically-representative SNNs, and achieved the first real-time simulation [RPR+19] of the Cortical Microcircuit network [PD12], a well-known benchmark in the field of computational neuroscience. To date, this represents the only complete Neuromorphic simulation of the network.

1.5.3 Chapter 4

Chapter 4 addresses the second research question, and expands the approach presented in Chapter 3 by applying it to a different context, targeting on-line learning [LDBK20] challenges. Multicompartmental neuron models from the literature are presented and simulated on SpiNNaker and the novel parallelisation strategy is adapted to a rate-based context. Performance of this new implementation is then tested in real-time supervised learning problems.

1.5.4 Chapter 5

Chapter 5 addresses the third research question and builds on the knowledge acquired from Chapters 3 and 4 to build an optimised framework which can adapt to support simulations of various ranges of SNNs: from extremely sparse connectivity patterns to denser networks, including support for biological learning mechanisms defined by plasticity rules. This method shows unprecedented throughput of neural operations and a reduction in the required hardware resources for a given simulation [PR22].

1.5.5 Chapter 6

Chapter 6 presents the conclusions of the thesis, summarising the research outcomes and providing insights for potential future work.

1.6 Summary

This thesis focuses on the development of parallelisation techniques to achieve real-time simulations of biologically-representative Spiking Neural Networks on Neuromorphic hardware. This is achieved by performing a more efficient use of the available hardware resources. First, the context is provided through a description of the relevant fields and the available tools. An implementation of a parallelisation technique is then provided for the SpiNNaker Neuromorphic platform, together with its extensions to different contexts and forms. Finally the results are presented together with considerations about future work.

Chapter 2

Neural Modelling and Dedicated Hardware

2.1 Introduction

The human brain is a complex system able to perform learning tasks efficiently with extremely low energy cost [LC20]. However, replicating such behaviour on conventional computer hardware is challenging, due to a lack of complete understanding of brain functioning and to inadequate simulation architectures. Both the neuroscientific and computer science communities are investing considerable effort in achieving full brain simulations, on the one side by providing biologically-representative models and on the other side by building dedicated hardware to simulate such models efficiently. This chapter presents an introduction to biologically-inspired neural network simulation and to the available platforms on the market to perform this task. This constitutes the background necessary to introduce the research presented in this thesis. Section 2.2 presents a description of neural networks and their structure, with focus on biologically-inspired networks. A description of spiking neuron models and biological learning mechanisms - such as synaptic plasticity - is provided, concluding with an introduction to neural compartments and multicompartment neuron models. Section 2.3 describes the tools and techniques used to simulate neural networks, both from hardware and software perspectives, providing evaluation of the advantages and tradeoffs of every solution. Section 2.4 presents a more detailed description of Neuromorphic platforms as dedicated hardware simulators which mimic the structure of the brain. Section 2.5 contains a detailed description of the

SpiNNaker Neuromorphic system, the chosen platform for this research, together with hardware and software features, and how neurons and neural networks are simulated on it.

2.2 Neurons and Neural Networks

Neural networks are circuits of neurons, which take inspiration from biology. The term takes its roots in the attempt to find mathematical representations of information processing in biological systems [Bis06]. Neural networks are gaining popularity in the field of Computer Science, thanks to the reduction in computational power consumption and advances in parallel computing paradigms, together with a larger presence of high quality databases and training sets [GBC16]. The most popular type of neural network is called Artificial Neural Networks (ANNs), which in general are circuits of artificial neurons. ANNs are a broad class employed in several Machine Learning (ML) tasks and commonly used by commercial applications in fields such as pattern recognition and inference. The basic building blocks of neural networks are neurons, which can be modeled following different abstraction levels from the biological neurons, according to the type of application. Multiple generations of neural networks have been proposed to date and they change according to the implemented type of neuron and connectivity. Figure 2.1 shows the evolution of neural networks through generations. The first row represents the neuron models employed by each generation, the second row shows how neurons interact with each other and the third row shows networks of neurons.

The first generation (shown in the first column in Figure 2.1), evolving from the first developed artificial neuron model (the McCulloch-Pitts neuron [MP43]), employed a computational model called the Perceptron [Ros58]. This sums together the input weights (w_1, w_2 and w_3 in Figure 2.1 top left) and then performs a threshold step function on the result of the sum. The output assumes binary values only. Multilayer combinations of these units generate networks called Multilayer Perceptrons (MLP) [Maa97].

The second generation (middle column in Figure 2.1) makes use of artificial neurons employing continuous nonlinear activation functions (such as the sigmoid function [Mey01]) to transform the inputs and generate real-valued outputs. This, from a biological interpretation of neural networks, corresponds to a

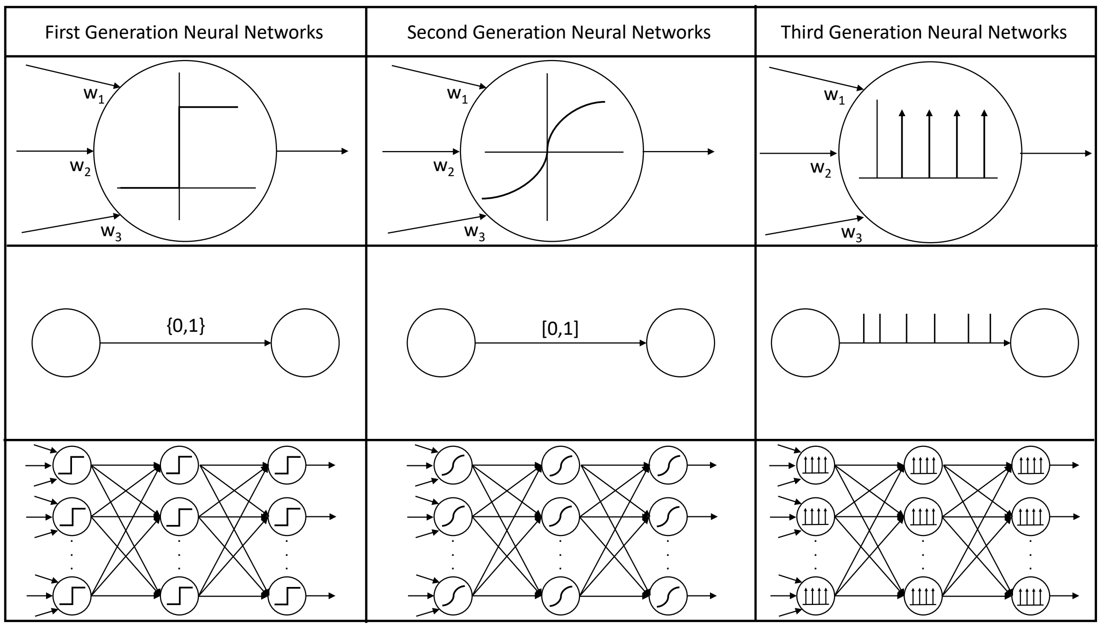

First Generation Neural Networks
Second Generation Neural Networks
Third Generation Neural Networks
w1
w2
w3
{0,1}
[0,1]

Figure 2.1: Neural networks generations. Each column shows a generation of neural networks, including Multilayer Perceptrons (left column), ANNs (central columns) and SNNs (right column). The first row contains the type of neuron employed by each generation, the second row the communication model and the third row the network structure.

representation of the current firing rate of a biological neuron [Maa97]. The activation function is applied, similarly to the first generation of neural networks, to a weighted versionsum of the inputs. Second generation neural networks support learning algorithms based on gradient descent, such as Error Backpropagation [RHW86, Maa97]. This generation is currently the most popular in the Artificial Intelligence (AI) field, being the basis of Deep Learning algorithms [GBC16].

Finally, the third generation (third column in Figure 2.1) is referred to as Spiking Neural Networks (SNN). This type of network takes inspiration from real neurons, by using the exact timing of spikes to encode information. This means that neurons do not output mean firing rates, as for the previous generations, but emit spikes at precise times and these times are used to encode information. Spikes (or action potentials) are commonly modelled as events at a precise time, therefore their mathematical representation is the Dirac delta function [Mey01]. SNNs present clear advantages over previous generations, including low power consumption and the capability of processing real-time temporal data. However the development of efficient learning algorithms for them is still at an early stage, therefore they have not yet fully replaced the previous generations of neural networks [LDC+20].

The simplest model of neural network is called feedforward and it is commonly used to approximate a chosen function. The goal for the network is to learn the values of some parameters to find the best approximation of the desired function. This type of network has the information flowing in one direction only (there is no feedback) and the neurons are organised in layers. Therefore at a higher abstraction level, each layer performs a task, and the whole network an algorithm. The most common representation is through a DAG (Directed Acyclic Graph), the first layer is called input layer, the last is the output layer and all the middle layers are called hidden layers [GBC16]. The introduction of feedback connections defines a more complex class of neural network called Recurrent Neural Networks (RNNs), which are employed for more complex tasks involving sequential data processing, such as language translation, natural language processing (NLP) and speech recognition. This type of neural network employs feedback connections to create a form of "memory" of the previous inputs which persists in the network's internal state, influencing the outputs. Additional details about RNNs are left as reference [Gra12].

This work focuses on algorithms and platforms related to SNNs simulations, aiming to provide design methodologies to inform the design of future dedicated hardware and algorithms. The following sections contain descriptions of SNNs and hardware platforms employed to simulate them.

2.2.1 Point Neuron Models

Biological neurons have been modeled through the years at different levels of abstraction, from simple Integrate-and-Fire models [GK02] to the ion channel modeling proposed by Hodgkin and Huxley [HH52]. The common aspect is the spiking nature of these models. Rate-based models, indeed, commonly encode the information through the firing rate of the neurons; spiking models, on the other hand, exploit the timings of the spikes to encode spatial and temporal information. Spikes are typically represented as Dirac delta functions, and the set of emitted spikes is represented by a train of delta functions. Networks of spiking neurons therefore communicate through this pulse-based mechanism, each connection is represented through a strength, commonly called weight and the inputs come in the form of synaptic currents, where the synapses allow connection between neurons. When two neurons are connected, the source neuron is called presynaptic, while the destination is denominated postsynaptic. A neuron takes its inputs from other presynaptic neurons on tree-like structures called dendrites and the junction between a presynaptic and a postsynaptic neuron is called synapse. The synapses allow the transmission of neurotransmitters between the two neurons. The core part of a neuron is called soma, or cell body, and it is the nucleus of the neuron. In the soma all the input currents are collected. The soma connects to the axon (a projection used to transmit information to other neurons) through a cone-shaped section called axon hillock. This is the generation site for action potentials, which are then propagated through the axon to reach the dendrites of the target neurons (the schematic of spiking neurons is shown in Figure 2.1 right).

Each spike contributes to postsynaptic neurons according to the weight of the connection, and connections can be either excitatory, when they contribute to a postsynaptic spike, or inhibitory, when they are opposed to postsynaptic spiking activity. Learning in SNNs happens through a mechanism called synaptic plasticity, which leads to weight modifications (more details about synaptic plasticity are presented in Section 2.2.2) under specific conditions.

The most basic representation of spiking neurons is given by the Leaky Integrate-and-Fire (LIF) model [GK02]. This model represents the neurons as simple electrical RC circuits, having a resistor (R) in parallel with a capacitor (C), driven by a current ($I_{syn}(t)$). The membrane voltage is measured as output voltage and evolves with time according to Equation 2.1, where R is the cell membrane resistance and τ_m represents the time constant (and is equal to the product between R and C).

$$\tau_m \frac{dV}{dt} = -(V - V_{rest}) + RI_{syn}(t) \tag{2.1}$$

$$\tau_{syn} \frac{dI_{syn}}{dt} = -I_{syn}(t) + \sum_j \delta(t - t_j) \tag{2.2}$$

V_{rest} is the resting potential and is the value the membrane voltage converges to when no synaptic input current (I_{syn}) is injected. The model acts in a way that when the membrane voltage reaches a threshold value, it emits a spike and then V is set to the resting potential level for an interval of time called the refractory period, during which the neuron cannot emit any spike. The current injected into the neuron through its synapses is modeled by Equation 2.2. This is determined by the incoming spikes (represented by the sum of δ functions, which include the weight contribution for each activated synapse j) and it is exponentially-shaped.

Another popular point neuron representation is given by the Izhikevich model [Izh03], whose creator claims it to be as biologically-plausible as the Hodgkin-Huxley [HH52] model, but with a complexity comparable to the LIF model [Izh03, Izh04]. The Izhikevich model dynamics are modeled by the two-dimensional system of equations shown in Equation 2.3.

$$\begin{aligned} \frac{dV}{dt} &= 0.04V^2 + 5V + 140 - U + I(t), \\ \frac{dU}{dt} &= a(bV - U) \end{aligned} \tag{2.3}$$

V and U are dimensionless variables representing membrane potential and a membrane recovery variable of the neurons respectively. $I(t)$ the synaptic current. a and b are dimensionless parameters used to tune the model dynamics and can be configured to modify the neuron spiking behaviour according to simulation conditions [Izh03].

Despite their simplicity compared to biological neurons, point neuron models represent a powerful way to model biologically-plausible SNNs and are nowadays employed in large networks used as benchmarks by the neuroscientific community, such as the Cortical Microcircuit model [PD12], or the more complex multi-area model [SBS+18], or even for cerebellar representations [BMC+21, CMM+19].

2.2.2 Synaptic Plasticity

The process that governs synaptic modification is termed synaptic plasticity and it is believed to be the main learning mechanism in biology, as well as the substrate of memory. Synaptic plasticity is also the main learning mechanism employed in SNNs. Most of the plasticity models originate from Hebb's description of synaptic changes [Heb49], which states that if a neuron's firing activity results in modifications of a connected postsynaptic neuron's firing activity, then some metabolic change happens in one or both the cells which causes the synaptic efficacy to increase. Therefore, correlations in pre- and postsynaptic firing activity drive synaptic modifications. Hebb's theory however does not take into account the fine temporal structure between pre- and postsynaptic spikes, stressing more on causation, rather than correlation. The discovery of this aspect led to the formulation of modern plasticity rules, in the form of Spike-Timing-Dependent Plasticity (STDP) [MGS12]. Differently from what Hebb thought, according to the temporal dynamics, both increases and decreases can be observed in synapses. These two phenomena are termed Long-Term Potentiation (LTP) and Long-Term Depression (LTD) respectively. An important feature of these rules is that only variables locally available at the synapses can be involved in the synaptic weight update. This means that effects caused by pre- and postsynaptic spikes, alone or in conjunction, and membrane voltage changes are the only parameters that can be taken into account when modeling synaptic plasticity [MDG08]. Several STDP rules have been formulated, which vary according to the neuron models and system type [AN00, BP01]. The most common STDP rule, as well as the simplest, is the pair-based formulation [MDG08]. According to this formulation, when a presynaptic neuron fires and, shortly after, a postsynaptic action potential is generated, then the involved synapse is potentiated, therefore the weight increases. If the opposite situation happens, i.e. a postsynaptic action potential is followed shortly after by a presynaptic spike, the synapse is depressed [BP98]. Therefore, a change in weight, according to pair-based STDP rules, depends on the temporal difference between pairs of pre- and postsynaptic spikes.

$$\begin{aligned} \Delta w^{+} &= F_{+}(w)e^{-\frac{|\Delta t|}{\tau_{+}}} \qquad if\ \Delta t > 0 \\ \Delta w^{-} &= -F_{-}(w)e^{-\frac{|\Delta t|}{\tau_{-}}} \quad if\ \Delta t \leq 0 \end{aligned} \tag{2.4}$$

Equation 2.4 shows the amount of weight modification according to the time difference between a single pair of spikes $\Delta t = t_j - t_i$. F_+ and F_- indicate functions expressing the dependence of the update on the current weight, and τ_+ and τ_- are time constants controlling the time windows over which synaptic modification can happen. A common way to simulate this rule is through a history trace of the spiking activity. This trace can be modeled through exponential decay (this is equivalent to a low-pass filter of the input spike train, therefore the function increases every time a spike arrives and then slowly decreases over time).

$$\frac{ds_i}{dt} = -\frac{s_i}{\tau_s} + \sum_{t_i} \delta(t - t_i) \tag{2.5}$$

The history trace evolution over time is shown in Equation 2.5, where s_i represents the trace for the synapse i, τ_s is the time constant for the synapse and the sum indicates the spike train. By keeping both a pre- and postsynaptic history trace, it is possible to separate the two contributions. Therefore the weight update can be modeled as the sum of two main contributions: a decrease induced by the arrival of a presynaptic spike, which is proportional to the momentary value of the postsynaptic trace, and an increase caused by the generation of a postsynaptic action potential, which is proportional to the presynaptic trace [MDG08].

$$\frac{dw_{ij}}{dt} = -F_-(w_{ij})s_i(t)\delta(t-t_j) + F_+(w_{ij})s_j(t)\delta(t-t_i) \tag{2.6}$$

Equation 2.6 shows the weight update relation for a synapse between a presynaptic neuron j and a postsynaptic neuron i, through an all-to-all pairing scheme.

The choice of the function $F(w)$ indicates the update dependence on the weight [BP98]. The case in which there is no dependence is called additive, and it is the simplest representation. In this formulation, the function $F(w)$ is replaced by a constant, which controls the maximum weight update [MDG08]. Other common weight dependence rules are multiplicative, where the weight update is proportional to the weight itself [RLS01], and power law, where the update is proportional to a power of the weight [MAD07]. Extensions to the pair-based rule have been provided to achieve more accurate results, however these are beyond the scope of this thesis and therefore are available as references [MDG08, PG06, Izh07, AE05].

2.2.3 Multicompartment Neuron Models

Section 2.2.1 addressed Spiking neurons in the form of point models. Point neuron models however, represent a simplification of real neurons, by assuming that the membrane potential is constant across the entire cell membrane of the neuron. This theory neglects the nature of dendrites as complex structures, simplifying them to wires carrying signals to the soma of the neuron. This limitation becomes evident in Pyramidal neurons, a type of neuron largely present in the cerebral cortex of most mammals, birds, fish and reptiles [Spr08]. Pyramidal neurons present dendritic trees which can be divided into two main categories: basal and apical dendrites, descending from the base and the apex of the soma respectively [Spr08]. The structure of these dendrites varies, and inputs can have different effects according to where they are received, which means that different dendritic domains have different integration properties for the inputs [Spr08]. This property does not only apply to different domains, but is also position-dependent: the farther the synapses are from the soma, the lower is their influence on the action potential generation [HM03]. Neurons commonly take advantage of this property to perform complex integrative processing [Spr08]. Multi-compartmental models try to take into account these phenomena by modeling neurons as collections of smaller independent compartments within which the membrane voltage is constant. Compartments are then connected together by passive conductance [HM03]. Different models present different numbers of compartments, ranging from the simplest, which condense the soma, axon and basal dendrites into a single somatic compartment and model the apical dendrites as a separate compartment [HM03], up to 100s of different dendritic compartments [SMKS00, SS01].

Multi-compartment modeling represents a tradeoff between efficiency and accuracy, since these types of models are more complex to simulate and therefore require more computational power, but are closer to biology than point models. A big benefit of multicompartment modeling, however, is given by the chance to build biologically-plausible learning rules, overcoming the limitations posed by point neuron models, therefore reducing the gap between Deep learning applications and biology [GLR17a].

2.3 Neural Network Simulators

The models and rules described in the previous sections need adequate tools to be simulated, and simulation is the key for further understanding and model development. Simulating neural networks is a complex task, which requires handling high parallelism and reliability. Scalability of systems is another important feature, since complex tasks require large network representations.

Different approaches are nowadays available for SNN simulation and they fall into different categories, presenting different advantages and drawbacks. The two main categories are: Hardware simulators and Software simulators. Hardware simulators are commonly ASICs (Application Specific Integrated Circuits) specifically designed with the aim of simulating neural networks. These tend to be very efficient, but often come with low flexibility, since the number of models or rules that can be implemented might be constrained by the hardware implementation of the simulator. Software simulators are tools designed to run on HPC (High Performance Computing) platforms. These typically come with a higher flexibility, however the performance is limited by the underlying hardware and its communication mechanisms, which can represent a major bottleneck.

2.3.1 Software Simulators

Software simulators are software tools commonly designed to run on CPU-based systems, which come with high flexibility, but performance constrained by the hardware they are executed on. Neurons are described as sets of Ordinary Differential Equations (ODEs) implementing the mathematical properties of the chosen model. Because of their flexibility it is generally straightforward to add new neuron models, and it is possible to simulate networks of arbitrary sizes. The implemented algorithms can follow either a clock-driven approach, or an event-driven approach or a hybrid one. A clock-driven approach consists of a discretisation of the simulation time into equally long intervals, called timesteps. These events are used to mark the advance of time and represent the points during the simulation where neural states are updated. This allows continuous time models to be discretised, and provided the timestep resolution is high enough, allows modelling of neuron state updates via exponential integration, calculating the dynamics timestep by timestep. The most commonly used resolutions, in accordance with biological times, are therefore 1 ms or 0.1 ms. A major drawback of these simulators, however, is that all the spikes arriving within a timestep are considered simultaneous in that timestep, reducing the maximum precision for the state update to the timestep resolution. The event-driven approach, on the other hand, performs the state update only when a spike is received. This property allows the temporal nature of SNNs to be precisely mimicked, by implementing synaptic and status changes when signals are received, instead of discretising them over a longer timestep. However, biologically-plausible SNNs present fan-ins so large that it becomes impractical to use this approach in many simulators [MMG+05]. The hybrid solution, which is the one preferred by most of the established simulators, employs a time-driven simulation for the neuron state update, and an event-driven approach for the synapses. Examples of the use of this approach are given by NEST [GD07], BRIAN [GB08] and NEURON [CH06].

An important requirement for software simulators is to perform real-time simulations. A software simulator is defined real-time when the simulation time matches the wall-clock time, therefore the time taken to perform the simulation corresponds to the biological time for the network. For biologically-representative complex SNNs this condition might become hard to meet, due to the high number of neurons and connections and the tight timing constraints. For this reason, some platforms perform soft real-time simulations, in which some timesteps are allowed to overrun, and then they recover in future timer periods where the computational load is reduced, yielding real-time performance on average. However this violates hard real-time requirements, which mandate that every timestep completes within the corresponding amount of wall-clock time (i.e. each 0.1 ms of biological time is completed in 0.1 ms). Where real-time performance is not possible, an operation called slowdown can be performed. This results in increasing the length of each timestep by a multiplicative factor (or slowdown factor). Therefore each timestep will have a longer duration compared to biological real-time, giving the resources enough time to perform all the necessary updates.

2.3.1.1 MPI and OpenMP

Software simulation is commonly performed on CPU-based supercomputers, as detailed in section 2.3.1. Supercomputers and HPC platforms typically consist of a large number of compute nodes connected by fast interconnect. Each of these nodes can contain a number of CPUs containing multiple cores. The memory system can be distributed or shared. In order to offer coding support to programmers, two main standards have been developed, which are exploited by many software simulators. These two standards are called OpenMP and MPI.

OpenMP is an API which provides directives for parallel programming in shared memory systems. Its aim is to provide a model for parallel programming which is portable across shared memory architectures. The standard allows multiple threads of execution to perform tasks defined implicitly or explicitly by OpenMP directives and provides a relaxed memory consistency mechanism, where the memory view of a thread can be modified without reflecting to memory until a flush operation is performed [Boa08].

MPI is a paradigm commonly used for multiprocess programming on distributed memory machines [For09]. This standard is based on a message-passing communication system. Similarly to OpenMP, the main idea is to guarantee portability, by defining a standard. This API allows communication mechanisms between different processes to be improved and provides abstraction for developers.

The two standards can be combined together in brain-scale network simulations [IEPD17, JIH$^+$18] to improve the exploitation of computer capabilities, employing both thread-based parallelism and process-based parallelism.

2.3.1.2 NEST

NEST (NEural Simulation Tool) [GD07] is an open source software tool, developed and maintained by the NEST initiative under the GNU General Public License. It has been designed to simulate large-scale networks of point neuron models or neurons with a small number of compartments. Simulations are controlled through a built-in scripting language or a Python module called PyNEST [JIH$^+$18]. Networks are modeled as sets of nodes and connections, where nodes can be neurons, devices or sub-networks. Connections are defined by sending node, receiving node, weight and delay, and communications are event-based. Models of synaptic plasticity can be implemented in the form of STDP, and high level functions to create connectivity schemes are provided. In order to optimise the use of resources, NEST supports hybrid parallelisation, therefore MPI is employed for inter-node communication, where each node is assigned to one MPI process and events are communicated between processes by collective MPI functions, and inside each MPI process multi-threading is supported through OpenMP, making better use of the available memory.

2.3.2 Hardware Simulators

Hardware simulators are dedicated circuits and platforms designed with the purpose of simulating neural networks. They commonly come with high performance, but limited flexibility compared to their software counterparts, as the hardware implementation of the neuron models limits the variety that can be simulated. The hardware simulators can be analog, digital or mixed-signal. The time scale can be continuous, discrete or abstract, and the simulators can be real-time, accelerated or non real-time [Dav12]. There are several classes of hardware simulator, which span from FPGAs, Graphic Processing Unit (GPU), hardware accelerators, to Neuromorphic platforms.

2.3.2.1 FPGAs

FPGAs (Field Programmable Gate Arrays) are an alternative to ASICs for hardware simulations. These systems allow the implementation of the desired logic function by programming them. However FPGAs are a viable choice only for small networks sizes, as the fabric process is much larger compared to ASICs [KR07] and the number of neurons that can be implemented is limited [RHTF03]. Due to their clock speeds, which are commonly around hundreds of MHz, it is possible to perform sub-realtime simulations of point neuron models and it is possible to connect multiple FPGAs together [MFM+12]. Scalability issues typical of these platforms [MMG+07] seem to have been overcome with the latest developments, however programmability still requires expertise in hardware design, reducing the accessibility of these platforms [NCA+20].

2.3.2.2 GPUs

GPUs (Graphics Processing Units) are nowadays well established neural network simulators. Through their high parallelism they provide a good framework for this type of application. The GeNN library [YTN16, KN21] allows code to be generated efficiently to accelerate SNN simulations on NVIDIA GPUs and demonstrated remarkable performance in simulating biologically-representative SNNs [KN18, KKN21]. Despite the rapid advances in technology, which continuously provide new more powerful platforms addressing issues such as sparsity (e.g. the Ampere architecture [NVI]), scalability still represents a limitation when simulating SNNs on GPUs. Multi-GPU simulations are problematic, as connections between separate GPUs still represent a bottleneck, being constrained by a single communication channel which therefore limits the performance.

2.3.2.3 Hardware Accelerators

The final subcategory of hardware simulators is identified by hardware accelerators. These are platforms commercially available from industry which aim to facilitate DNN and CNN algorithms [CBM+20]. Companies such as Intel (through the Xeon series) and Apple (though the Bionic series of CPUs) are investing in building new architectures commonly targetting inference and pattern recognition, and specific libraries are being developed to accelerate AI applications on these platforms. A detailed analysis of these platforms is, however, beyond the scope of this thesis, due to the heterogeneity of these technologies. Additional details are available as reference [CBM+20].

2.4 Neuromorphic Hardware

Neuromorphic Engineering is a field pioneered by Carver Mead in the 1980s, which approaches neural network simulations from a different perspective [Mea90, Mea89]. The original idea was, instead of employing standard computing systems, to implement a platform which was architecturally similar to the brain structure. Mead noticed that MOS transistors show very similar behaviours compared to ion channels, therefore he suggested to implement hardware simulators through the use of low-power sub-threshold analog circuits, instead of using digital computing systems, showing a reduction in power consumption up to 4 orders of magnitude [Mea90]. This was the first step towards a new field which is nowadays gaining more popularity, and several platforms, both in the digital and analog domains, have been developed [TMC+18, Fur16]. All these platforms share the basic idea of non conventional computing (they are often referred to as non Von Neumann, since their design deviates from the standard centralised approach commonly used in computing platforms), where, instead of a single central and powerful computational unit, a distributed approach is preferred, having small units acting like the neurons in the brain. The memory hierarchy is revised as well to be distributed and as close as possible to the computational units, to mimic the behaviour of synapses. The connectivity between computational nodes is enhanced, such that the units are allowed to form multiple connections as happens in the brain.

Analog and digital platforms present quite different architectures and approaches, where the first are generally more performance oriented, at the expense of lower flexibility, while the second allow more customisation and freedom in neural modelling.

Neuromorphic Engineering has two main objectives: first to better understand the computational properties of neural systems, by providing an electronic implementation of them, second to provide new alternatives to standard computing paradigms, to compensate for the end of Moore's law [Moo06].

2.4.1 Analog Neuromorphic Systems

Analog Neuromorphic platforms are the closest to Mead's idea. Neurons and synapses are implemented by physical circuits. This provides a very high efficiency in terms of simulation times, often guaranteeing sub-realtime performance, however this comes at the expense of flexibility. The physical implementation of the neural circuitry sets a limit on the number of neurons and on the variety of models that can be implemented. Well-known analog Neuromorphic platforms include BrainScaleS [SKMM17] and ROLLS [QMC+15].

2.4.1.1 BrainScaleS

BrainScaleS was developed at the University of Heidelberg and is based on the direct implementation of the neuron model equations in electronic circuits. The electrical parameters that can be read out represent the model variables. The system can run 10^4 times faster than biological real-time and the implemented neuron model is the Adaptive Exponential Integrate-and-Fire Model [BG05]. The neuron and all its synapses are implemented as a continuous-time analog circuit. Simulations including more than a few hundreds of neurons require a multichip implementation, which is achieved through wafer-scale integration [SFM08], a postprocessing wafer technique which allows chips manufactured on the same wafer to be interconnected without separating them. 128k synapses and 512 membrane circuits are grouped into ANNCOREs (Analog Neural Network CORE) and they can form neurons with up to 16k synapses each [SFM08, SKMM17]. Synaptic plasticity is available through a digital general purpose processor, implementing programmable STDP rules and connected to the synapse memory array [FSG+17, FFS+13].

At the moment of writing, a second generation of the BrainScaleS platform is under development, called BrainScaleS-2. This aims to provide a flexible tool for the machine learning and computational neuroscience communities, targeting scalability and bio-inspired learning rules [PBC+22].

2.4.1.2 ROLLS

The Reconfigurable On-Line Learning Spiking Neuromorphic processor (ROLLS) is composed of a configurable array of synapse and neuron circuits producing biologically realistic response properties and behaviours [QMC+15]. Neurons are represented as silicon circuits as a row in an array of 256 elements and they physically implement the Adaptive Exponential Integrate-and-Fire model. Synapse circuits are two blocks of the array of 256 $\times$ 256, the first modeling long-term plasticity mechanisms and the second short-term plasticity. Through a synapse de-multiplexer it is possible to choose how many rows of plastic synapses to connect to the neurons [QMC$^+$15, CSBI14]. Communication happens through the Address Event Representation (AER) packet format [Mea89] and the system supports STDP. SNNs are defined through custom software called PyNCS [SNSI14].

2.4.2 Digital Neuromorphic Systems

Digital Neuromorphic platforms commonly make use of components available on the market such as low-power processors, combining these with custom digital circuits to build a different type of architecture with enhanced connectivity and optimised for neural network simulations. Their structure guarantees a much higher programmability compared to the analog platforms, which results in more flexibility in the number and types of neurons that can be simulated. However digital circuits cannot often reach the performance of the analog implementation.

Some examples of digital Neuromorphic platforms are Intel Loihi [DSL$^+$18], IBM TrueNorth [ASC$^+$15] and SpiNNaker [FLP$^+$13]. The first two platforms are briefly presented below, SpiNNaker is addressed in the next section in more detail, as it is the main platform used in this work.

2.4.2.1 TrueNorth

TrueNorth is a digital Neuromorphic chip developed by IBM, with the aim of delivering a very dense, energy-efficient platform capable of supporting a range of cognitive applications [ASC$^+$15, MAAI$^+$14]. This plaftorm makes use of custom digital neurosynaptic cores, each implementing 256 neurons receiving 256 synapses each. A TrueNorth chip contains 4096 neurosynaptic cores which operate asynchronously. A 256 $\times$ 256 cross-bar connects source and destination neurons, implementing individually configurable point-to-point routes. 16 chips are connected to form larger boards, reaching a total of 16 million neurons and 4 billion synapses. Boards can be interconnected to provide higher scalability [Fur16].

2.4.2.2 Loihi

Loihi is Intel's contribution to the Neuromorphic field. It is a fully digital chip containing 128 digital Neuromorphic processors and 3 embedded x86 cores organised in a NoC, implemented as a two-dimensional mesh which allows asynchronous communication in the form of packetised messages [DSL+18]. The Neuromorphic cores implement the neural state update and handle the synapses in an event-driven manner, computing all the state updates in-memory. The general purpose cores, on the other hand, are used as service cores, loading the SNN configurations on the Neuromorphic cores, pre- or post-processing the input and output data and supervising the packet routing [LWC+18b]. Each Neuromorphic core can implement up to 1024 spiking neural units, or compartments, which are grouped into sets of trees building neurons. The state update is time-multiplexed and pipelined. Loihi supports synaptic plasticity in the form of STDP and the equations are stored as microcode in a memory block local to the Neuromorphic cores as part of a learning block [DSL+18, LWC+18b, LWC+18a].

SNNs are described in Python through a LoihiAPI library and a dedicated software toolchain maps the network onto the Loihi system [LWC+18b, LWC+18a]. To provide scalability Loihi comes in different configurations, ranging from a 2-chip USB device (called Kapoho Bay), to a system composed of 24 32-chip boards (for a total of 768 chips), called Pohoiki Beach [FOF+20].

A Python software framework called Lava [Int21] has been released to provide support to neuro-inspired applications and a second generation of the Loihi Neuromorphic platform is under development at the moment of writing [OFR+21].

2.5 The SpiNNaker System

This section provides details on the SpiNNaker Neuromorphic system [FGTP14], which is the main platform chosen for the work performed for this thesis. The system is described both from hardware and software perspectives. SpiNNaker is an acronym for Spiking Neural Network architecture [FLP+13, FGTP14]. It has been developed at the University of Manchester in the School of Computer Science by Prof. Steve Furber and his group. The system is a many-core GALS (Globally Asynchronous Locally Synchronous) system, composed of ≈ 1 million general purpose ARM cores and its aim is to perform real-time simulations of biologically-inspired SNNs.

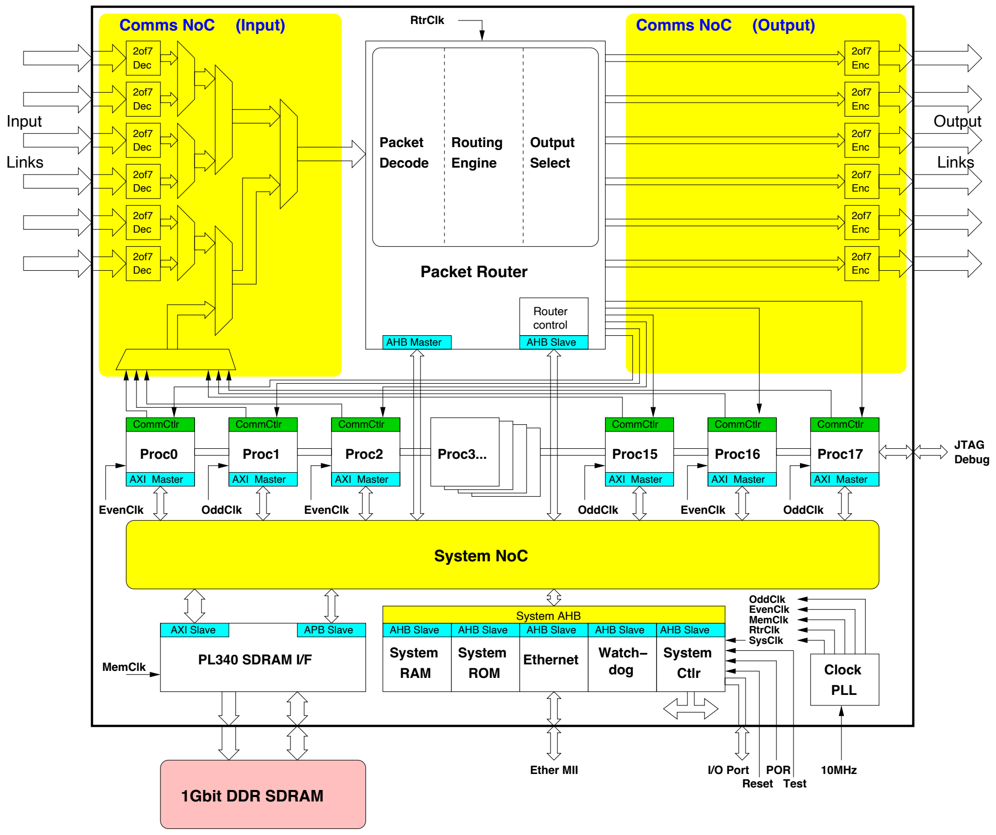


Figure 2.2: Block diagram of a SpiNNaker Chip

2.5.1 Hardware Overview

From a hardware perspective, the main building block of the SpiNNaker system is the SpiNNaker chip. Each chip contains 18 ARM968 cores, 128 MB of shared SDRAM memory, and a custom router allowing direct communication with 6 neighbouring chips. Among the 18 cores, 16 are assigned as Application processors and are used for simulation purposes, 1 is the Monitor Processor and has the role of supervising the other cores' behaviour and 1 is for fault tolerance purposes. A schematic of a SpiNNaker chip is shown in Figure 2.2, while Figure 2.3 shows the layout. SpiNNaker chips are arranged on boards, which can be accessed externally through an Ethernet interface.

2.5.1.1 The ARM968 Processor

The computational units employed by the SpiNNaker system are general purpose ARM968 processors. They feature the ARM9TDMI architecture, supporting the instruction set of the ARMv5TE architecture, an AHB bus interface and the

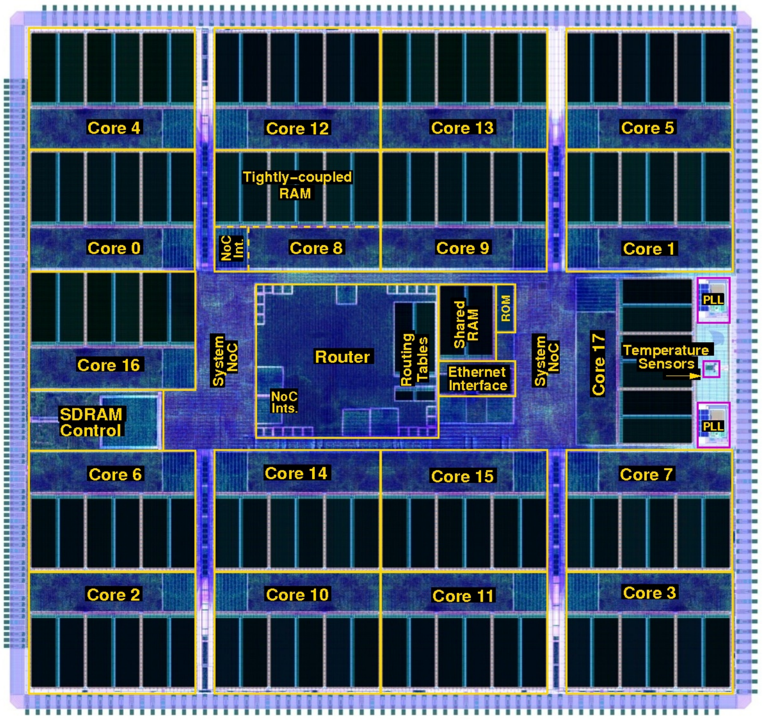


Figure 2.3: Layout of a SpiNNaker Chip

Thumb instruction set [ARM06]. The cores run at 200 MHz clock frequency and feature two local memories: a 32 KB ITCM (Instruction Tightly Coupled Memory) and a 64 KB DTCM (Data Tightly Coupled Memory). The first is used to store the instructions to be executed during a simulation, and the second for the data. Each core has direct access to its private memories and can access the shared SDRAM memory either via a core-specific DMA controller, or through bridge access. Each processor node has access to 2 independent counters, which can be used to trigger interrupts for real-time dynamics.

2.5.1.2 The Shared Memory

Each chip is connected to a 128 MB shared SDRAM. The access to it is through a single channel and the cores are distributed in a tree-based structure for getting access. A single core has access to the memory at a time, therefore when multiple requests are issued simultaneously the memory channel is contended. This results in different access times for different cores. The access tree consists of a binary tree having an arbiter at each junction and cores placed at the leaves. For every junction, the first core requesting a DMA transfer has the bus granted by the arbiter, the other cores get queued. This means that branches with multiple cores trying to access memory will experience longer access times [PPG+13]. In addition to the off-chip SDRAM memory, the SpiNNaker chips have an integrated 32 KB System RAM shared between the cores and a Boot ROM used during bootstrap.

2.5.1.3 The Routing System

The routing infrastructure is a unique feature of the SpiNNaker system. Each router allows each chip to connect directly with 6 neighbouring chips, building a hexagonal mesh. The whole SpiNNaker system is then represented as a toroid, by wrapping the extremities of the mesh, achieving full connectivity. Each router has an input tree, which filters input communications and internal links, and forwards data to 6 independent output links, as well as the 18 internal cores, as shown in Figure 2.2 [PFT+07, MLP+15, WFG09, WF10].

There are 4 different types of packet that the routing system is able to handle:

- **Point-to-Point (P2P)** packets having a single source processor and a single destination processor;
- **Multicast (MC)** packets having a single source and multiple destinations;
- **Nearest Neighbour (NN)** packets used to initialise the system and perform run-time flood-fill and debug functions;
- **Fixed Route (FR)** Packets having a predefined routing and as a target the nearest chip connected to the board Ethernet interface;

According to the type of packet arriving at a router, the way it is processed changes. Fixed Route packets have a predefined route which is defined inside a register accessible to the router. Incoming Nearest Neighbour packets are sent to the Monitor core for processing, while outgoing packets are sent either on a specific link or on all the links and will be received by the neighbouring chips.
Multicast packets are routed according to a set of rules defined in a table accessible by the router, known as the routing table. This has 1024 possible entries which can be used to define where a packet needs to be transmitted. A routing entry is composed of 3 fields: routing key, mask and direction vector. The key is used to identify the correct entry. Each multicast packet has a key, which is masked and

then compared to all the routing keys in the routing table. If the key matches with a routing entry, the action to take is specified by the direction vector and the packet can be routed to one or more output links as well as to the cores on the chip.
Point-to-Point packets are routed through a different routing table which encodes the direction of the packets in a 3-bit field. If the packet is directed to the local chip, this is sent to the Monitor core which will take care of delivering the packet to the correct destination processor [Spi11b, WFG09, Dav12].

2.5.1.4 The Boards

The SpiNNaker chips are available on boards. There are 2 types of board: SpiNN-3 and SpiNN-5. The SpiNN-3 board (Figure 2.4 bottom left) contains 4 chips, one Ethernet interface and Jtag ports. This can be used for small scale SNN simulations. The SpiNN-5 board (Figure 2.4 bottom right) contains 48 chips, one Ethernet interface for communications with an external host system and 3 FPGAs used to handle Board-to-Board communication, which happens through SATA links. The full SpiNNaker system, built at the University of Manchester, has one-million cores, and is composed of 1200 SpiNN-5 boards, arranged in 10 cabinets, each containing 5 racks, which contain 24 SpiNN-5 each.

2.5.2 Software Overview

The SpiNNaker software toolchain is composed of 2 main parts: host side software and board side software.

The host side software runs on general purpose computers which will be referred as *host* in this manuscript. The purpose of this software is to configure the SpiNNaker system correctly to run simulations, including resource allocation, results gathering and display.

The board side software runs on SpiNNaker and is in charge of executing the simulation and supervising the correct functioning of the system.

2.5.2.1 Host Side Software

The host side software is mostly written in Python. The 8 modules constituting the SpiNNaker toolchain are shown in Figure 2.5. The role of each module is described below.

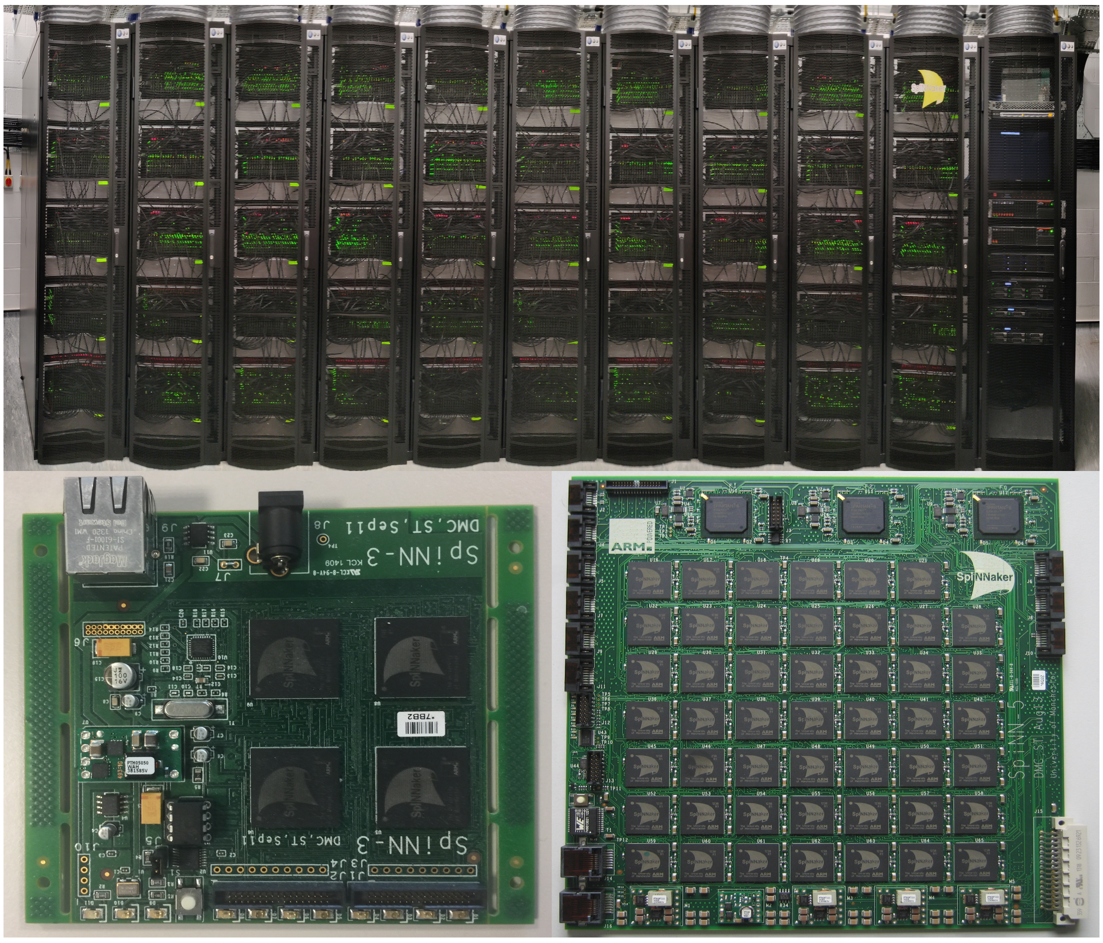


Figure 2.4: The SpiNNaker system. Top: the 1 million core machine; bottom left: SpiNN-3 board with 4 SpiNNaker chips; bottom right: SpiNN-5 board with 48 SpiNNaker chips.

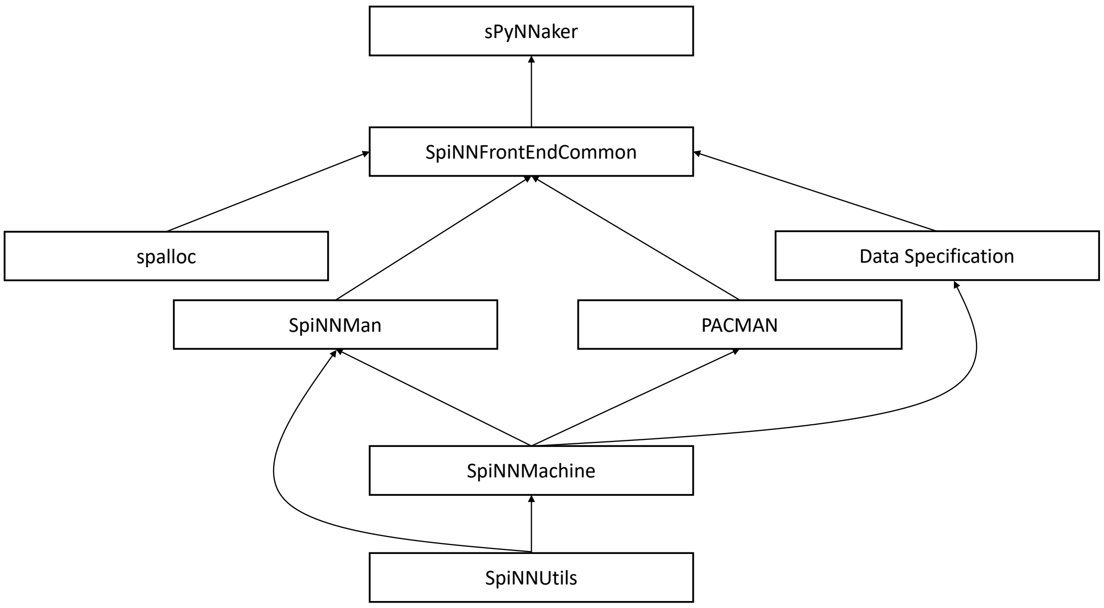


Figure 2.5: Structure of the SpiNNaker software toolchain, representing each module and their relations

- SpiNNUtils: this module contains a list of basic utility functions and classes which are used by the SpiNNaker toolchain.
- SpiNNMachine: this module presents a Python abstraction of a SpiNNaker machine. Its functionality is to create a representation of the current state of the machine allocated for the simulation in terms of chips, cores, routable links, available routing entries and available SDRAM.
- SpiNNMan: this module is used to communicate with a SpiNNaker board, by instantiating a UDP-based communication. Through this module it is possible to get the state of the machine, boot it, load application binaries and access the shared memories of individual chips from a host machine.
- PACMAN: this module performs the partitioning of the network to be simulated so that it fits the requirements imposed by the machine and then finds an optimal placement for the allocated SpiNNaker machine. It is possible to provide additional constraints both on the partitioning and the placement phases in order to best fit application requirements.
- spalloc: this module is used to access the 1 million core machine. This module allocates a portion of the machine large enough to run the simulation and it is responsible for the cleanup and reset after the simulation is

terminated.

- Data Specification: this module is used to generate memory images, which contain data needed during the simulation, for each SpiNNaker core involved in the simulation, starting from a specific set of instructions. The Data Specification tool is composed of two main submodules: the Data Specification Generator, in charge of creating the file containing the required instructions for generating the memory images, and the Data Specification Executor, which executes the instructions and generates the memory images to be loaded on SpiNNaker.
- SpiNNFrontEndCommon: this module provides functionalities common to front ends translating application level programs into executables for the SpiNNaker system.
- sPyNNaker: this module is the interface containing the implementation of the PyNN API [DBE+09], a specification language commonly used to describe SNNs, for the SpiNNaker system (more details in Section 2.6).

2.5.2.2 Board Side Software

The board side software is written in C. A schematic of it is shown in Figure 2.6. The Monitor processors (MP in Figure 2.6) run dedicated software called SC&MP (SpiNNaker Control & Monitor Program). This software supports the loading of routing tables, application software for the Application cores and the generation of memory structures in SDRAM. Furthermore SC&MP is able to determine blacklisted elements on the board and the shortest path to the Ethernet chip [RBD+19] and between chips.

The Application processors (AP in Figure 2.6) are provided with a hardware interface library acting as a kernel called SARK (SpiNNaker Application Runtime Kernel) [BFR+15, Tem16]. This library provides access to chip resources while keeping a reduced memory usage, as the instruction memory (ITCM) is limited and therefore it would not be possible to run a more complex OS such as Embedded Linux [RBD+19]. On top of SARK another library provides support for event-driven applications. This is called Spin1API [Spi11a]. The role of this library is to provide additional abstraction and support to neural modelling, through an event-driven approach. Spin1API is in charge of interrupt and timer handling, callback management and memory transfer requests [RBD+19].

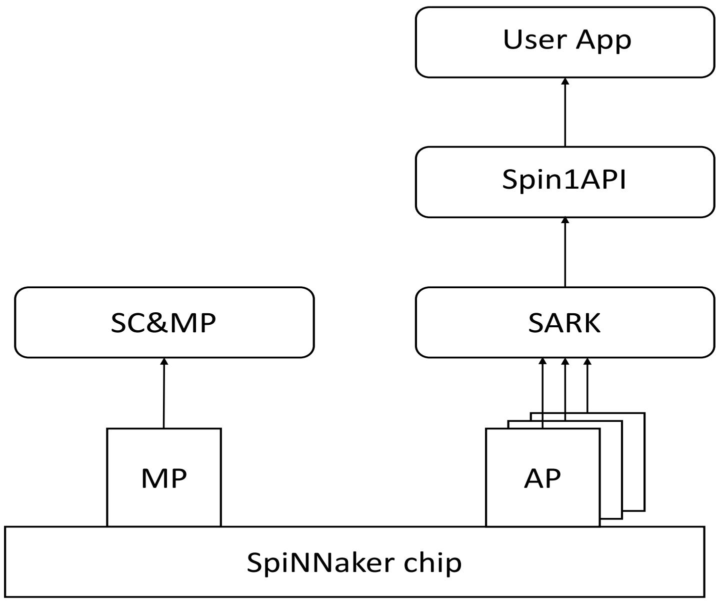


Figure 2.6: Structure of the software executed both on Monitor processors (MP) and Application processors (AP)

The application software is written on top of Spin1API and follows the event-driven approach. This means that Application cores remain idle most of the time (i.e. in a low-power state) and when an event is triggered, usually through an interrupt, the core starts to execute a callback, which is a function associated to that particular event. The SpiNNaker system, through Spin1API, allows events to be generated at different priority levels. These levels are associated with different types of interrupts and are summarised in Table 2.1. The highest priority level is labeled with -1 and is handled by the FIQ (Fast Interrupt Queue) Thread [ARM06], which has dedicated resources to allow highest performance, since this type of event requires to be executed as fast as possible. Callbacks associated to events at this priority level can preempt any other function, however it is possible to register only a single event and associated callback per application with this priority. Events registered at priority 0 are handled as IRQ (Interrupt ReQuest) and therefore will trigger an ISR (Interrupt Service Routine). Events with priority 1 and 2 are treated as soft interrupts and can be queued differently from events with priority 0 and -1.

Spin1API therefore maintains an event-driven framework with three main threads (as shown in Figure 2.7): a dispatcher thread, a scheduler thread and a FIQ thread. The scheduler thread, following an event, queues the tasks to be executed. The dispatcher thread dequeues tasks and executes them. When a

Spin1API Priority Levels		
Priority Level	Interrupt Type	Queueable/Not Queueable
-1	FIQ	NQ
0	IRQ	NQ
1,2	Soft	Q

Table 2.1: Priority levels allowed by Spin1API

priority 0 event is received, this is immediately executed directly by the scheduler, preempting any callback with priority > 0, and therefore it is not queued [SPGF11]. The FIQ thread allows callbacks with priority 0 to be preempted, by interacting directly with the scheduler thread.

SpiNNaker applications are built through a cross compiler which can either be the ARM armcc compiler [ARM16a] or the open source GNU gcc compiler for ARM [ARM16b]. Executables are generated using the APLX format [Tem11], which can be directly loaded onto a SpiNNaker core.

2.6 Neural Modelling and Simulations on SpiNNaker

This section describes how SNN simulations are implemented on SpiNNaker, together with technical details, in order to create a basis for the technical work described in this thesis. The main two references for this section [RBB+18, RBD+19] describe the mapping and simulation of SNNs on SpiNNaker at two different abstraction levels.

SNNs are simulated on SpiNNaker starting from a network description using the PyNN specification language [DBE+09], which is a high level language based on Python syntax that allows SNNs to be modeled abstracting from the hardware implementation. The main building blocks of PyNN SNNs are Populations and Projections. Populations are collections of neurons of the same type, typically representing layers or subregions of the brain [DBE+09]. Projections represent connections between two Populations, and this type of object carries all the properties of the connection (e.g. probability connectivity, weights and delays). A schematic of these is shown in Figure 2.8A, where 2 Populations (Source and Dest) connected through a Projection are presented. The network description is interpreted by the sPyNNaker module, which builds a translation

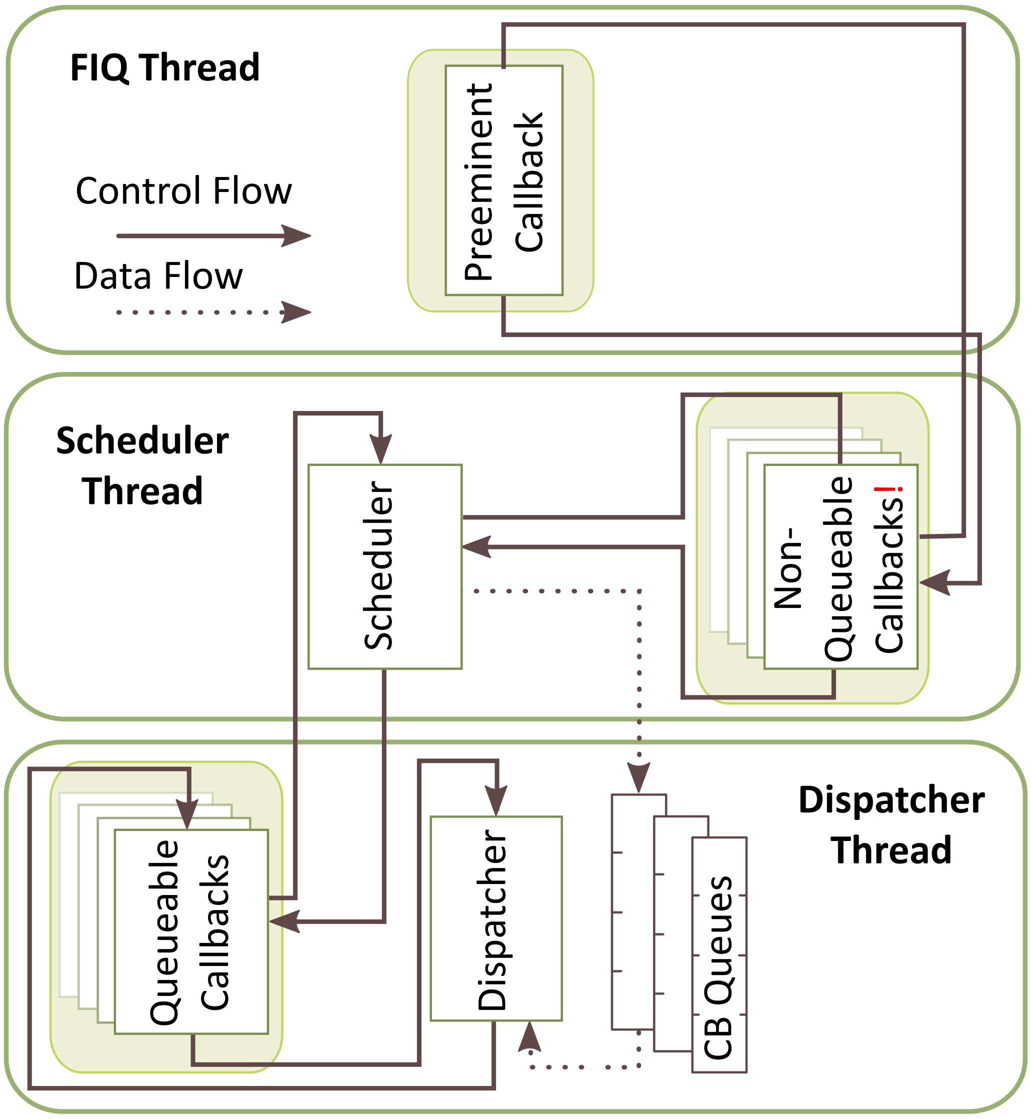


Figure 2.7: SpiNNaker event-driven programming framework: the FIQ thread handles high priority events, the scheduler thread queues callbacks and the dispatcher thread executes callbacks. Reproduced from [Spi11a] with permission.

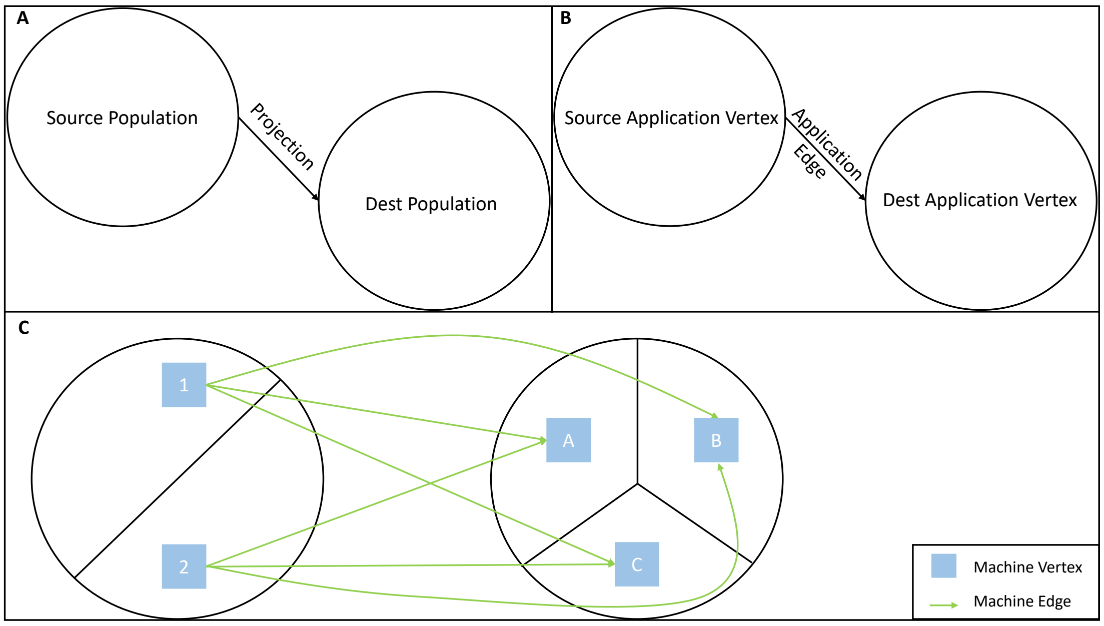


Figure 2.8: SNN partitioning under the SpiNNaker toolchain. **(A)**: Network composed of two PyNN populations. **(B)**: Application graph representing the network. **(C)**: Machine graph: the Source Application vertex is partitioned in 2 Machine vertices (blue), Destination is partitioned in 3. The Machine edges connecting the Machine vertices are represented (green)

of the network so that it can be loaded onto SpiNNaker. This phase consists of generating a graph representing the network, called Application graph, where the vertices (Application Vertices) correspond to the Populations and the edges (Application Edges) to the Projections, as shown in Figure 2.8B. The Application graph does not reflect the mapping of the network on a SpiNNaker machine, but it is a high level representation of the network with all the necessary constraints ready to be partitioned in order to fit on a SpiNNaker machine.

2.6.1 Network Partitioning

The Application graph cannot be placed onto a SpiNNaker machine as it is, this needs to be partitioned into what is called a Machine graph. A Machine graph is a lower level representation of the Application graph after all the constraints have been applied to it. A Machine graph representation is shown in Figure 2.8C. Each Application vertex is partitioned into one or more Machine vertices (in Figure 2.8C Source is partitioned into 2 Machine vertices and Dest into 3), this depends on the number of neurons the Application vertex contains and the maximum number of neurons allowed per SpiNNaker core according to the simulation parameters. Each Machine vertex represents a SpiNNaker core, therefore the resources allocated to it must be in line with the machine requirements. Application edges are then partitioned into Machine edges, in order to maintain the same connectivity at Machine level to the Application level (depicted in green in Figure 2.8C). Machine edges define the connections between cores on the SpiNNaker machine, which carry the synaptic connections linking neurons on different cores, and will instruct the generation of the routing tables.

Once the machine is allocated and the Machine Graph is generated, the mapping phase is started. First of all, a set of placements is generated, which details which core will simulate which Machine vertex; during this phase there might be additional constraints to meet, which can force some vertices to be on the same chip so that they can share memory structures, or instruct exactly where to place some vertices, or define a specific placement strategy based on chip coordinates. After the Machine vertices are placed, the routing tables are generated.

2.6.2 Routing Tables and Communications

Communication of spike packets in SNN simulations are performed through the multicast network. As detailed in Section 2.5.1.3, the multicast network, in order to correctly forward packets, uses a routing table stored on each chip. These tables are generated after the placement phase is completed and then compressed by the host side toolchain, by merging duplicates, in order to reduce their size. Communication of spikes during SNN simulations on the SpiNNaker system follow the AER (Address Event Representation) format [Mea89], which uses the key of the source neuron to route the packet. Therefore each multicast packet contains exclusively the key of the sender core (appropriately combined with the neuron ID), which is automatically generated during the mapping phase, and routers will use this value as key to be compared to the routing entries. This format allows the size of packets sent over the network to be reduced, by eliminating the need for a payload. It is furthermore not necessary to specify the list of receivers, since the receivers themselves will be the only processors accepting the packets.

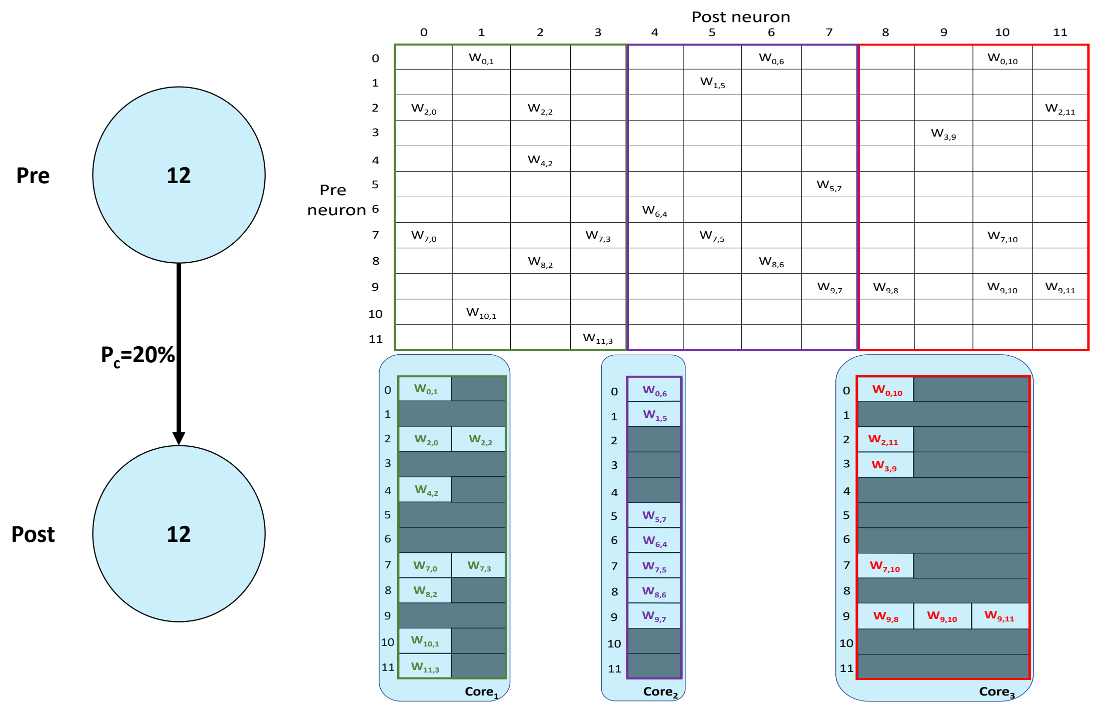


Figure 2.9: Synaptic matrix partitioning. The presented matrix comes from an example network composed of 2 Populations having 12 neurons each with 20% connectivity (schematic on the left). The full synaptic matrix is shown on top right. The sparse representation partitioned into 3 different cores is shown on bottom right (with colors matching the full synaptic matrix). The partitioning assumes a limit of 4 neurons per core, therefore 3 cores are required.

2.6.3 Synaptic Matrices

The final phase before the simulation is started is the synaptic matrix generation. Synaptic Matrices are structures stored in SDRAM, which contain data relevant to the connections, such as weights and delays. These structures are stored in the shared memory accessible from cores simulating the synapses. The choice of storing these structures in shared memory, instead of locally to postsynaptic cores, is motivated by the limited availability of local memory. By using the SDRAM it is therefore possible to simulate SNNs with much larger fan-ins. This implementation however requires synaptic information to be retrieved from shared memory whenever a spike is received, incurring in additional memory accesses which impact the simulation performance.

Synaptic matrix generation can happen either on the SpiNNaker machine or on the host. Typically, small networks using simple connectors have the synaptic matrices generated on the host, while more complex networks require the toolchain to preload some data onto each chip and then Application cores generate their subportion of the matrix using this data and connectivity information available locally. A representation is shown in Figure 2.9, where a network composed of 2 Populations (Pre and Post) with 12 neurons each, connected with 20% probability of connection (depicted on the left) is presented. The full synaptic matrix is displayed (top right), where each row corresponds to a presynaptic neuron and each column to a postsynaptic neuron. Where a connection is formed a weight is added to the respective cell. Figure 2.9 shows how synaptic matrices are partitioned and mapped to SpiNNaker cores. The right bottom representation shows 3 Application cores each with its own sparse representation of the synaptic matrix, assuming a limit of 4 neurons per core. This representation allows the size of the stored matrix to be reduced, only including the relevant information.

To allow efficient access, two additional structures are stored in the DTCM of each core as support, as shown in Figure 2.10. The first, called Master Population Table, is used to determine whether the presynaptic population has one or more connections with the neurons on the postsynaptic core and, if so, it contains the location of an entry into the second data structure, called Address List, and the length of this entry in terms of its number of rows. Figure 2.10 shows an example of the two structures from the perspective of the core simulating the Machine vertex Dest A in Figure 2.8C. The presynaptic population ID is obtained by masking the sender ID (Source 2 in the case presented in Figure

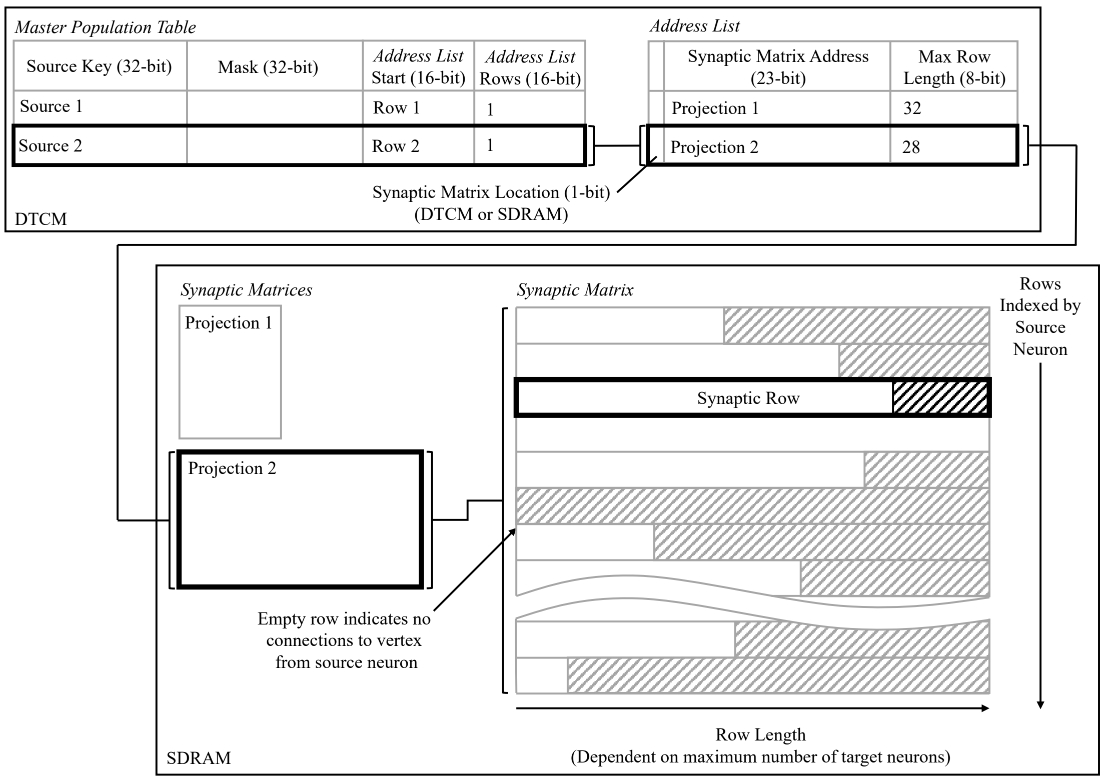

Master Population Table
Source Key (32-bit)
Mask (32-bit)
Address List Start (16-bit)
Address List Rows (16-bit)
Source 1
Row 1
1
Source 2
Row 2
1
Address List
Synaptic Matrix Address (23-bit)
Max Row Length (8-bit)
Projection 1
32
Projection 2
28
Synaptic Matrix Location (1-bit) (DTCM or SDRAM)
DTCM
Synaptic Matrices
Projection 1
Projection 2
Synaptic Matrix
Synaptic Row
Rows Indexed by Source Neuron
Empty row indicates no connections to vertex from source neuron
Row Length (Dependent on maximum number of target neurons)
SDRAM

Figure 2.10: Representation of the interaction between Master Population Table, Address List and Synaptic Matrices. The Master Population Table is shown from the perspective of the core simulating Dest Population A in Figure 2.8C. The path in bold shows the steps from when a spike packet is received from Source Population 2. Adapted from [RBB+18] with permission.

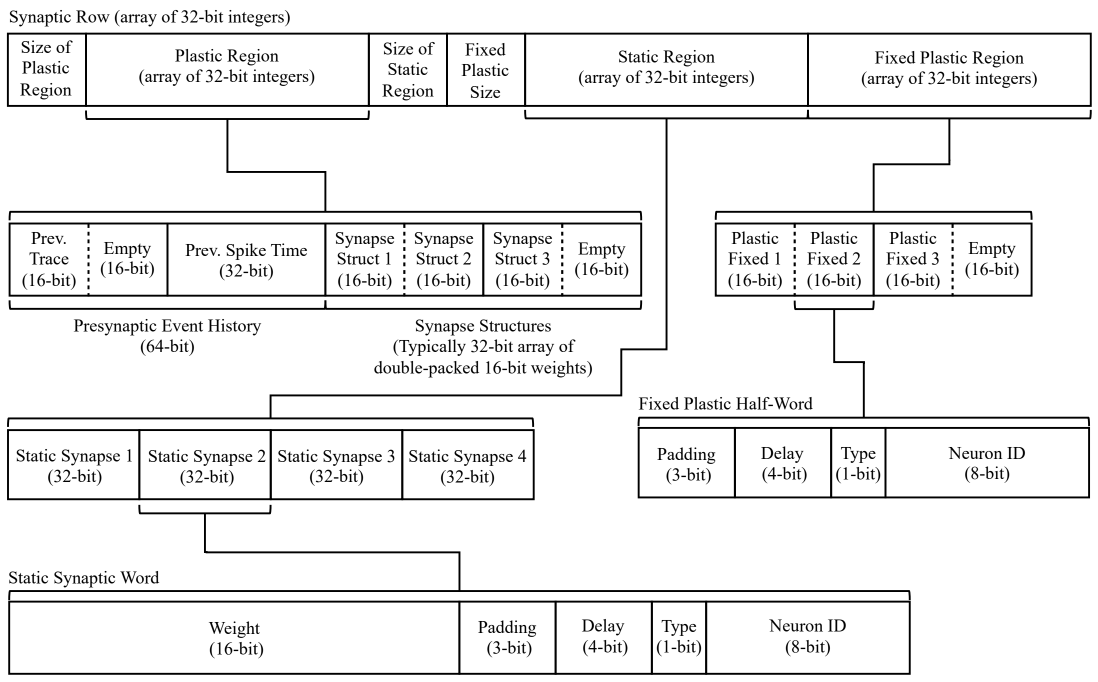

Synaptic Row (array of 32-bit integers)
Size of Plastic Region
Plastic Region (array of 32-bit integers)
Size of Static Region
Fixed Plastic Size
Static Region (array of 32-bit integers)
Fixed Plastic Region (array of 32-bit integers)
Prev. Trace (16-bit)
Empty (16-bit)
Prev. Spike Time (32-bit)
Synapse Struct 1 (16-bit)
Synapse Struct 2 (16-bit)
Synapse Struct 3 (16-bit)
Empty (16-bit)
Presynaptic Event History (64-bit)
Synapse Structures (Typically 32-bit array of double-packed 16-bit weights)
Plastic Fixed 1 (16-bit)
Plastic Fixed 2 (16-bit)
Plastic Fixed 3 (16-bit)
Empty (16-bit)
Fixed Plastic Half-Word
Padding (3-bit)
Delay (4-bit)
Type (1-bit)
Neuron ID (8-bit)
Static Synapse 1 (32-bit)
Static Synapse 2 (32-bit)
Static Synapse 3 (32-bit)
Static Synapse 4 (32-bit)
Static Synaptic Word
Weight (16-bit)
Padding (3-bit)
Delay (4-bit)
Type (1-bit)
Neuron ID (8-bit)

Figure 2.11: Structure of a synaptic row, including both static and plastic data. Reproduced from [RBB$^+$18] with permission.

2.10), in order to remove the least significant bits indicating the neuron ID. Each entry in the Master Population Table corresponds to a connected source vertex and contains the correct mask and the structure allows efficient binary search for faster access. The Address List contains information about where the synaptic data is stored. Each row of this data structure contains an address indicating the starting position of the projection into the synaptic matrix and the maximum number of postsynaptic neurons connected to a presynaptic neuron in the current projection (meaning the maximum row length for this projection). There might be multiple projections between two populations, therefore the Address List can have multiple lines associated with a master population table entry. The example presented in Figure 2.10 shows the steps necessary to retrieve the correct synaptic row when a spike is received, therefore, once the correct line for Source 2 is located in the Master Population Table, row 2 in the Address List is accessed (as it is the single row in the Address List containing information regarding the connections between the two Machine vertices). This row provides the address of the Synaptic Matrix in SDRAM, together with the maximum row length (28 in this case). At this point the correct synaptic row can be retrieved.

The synaptic matrices are indexed by presynaptic neuron, which means that a single row, in the sparse representation format, will contain all the connections that a presynaptic neuron has with the postsynaptic neurons implemented by the postsynaptic core for a specific projection. A synaptic row is composed of 3 main fields and the use of them depends on whether the connection is static or plastic (see section 2.2.2). The three fields are in order: plastic region, static region and fixed plastic region (as shown in Figure 2.11). Each region is preceded by an integer indicating its size, in terms of the number of postsynaptic connections. If a section is unused, the corresponding size is set to 0. Static connections make use of the static region only. This region is an array of 32-bit entries, each corresponding to a synapse. The most significant 16 bits are used to store the synaptic weight using fixed point arithmetic with right shift for maximum precision. The remaining 16 bits are used to store the neuron ID (8 bits), the synapse type (up to 4 bits) and the synaptic delay in timesteps (4 bits). Plastic connections use the two plastic fields (fixed plastic region and plastic region). The reason behind this choice is for efficiency purposes. Since plastic synapses require the row to be updated, only the plastic region will be changed, i.e. the fixed data doesn't change, and so there is no need to write it back. This separation therefore allows to perform shorter write-backs to SDRAM. The plastic region contains a header defining the Presynaptic Event History, used to keep track of the previous presynaptic spikes and an array of 16-bits values double packed into 32-bit words. Each of these values represents a weight, which is updated during the simulation and used to compute the synaptic contribution for the input current. The fixed plastic region is an array of 16-bit values containing neuron ID, synapse type and delay in the same structure as the 16 LSBs of the static region field. The indices in this region correspond to those in the plastic region, to allow direct access.

2.6.4 Simulation Flow

After the placement and the synaptic matrix generation phases are concluded, the cores are synchronised and the simulation starts. SNNs on SpiNNaker are simulated through a hybrid mechanism, where neurons are simulated following a time-driven approach, while handling of synapses is event-driven. This allows comparable efficiency to event-based schemes when considering biologically-representative spike rates, together with maintaining flexibility in neural modelling [RBB+18]. Each Application core involved in the simulation is in charge of implementing a

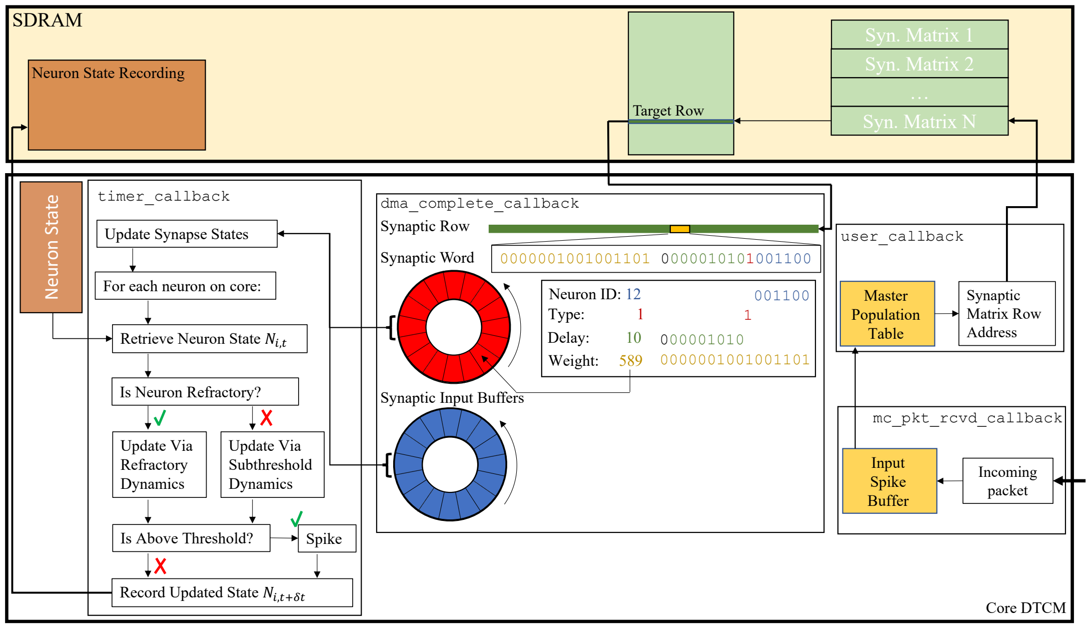


Figure 2.12: Simulation flow for a neural application on SpiNNaker. The *timer_callback* advances the neural state and generates output spikes, the *multicast_packet_received_callback* buffers incoming spike packets, the *user_callback* retrieves synaptic rows from shared memory, and the *dma_complete_callback* processes the synaptic events.

predefined number of neurons, defined by the toolchain on host. Time is discretised into fixed length timesteps, in which the neurons can fire and their internal state is updated. Timesteps are updated through the use of Timer1, which is one of the two counters available to the cores. A schematic of the simulation flow for neural applications is presented in Figure 2.12, while figure 2.13 shows the interactions between the callbacks for 2 simulation timesteps.

Every timestep is associated with a callback, called *timer_callback*, which sequentially updates the state for each neuron simulated on the core (as shown on the left in Figure 2.12), by computing the synaptic input currents and generating the state according to the model's equations. All the neuron state variables are held in DTCM, therefore no access to shared memory is required in this phase. After the state of a neuron is updated, the neuron can fire. If it does, a Multicast packet containing a key which is the result of a bitwise OR between the unique core's ID and the neuron ID is sent over the Multicast network, and the *timer_callback* moves to the next neuron implemented by the core, until all the neurons have been processed. After the last neuron is updated, the state is recorded, which means that some neural parameters, which can be specified by the host when configuring the simulation, are stored for the current timestep, including the generation of spikes.

While the *timer_callback* is processing the neural state update, spikes can arrive, either from spike sources or from other neurons. In order to prevent traffic from backing up in the network, as soon as a packet arrives at destination, the source ID is extracted and locally stored (as shown in Figure 2.13). This is performed through a *multicast_packet_received_callback* assigned to priority -1 (shown on the right in Figure 2.12). Each incoming spike is buffered in a circular buffer - the input spike buffer - which is typically 256 entries long. This allows to queue arriving spikes that will be processed when the core is available. If there are no spikes currently being processed, the *multicast_packet_received_callback* triggers another event called user event, which initiates a spike processing pipeline through a callback called *user_callback*, registered at priority 0. The spike processing pipeline extracts a spike from the input spike buffer and looks into the Master Population Table in order to locate the position of the synaptic information relative to that spike. Once an address is located in the Address List, it is possible to access the correct block containing all the synaptic information in the synaptic matrix. The synaptic row is retrieved by a DMA request to SDRAM. As shown in Figure 2.13, the *user_callback* is executed at a higher priority than the neural state update (which is registered at priority 2); this allows DMA transfers to start with minimum latency and the core can continue processing the neural state update while the DMA controller handles the memory transfer.

Upon completion of the memory transfer a *dma_complete_callback* is started by a DMA_complete event triggered by the DMA controller (registered at priority 0). This callback, which again has higher priority compared to the *timer_callback*, as shown in Figure 2.13, processes the synaptic information for the current spike. Each spike targets a synaptic row, therefore this callback will convert the synaptic row into individual postsynaptic neuron input. The amount of computation at this stage depends on whether the row is static or plastic. Regardless of the type of connection, the callback loops through the synaptic row, extracting the single weight for each synapse. Another data structure, called synaptic input buffer, inspired by previous works [MMG$^+$05], is used at this stage to determine the input contribution of each individual postsynaptic neuron (shown in blue and red for inhibitory and excitatory synapses respectively in Figure 2.12). Input

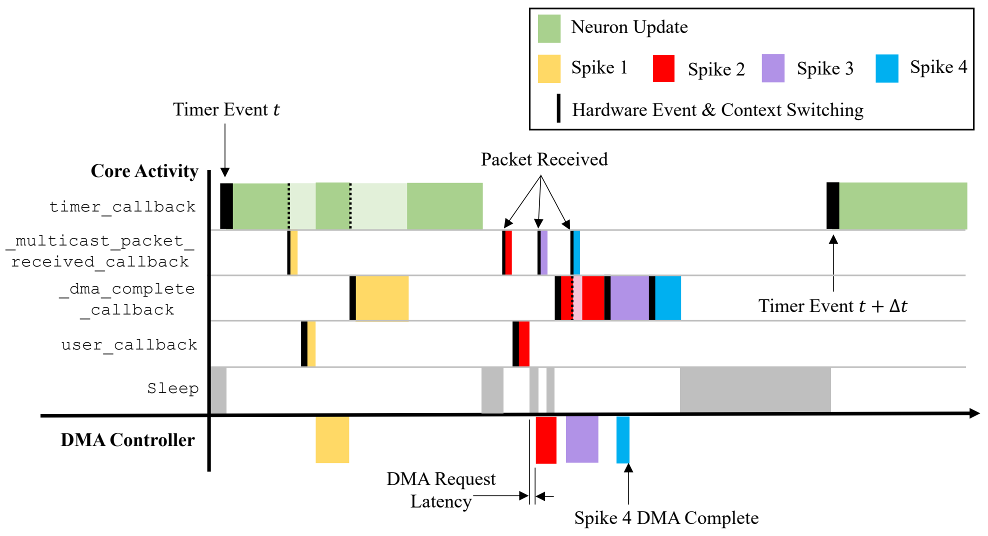


Figure 2.13: Interactions between callbacks for 2 simulation timesteps in the context of a neural application, showing 4 separate spike events. Lighter color shade indicates the preemption of the callback due to a higher priority task being executed. Reproduced from [RBB+18] with permission.

contributions are named synaptic events. A synaptic event is one spike innervating one synapse. Synaptic events (in the form of weights) are added by the *dma_complete_callback* to the correct postsynaptic neuron through the synaptic input buffers. The synaptic input buffers are two-dimensional circular buffers, indexed by postsynaptic neuron ID and delay. Every spike arriving at a postsynaptic core has a delay $\geq$ 1 timestep. Incoming spikes represent the output of neurons generated this timestep. Neural connections are characterised by a synaptic weight and delay, an efficient way to implement delays is by immediately sending the generated spikes and allowing postsynaptic cores to handle the delay. On the postsynaptic side, when the correct connection is identified, the postsynaptic event is added to the correct time slot in the synaptic input buffer for the correct postsynaptic neuron. At the end of the timestep, the synaptic input buffer rotates and creates space for a new time slot. The neuron state update at the beginning of the timestep extracts the information from the first time slot on the synaptic input buffer. Each synaptic type has a different synaptic input buffer (as shown in Figure 2.12), the values are combined together to obtain the input current on the neuron state update phase. Once all the incoming synaptic events related to an incoming spike are processed, the spike processing pipeline checks whether there are additional spikes waiting to be processed into the input spike buffer (some might have arrived during the processing of the last spike). If so, the first spike is processed according to the steps described above, otherwise the pipeline terminates and the core goes back into processing lower priority tasks or becomes idle.

Due to limited local memory, the maximum delay which can be stored into synaptic input buffers by default is 16 timesteps. For connections having larger delays, a different mechanism called delay extension is put in place. Delay extension employs application cores with the dedicated task of buffering spikes having larger delays. This mechanism is implemented by splitting delays into a multiple of 16 timesteps plus remainder. The remainder is handled by the core implementing the postsynaptic neurons through the synaptic input buffer mechanism described above, while the multiple is handled by delay extension cores. Each of these cores stores each spike for the required time and then forwards it. If the delay is larger than 8 $\times$ 16 timesteps (128 timesteps), additional steps are required, therefore the packet is sent to another delay extension core, otherwise the spike is forwarded directly to the postsynaptic processor [vARS+18, RBB+18]. This mechanism allows in principle any amount of delay to be implemented, by creating a chain of extensions.

$$E = \left(\frac{t_P - t_{1^{st}} - t_{last}}{t_{spike}} + 2 \right) Pn \tag{2.7}$$

$$t_P = \Delta t - t_{upd} \tag{2.8}$$

$$t_{spike} = m_s Pn + c_s \tag{2.9}$$

The maximum number of synaptic events that can be processed per timestep, while maintaining real-time performance is shown in Equation 2.7, where t_P indicates the fraction of the timestep available to process synaptic information, and is obtained (as shown in Equation 2.8) by subtracting from the timestep duration (Δt) the time required to update the neural state (t_{upd}) of all the neurons simulated on core. The amounts of time required to process the first and the last spikes in the spike processing pipeline are $t_{1^{st}}$ and t_{last} respectively. These differ from the generic spike processing time due to the pipelined spike processing approach performing different API calls at the beginning and end of the pipeline. The spike processing fraction of the timestep (obtained by substracting $t_{1^{st}}$ and t_{last} from t_p) is then divided by the time required to process a single spike (t_{spike}) and incremented by 2, to account for the two spikes previously subtracted ($t_{1^{st}}$

and t_{last}), resulting in the number of spikes that can be processed per timestep. The number of synaptic events is therefore given by multiplying the number of spikes that can be processed per timestep by the connectivity probability (P), which indicates the number of postsynaptic connections per spike, and then by the number of postsynaptic neurons on the core (n). The time necessary to process a single spike is expressed by Equation 2.9 and can be broken into a fixed contribution (c_s), which is paid once per spike packet, corresponding to context switches, synaptic row location in the shared memory and transfer time, and a variable contribution (m_s) which corresponds to the cost of processing a single synaptic event. The same rules apply to $t_{1^{st}}$ and t_{last}, however they have different values for fixed and variable costs [RBB+18].

2.6.5 Synaptic Update

Section 2.6.4 described the simulation flow related to static networks. For plastic networks, every time a presynaptic spike is received, or a postsynaptic action potential is generated, the synapses involved in the connection need to be updated, according to the mechanisms described in Section 2.2.2. In order to do so, postsynaptic neurons maintain local buffers, called postsynaptic buffers, including information regarding the last postsynaptic spike, together with a postsynaptic trace constructed from low-pass filtered postsynaptic spike trains. In a similar way, synaptic matrices, in their plastic regions, keep the last presynaptic spike time and the trace (see Section 2.2.2) for each row. These values combined allow to compute the weight update for every synapse. The SpiNNaker plasticity framework uses the deferred event-driven model [JRG+10, DC14], where weight updates are presynaptic-sensitive, therefore they happen when a synaptic row has been fully copied into DTCM. This mechanism allows to deal with the synaptic weights being stored in SDRAM and therefore not always locally available. Therefore, in simulations in which plastic connections are involved, when the *dma_complete_callback* (see Section 2.6.4) loops over a synaptic row to extract the synaptic events, before adding an event to the correct synaptic input buffer slot, it updates the corresponding synaptic weight according to the implemented plasticity rule. This entails evaluating potentiation by comparing the presynaptic trace with postsynaptic spike events and depression with the postsynaptic trace compared against presynaptic spike events. Updated weights are then written back to SDRAM, together with the updated trace and presynaptic spike time after the row has been processed. The remaining portion of the spike processing pipeline is analogous to the static case.

2.7 Summary

This chapter provided an introduction to Spiking Neural Network concepts. Neuron models and rules for biological networks were presented, together with biological learning mechanisms. A list of tools to simulate neural networks was provided, differentiating hardware and software solutions, and their strengths and weaknesses. Emphasis was given to Neuromorphic platforms and their potential, as a novel approach to address SNN simulations. Finally details about the SpiNNaker system, used to perform this work, were provided, both from a hardware and a software perspective, detailing how SNNs are simulated starting from their description to the execution phase.

The next chapter presents a novel approach to parallelise SNN simulations on the SpiNNaker Neuromorphic platform and its application to a use case popular among the neuroscientific community.

Chapter 3

Real-Time Simulations of SNNs

3.1 Introduction

Real-time simulations of large-scale SNNs have implications in several domains, spanning from artificial intelligence to neuroscience. On the one hand they allow the study of long-term learning tasks, on the other hand they facilitate the analysis of neural pathologies over meaningful periods, which otherwise would require much larger time scales. Achieving real-time performance is, however, challenging due to high input activity combined with very tight timing constraints. This chapter therefore delivers research in this direction and describes a study to tackle these issues, which allowed simulations of biologically-representative Spiking Neural Networks in real time on SpiNNaker, by providing all the necessary tools for the task. Section 3.2 presents the Heterogeneous Programming model, a method which evolved from a previous study performed on SpiNNaker [KF16] on synaptic matrix partitioning and neural event throughput. Section 3.3 describes the application of this Heterogeneous Programming model to a complex biologically-representative SNN, commonly regarded as a benchmark in the field, namely the Cortical Microcircuit [PD12]. The results and benchmarking are shown in section 3.4. The challenges presented in this chapter cover static SNN simulations, where synaptic weights are fixed at their initial level. Synaptic plasticity and the use of the Heterogeneous Programming model in that context are discussed in the next chapters.

Real-time simulations of biologically-representative SNNs present several challenges. First, the timing constraints: this type of application requires handling

simulation time resolutions usually between 1 ms and 0.1 ms. This allows discretisation of continuous time models, therefore time resolutions need to be high enough in order to allow modeling of neuron state updates via exponential integration, calculating the dynamics timestep by timestep. Within these ranges it is necessary to process all the information, including the incoming signals from presynaptic populations, the update of the neural states and the generation of action potentials to be delivered to the connected postsynaptic populations.
The connectivity patterns play a key role too. Two different populations commonly connect with a pre-defined connectivity probability. Long-range connections are characterised by a very low probability, meaning that the postsynaptic representation of the connectivity matrix will be very sparse; on the other hand, short-range connections will result in denser connectivity patterns. Very sparse connectivity patterns can cause performance degradation, as computation will be affected by the latency of memory reads, when retrieving the synaptic rows, which become the dominating term when processing an incoming spike packet. This happens because sparser networks generate synaptic matrices which contain only a limited number of synapses per row.
Finally, the high fan-in. Despite being characterised by relatively sparse connectivity patterns, in biologically-representative SNNs, postsynaptic neurons commonly exhibit very large numbers of incoming connections, due to the high number of neurons involved. This results in large numbers of spikes being delivered to postsynaptic neurons every timestep, which, because of the timing constraints, need to be processed within the timestep boundaries.

The standard SpiNNaker software toolchain (described in Chapter 2) struggles to deal with these characteristics while maintaining real-time performance. An example is given by a previous study [vARS+18], where, despite SpiNNaker being the first Neuromorphic platform to simulate a complex biologically inspired SNN, namely the Cortical Microcircuit model [PD12], the achieved performance was still 20× slower than real-time requirements.

Another study performed on SpiNNaker [KF16] however, demonstrated that it is possible to improve the simulation efficiency by better acting on the partitioning of the synaptic matrices and on the network placement. A horizontal fragmentation, as opposed to the vertical partitioning performed by the standard SpiNNaker software toolchain, of the synaptic matrices allows to maximise the

length of the processed postsynaptic rows and therefore to improve the performance when dealing with sparsity. This is achieved by performing the synaptic input processing on dedicated cores and by separating this phase from the neural state update, which is now not preempted by the arrival of packets containing spikes anymore (as described in Section 2.6). By allocating multiple Synaptic cores to each Neuron core, it is also possible to parallelise the input spike processing phase, further increasing the number of synaptic events that can be processed in a single timestep.
Starting from these principles it has been possible perform for the first time real-time simulations of the Cortical Microcircuit model, guaranteeing a 20$\times$ speedup compared to previous works [RPR$^+$19].

3.2 The Heterogeneous Programming Model

This section describes the Heterogeneous Programming model, detailing its first formulation and the implementation details on SpiNNaker. Profiling is also provided, showing its benchmarking on SpiNNaker, together with an updated cost model, compared to that presented in Equation 2.7.

3.2.1 The Synapse-Centric Mapping

The Heterogeneous Programming model is a novel technique designed to enhance the throughput of synaptic events on the SpiNNaker Neuromorphic platform. A similar idea was originally conceived with the aim of improving the handling of synaptic matrices [KF16], in order to increase the number of synaptic events which can be processed per timestep. This first approach splits the synaptic matrices in a way to maximise the length of each row. This improves the performance when dealing with sparse networks, by increasing the number of postsynaptic neurons per row, therefore amortising the cost of API calls and memory transfers over an increased number of neural operations.

As detailed in section 2.6.3, due to limited availability of local memory, the SpiNNaker system stores the synaptic matrices into the shared memory available to each chip (SDRAM). Through this mechanism, SpiNNaker allows to store much larger synaptic matrices and therefore handle higher fan-ins, by exploiting the large off-chip memory. The drawback of this mechanism is that, in order to access the synaptic data, it is necessary to copy these structures locally (to

DTCM). However, due to limited availability of local memory, it is possible to store only a limited number of rows at a time. This requires to access shared memory every time a spike is received, in order to obtain the postsynaptic data targeted by it. All the processors on a chip share a single memory channel. This characteristic causes memory contention, whenever multiple cores try to access memory simultaneously [RBB+18]. Memory contention causes performance degradation, which increases proportionally to the number of cores which try to access memory at the same time. Furthermore, shorter synaptic rows lead to a lower number of targets per postsynaptic core. This requires to allocate more processors which will contribute to memory contention. This aspect is even more significant when simulating sparse networks, and so biologically-representative SNNs, since presynaptic neurons connect with a small subset of the postsynaptic population, resulting in only a limited number of postsynaptic connections per spike. In this case, the standard partitioning performed by the SpiNNaker software toolchain generates synaptic matrices with very short rows, causing the cost of API calls and memory transfer to be even more significant [RBB+18].
On the other hand, through a horizontal partitioning of the synaptic matrices it is possible to include more synapses per core, resulting in longer synaptic rows, which, in turn, reduce the number of memory accesses.

The Synapse-centric mapping approach introduced the concept of *Synapse* cores and *Neuron* cores. As outlined in section 2.5.2, in the standard SpiNNaker software toolchain, each Application processor performs both the neural state update and the processing of all the incoming spikes for all the neurons simulated by that core. This causes the neural state update procedure to be interrupted multiple times per timestep, in order to accommodate the processing of incoming spikes, which are treated by the software toolchain at a higher priority level. This constitutes a hard limit on the number of neurons that can be simulated on a single core, or, a simulation slowdown is required. This type of mapping provides a vertical partitioning of the synaptic matrices. This means that a specific postsynaptic core will have access to all the synapses for all the neurons implemented by it. When simulating biologically-representative SNNs, the number of connections is generally very large, causing the total fan-in for each neuron to be very high. This affects the synaptic matrices, which become very large, as a new line is added for each presynaptic neuron. To ensure that all the incoming spikes are

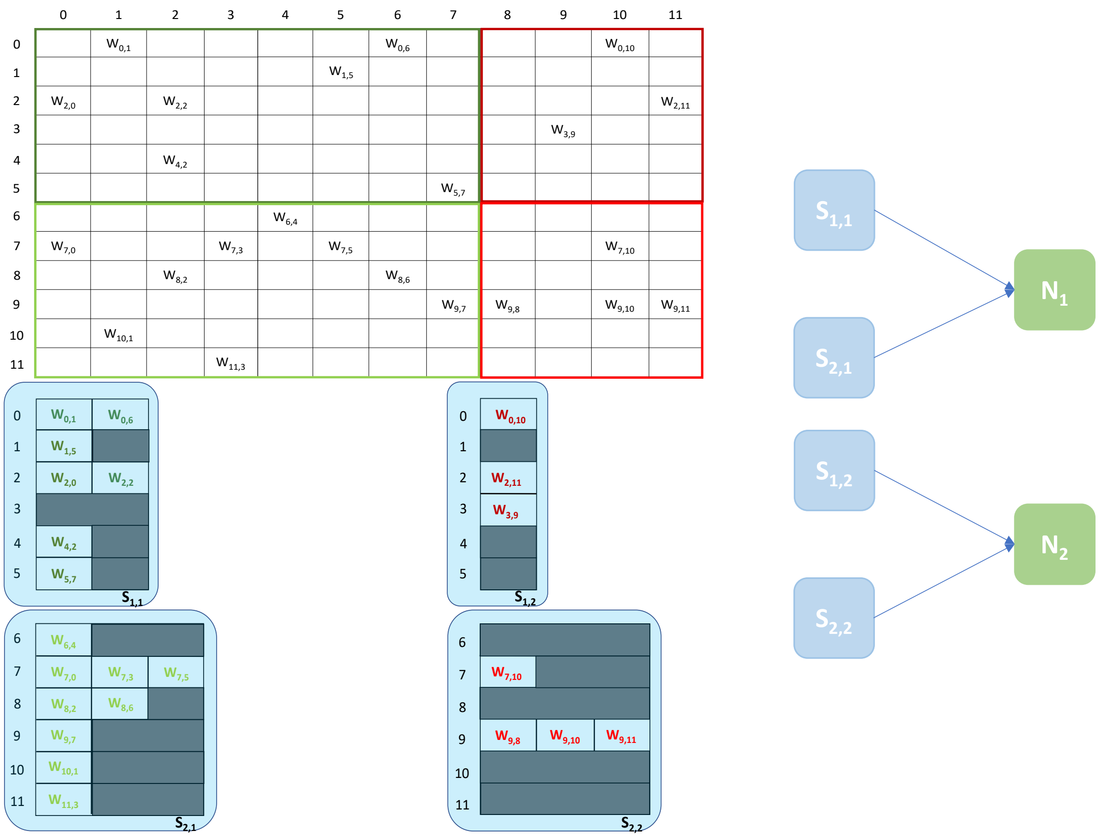


Figure 3.1: Synaptic matrix partitioning under the Heterogeneous Programming Model. The same matrix presented in Figure 2.9 is used. *Synapse* cores allow to partition the matrix by presynaptic index and to relieve *Neuron* cores from processing spikes, enabling the possibility of simulating more neurons per core. This, in turn, allows to increase the length of synaptic rows. A schematic of the ensembles generated by this partitioning is shown on the right, where each *Neuron* core receives inputs from two *Synapse* cores.

processed and the neuron states are updated by the end of the timestep, it is necessary to reduce the number of neurons per core. This, however, does not affect the fan-in of the neurons, but only reduces the length of the processed synaptic rows. The consequence is that the cost to API calls (interrupts and DMA transfers) becomes too high compared to the neural processing, since cores spend more time context switching and retrieving data from memory, than processing useful information.

By introducing *Synapse* cores and *Neuron* cores, it is possible to split the load over different computational units. The *Neuron* cores are dedicated to perform the neural state update, while the *Synapse* cores handle the spike processing side of the simulation. This allows longer synaptic rows, while preserving higher numbers of neurons per core, by moving the partitioning towards the synapses rather than neurons. Each *Synapse* core deals with a single synapse type and receives inputs from a subset of the presynaptic neurons. An example is shown in Figure 3.1, where the same network employed in Figure 2.9 is used. In Figure 3.1 the horizontal partitioning is performed on the synaptic matrix, resulting in postsynaptic cores doubling the number of synapses they can access in a synaptic row and reducing the number of empty rows per core. The allocated *Synapse* cores with their sparse synaptic matrix representations are shown on the bottom left. On the right, a mapping between *Neuron* and *Synapse* cores is shown. Through this approach the number of neurons per *Neuron* core is increased, since *Neuron* cores are in charge of the neural state update only.

3.2.2 Model Adaptations

The Synapse-centric approach was developed under a previous software infrastructure running on SpiNNaker [Spi15]. The current SpiNNaker toolchain, described in section 2.5.2, is structurally different and employs different techniques and languages. In order to implement an efficient parallelisation method, a new approach has been developed starting from the Synapse-centric mapping, instead of a simple porting of the original strategy. This new method is called the Heterogeneous Programming model. The principle behind it lies in the Synapse-centric mapping described in section 3.2.1. By splitting the synaptic matrices horizontally, and by introducing dedicated cores for the synaptic processing, it is possible to increase the number of synaptic events processed per timestep.

Compared to the Synapse-centric mapping, the Heterogeneous Programming model adopts a new optimised partitioning strategy, described in Section 3.2.2.1, which allows optimal placement of neurons per resource. This partitioning is furthermore integrated in the SpiNNaker software toolchain, which was conceived independently from the Synapse-centric mapping and did not offer any support for this type of simulation. Furthermore, the Heterogeneous model is well integrated into the SpiNNaker API, with additional emphasis on real-time simulations, therefore the employed memory and task interactions have been revised in order to maximise performance, as detailed in Section 3.2.2.2.

3.2.2.1 Network Partitioning

The Heterogeneous Programming model acts on various levels on the SpiNNaker software toolchain: first a different partitioning technique is applied, second the API is customised such that performance gain is maximised.

As described in section 2.6, the main building blocks of PyNN-based SNNs are Populations and Projections. Each Population is automatically translated into an *Application Vertex* by the software toolchain and then this is partitioned into one or more *Machine Vertices* according to the number of neurons belonging to that specific Population and the maximum allowed number of neurons per core. Similarly, Projections are translated into *Application Edges*, which are then partitioned into *Machine Edges* to match the partitioning of vertices. The Heterogeneous Programming model changes the way networks are partitioned, as shown in Figure 3.2. This approach generates an additional layer which lies between the PyNN description of a network and the *Application Graph*, which is called the *PyNN Partition Layer*. This additional layer allows the definition of the desired mapping between *Synapse* cores and *Neuron* cores and to connect them properly together. A PyNN Population is translated into a *PyNN Partition Vertex*, which will contain multiple Application Vertices (as shown in Figure 3.2B). Application Vertices can now represent *Neuron* vertices or *Synapse* vertices (respectively light blue and light green in Figure 3.2B). *Synapse* vertices are connected to *Neuron* vertices through Application Edges which are called Internal edges (in red). These edges allow to define constraints between these vertices, so that they are stored on the same chip and the neuron and synapse indices are correctly matching. At this stage, there are two levels of partitioning, the first is introduced with the PyNN Partition layer and is used to connect PyNN Partition Vertices, or Populations, together. This higher level fragments a PyNN Partition

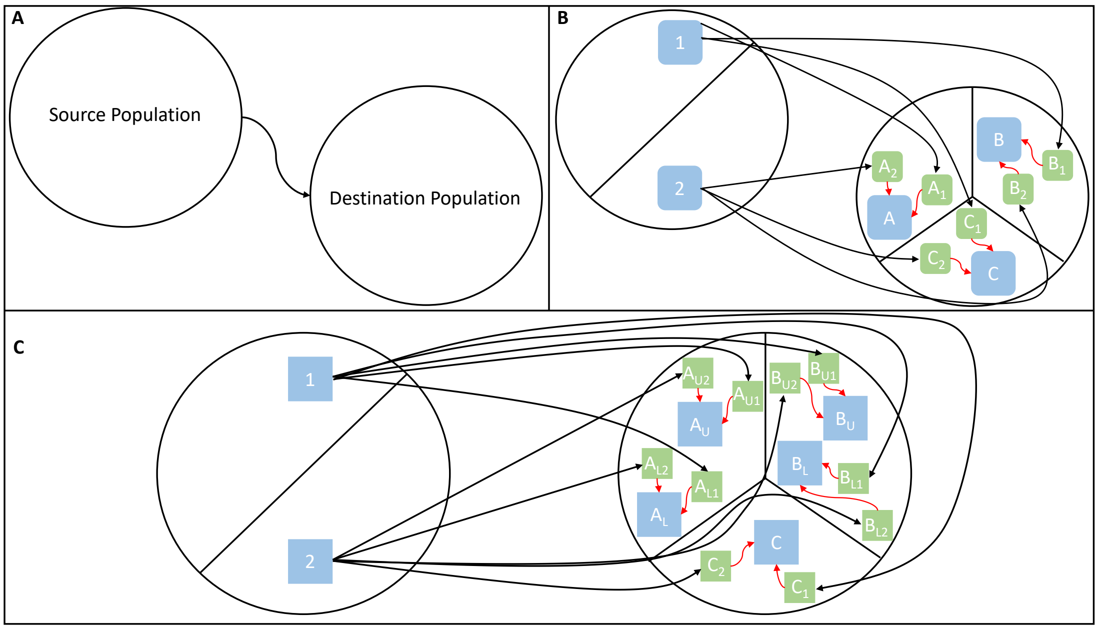


Figure 3.2: Network partitioning under the Heterogeneous Programming Model. **(A)**: Network composed of two PyNN populations. **(B)**: PyNN Partition layer; the source population is composed of 2 partitions, labelled “1” and “2”, the destination population has 3 partitions “A”, “B” and “C”. *Neuron* vertices are light blue, *Synapse* vertices light green. Internal edges are red and partition edges black. **(C)**: Machine Graph; the source population has the same number of machine vertices, the destination population has 2 machine vertices per Neuron vertex for the partitions “A” and “B” and 1 for the partition “C”. The represented edges here are the Machine edges resulting from the partitioning of the Application edges from **(B)**.

Vertex according to a source-based partitioning strategy. This mechanism allows to evenly spread the spikes among the receivers. Each of the receivers will implement the synapses for all the postsynaptic neurons of the associated Neuron Vertex, but receive only a portion of the presynaptic spikes. The reason behind this decision is to keep the synaptic rows as long as possible and to keep this approach effective even for unbalanced networks, where the communication might reach only a portion of the postsynaptic neurons. Each Population therefore will contain a predefined number of PyNN Partitions, which can be specified in the description of the network. This number will specify into how many *Neuron* vertices the population will be fragmented. According to the number of PyNN Partitions the presynaptic population has, the postsynaptic population needs the same number of *Synapse* vertices, and each of them will be the recipient of all the spikes coming from the presynaptic PyNN Partition having the same index. In Figure 3.2B, the presynaptic Population has two partitions, which means that each postsynaptic partition has two *Synapse* vertices. The presynaptic partition 1 communicates with all the three postsynaptic partitions A,B and C, but sends spikes only to the synaptic vertex with label 1 of each partition.
PyNN Partition Edges are partitioned similarly, such that each Partition Edge connects a presynaptic PyNN Partition to the correct postsynaptic *Synapse* vertices.

The second level of partitioning generates the Machine Vertices and Edges. The *Neuron* vertices are partitioned according to the standard software toolchain procedure; the afferent *Synapse* vertices will be partitioned to match the *Neuron* vertices and the imposed constraints. An example is shown in Figure 3.2C, where the *Neuron* vertices A and B are both partitioned into two Machine vertices labelled with the L and U subscripts. All the connected *Synapse* vertices are therefore partitioned to match the higher level structure. The same happens for the Edges in order to maintain the connectivity structure defined by the network.

On the SpiNNaker side, a different executable type is loaded according to the type of Vertex (*Synapse* or *Neuron*) running on a core. Figure 3.3, shows how the placement of executable changes from the standard toolchain (top), where single applications simulating both neurons and synapses are placed on a SpiNNaker chip, to the Heterogeneous Programming model (bottom), here simulating *Neuron* cores with four connected *Synapse* cores. Packet communication from presynaptic and postsynaptic populations are indicated by the blue arrows.

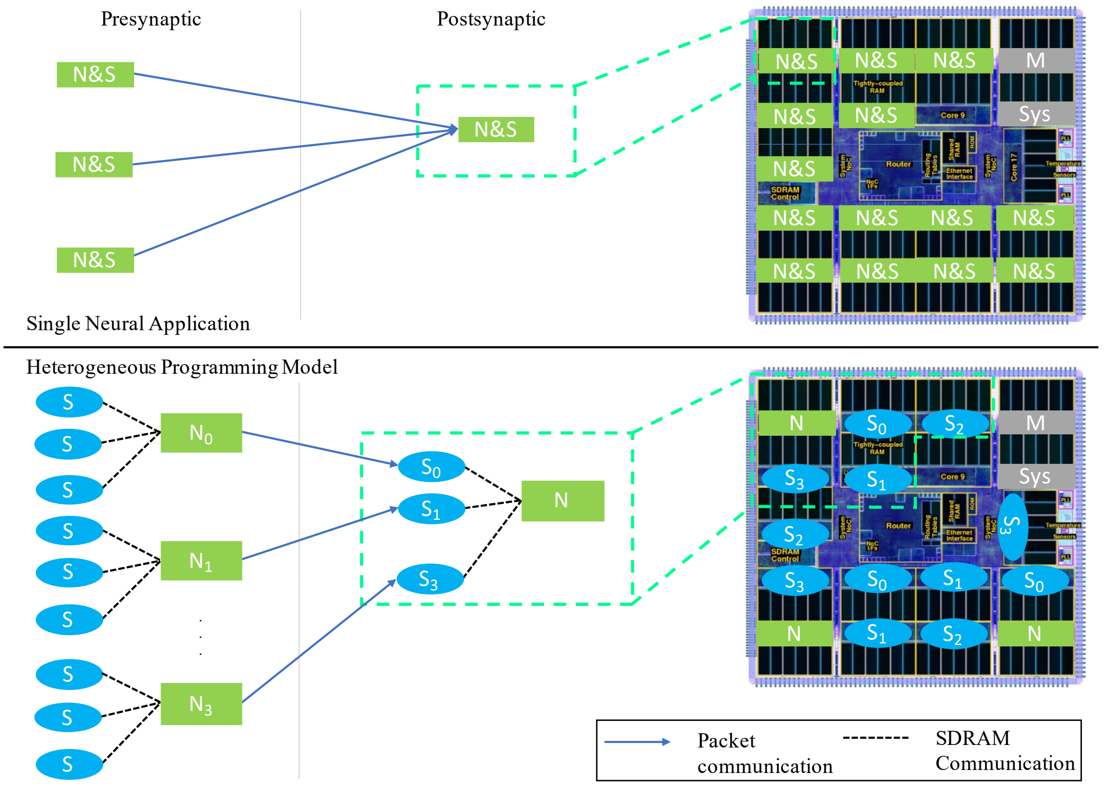


Figure 3.3: Application cores mapping on a SpiNNaker chip: (a) Standard toolchain placement, including simulation cores (N&S) and system cores (grey); (b) Heterogeneous programming model, including *Synapse* cores (S) and *Neuron* cores(N)

The number of neurons per PyNN Partition has a severe impact on the balance of the number of spikes arriving to each *Synapse* core in a postsynaptic population. In order to spread the neurons as evenly as possible, a preprocessing step, depending on the number of neurons and partitions per population, is applied before allocating the vertices for the PyNN Partitioning step. The procedure is described in Algorithm 1.

Algorithm 1: Population Partitioning

```
1 $neurons\_per\_partition = \frac{tot\_neurons}{tot\_partitions}$;
2 $cores\_per\_partition = Floor(\frac{neurons\_per\_partition}{neurons\_per\_core})$;
3 $tot\_neurons\_placed = cores\_per\_partition \times neurons\_per\_core$;
4 $unallocated\_neurons =$
    $tot\_neurons - (tot\_neurons\_placed \times tot\_partitions)$;
5 $additional\_cores = Floor(\frac{unallocated\_neurons}{neurons\_per\_core})$;
6 $remaining\_neurons =$
    $unallocated\_neurons - (additional\_cores \times neurons\_per\_core)$;
```

First, the minimum number of *Neuron* cores per partition is computed (*cores_per_partition*, line 2); this allows to obtain the minimum number of neurons allocated (*tot_neurons_placed*, line 3). The remaining neurons (line 4, *unallocated_neurons*) are then divided by the number of neurons per core and rounded down to compute how many partitions will have an additional *Neuron* core (*additional_cores*, line 5). Finally the remaining neurons are added to the last partition, which is the only one allowed to have a number of neurons not a multiple of the maximum number of neurons per core imposed by the toolchain. This means that the first *additional_cores* (line 5) partitions get *neurons_per_core* more neurons. The last partition gets *remaining_neurons* (line 6) more neurons. This allows the most even distribution of neurons, while meeting the listed constraints.

To allow higher flexibility, the Heterogeneous Programming model allows to connect together populations having different numbers of PyNN Partitions. This maintains the high efficiency of the model while running constrained applications, for example when there are multiple Populations with different partitionings connecting to a single destination population, or when multiple synapse types having different sources are involved. When the presynaptic Population has a lower number of PyNN Partitions, the connectivity follows a one-to-one pattern, leaving some postsynaptic *Synapse* vertices without an incoming Edge. On the opposite situation the PyNN Partitioner loops over the postsynaptic *Synapse* vertices, assigning the presynaptic partitions. This will result in postsynaptic PyNN Partitions having multiple incoming Edges from different presynaptic PyNN Partitions.

3.2.2.2 Memory Operations and Tasks Interactions

Connected *Neuron* and *Synapse* Machine Vertices need to be placed on the same chip, as the communication between them happens via SDRAM. During initialisation, each *Neuron* core allocates a portion of SDRAM that will be used for communication and tags this region with the core ID. This region needs to be large enough to contain the contributions that each *Synapse* core needs to transmit to the connected *Neuron* core. This approach allows to perform a single memory read per *Neuron* core per timestep, reducing the latency caused by multiple memory accesses. At the beginning of the simulation, each *Synapse* core retrieves the memory address of the full contribution region by using the connected *Neuron* core ID and adds an offset which indicates the beginning of its own subregion. Both the connected *Neuron* core ID and the memory offset are provided to the *Synapse* core during the second partitioning phase on the host side.

At the beginning of each timestep, every *Neuron* core, through a single DMA transfer from SDRAM, reads the synaptic contributions (Figure 3.4). Upon termination of the request, the core begins to process the synaptic inputs. This phase consists of reconstructing the input currents, by adding the partial contributions of all the connected *Synapse* cores of the same type sequentially for each neuron and then for each synapse type. With the possibility of having multiple connected *Synapse* cores of the same type, this operation needs to be performed through a loop. Once this process is finished, the *Neuron* cores proceed to update the neuron state, and determine whether to generate a spike or not, by following the standard toolchain procedure and neuron model specifications.

On the synaptic side, at the beginning of each timestep, Timer2 (the second counter available to SpiNNaker cores, as desrcibed in Section 2.5.1) is configured to trigger an event towards the end of the timestep to signal to the core the end of the spike processing phase. The spike processing procedure is similar to the standard software toolchain. The *Synapse* cores receive packets containing

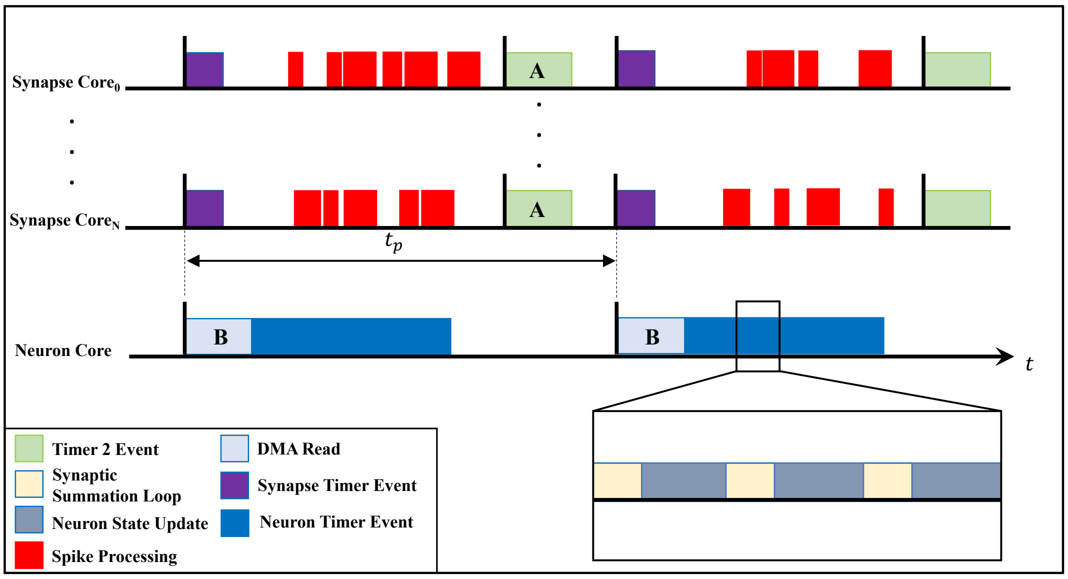


Figure 3.4: Heterogeneous Programming model scheduling.

spikes, which are locally buffered and kickstart the spike processing pipeline if the core was idle. The correct synaptic row will be retrieved from SDRAM and the synaptic events related to each packet will be processed, by adding the weight contribution to the appropriate slot in the synaptic input buffers. When Timer2 triggers its interrupt, a high priority callback is started, which halts the spike processing and flushes all the unprocessed buffered spikes. At this point the synaptic input buffer relative to the next timestep is written to SDRAM through a DMA write operation (see Figure 3.4). The interrupt triggered by Timer2 takes into account the time necessary to write the buffer to memory, plus the contention caused by multiple *Synapse* cores simultaneously writing to memory at this stage (for details about these timings see Section 3.2.3). If there are unprocessed spikes when this event arises, these are flushed, which means that they are lost. This operation is necessary because the information processing for this timestep is concluded with the contribution writing to memory, which means that processing remaining spikes after this point would cause previous spikes to be added into a later timestep, resulting in incorrect delay values. Furthermore this would violate the hard real-time requirement. Therefore all the information which is not fully processed by the Timer2 event is considered lost.

All the cores are synchronised at the beginning of the simulation, and the timer events have been set to have higher priority compared to the standard software toolchain, to guarantee that the timestep update is not preempted by

Synapse Cores Priority Levels	
Spike Packet	-1
Timer and Timer2	0
Spike Processing Pipeline	0

Table 3.1: Priority levels for the *Synapse* cores events.

other tasks and therefore that the simulation remains synchronous. The callback interaction is shown in Figure 3.4, where two simulation timesteps are presented for a generic N number of *Synapse* cores and one *Neuron* core. The purple blocks on the *Synapse* cores represent the synaptic timer event callback, containing the scheduling of the Timer2 event. The red blocks are the spikes arriving on each core. The memory write phase is in light green and labelled **A**, while memory reads on the *Neuron* core are light blue and labelled **B**. All the processing for the *Neuron* core happens in a single callback. The neuron state update phase is highlighted at the right bottom, where the yellow blocks represent the sum of the partial contributions described above, while the grey the neuronal variables update. The size of the yellow blocks changes according to the number of *Synapse* cores contributing to each *Neuron* core and, as shown in Section 3.2.3, it has a significant impact on the duration of the Neuron timer callback.

The events priorities for the *Synapse* cores are detailed in Table 3.1. The priority levels are those defined by the SpiNNaker toolchain. As described in Section 2.5.2.2, a priority level of -1 is the highest and handled as FIQ; priority level 0 is handled as IRQ and can be preempted only by an event with priority -1. Finally priority levels > 0 are handled as soft interrupts. The highest priority event is the reception of a spike packet and this is inherited from the standard software toolchain. Events triggering the spike processing pipeline have the same priority as timer events, however the latter are assigned to a higher priority slot in the VIC, meaning that, if the spike processing pipeline is active when a Timer2 event is received, when the currently processed spike is completed, the Timer2 event is executed before processing any other spike. The scheduling of Timer2 takes into account the possibility of a higher spike input activity; this means that Timer2 events will be triggered early enough to ensure their completion before the end of the timestep, preserving real-time operation.

3.2.3 Model Benchmarking

The most useful metric that can be used to determine how well the Heterogeneous Programming model performs is the synaptic events throughput. This is measured in terms of processed synaptic events per timestep. A synaptic event corresponds to one spike innervating one synapse. By defining a mathematical description of this metric under the Heterogeneous Programming model, it is possible to determine the best fit for each application.

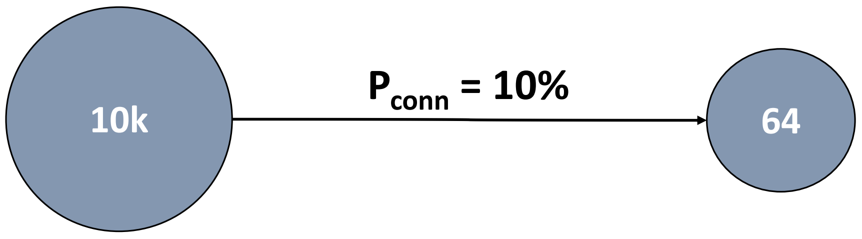
The mathematical model indicating the maximum number of synaptic events that can be processed per timestep evolves from that presented in Equations 2.7 and 2.8 for the standard SpiNNaker toolchain, and is defined by Equation 3.1.

$$E = [\frac{t_p - t_{1^{st}} - t_{last}}{t_{spike}} + 2] \times P \times N \tag{3.1}$$

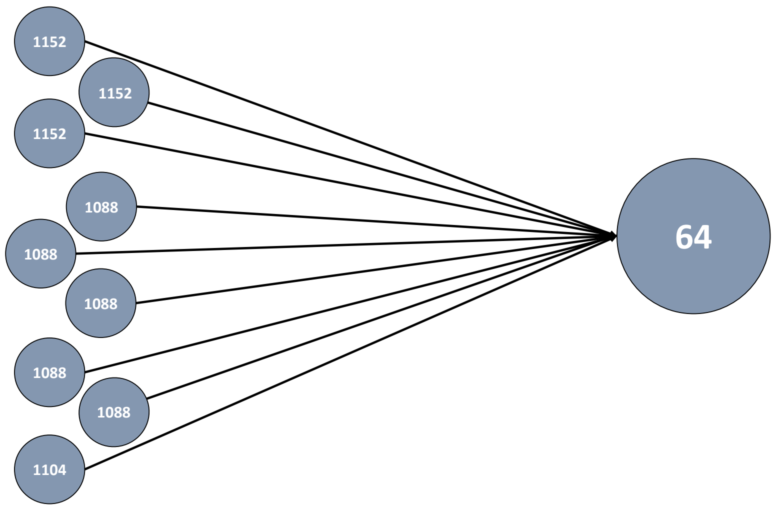
$$t_p = \Delta t - t_w - t_r \tag{3.2}$$

E is the total number of synaptic events per timestep, t_P is the spike processing window, and is obtained by subtracting from the timestep duration (Δt) the time required for the *Synapse* cores to write the synaptic contributions (t_w), minus the time taken by the postsynaptic *Neuron* core to read the contributions from shared memory (t_r). Analogous to the standard toolchain case, $t_{1^{st}}$ and t_{last} are the amounts of time required to process the first and last spike respectively, which are different from the other spikes due to the pipelined spike processing approach. This quantity is divided by the time required to process a single spike (t_{spike}) and incremented by 2, to account for the two spikes previously subtracted, resulting in the maximum number of spikes that can be processed in a single timestep. The number of synaptic events is given by multiplying the number of spikes by the connectivity probability (P), and then by the number of postsynaptic neurons on the *Neuron* core of the ensemble (N). The read and write times are calculated through experimental evaluation and are detailed in Sections 3.2.3.1 and 3.2.3.2 respectively. The value described in Equation 3.1 represents the number of processed synaptic events per *Synapse* core, therefore the number of synaptic events per ensemble (or per *Neuron* core) is obtained by adding together the values for each *Synapse* core in the ensemble.

In order to test the effectiveness of the Heterogeneous Programming model, a configurable testbench SNN was defined, as shown in Figure 3.5(a). The presynaptic population contains 10000 neurons and the postsynaptic population 64



((a))



((b))

Figure 3.5: Schematic of the test network. (a) Shows the full network; each neuron in the source population has 10% probability to connect to each neuron in the destination population. (b) Shows the partitioning of the network when using 9 *Synapse* cores and 64 neurons per core. According to Algorithm 1, each partition has 1088 neurons, the remaining 208 can be placed on 3 full cores with a remainder of 16 neurons, therefore the first three partitions have 64 additional neurons (for a total of 1152, which results in an additional core per partition), and the remaining 16 neurons are included in the last partition.

(this is the maximum number of neurons that can be simulated per *Neuron* core in real-time using 0.1 ms timesteps, as shown in Section 3.2.3.1). The Projection is implemented through a fixed probability connector with 10% connectivity, meaning that each neuron in the source Population connects with a neuron in the destination with 10% probability. Some presynaptic neurons are initialised with the internal voltage above threshold. This allows those neurons to fire during the first timestep. Each spike is delivered with 1 timestep delay. This initialisation allows measurement of the neuron state update times in all three possible scenarios:

- idle neurons;
- spiking neurons;
- refractory period dynamics;

The number of source spikes is set such that the *Synapse* cores are saturated (i.e. they are busy processing spikes for the duration of a whole simulation timestep). In order to explore peak throughput, neurons are chosen at random within the presynaptic Population to ensure that spikes are evenly distributed throughout the *Neuron* cores state update, replicating the random distribution of biologically-plausible models [PD12]. The presynaptic Population is partitioned according to Algorithm 1, assuming a limit of 64 neurons per core, and the number of neurons per Partition in the presynaptic population is presented in Figure 3.5(b), showing the optimal spread when using 9 partitions. Each partition contains at least 1088 neurons. The remaining 208 neurons can be placed on 3 cores with a remainder of 16 neurons. Therefore each of the first three partitions received 64 additional neurons (resulting in 1152 neurons per partition and therefore 1 more *Neuron* core) and the final partition will contain the remaining 16 neurons (with a total of 1104 neurons). The first 8 partitions therefore contain multiples of 64 neurons, providing a Machine graph with complete cores, the remainder is added to the last partition, which is the only one allowed to have a core with fewer than 64 neurons. This approach grants maximum efficiency on the machine side, allowing the partitioner to assume that all the cores are complete, therefore simplifying calculations such as indices among different cores and ensuring that all the transfers are memory aligned, so that there is no need to add padding. According to previous work [RBB+18], the number of synaptic events which can

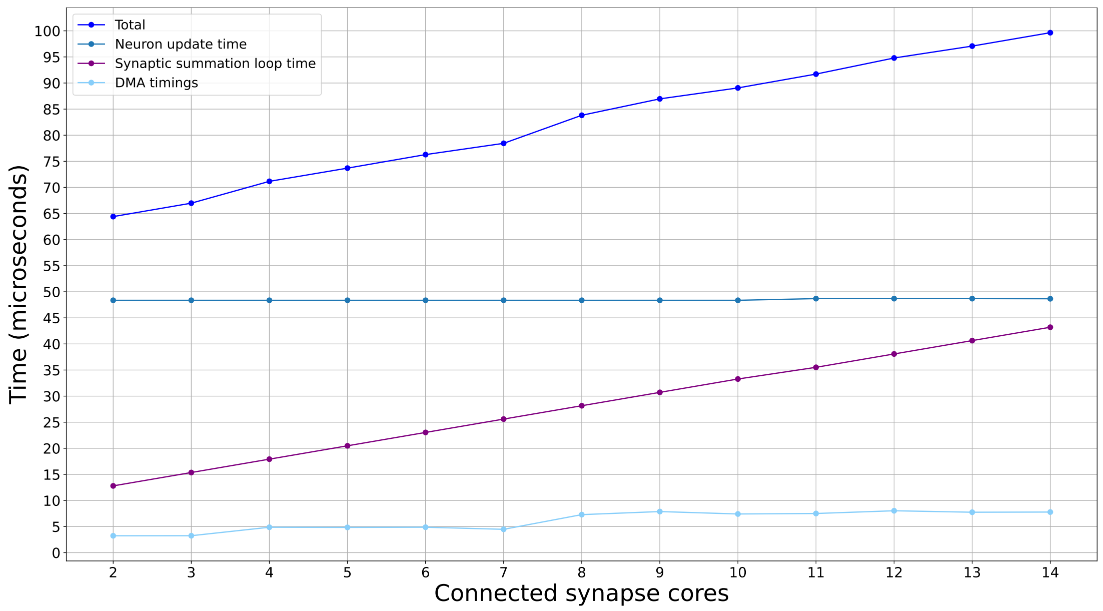


Figure 3.6: The *Neuron* core performance: DMA read time; synaptic summation loop time; neuron state update; and total update time with increasing numbers of *Synapse* cores.

be processed per timestep, simulating 64 neurons per core and having an incoming connectivity probability of 10% is ≈ 1300 for 1 ms timesteps (corresponding to ≈ 130 when using 0.1 ms timesteps). Assuming the case in which the number of processed synaptic events per timestep increases linearly with the number of partitions, the test has been designed to generate a number of spikes which is higher than these values at every iteration, with any excess spikes left unprocessed and discounted from throughput profiling results.

3.2.3.1 *Neuron* Core Profiling

The time required to update the neuron state is independent from the duration of the simulation timestep; the only difference is that, with a smaller simulation Δt, more neural updates are performed in a given simulation time. The presented results are extracted from the *Neuron* core implementing the postsynaptic population from Figure 3.5. The number of *Synapse* cores connected to a *Neuron* core affects its performance, since the *Neuron* core has to perform a larger DMA transfer each timestep to retrieve the synaptic contribution of each *Synapse* core and then sum these contributions for each simulated neuron.

Figure 3.6 shows timing measurements for different stages of the neuron state

update phase, and how these vary with the number of *Synapse* cores. The horizontal axis contains the number of *Synapse* cores connected to the *Neuron* core, while the vertical axis the time, measured in microseconds. The light-blue line at the bottom shows the DMA read time, therefore the time required to retrieve the memory block containing all the synaptic contributions for all the implemented neurons on the core for the current timestep. The efficiency of performing a single DMA transfer for the whole block can be seen, as the read time remains constant for high numbers of *Synapse* cores, with a plateau of 8 μs. This time, added to that allocated to *Synapse* cores to write the same values to memory (10 μs), lies between 13 μs with 2-3 *Synapse* cores and 18 μs with 14 *Synapse* cores every timestep. This time interval however, cannot be used for neural computation, to ensure data correctness, and represents a significant proportion in the case of 0.1 ms timesteps.
The purple line represents the synaptic summation loop time: this is the time that the *Neuron* core takes to sum the synaptic contributions from the *Synapse* cores for all 64 neurons. This number increases linearly with the number of *Synapse* cores up to 43 μs, representing again a large fraction of the timestep in the case of 0.1 ms timesteps. The central blue line is the effective neuron update time, for all the 64 neurons, including all the necessary operations to update the neuron state variables (measurements were performed using LIF neuron models with current-based synapses) and it is, as expected, constant at 48 μs.
The dark blue line (on top) represents the total update time, therefore it is a sum of all the previous contributions. This line is useful to indicate when the last neuron on the core will spike. With a large number of *Synapse* cores, it reaches 100 μs. This is problematic for 0.1 ms timestep simulations, since the *Synapse* cores cannot receive these spikes before the time limit for the DMA write is reached, meaning they will never be processed. For this reason, to achieve real-time execution using 0.1 ms timesteps, it is necessary to limit the number of *Synapse* cores per *Neuron* core (as further detailed in Section 3.2.3.2). These values show that a limit of 64 neurons per core is the best choice for 0.1 ms timesteps simulations, 64 being the highest power of 2 that can meet the real-time requirements. 128 would not be a feasible choice as the neural state update phase would require 96 μs, excluding the reading time and the synaptic summation time.

Using the measured values it is possible to obtain the reading time (t_r) shown in Equation 3.1. This value depends on the number of *Synapse* cores in the ensemble and is expressed by Equation 3.3. This dependency is due to the increase in size of the memory block that is read by the *Neuron* core every timestep.

$$t_r = 0.3 \times S_{cores} + 3.93 \tag{3.3}$$

3.2.3.2 *Synapse* cores Profiling

Regarding *Synapse* cores profiling, the important metrics are the synaptic contributions writing times and the synaptic events processed per timestep. These values are reported in this section, distinguishing between processed synaptic events per timestep with 1 ms and 0.1 ms timestep simulations.

As described in Section 2.5.1 and Section 3.2, each SpiNNaker chip has a single access channel to SDRAM, meaning only a single core at a time can perform a DMA transfer. The access to memory is tree-based, causing different cores to have different access times and performance according to their position. This means that cores belonging to branches with more access requests experience longer access time. By increasing the number of *Synapse* cores, the likelihood of memory contention becomes much higher. Each *Synapse* core, implementing synapses for 64 postsynaptic neurons, has to write 64 $\times$ 16 bits = 128 Bytes per timestep, as the size of the input contributions for a single postsynaptic neuron is 16 bits. In order to define how this impacts the simulation, DMA times were profiled from 2 to 14 contending cores, and the results are shown in Figure 3.7.

Each value represents the worst of 128 tests, and is taken from the core having the longest access time. This is important, since it is not possible to predict which cores the application will be assigned to, as the assignment is automatically performed by the software toolchain to take into account failed cores. Therefore a worst case time allocation becomes necessary. Figure 3.7 shows that, in the worst measured case (in blue), the DMA write is processed in less than 7 μs, which happens with 14 contending cores. This means that *Synapse* cores can set the Timer2 event to be triggered 10 μs before the end of the timestep, being sufficiently conservative to allow the current spike under processing to be completed and to perform the memory write on time.

Through these measurements it is possible to obtain the writing time (t_w) shown in Equation 3.1. This value, which depends on the number of *Synapse* cores in the ensemble, is expressed by Equation 3.4 and plotted in orange in Figure 3.7. This dependency is due to the growing contention when multiple *Synapse*

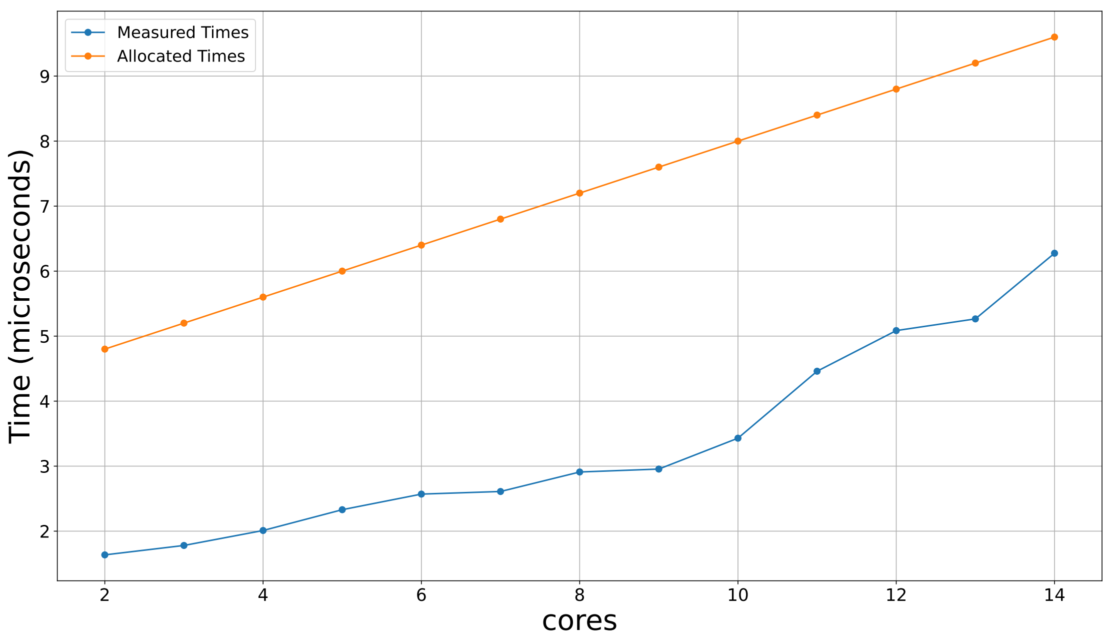


Figure 3.7: DMA write latency: worst case measured write time when writing 128 Bytes to SDRAM simultaneously from different numbers of cores on the same chip in blue. Allocated DMA writing time for the synaptic contributions, according to Equation 3.4 in orange.

cores try to write to shared memory simultaneously at the end of each timestep. This relation is obtained through linear interpolation between the first and last values presented in Figure 3.7 with a constant increase of 3.1 μs. This added quantity represents the time necessary to process a full spike when simulating 64 neurons per core [RBB+18, RPR+19]. The choice of limiting the interpolation to the first and last values allows to keep the writing times above each measured value, and at the same time simplifies the allocation of writing times according to the number of *Synapse* cores.

$$t_w = 0.4 \times S_{cores} + 4 \tag{3.4}$$

Throughput profiling results for the *Synapse* cores when simulating with a 1 ms time scale resolution are shown in Figure 3.8.

The plot presents error bars measured over 4 executions of the same network, on a different SpiNNaker machine allocation, in order to prove the robustness of the approach.

The number of *Synapse* cores connected to the *Neuron* core in the postsynaptic population is increased from 2 to 14, in order to test the allocation of

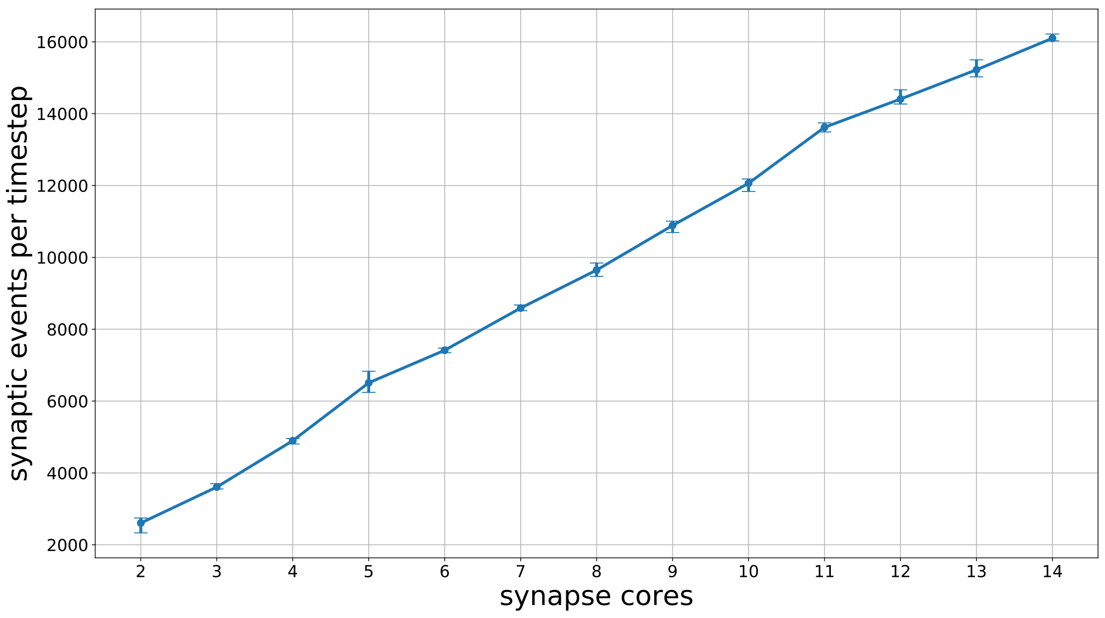


Figure 3.8: Processed Synaptic Events per timestep for 2 to 14 contributing *Synapse* cores using 1 ms timesteps

a complete chip. The quantity measured is the number of synaptic events per timestep. As outlined in Section 2.6, one spike packet on SpiNNaker carries synaptic events equal to the number of connections that the presynaptic neuron that generated the spike has on the postsynaptic core.

This metric is more meaningful than measuring the number of processed spikes, as it gives a precise idea of the number of updated synapses per timestep. The number of processed synaptic events shown in Figure 3.8 grows linearly with the number of *Synapse* cores, reaching up to 16000 processed synaptic events, representing an increase of 12.3× compared to previous work [RBB+18]. The plot presents some regions where the rise in processed synaptic events seems to be smaller; this is due to a different allocation of processors, which causes different DMA timings, affecting the number of generated and processed spikes.

Figure 3.9 presents the processed synaptic events per timestep with 0.1 ms timesteps. In this case, the same network has been used, however the number of generated spikes has been scaled down by a factor of 10, to represent the number of expected spikes in the reduced time frame.

Because of the time necessary to update the neuron state (more specifically the growing synaptic summation loop, shown in Figure 3.6), with more than 8 *Synapse* cores per *Neuron* core it is not possible to run the simulations with

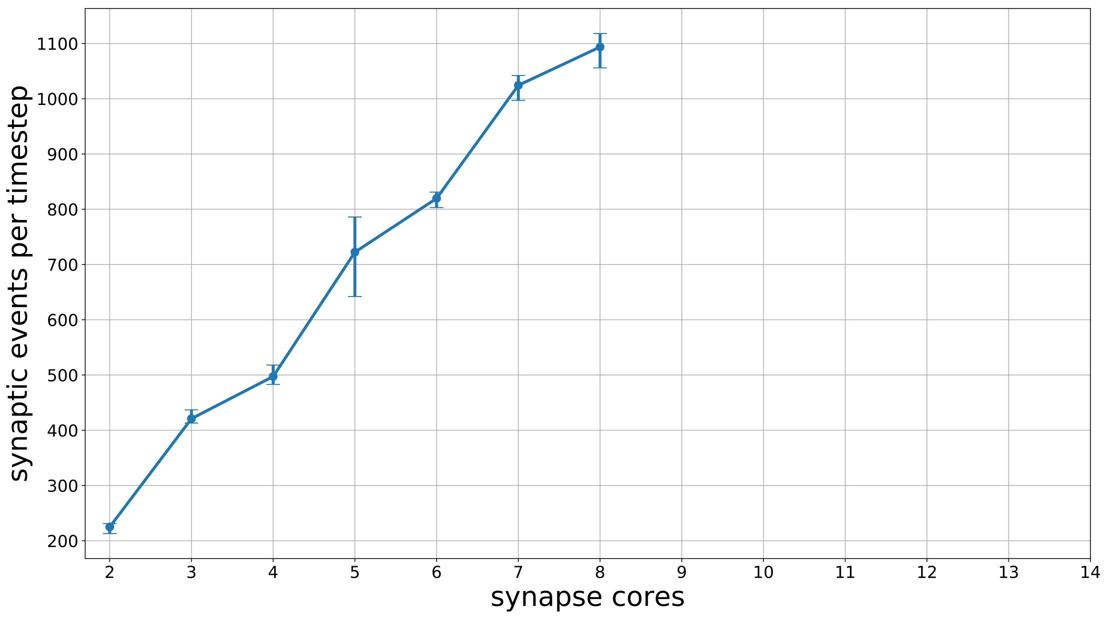


Figure 3.9: Processed Synaptic Events per timestep for 2 to 8 contributing *Synapse* cores using 0.1 ms timesteps

0.1 ms timesteps. This is because the computation that *Neuron* cores have to perform during the timestep exceeds the length of the timestep itself, failing to fulfill the real-time requirements. For this reason, only the synaptic events with up to 8 *Synapse* cores per *Neuron* core have been reported here. Similarly to the 1 ms timesteps case, the plot presents error bars over 4 executions of the same network over a different SpiNNaker machine allocation. The results achieved here show a linear increase of processed synaptic events per timestep when increasing the connected *Synapse* cores, with the highest number of synaptic events at 1100 when using 8 *Synapse* cores.

These profilings allow to make predictions on the required machine according to the application that needs to be simulated. High fan-in networks require a larger number of *Synapse* cores, in order to be able to process all the incoming spikes. On the other hand, increasing the number of *Synapse* cores not only increases the memory contention, which from the *Neuron* core's perspective means generating output spikes later which are therefore delivered later, but also increases the size of the machine involved in the simulation, resulting in higher power consumption. By comparing the processed synaptic events between 1 ms and 0.1 ms simulations it can be observed that, in some cases, the efficiency of the *Synapse* cores when using 1 ms timesteps appears to be lower than that with

0.1 ms (the 1 ms throughput appears to be $\approx 9\times$ 0.1 ms, with a $10\times$ longer timestep). This can be explained by the selection of presynaptic firing neurons. The tests performed for the 1 ms cases resulted in a much higher quantity of spikes delivered to the postsynaptic cores, compared to the ensemble processing capability. The number of generated spikes was as close as possible to the saturation limit of the routing infrastructure. The same situation did not happen for the 0.1 ms case, as the saturation limit at routing level appears to be closer to the *Synapse* cores processing limit. As it can be observed in Figure 3.8, when the input activity is much higher than what the cores in the ensemble can handle, this results in performance degradation, as *Synapse* cores have to spend additional time buffering incoming packets that won't be processed due to timings constraints. It is however challenging to predict the maximum input activity which allows *Synapse* cores to achieve the highest throughput for every case, therefore an approach which generated spikes below the routing saturation point has been chosen.

Section 3.3 shows the use of the Heterogeneous Programming model applied to a biologically-representative SNN, in which the number of *Synapse* cores per ensemble is limited to 3 and demonstrates a performance gain of $20\times$ compared to previous published results [vARS+18, RPR+19].

3.3 Real-Time Simulation of the Cortical Microcircuit Model

This section focuses on the application of the Heterogeneous Programming model to a real-world use case. First a description of the problem is presented, together with the details of the network and why this is considered a benchmark by the neuroscience community, then the improvements and adaptations to the Heterogeneous Programming model to solve the problem are described, followed by the results achieved.

3.3.1 The Cortical Microcircuit Network

The Cortical Microcircuit Model was developed by T. Potjans and M. Diesmann in 2012 [PD12]. This is the first model to present full density of connectivity of 1 $\mathtt{mm}^2$ of generic early sensory cortex of the mammalian brain. The model

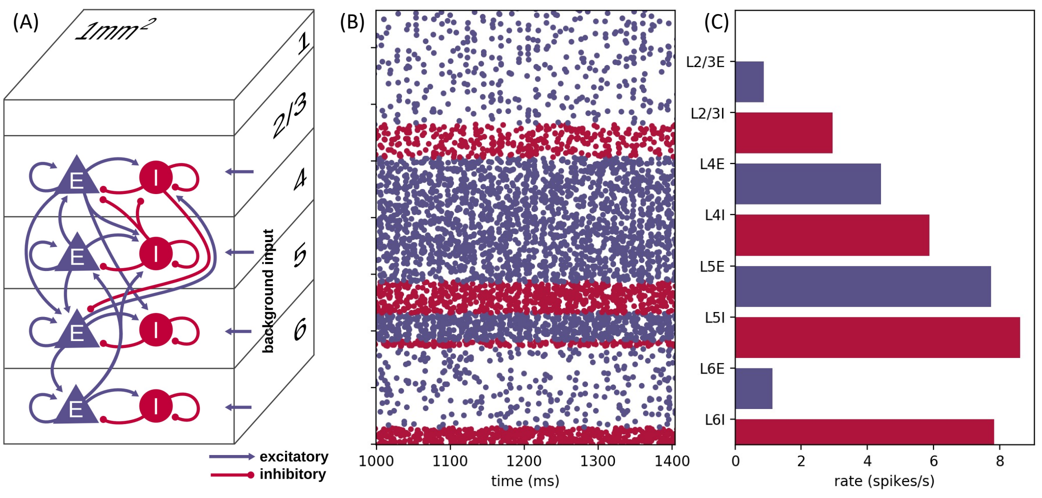


Figure 3.10: The cortical Microcircuit: **(A)** The network structure, including layers and respective populations; **(B)** 400 ms of activity at steady state; **(C)** Firing rates per population. Reproduced from [RPR+19] with permission.

combines data from rat primary visual and somatosensory areas and cat area 17 [PD12] and is the smallest network size where a realistic number of synapses and a realistic connection probability are simultaneously achieved. Larger cortical models are less densely connected, with a limited increase in the number of synapses per neuron for increased model size [vARS+18]. These characteristics make the network an accepted standard across the field of computational neuroscience. Furthermore, this network represents a good test case for comparison between different simulators as it produces typical firing rates and peak synaptic fan-in and fan-out experienced in the brain and sets constraints on communication, available synaptic memory and processing speed [RPR+19].

The network model is available in the PyNN specification language [DBE+09], providing abstraction between the network specification and the hardware/software implementation, facilitating comparisons between different simulation platforms. Connections are defined according to probabilities, instead of being hand-tuned. Each probability represents the distance between two populations (the higher, the closer).

The network structure, together with firing rates, is shown in Figure 3.10. The network consists of 77,169 current-based Leaky Integrate-and-Fire (LIF) point neurons, connected through 3×10^8 synapses. Synapses are modeled as current-based with exponential decay and are static (i.e. no plasticity is modelled). The

Populations Parameters								
Size								
Name	L2/3e	L2/3i	L4e	L4i	L5e	L5i	L6e	L6i
	20683	5834	21915	4579	4850	1065	14395	2948
Connectivity								
	L2/3e	L2/3i	L4e	L4i	L5e	L5i	L6e	L6i
L2/3e	0.101	0.169	0.044	0.082	0.032	0.0	0.008	0.0
L2/3i	0.135	0.137	0.032	0.052	0.075	0.0	0.004	0.0
L4e	0.008	0.006	0.050	0.135	0.007	0.0003	0.045	0.0
L4i	0.069	0.003	0.079	0.160	0.003	0.0	0.106	0.0
L5e	0.100	0.062	0.051	0.006	0.083	0.373	0.020	0.0
L5i	0.055	0.027	0.026	0.002	0.060	0.316	0.009	0.0
L6e	0.016	0.007	0.021	0.017	0.057	0.020	0.040	0.225
L6i	0.036	0.001	0.003	0.001	0.028	0.008	0.066	0.144

Table 3.2: Cortical Microcircuit Parameters

neurons are organised into 8 populations (4 excitatory and 4 inhibitory) spread over 4 layers (L2/3, L4, L5, L6). Connections are classified into 3 main groups: recurrent, intralayer and interlayer connections. Multiple connections between two neurons are allowed. The network topology is presented in Figure 3.10(A). Furthermore, each population receives background stimulation, which represents connections from adjacent cortex, other cortical areas and subcortical regions. This additional input is either represented as Poisson input sources whose firing rates depend on the target population or as direct currents injected into the neurons, corresponding to the mean current generated by the Poisson sources per population. The synaptic weights and time constants are chosen such that an average excitatory postsynaptic potential has an amplitude of 0.15 mV with a rise time of 1.60 ms and a width of 8.80 ms, mimicking in vivo measurements [PD12]. To introduce heterogeneity into the network, the synaptic weights are drawn from Gaussian distributions, with mean $\pm$ standard_deviation equal to 87.80 ± 8.78 pA for excitatory source neurons and 351.20 ± 35.32 pA for inhibitory source neurons (except for connections from L4 to L2/3, which have weights 175.60 ± 8.78 pA). Transmission delays are similarly distributed, but truncated to the nearest simulation timestep, with parameters 1.50 ± 0.75 ms for excitatory sources and 0.75 ± 0.37 ms for inhibitory sources. The simulation timestep is $\Delta t = 0.1$ ms and the model is simulated for 10 s of activity. Populations sizes and connectivity probabilities are summarised in Table 3.2.

3.3.2 Challenges and Previous Works

Simulations of the Cortical Microcircuit Model set important challenges common to large-scale biologically-representative SNNs. By understanding and addressing these challenges, it is possible to find the best fit for such models on Neuromorphic hardware and to pave the way for the design of future Neuromorphic chips with the target of real-time biologically-plausible SNN simulations. Furthermore, successful real-time simulations of the model can lay the foundations for scaling up and modelling additional and more complex brain regions in the future.

The Cortical Microcircuit is a computationally intensive network with tight timing constraints. As described in Section 3.3.1, the time resolution is sub-millisecond, meaning that a hard real-time simulation requires all the processing to complete in the time scale of 0.1 ms. This includes handling the synaptic inputs, updating the state for all the neurons and generating the action potentials.

Different flavours of real-time performance can be achieved, namely hard real-time and soft real-time simulations. In this context the target is hard real-time, which mandates that all the information is processed within the boundaries of the timestep. However, some simulation platforms allow a more relaxed constraint achieving soft real-time performance, meaning that the overall simulation completes within biological real-time, but timesteps are allowed to overrun, resulting in time intervals having different lengths. This means that, when the computational load is higher, timesteps run for longer in order to process all the information, this is then averaged out on shorter timesteps when the activity is lower.

The large fan-in makes meeting hard real-time requirements even harder. Because of network oscillations, instantaneous rates produced by the network tend to vary, increasing the complexity of a hard real-time simulation, compared to a soft real-time execution, where it is sufficient to handle mean spike rates.

An example of this is shown in Figure 3.11. Here the total number of spikes generated during 212 ms of simulation is presented. Since the connectivity in the Cortical Microcircuit obeys Dale's Law [SH99], which states that each excitatory neuron will only create excitatory connections, and inhibitory neuron inhibitory connections, excitatory and inhibitory spikes generated per timestep are also presented. The right inset of Figure 3.11 shows the variation in instantaneous and mean activity. While mean excitatory and inhibitory activity correspond respectively to 17 and 8.3 spikes per timestep, the instantaneous activity reaches

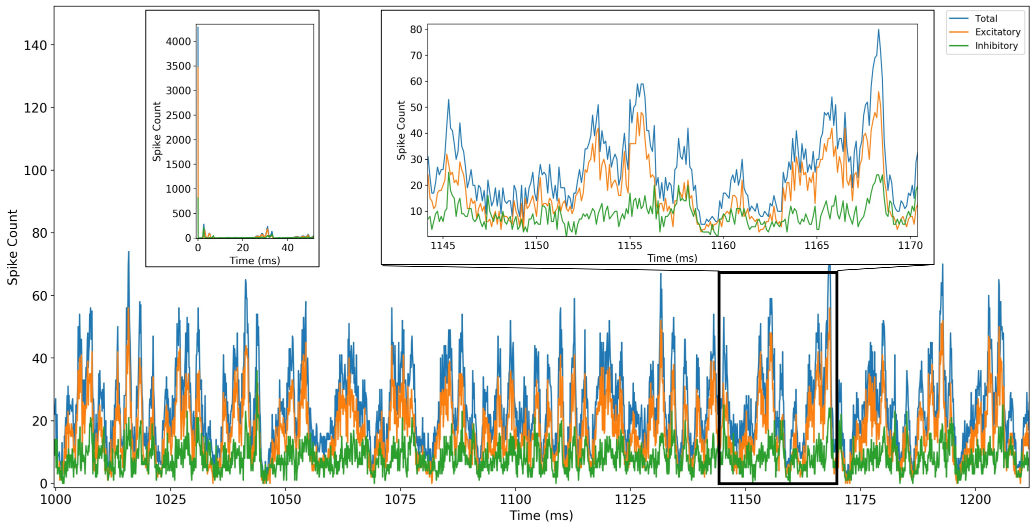


Figure 3.11: The cortical Microcircuit output activity. Reproduced from [RPR+19] with permission.

peaks of 55 excitatory spikes and 38 inhibitory spikes per timestep, making the worst case scenario for a hard real-time simulation 3× worse than average firing activity. This analysis does not include the initial transient response of the model (Figure 3.11 left inset), where, because of an above-threshold initialisation of the neurons for the initial conditions, the network generates more than 4000 spikes in the first timestep. This high activity however is not meaningful for the network simulation, as the network quickly converges to steady-state behaviour, and this initial transient does not represent meaningful biological activity.

The Cortical Microcircuit Model has been previously simulated on SpiNNaker [vARS+18], through the standard software toolchain described in Section 2.5.2. The number of neurons simulated per core was 80, as this was the smallest number of neurons which still allowed to generate routing keys such that routing tables would fit on the machine under the toolchain implementation at the time. The activity in the SNN gave limitation on the maximum simulation speed, which required a 20× slow-down factor compared to real-time. Although this constituted a breakthrough, resulting in SpiNNaker being the first Neuromorphic platform able to simulate the Cortical Microcircuit model, performance was significantly lower than competitors running on a HPC cluster through the NEST simulation software [GD07], which were able to achieve a complete simulation 3× slower than real-time [vARS+18]. GPU-based simulators, through the use of customised software for SNNs simulations [YTN16], were able to achieve even better results, performing simulations 2× slower than real-time [KN18].

Similarly, energy consumption figures were not in favour of the SpiNNaker platform, which was able to achieve 5.9μJ per synaptic event compared to the 5.8μJ registered on HPC or even worse, with the 0.47μJ achieved by GPU simulators. These numbers were caused by the spread of the simulation over 6 boards, which caused the baseline power to be amortised across many fewer synaptic events, combined with the large slowdown factor, which forced the system to be powered on for a much longer time.

The porting of the Cortical Microcircuit model to SpiNNaker inevitably added additional challenges in adapting the mapping of the network to the architecture. The first challenge is caused by the background input stimulation, as described in Section 3.3.1. This input can either be directly injected into the neurons as mean DC current, or delivered as spike trains from Poisson sources through the synapses. The latter version, because of an intensive background activity, results in an increased input activity, which, in the worst case, results in 300 additional incoming spikes per timestep [vARS$^+$18, RPR$^+$19]. Furthermore, additional cores are necessary to simulate the Poisson sources, increasing the size of the required machine. Another challenge is given by how synaptic delays are handled within the SpiNNaker system. As described in Section 2.6, each application core keeps a synaptic input buffer, which allows it to store the synaptic contributions with delays greater than 1 simulation timestep. However, the toolchain generally limits the number of delay slots to 16, for memory reasons. This means that spikes with synaptic delays over 16 timesteps (corresponding to 1.6 ms), need to be routed through delay extension cores [RBD$^+$19], which buffer the packets for the required amount of time, as further detailed in Section 2.6. This aspect not only increases the number of cores required to run the simulation, but also generates additional network traffic, since packets need to be delivered both to the delay extension cores and the postsynaptic cores, as not all the synapses require such a delay. This reduces efficiency, since the number of postsynaptic targets per spike is also reduced resulting in even shorter synaptic rows, due to the intermediate step added by delay extension. Furthermore, as the synaptic connections are defined according to probability of connectivity, it is not possible to perform simplifying assumptions based on distance-dependence and proximity to improve performance during the synaptic matrix generation and routing phases. This means that routing must be performed on an all-to-all basis, resulting in an even higher incoming traffic.

3.3.3 Real-Time Simulation

This section focuses on the modifications to the Heterogeneous Programming model to achieve hard real-time simulations of the Cortical Microcircuit network on SpiNNaker. These changes reflect the challenges presented in section 3.3.2.

The main requirement for hard real-time simulations is the capability of processing the neural state updates and the incoming synaptic events within the timing constraints. The biggest challenges in this can be found in the peak spike rates shown in Figure 3.11, which need to be combined with the packets coming from the delay extension cores and the Poisson sources to get full figures of the incoming packets. The traditional software toolchain is not suited to handle such an intensive incoming traffic in such a tight time resolution. Single Application cores cannot process all the incoming spike packets while they are busy updating neural states, therefore they spend most of the timestep performing context switches or waiting for memory to respond. This is also exacerbated by the fact that there are too many incoming spike packets to be processed per timestep. For this reason, reducing the number of neurons per core is not a viable solution. Since API activities such as the spike packet reception dominate the processing time, a reduced number of neurons per core would reduce the spike processing efficiency, because the fixed cost of turning a spike into neural input would be amortised over fewer individual neuron contributions [RBB+18]. Furthermore, a reduced number of neurons per core does not mean fewer incoming spikes, as the neurons' fan-in does not change. The Heterogeneous Programming model provides a different approach to the problem by dedicating some cores to the synaptic processing and others to the neural state update. The approach used for the Cortical Microcircuit uses a fixed number of *Synapse* cores per *Neuron* core. An example allocation of the Heterogeneous Programming model targeting the Cortical Microcircuit on a SpiNNaker chip is shown in Figure 3.12. Each Synapse processor implements a single synapse type. The partitioning is source-based as described in Section 3.2. The excitatory activity of the Cortical Microcircuit model is double that of the inhibitory (see Figure 3.11). For this reason the number of excitatory *Synapse* cores per *Neuron* core is twice that of the inhibitory cores. This results in allocating two excitatory *Synapse* cores (the blue $\mathtt{S}_L$ and $\mathtt{S}_U$ in Figure 3.12) and one inhibitory *Synapse* core (the red S in Figure 3.12) per

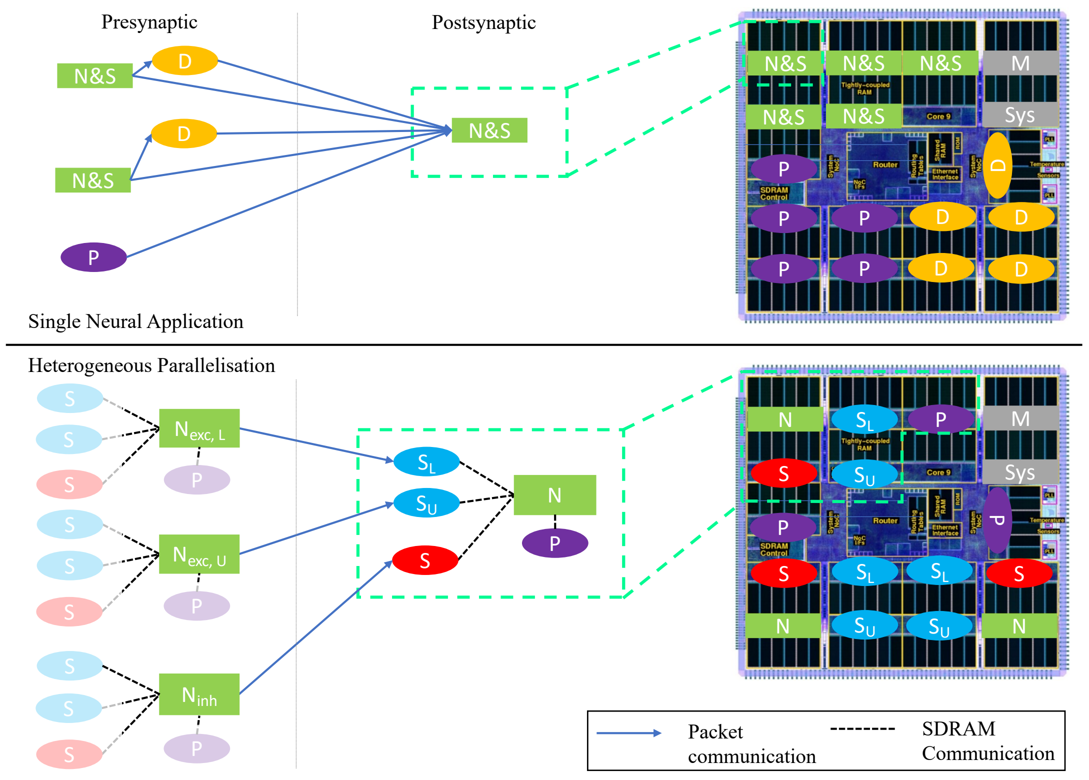


Figure 3.12: Application cores mapping on a SpiNNaker chip for the Cortical Microcircuit case: (a) Standard toolchain placement, including delay extension cores (D), Poisson sources (P) and simulation cores implementing neurons and synapses (N&S). Grey cores are used for system purposes; (b) Heterogeneous programming model applied to the Cortical Microcircuit, including *Synapse* cores (S), *Neuron* cores(N) and Poisson cores(P). Reproduced from [RPR^{+}19] with permission.

Neuron core. Each of the two excitatory *Synapse* cores will be the target of one of the two halves of the presynaptic population. This type of partitioning is the most effective way to spread the number of spikes over destination *Synapse* cores, as all the neurons belonging to the same presynaptic population are equally likely to spike. This also means that all spike packets no longer need routing to every receiver, but, instead, causes excitatory and inhibitory spikes to be split among the corresponding destination *Synapse* cores and excitatory packets to be split again between the upper and lower destination excitatory *Synape* cores.

The *Synapse* cores have now an increased local memory, since the structures containing the neural state variables are locally stored into *Neuron* cores' DTCMs. This means that it is possible to maintain additional synaptic input buffers locally, eliminating the need for delay extension cores. Through the use of independent

Synapse cores, it is possible to store 255 delay slots in local memory, which allows to accumulate synaptic inputs for delivery over the next 25.5 ms, using 0.1 ms timesteps. This quantity allows to simulate any delay present in the network. By employing this approach it is possible to remove all the additional traffic caused by the delay extension cores, together with eliminating the allocation of these cores themselves.

Other elements heavily contributing to the incoming peak activity are the Poisson source generators. These, on the standard software toolchain, are represented as independent cores generating spikes according to Poisson processes running on them. This, as outlined in section 3.3.2, increases the traffic through the Network-on-Chip and puts additional pressure on the postsynaptic cores, which need to process additional incoming spikes. To address this issue, this version of the Heterogeneous Programming model, employs local Poisson source generators, which communicate to *Neuron* cores in the same way as *Synapse* cores. These can be called *Poisson* cores and are part of the main building block of the model employed here, as shown in Figure 3.12.

The main building block of this version of the Heterogeneous Programming model is, therefore, an ensemble of 5 cores: lower and upper excitatory *Synapse* cores, an inhibitory *Synapse* core, a *Poisson* core and a *Neuron* core. This ensemble communicates through shared memory, using SDRAM, therefore all the cores need to reside on the same SpiNNaker chip. This allows to store 3 ensembles per chip. A schematic of the updated callback interactions is shown in Figure 3.13, where again 2 timesteps of execution are shown, here detailing the activity of a *Neuron* core, a *Poisson* core and 2 *Synapse* cores (1 excitatory and 1 inhibitory).

For the *Neuron* cores, consistently with the standard Heterogeneous model, the timestep begins with a DMA read (transfer D in Figure 3.13), in order to retrieve the synaptic contribution that will constitute the input currents. The block of contributions is retrieved in a single transfer and contains all the input values from all the *Synapse* cores of the ensemble and from the *Poisson* core. After the completion of this operation, the neurons are updated in a sequential fashion by adding together the two excitatory contributions and the Poisson background and by subtracting the inhibitory value. If the model mandates it, spikes are then generated. Finally the state for each neuron is recorded and written back to the dedicated memory region (transfer E). The neuron state update phase is critical, as the last output spike will be sent after all the neuron states have been

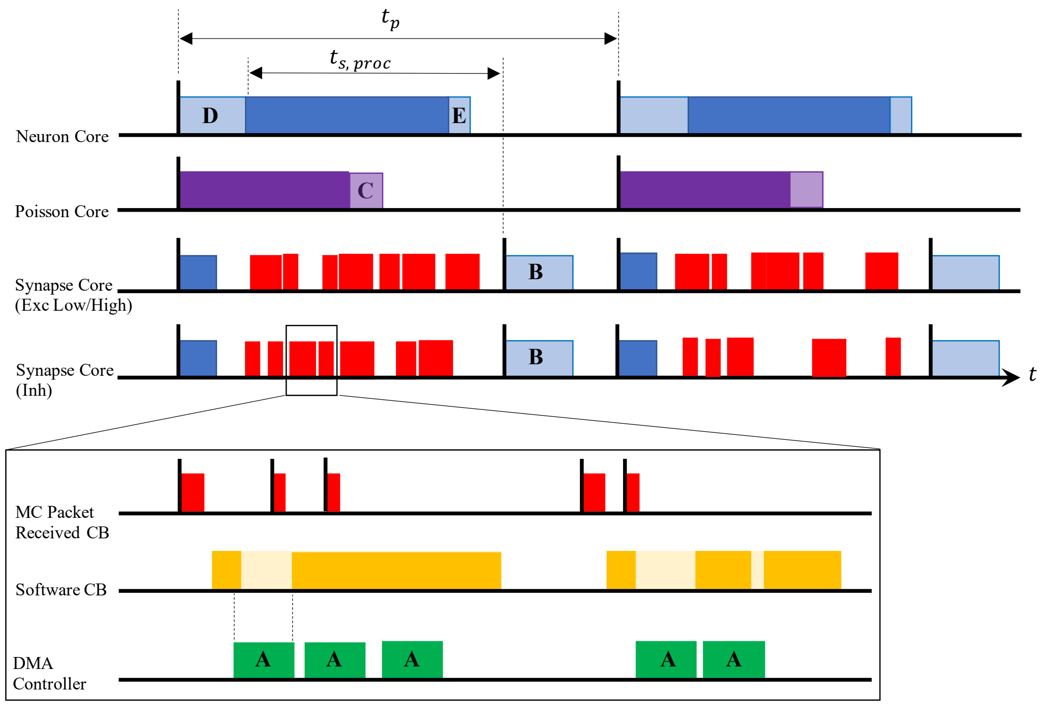


Figure 3.13: Callback interactions for the real-time execution of the Cortical Microcircuit. Reproduced from [RPR+19] with permission.

updated. The measured update time for a single neuron is 1.05 μs. To guarantee that all the spikes are generated within 70 μs from the beginning of the timestep, in order to allow enough time for the *Synapse* cores to process them, the number of neurons per core must be limited to 64. The choice of a power of 2 is to ensure word-aligned memory transfers without padding and ease of computation for the offset on the *Neuron* core side. This also allows increased efficiency, enabling the use of shifts and bitwise operations with masks. *Poisson* cores internally generate the background input for each postsynaptic neuron for the subsequent timestep. This value is written into shared memory in the middle of the timestep (transfer C). This allows reduction of the memory contention problem, having *Synapse* cores writing to memory in a separate phase. *Synapse* cores receive input spike packets throughout the timestep and compute the synaptic contributions to be sent to the *Neuron* core. The contributions are written to shared memory at the end of the timestep (transfer B).

To facilitate the state update operation as much as possible, the block read by the *Neuron* core contains the contributions arranged in the following order:

- lower excitatory

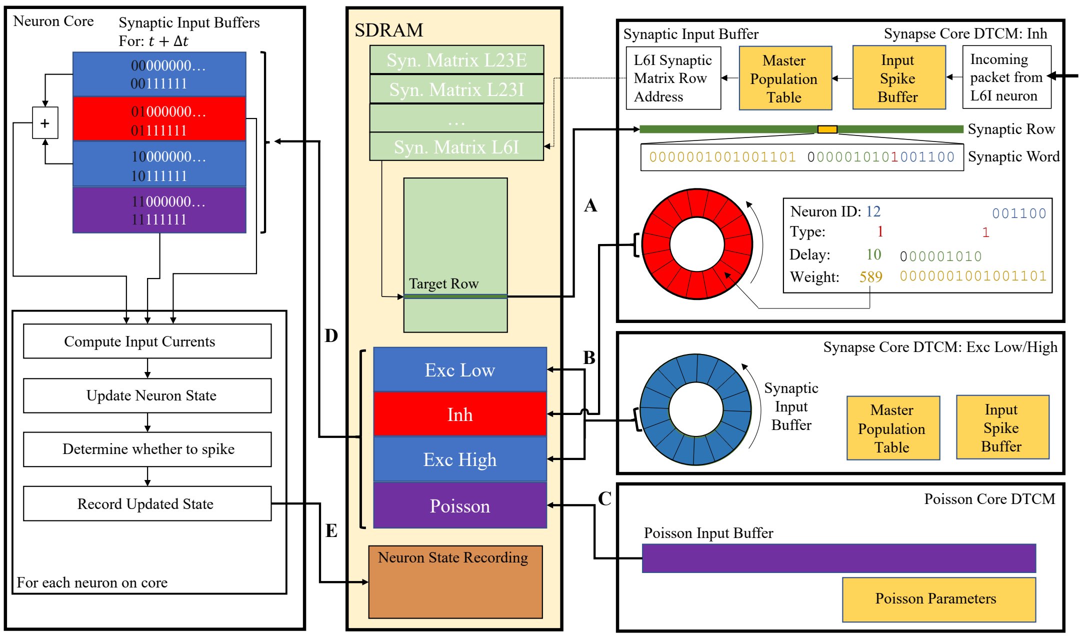


Figure 3.14: Memory Usage and data structures for the Cortical Microcircuit network on SpiNNaker. Shared memory is used for the data transfer within a neural processing ensemble; the ordering of the regions match the one used on SpiNNaker. Bold capital letters mark memory transfers corresponding to the same labels in figure 3.13. Adapted from [RPR+19].

- inhibitory
- higher excitatory
- Poisson

When calculating the inhibitory contribution, the input current is equal to the value read from memory. The excitatory values on the other hand, require summing together the lower and higher excitatory contributions together with the Poisson input.

A schematic of the memory interactions is shown in Figure 3.14, where the transfer labels match those in Figure 3.13. The choice of the synaptic contribution ordering described above is motivated by efficiency during the summation phase. The contributions block is a large array, where indices are represented on 8 bits, where the 2 MSBs (in black in Figure 3.14) represent the synapse type and the 6 LSBs (in white) the postsynaptic neuron ID. By masking the synapse type, it is possible to access the contributions efficiently, having the Poisson and inhibitory values converted separately and the two excitatory contributions accessed with cost O(1), added together and then converted.

Similarly to the general Heterogeneous Programming model, the shared memory regions are allocated, during initialisation, by the *Neuron* cores as a single memory block large enough to contain the data for all the other cores belonging to the ensemble. Once this is performed, the *Synapse* and *Poisson* cores retrieve the pointer to the memory block, which is tagged with the ID of the *Neuron* core, and add their offset to it. Through this operation they get the starting position of their own subregion. This offset is calculated from the host side software during the partitioning of the Population and communicated to each core before the beginning of the simulation.

3.4 Results

This section presents results of the first hard real-time simulation of the Cortical Microcircuit [RPR+19]. This result has been achieved by simulating the network on SpiNNaker through the Heterogeneous Programming model with the adaptations presented in Section 3.3.3. The important metrics here reported are accuracy of the simulation and energy consumption. Model and system performance are also measured.

Hard real-time simulation of the Cortical Microcircuit model requires the simulator to execute with a time resolution of 0.1 ms. The model has been simulated for 10 s of biological simulation time, resulting in the system running for a wall-clock simulation time of 10 s. Previous works [vARS+18] initialised the neurons' membrane voltages above threshold, causing very high firing activity during the first timestep (as shown in Figure 3.11). This work relaxed this condition, since this has no biological relevance and does not affect the steady state behaviour of the network, but only generates excessive traffic at the beginning of the simulation. Both the DC and the Poisson versions of the Cortical Microcircuit model have been simulated in hard real-time.

3.4.1 Simulation Accuracy

Figure 3.15 shows profiling data recorded from an upper excitatory *Synapse* core from the L23E population for the duration of 200 ms of simulation. The solid red line shows the total number of spikes arriving to the core and is in line with the spikes produced by the model (grey dashed line, which is mostly covered by the

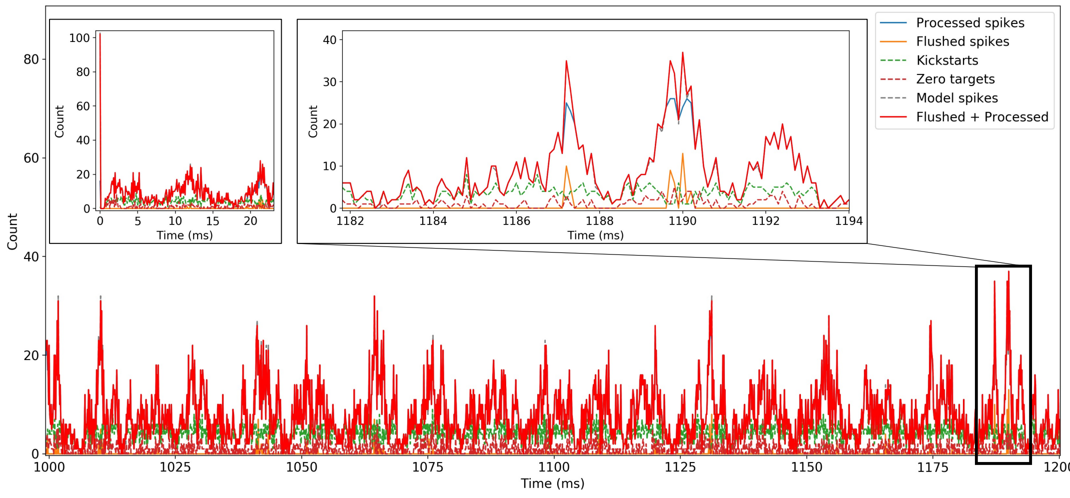


Figure 3.15: *Synapse* core profiling during simulation of the Poisson input Cortical Microcircuit. The total spike packets received per timestep are plotted, together with the total processed and flushed spikes: left inset details the response around the initial transient; while the right inset details the handling of extreme peak spike rates during steady-state oscillations. Reproduced from [RPR+19] with permission.

solid red line). This first equality demonstrates the correctness of the simulation, since the number of received spikes at every timestep matches the expected ones. The total number of spikes can be divided into processed spikes (blue line in the plot) and flushed spikes (orange line in the plot). The number of processed spikes matches that received throughout most of the simulation. In certain timesteps, when the incoming spikes are above the number that can be processed (≈ 26 spikes per timestep), the core flushes the inputs that cannot be handled. This causes some peaks in the number of flushed spikes as it can be observed in the right inset (e.g. timestep 1190). By flushing spikes, the core maintains hard real-time requirements, as it moves to the next timestep when the timer dictates. The last two lines indicate zero target packets and the number of kickstarts of the spike processing pipeline per timestep in dashed red and dashed green respectively. The zero target packets constitute an issue, as the system can only determine one of these when accessing the synaptic row, causing wasted computation to extract this data. This issue is more common when low probability connections are used, amounting, in some cases, to up to half the incoming spike traffic. The number of kickstarts of the spike processing pipeline, on the other hand, has a low impact on the execution, as timesteps where the input activity is higher maintain the pipeline active for longer, not incurring this penalty.

Since spikes are occasionally flushed from *Synapse* cores, it is important to evaluate the accuracy of the simulation, in order to determine whether this impacts the results. All flushed spikes represent information which is effectively lost. The simulation accuracy is checked against the results obtained from NEST simulations. The total number of processed synaptic events on SpiNNaker, during the 10 s simulations, is 9.135 $\times 10^9$ for the DC version and 9.343 $\times 10^9$ for the Poisson version. Compared to the NEST results, losses of 2.98% for the DC version and 0.76% for the Poisson version are registered. A higher loss for the DC version can be explained by a higher density of *Synapse* cores per chip (as there are no *Poisson* cores in this case), which causes a higher memory contention during the contribution writing phase, which results in a longer writing time per core and therefore a shorter spike processing window.

To assess the effects of flushed spikes, a statistical analysis comparing the within-population distributions of firing rates between SpiNNaker and NEST simulations is performed. The results are presented in figure 3.16. The three plots represent a comparison of firing rates, coefficients of variation of inter-spike intervals and correlation coefficients for binned spiketrains. The blue line represents results obtained from the real-time execution of the Poisson version on SpiNNaker, while the yellow line comes from the NEST simulator, running with a 3$\times$ slowdown factor. The results appear to be coherent and slight variations are in line with previous observations [vARS+18]. The analysis was performed using the Elephant toolbox [YDH+15], by using output spiketrains from both the simulators. The correctness of the simulation is therefore proven for the real-time simulation on SpiNNaker.

3.4.2 Machine Allocation and Energy Consumption

Because of the low-power nature of Neuromorphic hardware it is interesting to perform an analysis of the energy consumption, to check how the system performed and to compare it against other platforms. Compared to previous work [vARS+18], the network allocation on SpiNNaker requires a higher number of cores and, therefore, a bigger machine. The Poisson version is mapped to a 8.4 boards machine (6050 cores, spanning to 404 chips), while the DC version fits on 6.6 boards (4840 cores, with 318 chips). The Spalloc utility from the software toolchain (see Section 2.5.2) is in charge of providing the correct machine which,

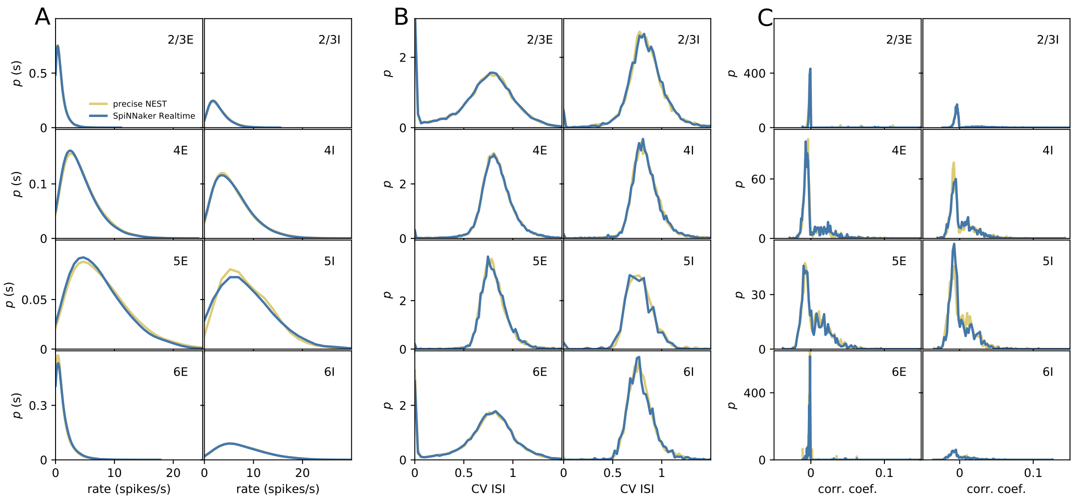
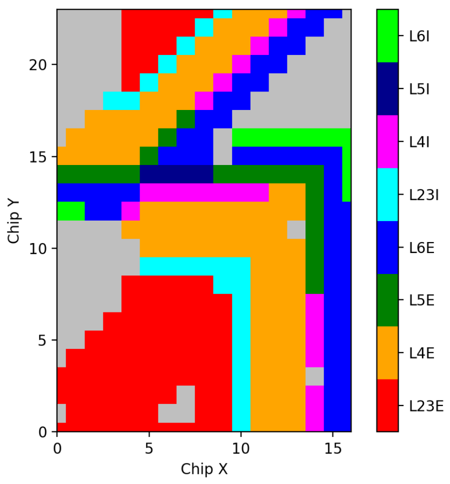


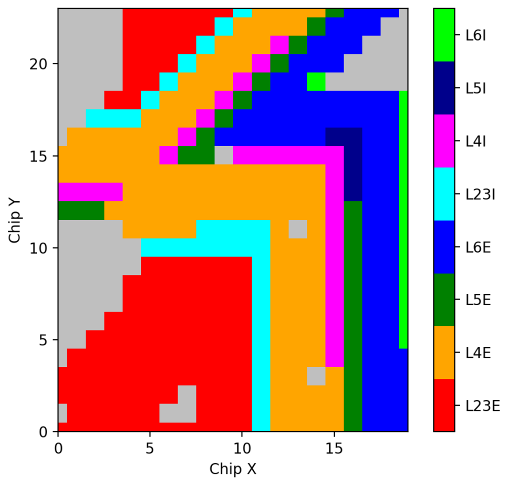
Figure 3.16: Comparison of spiking output from the last 10 s of cortical microcircuit simulations with Poisson input, executed using SpiNNaker in real time, and NEST at 3× slow-down (all averaged over the last 9 s of simulation): A, single neuron firing rates; B, coefficient of variation of inter-spike interval; C, correlation coefficient between binned spiketrains. Reproduced from [RPR+19] with permission.

in both cases, results in a 12-board SpiNNaker machine (having a total of 576 chips arranged in a 24×24 grid). Figure 3.17 details the core mapping for both the DC (left) and the Poisson (right) versions. There is a 25% increase in machine size for the Poisson case, which is caused by the addition of the *Poisson* cores (one allocated per ensemble). From these figures it is possible to calculate the neuron density per chip, which depends on the number of ensembles per chip, multiplied by the neurons per core, amounting to 256 neurons per chip in the case of the DC version (4 ensembles per chip and 64 neurons per core) and 192 for the Poisson version (3 ensembles per chip and 64 neurons per core).

Real-time simulations run 20× faster than previous achievements on SpiNNaker, causing the system to be powered on for much less time. It is therefore important to obtain the energy consumption under these new conditions. The most useful metric is the energy per synaptic event, which allows comparison of how well SpiNNaker performs with respect to HPC platforms and GPUs. In order to obtain these figures, the same technique used in previous studies [vARS+18] was replicated, where the wall-socket power of the entire SpiNNaker system is recorded. This is necessary, since there is no power monitoring for multi-board systems. The energy is therefore measured through an energy meter (with an



(a)



(b)

Figure 3.17: Cortical Microcircuit Placements on SpiNNaker
Layer-wise placements of the Cortical Microcircuit network on SpiNNaker: (a) DC version, (b) Poisson version. Reproduced from [RPR+19] with permission.

accuracy of 0.01 kWh) which measured wall-socket power of the entire SpiNNaker system. A software-controlled camera took readings from the meter at the beginning and end of the simulation, and the difference between these two values indicates the total energy consumption. To obtain the energy per synaptic event it is necessary to divide this value by the total number of synaptic events generated by the model.

The measured values are detailed in Table 3.3, where a 24-board SpiNNaker system has been used for the measurements. Both the simulated versions of the Cortical Microcircuit (DC and Poisson) resulted into a 12-board machine allocation, leaving the unused boards powered off. The unused chips on the allocated machine, on the other hand, remained idle all the time. The presented results show, in order, the energy consumption for the auxiliary system components (e.g. cooling fans, communication switch and rack power supplies) with all the boards powered off, the values for 12 booted SpiNNaker boards and the consumption for the Cortical Microcircuit simulations. Measurements are taken over a 12h period. By comparing the energy figures for the Microcircuit and the booted system, it is observed that energy consumption contribution of the network amounts to 25% of the total consumption in the case of the DC version, and 30% in the case of the Poisson version, as they show an increase in energy demand of 1.34× and 1.44×

System configuration	Time	Total energy (kWh)	Synaptic events	Energy per synaptic event (μJ/syn-event)
system only	12h	0.93	–	–
12 booted boards	12h	4.93	–	–
Cortical Microcircuit, DC	12h	6.59	–	–
Cortical Microcircuit, Poisson	12h	7.11	–	–
Cortical Microcircuit, DC	10s	0.001525	9.135×10^9	0.601
Cortical Microcircuit, Poisson	10s	0.001646	9.343×10^9	0.628

Table 3.3: Cortical Microcircuit Energy Measurements.

respectively. Finally, the bottom two lines indicate the energy per synaptic event, calculated over 10 s simulations of the network. The obtained values amount to 0.601 μJ and 0.628 μJ for the DC and Poisson version respectively. These values are one order of magnitude smaller than previous results on both SpiNNaker and HPC platforms [vARS+18]. Furthermore these values demonstrated that SpiNNaker can compete with a range of modern GPUs running optimised SNN libraries [KN18].

3.5 Discussion

The capability of processing all the information within each simulation timestep is a necessary requirement for hard real-time simulations of biologically-inspired SNNs. Large networks are affected by very high fan-ins, causing large numbers of spikes to be delivered every timestep. The number of synaptic events that can be processed per timestep therefore becomes a critical factor. SNN simulators struggle when dealing with such heavy activity, because of their implementation. The proposed Heterogeneous Programming model, through parallelisation of incoming spike traffic, provides a different perspective for the task. The implementation for the SpiNNaker Neuromorphic platform of this approach allows improvement of the performance of SNNs simulations, by achieving 12.3× more synaptic events processed per timestep compared to previously published work, and additional flexibility in targeting different computational loads.

When applied to a neuroscience benchmark, namely the Cortical Microcircuit model, the new approach achieved unprecedented results, performing the first hard real-time simulation of the model, surpassing any previously published result on any other platform. The robustness of the approach has also been demonstrated by successfully performing multiple 12h simulations, placing Neuromorphic hardware as a viable choice for modelling long-term effects in SNN models representing brain activity at the same speed and scale as they occur in biology.

Other simulators recently attempted to accelerate the simulation of the network, trying to achieve faster than real-time results. Sub-real time simulations have been achieved respectively by HPC using NEST [KST$^+$21] and on GPUs through the GeNN framework [KKN21]. Even faster simulations have been achieved by FPGA-based neural supercomputers, reaching 4× faster than real time simulations [HPT$^+$22]. SpiNNaker however still remains, at the moment of writing, the only Neuromorphic platform able to simulate such benchmark in hard real-time.

3.6 Summary

The aim of this Chapter was to address the first research question presented in Section 1.2, which states: *How can the process of mapping biologically-representative SNNs be optimised on Neuromorphic Hardware?*

This can be achieved through the Heterogeneous Programming model, a novel parallelisation technique developed for the SpiNNaker Neuromorphic platform, in order to improve the mapping of SNNs on Neuromorphic hardware, consequently increasing the processed synaptic events per ms. By using dedicated components to different aspects of SNNs simulations, namely the synaptic input processing and the neural state update, and by performing a heterogeneous parallelisation of them, it was possible to increase the processed synaptic events per timestep up to 12.3× compared to previous results. This enabled the first real-time simulation of the Cortical Microcircuit model, constituting a 20× speedup compared to previous simulations, which, de facto, placed digital Neuromorphic platforms as key competitors in the landscape of real-time simulators for biologically-representative Spiking Neural Networks.

The presented approach, through its flexibility, allows to target different levels of sparsity and throughput, by allowing to better adapt the allocated machine to the network requirements.

Several challenges still remain related to biological learning mechanisms and very sparse SNN simulations, which will be addressed in Chapters 4 and 5.

Chapter 4

On-line Learning on SpiNNaker

4.1 Introduction

Traditional learning mechanisms such as gradient descent with error backpropagation [RHW86], are energy and resource consuming and far from biological learning models [IGB19, Hun]. Several biologically-plausible alternatives have been presented to date, in order to provide valid power-efficient strategies [SPCBS18, BSS^{+}20, LCTA16a]. These techniques commonly aim at replacing the error backpropagation phase, where the network does not produce useful outputs, with alternatives which allow the network to be always active. These approaches are categorised under the definition of on-line learning algorithms [LDBK20]. It is, however, problematic to simulate such mechanisms in real time, due to the challenges presented by large biologically-representative SNN simulations shown in Chapter 3, combined with the addition of intrinsic learning mechanisms. Thanks to the possibilities offered by this unconventional hardware, such as great scalability and limited power consumption, a Neuromorphic implementation might provide an efficient framework to develop novel learning algorithms, with potential to replace the power-hungry solutions which still are the main learning tools currently used.

This chapter therefore describes the implementation of on-line learning tasks on SpiNNaker. The approach used for this problem employs the Heterogeneous Programming model presented in Chapter 3, with some adaptations. The chosen learning strategy was originally proposed by Urbanczik and Senn [US14] in the form of a multicompartmental error-correction rule, and then expanded by Sacramento et al. [SPCBS18] to be applied to classification problems in the field of machine learning as an on-line replacement for the costly error backpropagation rule [RHW86, LBH15]. This learning rule employs more biologically-plausible neuron models, combined with plasticity rules, to obtain results comparable to the standard backpropagation of errors algorithm [SPCBS18]. Because of the type of employed neuron models, this approach represents a good fit for Neuromorphic hardware.

Despite its biological plausibility, there are several challenges that arise when porting this rule to Neuromorphic hardware. The networks described, together with the learning rule itself, use rate encoding for the communications, requiring frequent transmissions of neurons' firing rates. Furthermore, the connectivity patterns are very dense, resulting in additional requirements when placing the neural networks, in order to maintain real-time performance. The standard SpiNNaker toolchain implementation fails to handle such constraints, therefore an adapted version of the Heterogeneous Programming model is required. This extension is presented throughout this chapter, including the added support to synaptic plasticity. Additional strategies to improve the handling of rate-based communications required by this task are also presented.

Section 4.2 presents the first two-compartment model described by Urbanczik and Senn [US14], together with its rate-based implementation on SpiNNaker and the adaptation of the plasticity framework to handle this new rule. Section 4.3 describes a three-compartment model [SPCBS18] representing pyramidal neurons, used as the main building block for the learning rule. Section 4.4 presents the learning rule combining the two neuron models and the neural network structure, together with the implementation challenges. Section 4.5 describes how the Heterogeneous Programming model has been adapted to this context, including the addition of a plasticity handling mechanism and other optimisations to speed up the synaptic events processing. Section 4.6 shows the results of the implementation of the models on SpiNNaker, first by proving the correctness and then by simulating larger networks addressing specific tasks.

4.2 The Urbanczik-Senn Neuron Model

The Urbanczik-Senn (US) neuron model [US14] (shown in Figure 4.1) is a two-compartment neuron model, composed of Somatic and Dendritic compartments coupled via a fixed conductance called "Coupling Conductance". The model

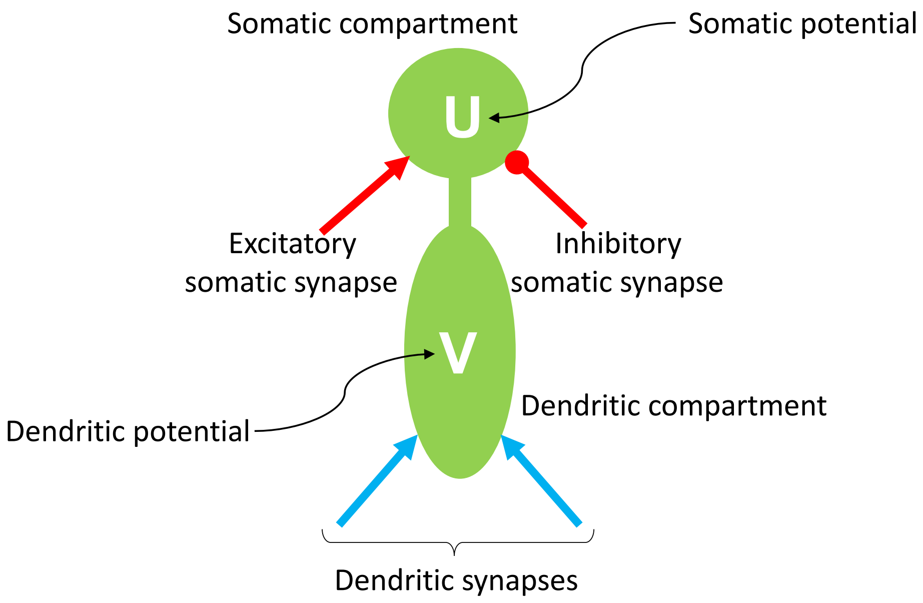


Figure 4.1: The Urbanczik-Senn neuron model. The two compartments with the respective potentials are shown. Red synapses are static, blue synapses are plastic. Dendritic synapses are current based, somatic synapses are conductance based.

is rate-based and the output rate, emitted by the somatic compartment, is a sigmoidal function of the somatic voltage (U in Figure 4.1). The somatic synapses (red in Figure 4.1) are conductance-based and static and can be either excitatory or inhibitory. The somatic input current, result of the combination of the somatic inputs, is termed Teaching current. The dendritic compartment produces its own internal voltage (V in Figure 4.1) as a linear combination of the input rates and the synaptic weights. The dendritic synapses (blue in Figure 4.1) are current-based and plastic. The role of the coupling conductance is to low-pass filter the dendritic voltage that contributes to the calculation of the somatic voltage. All the connections represented in red are modelled by static synapses, while the blue connections are associated with plastic synapses. This convention is followed throughout this chapter.

The instantaneous representation of the model's equations is given by Equations 4.1 to 4.5.

$$U(t) = \frac{g_{SD}V_w(t) + I_{teach}(t)}{g_{tot}} \tag{4.1}$$

$$g_{tot} = g_L + g_{SD} + g_E + g_I \tag{4.2}$$

$$V_w(t) = \frac{I_{dnd}(t)}{g_L} \tag{4.3}$$

$$I_{dnd}(t) = \sum_i w_{xi} r_i \tag{4.4}$$

$$I_{teach}(t) = g_E(t)(E_E - U(t)) + g_I(t)(E_I - U(t)) \tag{4.5}$$

I_{dnd} is the dendritic current, g_L is the leak conductance (fixed), $V_w(t)$ is the dendritic voltage, while I_{teach} is the somatic teaching current, which is obtained by adding together the excitatory and inhibitory components. g_E and E_E are, respectively, the excitatory conductance and reversal potential; g_I and E_I are the inhibitory ones. The reversal potentials are fixed, while the conductances are obtained via the synaptic inputs. g_{SD} is the coupling conductance between dendrite and soma, which is fixed, and g_{tot} is a support variable expressing the total conductance. Finally $U(t)$ is the somatic potential.

The plasticity rule is an error-correcting rule and aims to minimise the difference between the somatic and dendritic compartments. In a supervised learning scenario, nudging currents are applied to the somatic synapses, in order to force a difference between the two compartmental voltages. This difference triggers the dendritic plasticity, nudging the dendritic potential towards the somatic potential. When the teaching current is removed, the output is solely driven by the dendritic input values. The weight change in time is represented by Equation 4.6.

$$\Delta w_{di} = \eta_{di}(\phi(V_w^{tgt}(t)) - \phi(V_w(t)))r_i = \eta_{di}(\phi(\frac{g_L + g_{SD}}{g_{SD}}U(t)) - \phi(V_w(t)))r_i \tag{4.6}$$

w_{di} is the weight for the i^{th} dendritic synapse. η_{di} is the learning rate, which scales the weight modification and is fixed. r_i is the incoming rate for the i^{th} synapse. $\phi(V_w(t))$ is the output rate calculated on the dendritic voltage and $\phi(V_w^{tgt}(t))$ is the rate calculated on the target dendritic voltage. The target dendritic voltage is obtained by filtering the somatic potential, and therefore represents what would the dendritic pontential be if the somatic synapses were silent. The difference between these rates amounts to the error the dendrite makes in predicting the somatic voltage. Such an error arises when a teaching signal is applied on the somatic synapses, therefore the dendritic weights' values are corrected in order to eliminate this misprediction, resulting in correct operation when the supervision is removed.

4.2.1 SpiNNaker Implementation

The SpiNNaker US model implementation requires some structural changes to the software toolchain. The first change is motivated by the rate-based nature of the model. For efficiency, from a computational point of view, it is preferrable to have neurons communicating rates with each other, instead of having processors to convert the firing rate into spikes and then the received spikes back into firing rates. This choice however requires the use of multicast packets with payload, in order to carry the rates over the communication network. This requirement adds challenges in terms of real-time performance however, as packets with payload are slower and can cause more traffic congestion over the SpiNNaker network in a dense communication scenario.

The best option to represent output rates is through the *accum* data type provided by the fixed point C standard library. The SpiNNaker system does not contain a Floating Point Unit (FPU), therefore it is preferrable to handle this data through fixed-point arithmetic, as software simulation of floating point operations would be inefficient. The firing rates are positive numbers which can take values greater than 1 (which excludes the use of the fract S031 data type), therefore over 32 bits a S1615 (16 bits for the integer part and 15 for the fractional part, plus one bit for the sign) is the most appropriate data type. The same data type is also used by the SpiNNaker system to represent input currents and voltages, therefore adopting this simplifies the data handling.

Input currents to the dendritic compartments are obtained by multiplying the input rate by the corresponding weight. This operation considerably increases the time per synaptic event, compared to standard spike-based SNN simulations, as it involves a multiplication and sum (the input rate value is multiplied by the weight and then added to the correct synaptic input buffer slot), instead of a single sum per event (the weight is simply added to the correct synaptic input buffer) as described in Section 2.6. Given the required operations, it is sensible to represent weights as *accum* as well: this also allows greater precision. The synaptic input buffers therefore need to be adapted to store *accum* values. However, since each input rate contributes to the postsynaptic voltage of the next timestep only, there is no need for delay slots. On the other hand, the input spike buffer (the data structure used to store incoming packets before processing synaptic events) requires some adaptations as well, as each packet now carries a firing rate, which needs to be stored for the synaptic event processing.

The somatic voltage update is performed sequentially after the dendritic potential calculation for each neuron. This is due to the dependency of the somatic potential on the dendritic one, as shown in Equation 4.1.

4.2.2 The Plasticity Framework

The plasticity rule requires adapting the existing structures provided by the SpiNNaker toolchain to accommodate the new parameters used by the rule. Therefore a modified version of the STDP framework has been used. This choice is motivated by the use of the event-driven processing of synaptic events described in Section 2.6, combined with the chance of implementing a large pool of incoming synapses.

The new protocol requires the neurons to save the difference between the output rate calculated over the somatic voltage and the rate evaluated from filtered dendritic potential. This difference is used to determine the amount of the correction which is applied to the synapses. Upon the reception of an input rate, after the correct synaptic row is retrieved from shared memory, the incoming value is stored in the row and the weights are updated according to Equation 4.6. For this operations, the previous input rate (saved from the synaptic row) is used in combination with the stored postsynaptic difference. Because of the teaching current, which can modify the somatic potential through the somatic synapses, and therefore independently from the dendritic input, the dendritic weights can be subject to synaptic plasticity even in the absence of dendritic input (as shown by Equation 4.6). This requires to constantly check whether the dendritic weights need to be updated. In order to perform this check, the SpiNNaker toolchain needs to have the synaptic rows locally available. Therefore each neuron will emit a packet every timestep, allowing to retrieve the rows one at a time. This proved to be the most efficient solution on SpiNNaker to make sure all the weights are correctly updated. A calculation based on the difference between the input rates on different timesteps requires additional operations for each weight update, which results in performance degradation.

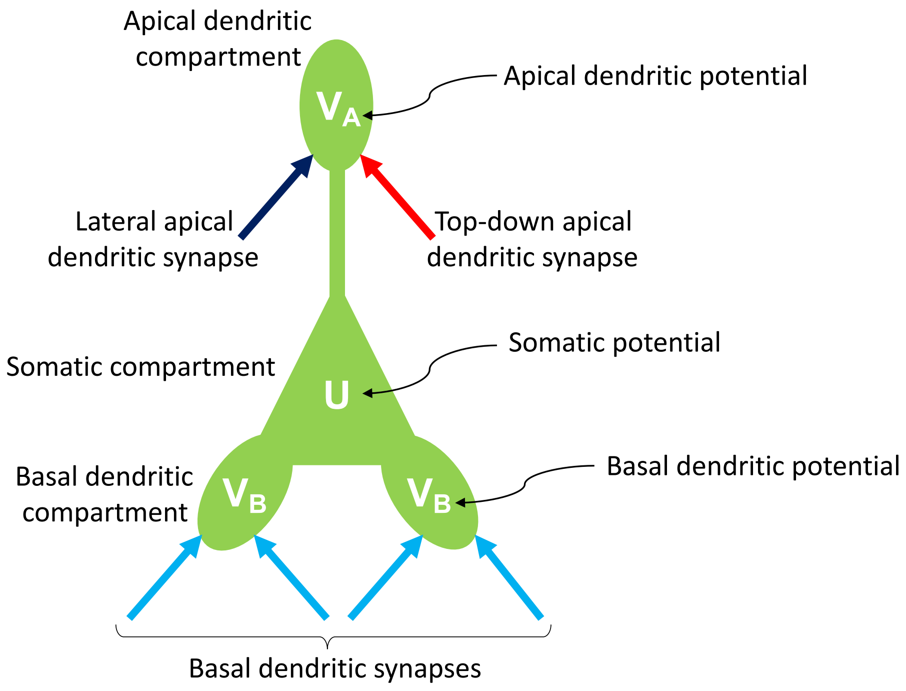

Apical dendritic compartment
V_A
Apical dendritic potential
Lateral apical dendritic synapse
Top-down apical dendritic synapse
Somatic compartment
U
Somatic potential
Basal dendritic compartment
V_B
V_B
Basal dendritic potential
Basal dendritic synapses

Figure 4.2: The 3-compartment pyramidal neuron model. The three compartments with respective potentials are shown. Blue synapses are plastic, red synapses are static. All the synapses are current based

The plastic region of the synaptic row keeps a similar structure compared to the standard SpiNNaker toolchain. The presynaptic history trace however is now replaced by the last presynaptic input rate, and weights are now represented on 32 bits as described in Section 4.2.1. A schematic of the new structure of the synaptic matrix is shown in Figure 4.5.

4.3 The Three-Compartment Pyramidal Model

Evolving from the US model, a three-compartment representation of pyramidal neurons was modelled by Sacramento et al. [SPCBS18]. This model presents a simplified structure of the basal and apical integration zones which define neocortical pyramidal cells [Spr08, Lar13], through separate dendritic compartments. Bottom-up and top-down synapses are handled separately by basal and apical compartments respectively. The two dendritic compartments are coupled to a somatic compartment via fixed conductances. There are no proximal synapses to the somatic compartment, therefore the somatic voltage is exclusively shaped by the dendritic compartments. The model state evolves according to Equations 4.7 and 4.8. g_A is the coupling conductance between the apical dendritic compartment and the somatic compartment, g_B between the basal and the somatic compartments. g_L is the leak conductance, $V_A(t)$ and $V_B(t)$ are the dendritic potentials and are obtained similarly to Equation 4.3. All the synapses are current based.

$$U(t) = \frac{g_A V_A(t) + g_B V_B(t)}{g_{tot}} \tag{4.7}$$

$$g_{tot} = g_L + g_A + g_B \tag{4.8}$$

Both compartments have plastic synapses, however the plasticity rule changes according to the compartment type. The basal compartment acts similarly to the US model, trying to match the somatic potential, while the apical synapses aim at silencing the apical voltage, therefore nudging the apical synapses to a state where the apical dendritic potential matches the resting potential. Equations 4.9 and 4.10 show the weight changes for the basal and apical compartments respectively.

$$\Delta w_{di} = \eta_{di}(\phi(V_B^{tgt}(t)) - \phi(V_B(t)))r_i = \eta_{di}(\phi(\frac{g_B + g_A + g_L}{g_B}U(t)) - \phi(V_B(t)))r_i \tag{4.9}$$

$$\Delta w_{di} = \eta_{di}(v_{rest} - V_A(t))r_i \tag{4.10}$$

Similarly to the Urbanczik-Senn case, in Equation 4.9, the error is computed towards a target voltage, which, in this case, takes into account both the coupling conductances. Equation 4.10 on the other hand, nudges the weights towards the resting potential, by computing the error with respect to this value. For the apical compartment plasticity, the error is calculated directly on the voltages, instead of the rates.

4.3.1 SpiNNaker Implementation

The SpiNNaker implementation of the three-compartment model follows the same approach adopted for the US model and presented in Section 4.2.1. The model is rate-based, therefore communication happens through Multicast Packets with payload carrying rate values. Each rate is a fixed-point value represented according to the S1615 format. All the compartment voltages are updated timestep by timestep sequentially according to the following order: basal dendritic compartment, apical dendritic compartment and somatic compartment. Separate synaptic input buffers are maintained for separate compartments. There are no proximal synapses to the somatic compartment, and all the implemented synapses are current-based.

4.3.2 The Plasticity Framework

The plasticity rule presented for the three-compartment model is of the same type as the US model, however the behaviour is slightly different according to the involved compartment. Given the similarities with the US model, the plasticity framework adopted for the three-compartment model follows the same adaptations presented in Section 4.2.2, with the addition of a second rule (the apical plasticity rule) which is selected through a flag indicating the compartment in which the weight update is happening.

4.4 Self-Predicting Learning Approximates Error Backpropagation

By combining the two models presented in Sections 4.2 and 4.3, it is possible to build a learning rule that approximates the error backpropagation algorithm. The idea originated from a previous study [KK01] which demonstrated that it is possible to implement backpropagation-like strategies, by using neurons with two separate integration sites. A similar approach was presented in a more recent study [GLR17b], where it was demonstrated that, by combining neuron models with segregated dendritic compartments into multilayer networks, it is possible to obtain accurate results when performing deep learning tasks. Therefore, the backpropagated error is represented by the activity of the apical compartment in the presence or absence of teaching input. The learning algorithm presented by Sacramento et al. [SPCBS18] evolves this idea, by explicitly encoding the error, by having the apical compartment integrating two different types of signal. This removes the need for representing two separate phases and therefore having the

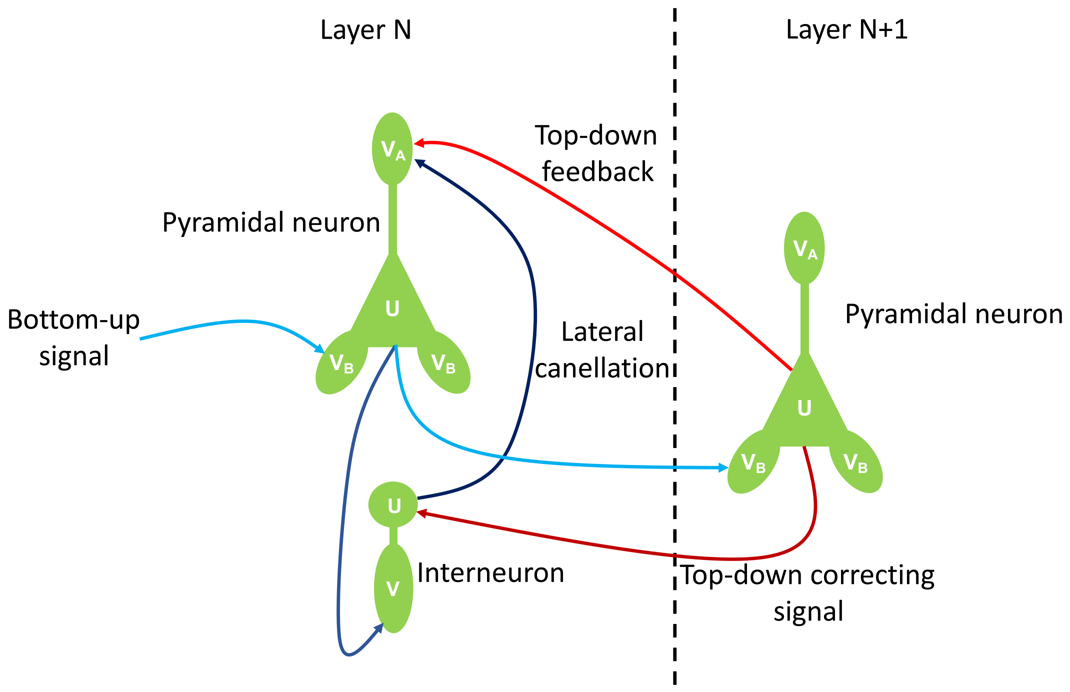


Figure 4.3: Schematic of the Self-predicting learning mechanism. Intralayer and interlayer interaction between pyramidal neurons and Interneurons is shown.

plasticity always active.

The reason for different plasticity rules for separate integration zones comes exactly from this necessity. Input signals are provided bottom-up to the basal synapses, which propagate them to the somatic compartment. Teaching signals are handled by the apical compartment, which receives both top-down and lateral values, as shown in Figure 4.3. The authors termed these types of network Dendritic Microcircuits.

4.4.1 Dendritic Microcircuits

Figure 4.3 represents both the inter-layer and intra-layer interactions for the Dendritic Microcircuits. For clarity, by keeping the same convention through this chapter, all the red connections are modelled by static synapses, while the blue ones are plastic. Each layer contains a population of pyramidal neurons, as well as a population of two-compartment US neurons, here termed Interneurons (or SST neurons) by the authors [SPCBS18]. The inputs are propagated bottom-up through the basal compartments of pyramidal neurons. The somatic compartments generate the output rate, which is sent to the next layer and to the lateral

Interneurons. The somatic potential is also backpropagated to the previous layer as somatic proximal input for the Interneurons (acting as a teaching signal) and as apical input for the pyramidal neurons (red arrows in Figure 4.3). The role of the Interneurons is to match the behaviour of the pyramidal neurons in the layer above, by integrating the somatic input of the pyramidal neurons of the same layer (dark blue arrow between the pyramidal neuron and the Interneuron) through the dendritic compartment. Pyramidal neurons from the next layer then transmit the correct somatic potential back to the Interneurons, eventually correcting misprediction errors. The apical compartment of pyramidal neurons therefore receives lateral input from the somatic compartment of the Interneurons (dark blue arrow) and top-down input from the layer above (light red arrow). The difference between these two signals corresponds to the backpropagated error. Therefore synaptic plasticity intervenes to silence the apical compartment, by acting on the top-down synapses, and so correcting this error. This network state, when apical compartments are silent and there is therefore no learning, is referred to as the self-predicting state by the authors [SPCBS18].

Because of the nature of the backpropagated signal, the teaching signal for the Interneurons (dark red arrow) needs to be in the form of a balanced teaching excitatory and inhibitory input, which is represented by the backpropagated somatic voltage, instead of separate components as described in Section 4.2. This is achieved by defining a fixed somatic conductance, called g_{som}. Therefore the somatic teaching current for Interneurons becomes:

$$I_{teach} = g_{som}(U^{tgt}(t) - U(t)) \tag{4.11}$$

where $U^{tgt}(t)$ is the backpropagated target potential and $U(t)$ is the somatic potential.

The nudging of Interneurons happens on a one-to-one basis, therefore the number of Interneurons in each layer of a dendritic microcircuit must match the number of pyramidal neurons in the next layer. All the other connections follow an all-to-all connectivity pattern. Connections are formed between neurons of contiguous layers only. An example of the connectivity pattern for the Dendritic Microcircuits is given by the network in Figure 4.4, where all the connections are all-to-all, except for those explicitly labeled as 1-to-1.

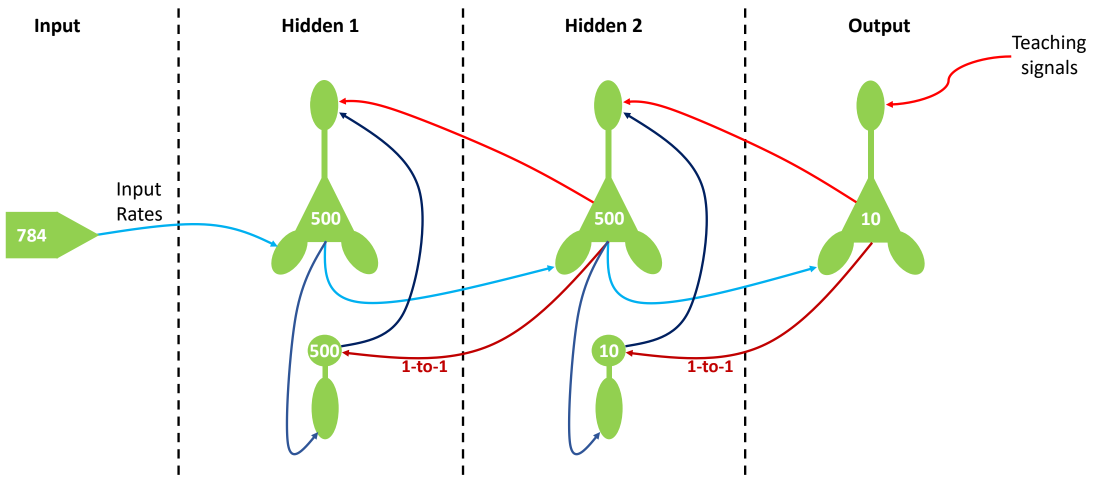


Figure 4.4: Topology of the MNIST microcircuit.

4.5 Application of the Heterogeneous Partitioning

Currently, the most challenging application of the Dendritic Microcircuits is the classification problem for the MNIST handwritten digits dataset [LBBH98]. The authors achieved a test error of 1.96% [SPCBS18], which is comparable to state of the art results for the dataset using non-convolutional ANNs optimised with Error Backpropagation [LCTA16b]. This section describes the network used by the authors, together with the steps necessary to implement it on SpiNNaker.

4.5.1 Microcircuit Network for MNIST

The network used for this classification task (shown in Figure 4.4) consists of four layers (1 input layer, 2 hidden layers and 1 output layer). The input layer contains the rate generators for the pixels composing each image in the dataset, therefore 784 sources are required (each image contains 28×28 pixels). The first hidden layer contains 500 pyramidal neurons and 500 Interneurons, the second hidden layer has 500 pyramidal neurons (matching the Interneurons of the first hidden layer), together with 10 Interneurons. Finally, the output layer has 10 neurons, which match the number of possible classes. The 10 output neurons receive direct feedback during the learning phase from each of the classes. Having mostly all-to-all connections (except for the ones labeled 1-to-1 in Figure 4.4), the resulting synaptic matrices are fully connected, strongly impacting the requirements on

SpiNNaker.

The worst case is given by the first hidden layer, where each pyramidal neuron is fed by 784 bottom-up connections from the input layer, 500 lateral connections from the Interneurons in hidden layer 1 and 500 top-down connections from the pyramidal neurons from hidden layer 2. It is important to notice that, differently from spike-based SNN where firing is irregular, for rate-encoded networks each neuron receives inputs from each connection every timestep. This type of encoding requires to process all the incoming information, as missing an input rate has a stronger effect on the network behaviour, compared to not processing a single spike.

The worst case is therefore represented by the hidden 1 pyramidal population. A real-time implementation therefore needs to meet the requirements for this case.

4.5.2 Placements and Constraints

The MNIST dendritic microcircuit involves the use of synaptic plasticity. This requirement, together with the large number of input signals, requires a timestep resolution not higher than 1 ms. It is indeed a challenge to perform the weight updates on SpiNNaker for large networks with tighter resolutions, due to the write back of updated synaptic rows to shared memory. For this case a timestep-based structure is preferred. The task involves real-time on-line learning, therefore, to enforce real-time execution, it has been decided to maintain a timestep-based structure analogous to that presented in Chapter 2 and 3, where the timesteps are advanced through the use of a timer. This is preferred to a more general implementation, where there are no explicit time mechanisms, but the simulation state is advanced when all the neurons have been updated and the firing rates are generated. Because of the large number of input signals received every timestep, it is not possible to process all the necessary information within the timestep boundaries, by using the standard SpiNNaker toolchain. Therefore an adaptation of the Heterogeneous Programming model, presented in Chapter 3 is necessary to maximise the processing capabilities of the system. In order to correctly model the approach, an analysis of the synaptic update times according to different numbers of target neurons is necessary. The number of postsynaptic neurons implemented per core, indeed, impacts the number of synaptic events carried per packet and therefore the processing time per packet. The connectivity pattern is mostly all-to-all (which is analogous to a 100% probability of connectivity),

Synaptic Processing Time			
Target neurons	Basal Time(μs)	Apical Time(μs)	Static Time(μs)
1	6.69	6.675	4.975
2	7.725	7.695	5.51
4	9.8	9.74	6.57
8	14.03	13.905	8.775
16	22.33	22.1	12.945
32	39.875	39.395	22.49
64	73.0	72.045	39.455

Table 4.1: Synaptic processing time for 1 rate packet, with different neurons configurations. From left: basal, plastic apical and static apical synapses.

therefore this means that by adding one neuron per core, each received packet will contain one more synaptic event. Such analysis is shown in Table 4.1, showing total synaptic processing time.

The table shows the processing time in μs for a single input rate packet, measured on SpiNNaker, with different numbers of neurons per core. Each column corresponds to a specific synapse type. The plastic basal synaptic type is the leftmost column, followed by the plastic apical and then static apical. Static synapses are processed faster, since no weight update is required; similarly the plastic apical are faster than the plastic basal, as the error is calculated over the compartment voltage, therefore there is no need to calculate the output rates.

By combining these numbers it is possible to determine the time required to update the synapses for the pyramidal population in hidden layer 1. The relation is expressed by Equation 4.12.

$$T = 784 \times t_{basal} + 500 \times t_{apical} + 500 \times t_{static} \tag{4.12}$$

T represents the time required to process all the synaptic inputs for a timestep per core, t_{basal} is the time necessary to process one basal input rate packet, t_{apical} is relative to plastic apical synapses and t_{static} to static apical synapses. Therefore, by having 16 neurons per core, the synaptic processing time amounts to 35 ms, which is 35 times larger than the allowed timestep (for real-time simulations), de facto requiring a 35$\times$ slowdown factor. Similarly with 8 neurons per core, the synaptic processing amounts to 22.33 ms, and to 15.83 ms with 4 neurons per core. These numbers are far from the real-time requirements and, furthermore, they do not include the neuron update times. Since the Heterogeneous model allows equal splitting of the number of incoming synaptic inputs by performing a source-based partitioning, it is possible to fragment these synaptic times according to the number of associated synaptic cores. This is guaranteed by the uniform all-to-all connectivity pattern. By using 14 *Synapse* cores therefore, which is the maximum available on a chip (3 cores are reserved for system purpose as Monitor core, Extra Monitor core, and Fault tolerance and 1 core is used as *Neuron* core), the total synaptic time with 8 neurons per core amounts to 1.60 ms. This, combined with synaptic processing optimisations, represents a good target for real-time processing.

4.5.3 Synaptic Matrices in Local Memory

Previous work [RBB+18] has shown that there is a large cost in retrieving synaptic rows from shared memory, which includes setting up a DMA request, performing the memory transfer, and then responding to the transfer completion event with the associated callback. This cost is partially amortised by the spike processing pipeline, which, when triggered, allows cores to process synaptic events while other rows are transferred. The fixed cost of setting up DMA transfers and the interrupt response, however, must be paid regardless. Furthermore, transfer times can increase due to multiple cores contending the access to memory. Finally, plastic networks need to write back the updated plastic region of each processed synaptic row (as detailed in Section 2.6.5), effectively doubling the number of memory transactions.

Although having the synaptic matrices stored in shared memory is a necessity in standard SNN simulations, where it is complex to foresee the size of synaptic matrices, the connectivity patterns are irregular and the number of neurons per core can be considerably large. For this specific case, the timing constraints limit the number of neurons per core to 8, which amounts to a total of 1784 presynaptic units for the pyramidal population hidden layer 1, resulting in a fixed number of synaptic events of $1784 \times 8 = 14272$. This means that the synaptic matrix for each ensemble (block of *Neuron* core with the afferent *Synapse* cores), for the hidden 1 pyramidal population, contains 1784 rows (1284 plastic and 500 static). Each plastic row (according to the synaptic row structure presented in Figure 4.5) requires 12 Bytes for the regions' sizes (Plastic, Static and Fixed Plastic), 4 Bytes for the presynaptic buffer, $4 \times 8 = 32$ Bytes for the weights, plus $1 \times 8 = 8$ Bytes for the synapse type and Neuron ID, which amounts to a

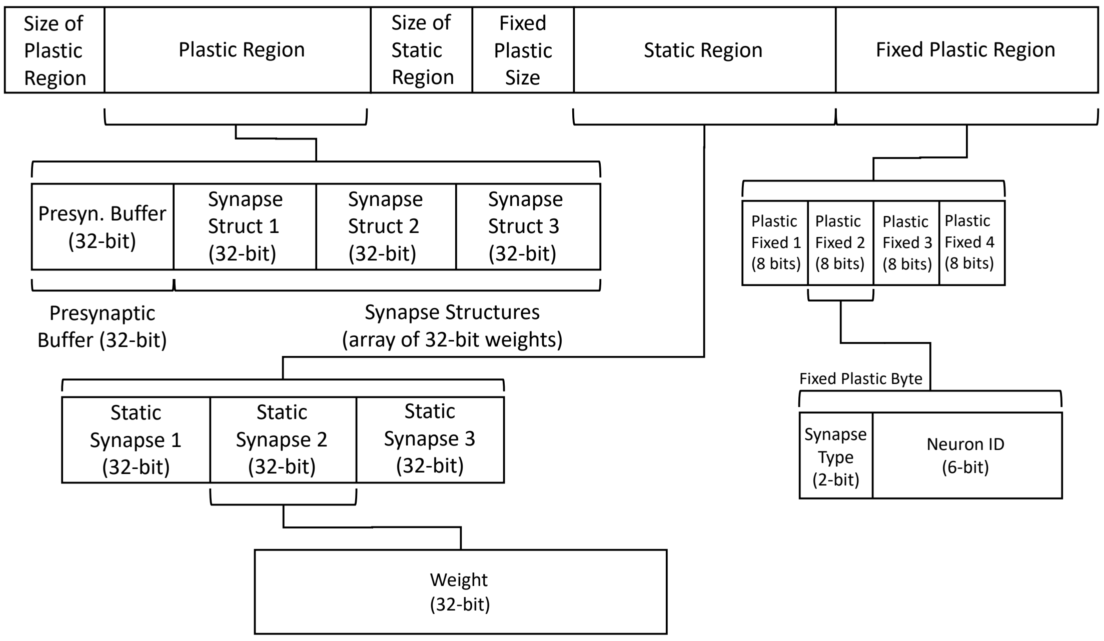


Figure 4.5: Updated synaptic row for rate-based models to be stored in DTCM.

total of 56 Bytes. Similarly, each static row requires 52 Bytes (12 for the sizes, 32 for the weights, 8 for synapse and neuron ID). Therefore, the total synaptic matrix will be $56 \times 1284 + 52 \times 500 = 97.9$ KB, which, if divided by the number of synapse cores for that chip (14) gives 7 KB, which can easily be stored into DTCM, therefore eliminating the need for transfers to SDRAM and of responding to the DMA complete interrupt.

One limitation of the all-to-all connector provided by the standard SpiNNaker toolchain is to assign the same weight value to all the weights. The MNIST task requires random weight matrices [SPCBS18], with uniform distribution between -1 and 1. Support for this feature has been added to the matrix generator, providing the capability of generating random numbers in the range provided by the user when the connections are specified, by using the support functions from the math library available to SpiNNaker.

4.5.4 A Plasticity Framework for the Heterogeneous Model

The Heterogeneous model described in Chapter 3 addressed static spike-based networks exclusively. In order to adapt this approach to on-line learning tasks it is necessary to include a framework for synaptic plasticity.

Synapse cores require postsynaptic information to update the synaptic weights

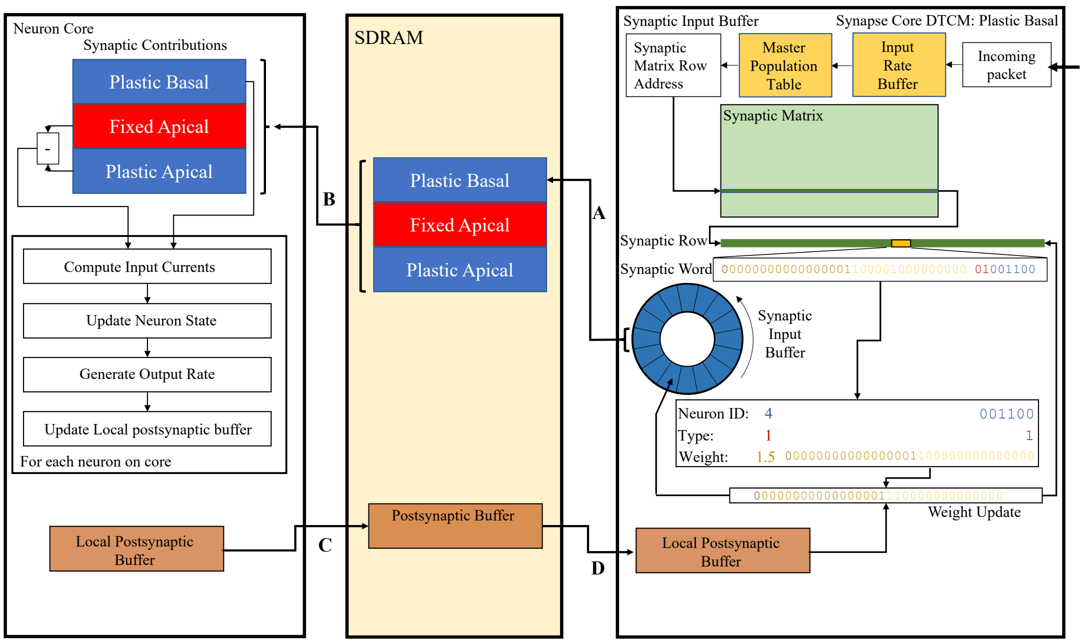


Figure 4.6: Memory usage and data structures for the pyramidal model on SPiNNaker. SDRAM contains both the synaptic contributions regions and the postsynaptic buffer. Represented cores are one neuron core (left) and one Synapse core implementing plastic basal synapses (right). Bold capital letters mark memory transfer corresponding to Figure 4.7.

correctly (similarly to what is described in Section 2.6.5). The data required by the *Synapse* cores from the Neuron cores changes according to the implemented neuron model and the compartment type. In all the cases, however, this data can be defined as a fixed point S1615 value representing the difference between two rates (or voltages in the case of pyramidal dendritic apical compartments). The *Neuron* cores are therefore required to provide the *Synapse* cores with an array of these values, where each value is related to a postsynaptic neuron. This array is called postsynaptic buffer. This aspect represents a challenge in the implementation of this mechanism, since *Synapse* cores cannot proceed updating the synaptic weights, and so processing the synaptic events, until the postsynaptic buffer is received. The postsynaptic buffer, on the other hand, can be transmitted only after all the neurons on the *Neuron* core have been updated for the current timestep, which subtracts an additional portion of the timestep from the synaptic update time.

A schematic of the memory interactions for the Heterogeneous partitioning is shown in Figure 4.6. The execution flow for one *Neuron* core and one plastic basal *Synapse* core is presented. The callback interactions between one Neuron core and one *Synapse* core is shown in Figure 4.7, where the memory transfers are labeled according to Figure 4.6. The communication of the postsynaptic buffer is handled similarly to the contribution regions via shared memory (the contribution regions are shown in Figure 4.6, keeping the convention of plastic synapses in blue and static synapses in red, while the postsynaptic buffer is in brown). The *Neuron* cores allocate a second region at the beginning of the simulation large enough to accommodate all the data belonging to the postsynaptic buffer. *Synapse* cores will retrieve the memory addresses for both their contribution region and the postsynaptic buffer. The *Neuron* cores, after having updated all the neurons, start a DMA transfer to SDRAM (labelled with **C** in Figure 4.6) for the postsynaptic buffer, copying the content of the local postsynaptic buffer to shared memory. The *Synapse* cores use Timer2 to schedule a second event (during the synapse timer event, as shown in Figure 4.7), which will inform the core about when to start reading the postsynaptic region from memory. This amount of time can easily be precalculated by the Python toolchain, since the neuron update time is fixed and related to the neuron model, and the total time depends on the number of neurons per core. Therefore, when this time elapses all the *Synapse* cores start reading the postsynaptic buffer from shared memory (labeled with **D** in Figure

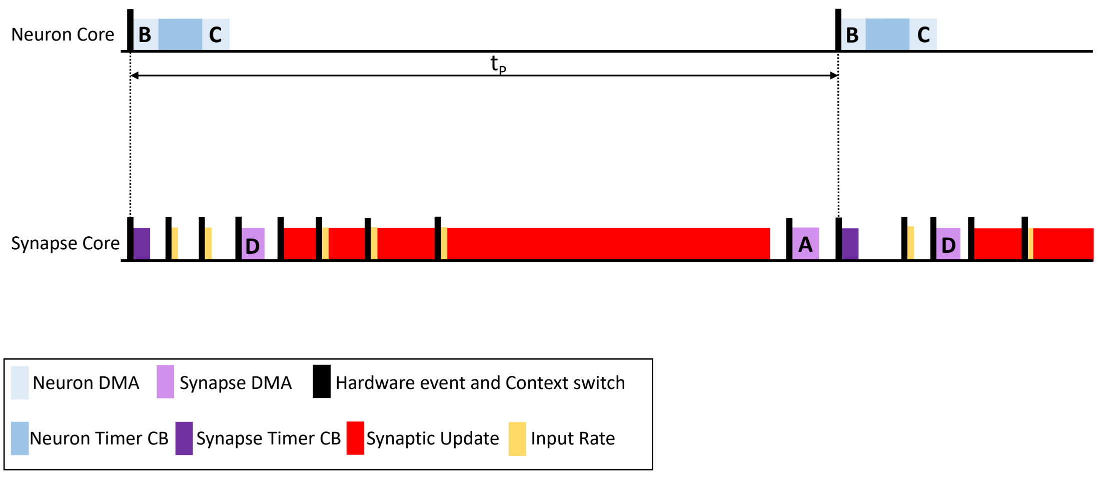


Figure 4.7: Callbacks interaction for the pyramidal model showing one *Neuron* cone and one *Synapse* core.

4.6, and 4.7).

In order to optimise the use of the timestep, all the incoming rates, before reading the postsynaptic buffer, are buffered and preprocessed (yellow events in Figure 4.7). This operation allows to receive all the packets and have the data related to the synaptic event processing ready for when the postsynaptic buffer is received. This is achieved by stopping the rate processing pipeline from processing synaptic rows, until the read event is completed, but the core is allowed to buffer locally the packet information. Therefore, for each packet received before the postsynaptic buffer read, the top right line in Figure 4.6 is executed, but the synaptic row is not retrieved from the synaptic matrix. After the read event is complete (light purple block in Figure 4.7), an event triggers the synaptic event processing (red block in figure 4.7). Some input rates can still be received at this stage, which will be treated the same way as spikes in the standard SpiNNaker toolchain (therefore unpacked and locally queued for processing). For this type of application, where the connectivity is very dense and the firing rate high, the rate processing pipeline takes most of the remaining portion of the timestep. Furthermore, the accumulated input packets allow to perform all the synaptic processing without stopping and restarting the pipeline most of the times, eliminating the start-up cost.

4.5.5 Rate Live Injector

Given the on-line nature of the learning task, it is beneficial to have a source population able to take live input and feed it to the network. Furthermore, given the sizes of common datasets in the field, such as Imagenet [DDS$^+$09] or eMNIST [CATvS17], it would become impractical, if not impossible to preload locally the full set onto systems with limited memory such as Neuromorphic hardware.

This led to the development of the Rate Live Injector, which allows to sequentially read one input image and then split this into the correct number of inputs required by the postsynaptic partitioning. Therefore, while one input image is presented to the network, the injector acquires the next image and prepares it. This is performed via alternating two shared memory regions. This system is composed of a single controller and multiple sources. The controller preprocesses each image and makes it available for the sources, while each source transmits one pixel in the form of a rate, with multiple sources on each Source core. Each SpiNNaker chip implementing a Rate Live Injector contains one Injector core and up to 14 Source cores. Each Source core can implement multiple sources which are handled similarly to neurons, therefore they will fire within the boundaries of the simulation timestep. The mechanism is shown in Figure 4.8.

The Controller core (left) receives the next image to be presented to the network and fragments it according to the number of Source cores on the chip. The fragmented image is then written into a shared memory region (labelled Future Rates in Figure 4.8) to be processed by the Source cores. In the MNIST case, each image is stored into SDRAM in the form of an array of images given the limited size of the dataset and for simplicity, however it is possible to receive each image from external sources such as sensors or DVS cameras [LPD08]. The Source cores read the current rates to be presented to the network from shared memory (Current Region in Figure 4.8). Each Source core reads from its own subregion from the memory buffer and then starts sending the rates according to the number of sources they implement, by extracting each pixel value and sequentially generating Multicast packets with payload to be sent to the network. Each image needs to be kept stable on the network inputs for a fixed minimum amount of time, which depends on the simulated network. This amount of time is received by both the Controllers and Sources as a refresh rate in the form of a number of timesteps. Therefore, the Source cores maintain a counter for each image they present to the network. When this timer elapses, the Source cores

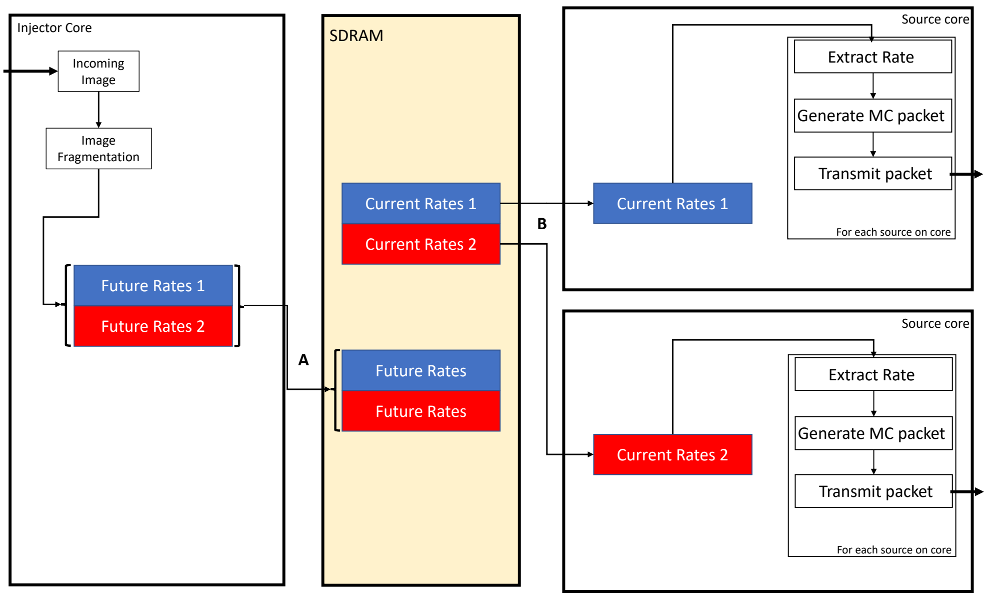


Figure 4.8: Memory usage and data structures for the Rate Live Injector implementation.

perform a new read to shared memory to retrieve the next image. This allows to reduce the number of shared memory accesses. In order to maintain efficiency, at the beginning of the simulation, the Controller core of each chip allocates two memory areas (Current Rates and Future Rates in Figure 4.8). This allows to process the next image while the current image is still being transmitted, especially when the exposition time is very short. The two memory buffers are then swapped every time a refresh occurs (which means that Current Rates becomes Future Rates and vice versa at every memory read/write).

By allowing up to 14 Source cores per chip, it is possible to match the number of postsynaptic partitions, therefore controlling the output traffic. Multi-chip implementations of Rate Live Injectors are also allowed. The Python side of this module allows specifying how many resources to allocate for it. This is convenient for datasets having larger images, as it spreads the number of sources over a larger number of cores. The toolchain is able to configure the application according to the number of required partitions (on the postsynaptic side) and required source chips. The dataset will then be fragmented to be sent to the required number of chips and the controllers will be informed about their respective offset in the image.

4.5.5.1 Packet Compression

The rate injector implementation presented in Section 4.5.5 generates very intense traffic. For the MNIST microcircuit the input layer sends 784 packets per timestep, which will be routed to all the chips implementing the pyramidal population of hidden layer 1. The SpiNNaker communication network struggles to deal with such intense traffic of packets with payload, resulting in a large number of packets dumped by the routers and then reinjected, which get delivered late to postsynaptic cores affecting the synaptic event processing. Given the presented issue, in this section, a compression mechanism to alleviate communication network overloading is described.

Each pixel intensity value for the MNIST dataset can be expressed between 0 and 255 and this value is then normalised to be between 0 and 1. Such information can be represented in 8 bits. The payload of a SpiNNaker multicast packet is 32-bits wide, which means that, since the input connection is all-to-all (all postsynaptic neurons need to receive all the packets) it is possible to store 4 pixel values per packet, by correctly shifting them into the payload. This operation reduces the peak transmission for the input layer to a quarter, as well as reducing the number of sources to be implemented to a quarter. This also results in fewer packets being received at the destination, resulting in having the information locally earlier in the timestep. A schematic of the mechanism is shown in Figure 4.9.

Source cores (left in Figure 4.9) retrieve 4 input rates from the current local buffer and through shift and mask pack them into a single 32-bit value. This value is then used as payload for multicast Packets. *Synapse* cores (right in Figure 4.9), receive the packet and through shift and mask convert the payload into four S1615 values to be stored into the Input Rate Buffer for standard processing.

The only drawback of this operation is a higher cost upon packet reception for the postsynaptic *Synapse* core. Each core needs to unpack the data into 4 values, instead of simply converting and storing the payload. This means performing 3 additional mask and shift operations per packet. This operation is, however, amortised by the reduced number of packets (only a quarter are received compared to previous implementations), therefore by fewer event responses. This compression strategy, however, can only be applied to the input layer, where the information is encoded on 8 bits. For the neurons' output rates the maximum precision given by the S1615 data type is required, therefore each packet can only

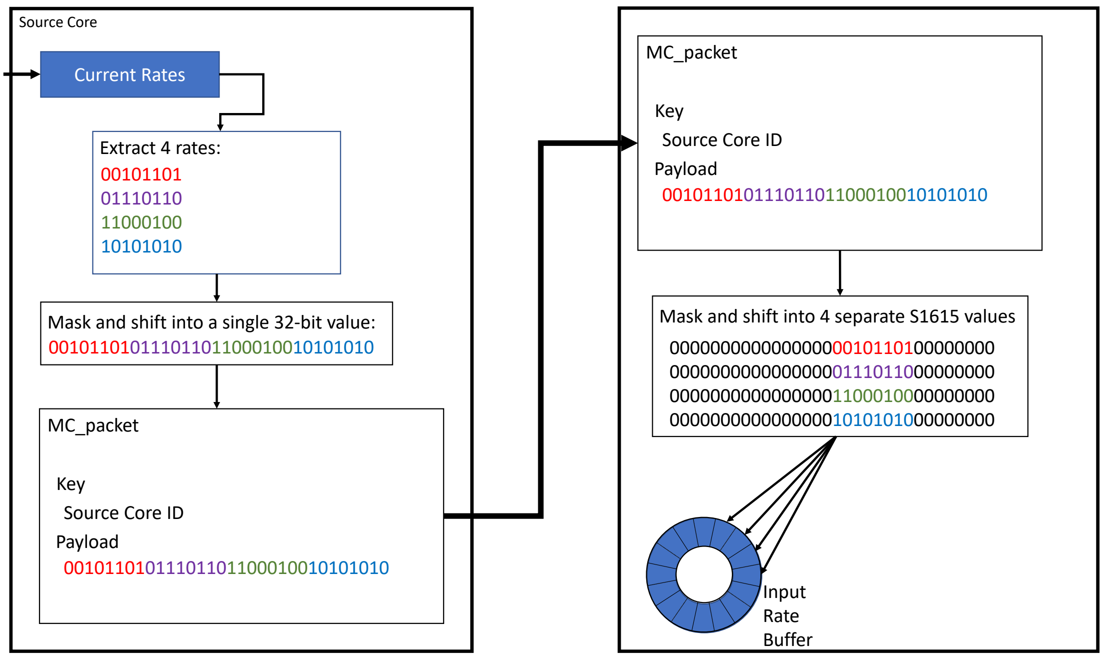


Figure 4.9: Rate compression mechanism. Source cores (left) compress 4 values into one packet and send it over the network. Synapse cores (right) receive these packets, uncompress the values and store them into the Rate input buffer.

carry a single rate.

4.6 Results

This section presents results of the execution of the multicompartment models on SpiNNaker. First the correctness of the implementation is addressed, by replicating the experiments shown by the authors, then the MNIST classification problem is investigated.

4.6.1 Correctness of the US Learning Rule

The first experiment aims at verifying the correctness of the learning mechanism of the Urbanczik-Senn model (described in Section 4.2). In order to demonstrate the correct behaviour of the model, the original experiment presented by the authors is addressed [US14]. This experiment consists of a supervised learning task, where a single two-compartment model receives random inputs on its dendrites through a repeating 200 ms pattern. The replica presented here receives input

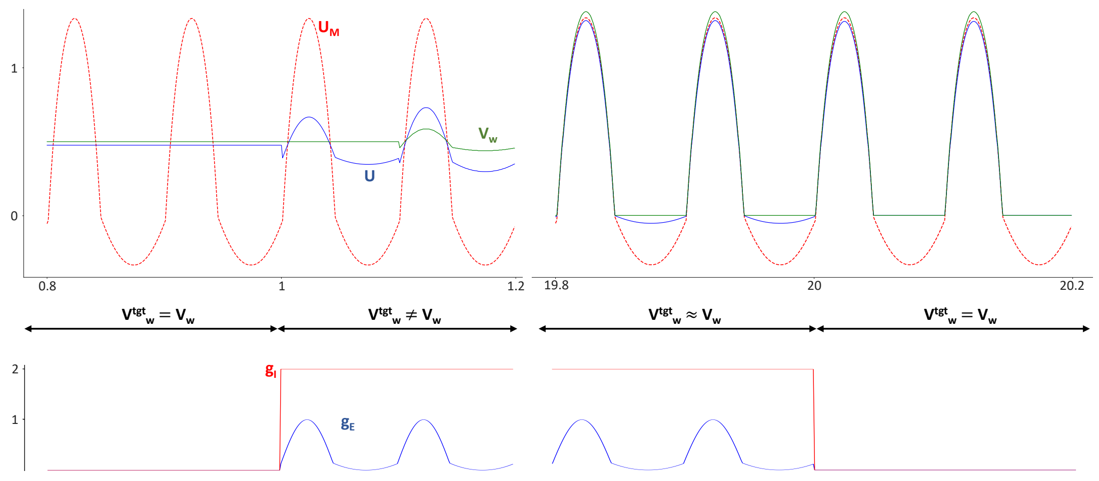


Figure 4.10: SpiNNaker simulation of the Urbanczik-Senn plasticity experiment.

from 200 different input sources as constant input rate, matching the value presented in the original experiment. The total simulation time is 20.2 s. After 1 s of simulation the somatic synapses are activated providing an oscillatory excitatory input and a constant inhibitory input. The combination of these two inputs encodes a target potential called U_M (or matching potential) by the authors. The activation of the somatic synapses creates a voltage difference between the two compartments, as the somatic potential starts to converge towards the matching potential. This difference triggers the synaptic plasticity in the dendritic synapses and the dendritic voltage slowly converges to the target potential. After 20 s the somatic input is removed and the target output is still produced by the neuron, but in this case is solely driven by the dendritic input, as the somatic synapses are silent, therefore the learning happened.

Results of the execution of the experiment on SpiNNaker are shown in Figure 4.10. The red dashed line is U_M, the somatic potential is blue, while the dendritic potential is green. The four phases are indicated below the voltage plot, where originally V_w matches the target potential, then the difference is introduced creating divergence between the two potentials. V_w starts then to converge to the target potential and finally reaches it. The bottom side of the plot shows the evolution of the excitatory (g_E) and inhibitory (g_I) conductances over time. The inhibitory value is static, while the excitatory is obtained by combining a sinusoid between 0 and 1 with a parabola for the lower phase, in order to match as closely as possible the results of the authors in the absence of a defined curve. The experiment shown here is rate-based and the input rate is denoised (fixed to the average value shown in the paper), to ensure that the model behaves correctly. The adopted colour scheme matches the results presented in the paper. In the final stages of the learning phase, the dendritic potential converges to 0 and does not reach the minimum value expressed by U_M, in accordance to what shown in the original experiment. This can be explained by the nature of the learning rule (expressed by Equation 4.6), which employs output rates, which cannot be negative, as they are obtained as rectified linear unit or sigmoid of the compartments' voltage. This means that when the somatic potential is negative, the rate calculated over the target voltage becomes 0. Since the dendritic potential is also 0, there is no learning happening, therefore the dendritic compartment keeps its voltage value.

It is possible to obtain the same results with the adapted version of the US model for the dendritic microcircuits. The experiment in this context changes by directly providing U_M as a target to the somatic synapses, instead of the two separate excitatory and inhibitory components.

4.6.2 Correctness of the Dendritic Microcircuits Learning

After showing the correctness of the plasticity implementation of the Urbanczik-Senn model on SpiNNaker, the next step consists of simulating the learning mechanism presented for the dendritic microcircuits (detailed in Section 4.4). In order to perform this experiment a network analogous to that presented in Figure 4.3 is used. This experiment is based on the first microcircuit presented in the paper [SPCBS18]. In this case, the source is a simple rate source transmiting constant input values to the basal dendrites of the pyramidal neuron. The network is a two layer network having three populations with a single neuron each. The first layer contains one pyramidal neuron and one Interneuron (or SST neuron), the second layer contains one two-compartment neuron, so that it is possible to feed a teaching current directly to it (as indicated by [SPCBS18]). The bottom-up basal input is stable, while the teaching current is deactivated originally and then set to the value 1.0.

The output of the network is shown in Figure 4.11. The top plot represents the somatic voltage of the pyramidal neuron, the second plot contains the apical voltage, which increases when a teaching signal is presented, and then slowly decreases to 0, when the apical plasticity silences the compartment voltage. The basal compartment is subject to plasticity as well, therefore the voltage value

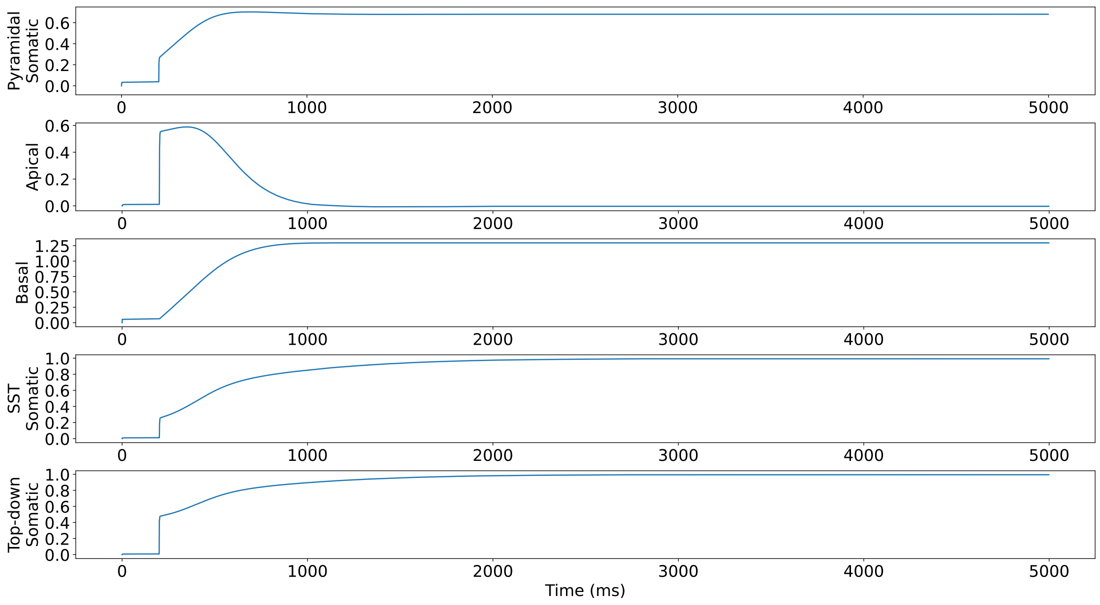


Figure 4.11: SpiNNaker simulation of the dendritic microcircuit learning rule. The simulated network is a two-layer network with 1 Pyramidal neuron, 1 Interneuron and a Top-down 2-compartments neuron, replicating the first experiment presented in [SPCBS18]. From top to bottom: Somatic potential, Apical Dendritic potential, Basal Dendritic potential of the Pyramidal neuron, Somatic potential of the Interneuron, Somatic potential of the Top-down neuron. The network slowly converges to the self predicting state.

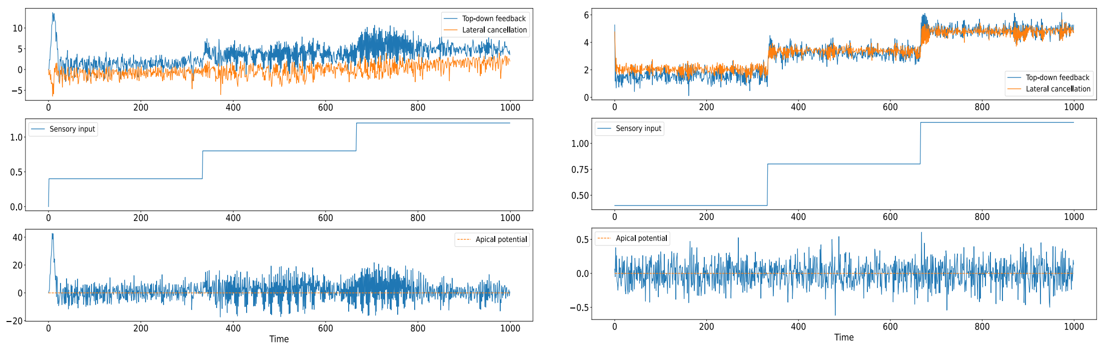


Figure 4.12: SpiNNaker simulation of the self-predicting state convergence. The simulated network is a three-layer network (30-20-10). Left side shows pre-learning outputs, right side shows post-learning values. From top to bottom: top-down output voltage compared to lateral cancellation, sensory input and apical voltage. Values sampled from random neurons in the network.

is initially stable when there is no teaching input, and then starts increasing until the network stabilises due to the learning rule (middle plot). The last two plots represent the somatic potential of the Interneuron (SST) and of the two-compartment output neuron respectively. The Interneuron tracks the behaviour of the output neuron and they both slowly converge to the teaching value, which is 1.0. The network is simulated in the absence of background noise, to prove the correctness of the mechanism.

4.6.3 Dendritic Microcircuit and Self-Predicting State Simulation

After having isolated the learning rule and shown the correct behaviour on SpiNNaker, the next experiment consists in simulating the convergence to the self-predicting state for a network. To demonstrate this, the experiment conducted in the supplementary material of the original paper [SPCBS18] was replicated. A three-layer network with one hidden layer and 30-20-10 pyramidal neurons (input-hidden-output) was simulated. The network does not receive any teaching signal, resulting in the output values being driven solely from the inputs. Background noise was introduced in this context as described by the paper in the form of input current injected into the neurons as white Gaussian noise having parameter σ=0.1. The results are presented in Figure 4.12.

The left side of Figure 4.12 shows pre-learning values, while on the right side post-learning values are shown. The top line presents top-down somatic voltage (blue) compared with lateral cancellation (orange). The values are sampled from a random neuron from the output layer and the corresponding Interneuron from the hidden layer, consistently with the original experiment. The middle line shows the sensory input presented to the network. The same pattern was presented 1000 times before sampling the post learning values. Finally, the bottom line shows the apical compartment voltage from a randomly sampled neuron. It is possible to notice that, in the pre-learning stage, the top-down feedback and lateral cancellation voltages are misaligned, together with large oscillations of the apical voltage. When the network starts to converge to the self predicting state, apical fluctuations are reduced (oscillating between -0.4 and 0.6 in a post learning situation) and the lateral Interneuron learns to match the top-down signal. The remaining small oscillations are caused by the background noise acting on the compartment potentials.

4.6.4 Nonlinear Regression Task

A more complex task consists of approximating non-linear output generated by another network. Therefore a 3-layer network of size 3-4-2 (input-hidden-output) can learn to associate sensory input to the output of a second network (having the same size) that transforms the same input.

The first network will be referred to as “student” from here on, while the second one as “teacher”. The teacher network does not receive any teaching signal, therefore the outputs are solely driven by the input, while the student network receives the outputs of the teacher network as teaching signals. The two networks have the same structure, but different weight initialisation, but in both cases the values are drawn from a uniform distribution in the range [-1, 1]. Results of the execution are presented in Figure 4.13.

The structure of Figure 4.13 is analogous to Figure 4.12, where the left side shows pre-learning outputs and the right side shows convergence after the learning phase. The sensory input is shown in the middle line and has a period of 1000 timesteps (1 s). The same pattern was presented 10000 times at the inputs of both networks and a convergence is reached after the learning phase. The background noise is activated for the student network and the convergence of the output is observed (the orange line shows the student’s outputs, while the blue line the teacher’s). Similarly to the previous experiment, the presented output is sampled

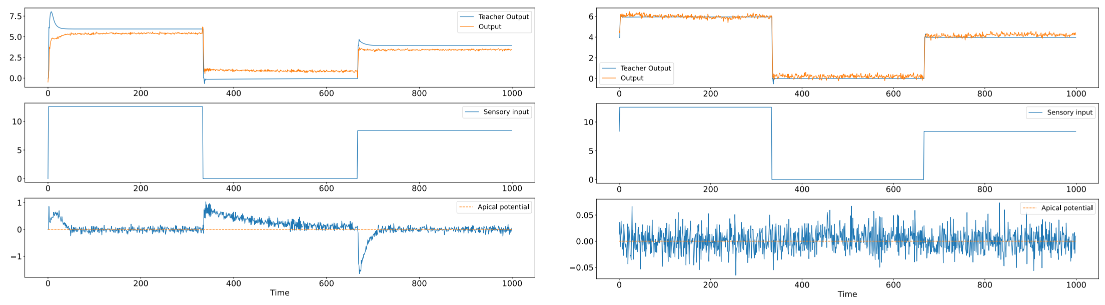


Figure 4.13: SpiNNaker simulation for a nonlinear regression task. The two simulated networks are 3-4-2 size. Left side shows pre-learning outputs, right side post-learning. From top to bottom: comparison between teacher output and student output, sensory input and apical compartment convergence; samples are taken from randomly chosen neurons.

from random neurons (matching the indices between the two networks). The self-predicting state is reached, as shown by the bottom line, as the apical voltages converge to 0 and oscillate between -0.06 and 0.06.

4.6.5 The MNIST Challenge

The most challenging task explored here is the classification of the MNIST dataset. Several features introduced in Section 4.5 are developed to target this task. The full network, shown in Figure 4.4, can be placed on 159 SpiNNaker chips (8 neurons per chip for the Pyramidal populations, resulting in 125 chips and 16 neurons per chip for the Interneurons, resulting in 34 chips), plus 2 chips for the sources and 1 for the teacher, resulting in 162 chips. This requires 4 SpiNNaker boards. According to the authors a good convergence (below 2% error) is reached after 200 trials of the dataset [SPCBS18], however the authors do not specify how long every image should remain stable at the inputs of the network.

The first simulations were performed with 100 timesteps per input, running 200 trials of the dataset. The SpiNNaker system, however, struggles with such a network, as the communication network cannot deal with this traffic. For the first trial the system seems to cope with the amount of information (while recording some packet losses), however, running the network for longer results in these numbers largely growing, having packets delivered several timesteps later than expected. This is exacerbated by the board-to-board connections handled by the FPGAs, infact smaller networks spanning over a single board experience fewer critical losses. Ad-hoc placement strategies might mitigate the issue, however these are not explored in this thesis.

4.7 Discussion

Learning tasks are computationally expensive and power hungry. Traditional approaches such as error backpropagation with gradient descent require considerable time for the error backpropagation phase, where the simulation needs to be paused until the gradient descent phase is completed. This becomes impractical for large networks, where learning times become dominant over simulation times and power consumption grows. These methods furthermore are believed to be biologically-implausible [IGB19, Hun]. Several attempts at finding biology-like learning rules are presented nowadays. One example is given by the Dendritic Microcircuits presented in this chapter and developed by Sacramento et al. [SPCBS18]. The learning approach can be potentially implemented on Neuromorphic hardware, by exploiting strategies such as the Heterogeneous model in order to process all the incoming information within the timestep boundaries. However, the dense nature of this problem shows additional limitations of Neuromorphic hardware, which struggles to deal with such intensive communication at the network level. Therefore more effective placement strategies are required to tackle the bottleneck caused by the SNN, in order to correctly simulate more complex tasks.

4.8 Summary

The aim of this chapter is to address the second research question presented in Section 1.2, which states: *What are the challenges of implementing on-line learning algorithms in real time on Neuromorphic hardware?*

These are presented throughout the chapter, together with possible approaches targeting them. The implementation of such rules is achieved through the application of the Heterogeneous Programming model to the context of on-line learning strategies employing multicompartment neuron models. The presented learning rule uses rate-based networks to approximate the error backpropagation algorithm. The correctness of the approach has been explored on SpiNNaker through isolated simulations of a controlled network, and through the use of the Heterogeneous model, together with optimisation techniques enabling simulation of neural dynamics in real-time. However, because of the dense connectivity patterns, combined with intensive communications, the underlying communication network struggles to deliver all the information to the destinations in time. This prevents the correct execution of more complex tasks, which will require more targeted placement strategies to optimise the use of forwarding points.

Chapter 5

Multi-Target Partitioning

5.1 Introduction

Large biologically-representative SNNs, such as the Multi-Area model of macaque's visual cortex [SBS$^+$18], simultaneously present extremely sparse connectivity patterns among different regions, representing long-range connections, and very high neuron input activity, due to the massive number of simulated neurons. Despite the improvements shown in Chapter 3, when scaling up parameters such as sparsity of connections and neurons fan-in to these levels, the Heterogeneous Programming model fails to achieve real-time performance. This Chapter therefore presents a new partitioning strategy evolving from the Heterogeneous Programming model, to address higher sparsity levels for SNN simulations and to further increase the peak synaptic event processing per timestep, allowing a higher density of neurons per chip compared to previous results.

This new model originates from the idea of extending the concept of neural ensembles presented in Chapter 3, by employing Multi-Target *Synapse* cores. This means that each *Synapse* core can now receive synaptic inputs for more than one postsynaptic target *Neuron* core, instead of employing a single *Neuron* core per ensemble as for the Heterogeneous Programming model. This work builds on all the previously presented aspects, starting from the Heterogeneous model, including plasticity. The focus of this study is real-time simulation of spike-based biologically-representative SNNs addressing extremely sparse examples. The work presented in this chapter has been published in the form of journal article [PR22].

Section 5.2 presents an introduction to this work and the motivations that led to it. Section 5.3 describes the SpiNNaker implementation of the Multi-Target partitioning, detailing the used memory structures and the changes to the software toolchain both for the static and plastic case. Section 5.4 shows the results of this approach, in terms of memory contention and response time, peak synaptic event processing and handling of sparsity, together with comparisons with previous results.

5.2 Why Multi-Target *Synapse* Cores?

The improvements offered by the Heterogeneous Programming model presented in Chapter 3 allowed to increase efficiently the number of synaptic events that can be processed per timestep, by a performing targeted parallelisation of tasks. The introduction of *Synapse* cores allows decoupling of the spike processing phase from the neural state update, through the use of dedicated units for each different task. Through this approach, it is possible to exploit the parallelism of digital Neuromorphic hardware, such as SpiNNaker, to allocate multiple synaptic processing units, in order to increase the input processing capabilities per neuron. This results in a linear increase in the synaptic event throughput, according to the number of employed synaptic units. Thanks to the synapse and neuron decoupling it is also possible to perform a more efficient partitioning of the synaptic matrices, generating longer synaptic rows, as *Neuron* cores can simulate a larger number of neurons, having the sole task of updating their internal state. The matrices are therefore partitioned horizontally, and so by presynaptic id, resulting in the *Synapse* cores implementing all the synapses (for all the postsynaptic neurons implemented by the connected *Neuron* core) for a given presynaptic neuron, and therefore the full synaptic row. Longer synaptic rows mean fewer accesses to memory, which in turn result in fewer context switches and contention, and in amortising the memory access time over a larger number of synapses. Longer synaptic rows improve the handling of sparser networks too, as they reduce the probability of zero target spikes, which are spike packets carrying a spike with no target neurons when received on a postsynaptic core (i.e. the synaptic row is empty). These are common in sparse SNN simulations, where the connectivity is low and therefore presynaptic neurons form few connections with postsynaptic neurons. This results in spike packets to be generated and delivered, but only a limited number of postsynaptic processors will produce synaptic events, due to

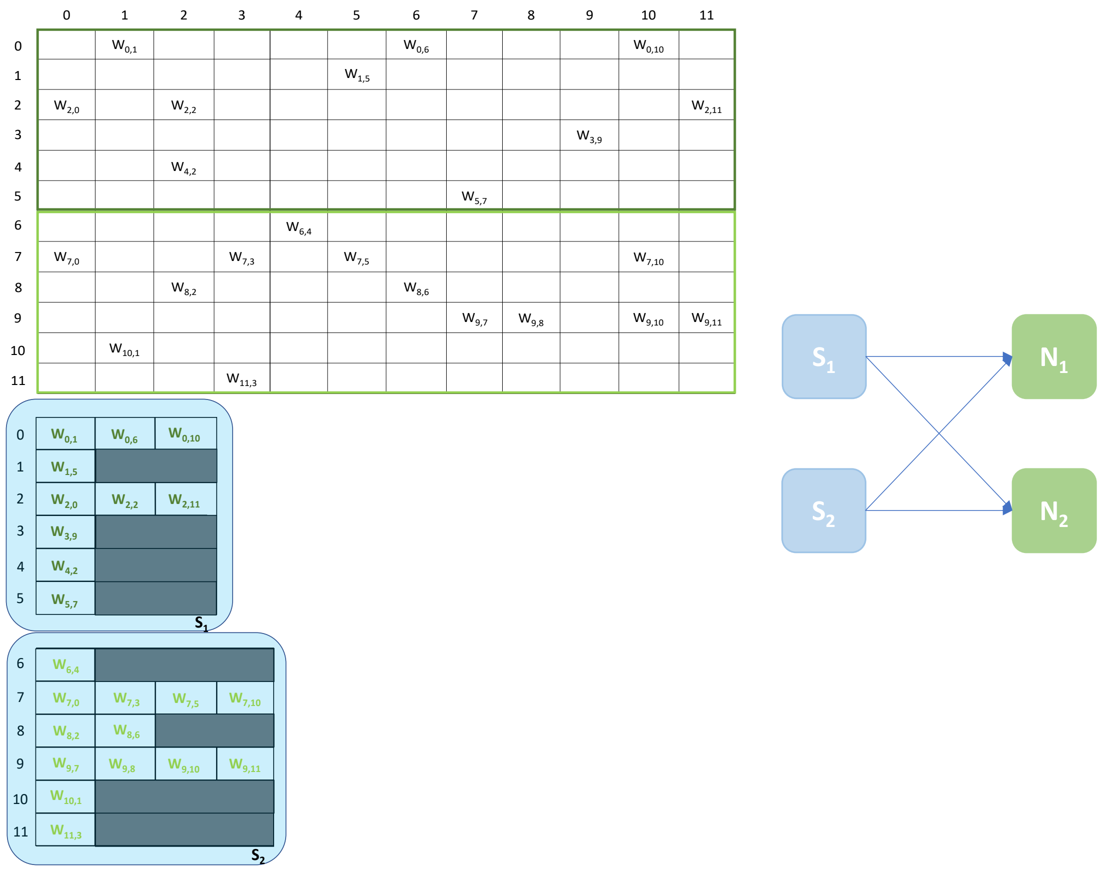


Figure 5.1: Synaptic matrix partitioning for the Multi-Target approach. The used network is the same shown in Figure 2.9 and Figure 3.1 Here *Synapse* cores have much longer synaptic rows, further reducing the risk of empty rows, therefore fewer resources are required. The generated ensemble is shown on the right.

the low connectivity probability. These packets represent an issue, as they can only be detected after a row has been retrieved from SDRAM, resulting in wasted computation. Furthermore this type of packet represents additional information which is propagated through the communication infrastructure, increasing the risk of routers congestion.

The Heterogenous Programming model only partially addresses the issue of zero target spike packets, as the number of synapses per *Synapse* core is limited to the number of neurons that can be simulated on a single *Neuron* core. This limits the maximum synaptic row length to the number of neurons per *Neuron* core.

This chapter presents an evolution beyond this limitation. By allowing *Synapse* cores to target multiple *Neuron* cores, it is possible to generate even longer synaptic rows, further reducing the chance of zero target packets. This in principle extends the synaptic row length limit to the sum of the postsynaptic neurons that can be simulated on a chip, instead of a single core. This approach is also advantageous because it offers a customised strategy, which can adapt to the network to be simulated. Therefore sparser SNNs can benefit of a reduced number of *Synapse* cores, in favour of more target *Neuron* cores and, viceversa, networks presenting a high fan-in will map ensembles with a predominance of *Synapse* cores.

An example of matrix partitioning according to the Multi-target approach is shown in Figure 5.1. The synaptic matrix shown in this example is the same used for Figure 2.9 and Figure 3.1. In the Multi-target case, by allowing *Synapse* cores to target 2 *Neuron* cores each, the length of the implemented synaptic rows per *Synapse* core doubles, therefore only 2 *Synapse* cores are now necessary, resulting in no empty rows. This new partitioning approach also performs a more efficient use of the available hardware resources (as shown in Figure 5.1), reducing the number or processors necessary to simulate the same network, compared to the Heterogeneous Programming model. This results in a higher density of neurons per chip compared to the example shown in Figure 3.1.

$$E = [\frac{t_p - t_{1^{st}} - t_{last}}{t_{spike}} + 2]PN \tag{5.1}$$

$$t_p = \Delta t - t_w - t_r \tag{5.2}$$

$$t_w = 1.34 + 0.95S_c - 0.02N_c + 0.58N_cS_c \tag{5.3}$$

$$t_r = 1.22S_c + 0.4N_c - 0.21 \tag{5.4}$$

$$N = nN_c \tag{5.5}$$

$$t_{spike} = m_sPN + c_s \tag{5.6}$$

To evaluate the performance of the Multi-target approach, an updated cost model is required. The maximum amount of synaptic events that can be processed in a timestep, can be obtained by evolving the models presented in Chapter 2 (Equation 2.7) and Chapter 3 (Equation 3.1). This value is now modeled by Equations 5.1-5.6.

The components are similar to the Heterogeneous Model case, where E is the number of synaptic events per timestep per *Synapse* core, t_p is the fraction of the timestep available for spike processing, $t_{1^{st}}$, t_{last} and t_{spike} are the times necessary to process the first, the last and a generic spike in the pipeline respectively. P represents the connectivity probability. However, differently from the Heterogeneous partitioning case (Equation 3.1), N depends now on the number of *Neuron* cores connected to each *Synapse* core. This value is obtained by multiplying the number of neurons per core (n) by the number of connected *Neuron* cores (N_c), as shown in Equation 5.5. This reflects also on the spike processing times, as shown by Equation 5.6, where the variable cost is now multiplied by the total number of neurons targeted by the spike, therefore by the *Synapse* core, as the number of postsynaptic targets per spike is now larger. t_p is obtained by subtracting from the full timestep duration (Δt) the reading and writing contributions times (t_r and t_w respectively), similarly to the Heterogeneous case. Reading (t_r) and writing (t_w) times however now depend on the structure of the ensemble, as both contention and size of the transfer play a key role. The values reported here are obtained by profiling execution. These reflect the measurements presented in Section 5.4.1.1 for t_r and 5.4.1.2 for t_w. Both the equations are obtained as linear regression of the measured values and take into account the number of involved *Synapse* and *Neuron* cores of the ensemble.

5.3 Multi-Target *Synapse* Cores Implementation

This Section describes the implementation of the Multi-target partitioning on SpiNNaker. This approach requires a restructuring of the used memory structures, as well as a reorganisation of tasks among different cores. The network partitioning is performed in the same way as for the Heterogeneous model, however, the core interactions and dynamics change significantly. These changes are addressed separately in Section 5.3.1 for the memory allocations and Section 5.3.3 for the cores interactions. Section 5.3.4 presents the adaptations required to include the synaptic plasticity framework into the Multi-target implementation of the neural ensembles.

5.3.1 Memory Usage

The shift in paradigm presented in this chapter requires a different memory arrangement, and consequently results in an increase in shared memory accesses per timestep, to retrieve the synaptic contributions. Shifting from a many-to-one (neural ensembles for the Heterogeneous model, having multiple *Synapse* cores targeting a single *Neuron* core) to a many-to-many scenario (the Multi-target partitioning) brings additional complexity on the shared memory regions handling. Furthermore, more accesses results in increased memory contention, with a further reduction in the effective spike processing time (as shown in Equations 5.2-5.4).

An efficient memory allocation and use is therefore required. This section addresses two steps taken to mitigate this potential memory issue. The first consists in a new distribution of the shared regions for the neural ensemble, while the second proposes the use of multiple memories in order to reduce access contention.

5.3.1.1 Memory Regions Redistribution

The Heterogeneous model shared memory interaction was based on a larger shared memory block allocated in SDRAM by the *Neuron* cores, which was accessed in form of smaller subregions by the *Synapse* cores (as detailed in Section 3.2.2.2). For the Multi-Target partitioning, the memory allocation is reversed. An example of the data structures employed for this new method, using 2 *Synapse* cores and 3 *Neuron* cores, is shown in Figure 5.2. Each *Synapse* core allocates a memory region which is large enough to store all its synaptic contributions. Therefore, this contains all the synaptic input buffer slots for all the neurons implemented by the targeted *Neuron* cores. The *Neuron* cores access their own subregions by retrieving the subset of contributions for their neurons for the current timestep. This allows each *Synapse* core to perform a single write operation to shared memory per timestep, instead of one per target *Neuron* core. The cost is however paid on *Neuron* side, as *Neuron* cores have to perform as many reads as the number of afferent *Synapse* cores. The reason behind this choice is due to the higher efficiency on SpiNNaker of read operations compared to write operations [PPG$^+$13]. Therefore it is more efficient to have a single larger write on one side combined with multiple smaller reads on the other side, than vice versa.

5.3.2 Dual Memory Use

The memory contention problem has a large impact on the portion of the timestep available for spike processing. Previous works demonstrated that multiple cores accessing shared memory at once show faster degrading performance than performing fewer longer transfers [RBB+18, RPR+19]. A viable solution to tackle this problem is to make use of both the shared memories provided on chip. As described in Section 2.5.1, each SpiNNaker chip as an off-chip SDRAM 128 MB wide and an on-chip SysRAM 32 KB wide. Part of the SysRAM is used by the Monitor core for system operations such as handling of Point-to-Point packets or monitoring purposes. However, from experimental readings, 22 KB are available and can be used by Application cores. One adidtional aspect that makes SysRAM potentially a good choice to alleviate memory contention is that the two shared memories (SDRAM and SysRAM) are on different bars of the cross-bar composing the System NoC, therefore two separate cores accessing the two different memories at the same time would not create contention. A drawback of using SysRAM however is that, although the access is faster than SDRAM (being SysRAM on-chip), the transfer data rate is lower, as the SDRAM memory has a DDR implementation. This can however be averaged out with the faster access when transfers are relatively small, such as the case of synaptic contributions.

A dual memory system is therefore employed (as shown in Figure 5.2), which informs the *Synapse* cores on which memory to use as destination for their synaptic contributions. The shared memory is selected according to the physical *Synapse* core ID. This allows to spread evenly the contributions between the two memories. Both memories are part of the system memory map, therefore the allocation can be performed simply by specifying the correct memory heap, and the address retrieval is transparent to this operation.

5.3.3 Cores Interaction

Together with a different memory allocation, the cores interaction for the Multi-Target partitioning presents some differences. A schematic of the employed memory structures and the interactions is presented in Figure 5.2. The task scheduling is analogous to that shown in Figure 3.4, however the memory structures and the interactions between them and the cores change according to Figure 5.2.

The example shown in Figure 5.2 presents 2 *Synapse* cores targeting 3 *Neuron*

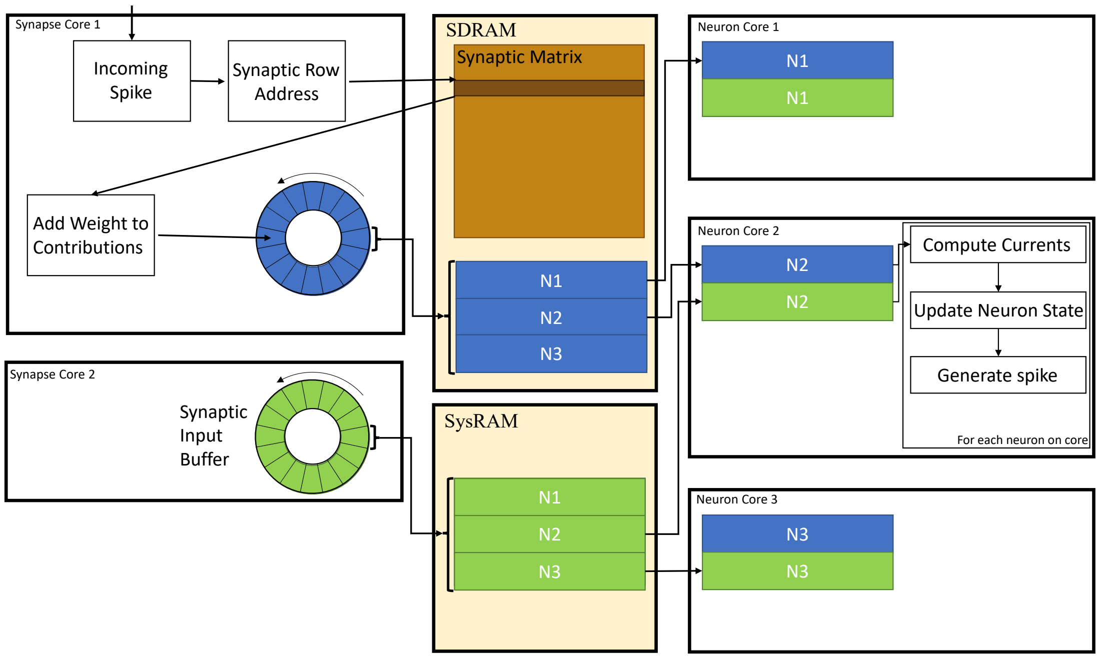


Figure 5.2: Memory usage and data structures for the Multi-Target partitioning approach on SpiNNaker (static version), for the case of 2 *Synapse* cores targeting 3 *Neuron* cores each. Both shared memories are shown with region allocation.

cores each. The host side SpiNNaker toolchain is provided with an additional parameter per Population which indicates the number of target *Neuron* cores per *Synapse* core. According to this value, a list of parameters is generated. This includes the correct offset for neuron indices and the list of connected core IDs (which means the afferent *Synapse* core IDs for each *Neuron* core and the postsynaptic *Neuron* core IDs for each *Synapse* core). The whole ensemble needs to be placed on a single chip, as for the Heterogeneous partitioning. Therefore this constraint is specified to the mapper in the SpiNNaker toolchain. Each *Neuron* core receives one offset parameter. This indicates its position in the neural ensemble from the *Synapse* cores perspective. This parameter is useful to locate the memory subregion from which the *Neuron* core will read the contributions.

At the beginning of the simulation, each *Synapse* core allocates its contribution region in one of the two shared memories, corresponding to its core ID, as described in Section 5.3.2 (these correspond to the blue region for *Synapse* core 1 and to the green region for *Synapse* core 2 in Figure 5.2). Each region is tagged using the core ID as unique identifier. During the first simulation timestep, each *Neuron* core retrieves the array of memory addresses corresponding to each afferent *Synapse* core, by using the list of connected *Synapse* core IDs provided by the Python toolchain (therefore *Neuron* core 1 has N1 from both the blue and green regions, *Neuron* core 2 has N2 and *Neuron* core 3 has N3 in Figure 5.2). The SpiNNaker API allows retrieval of memory addresses from different heaps, by just specifying a memory tag, provided that this memory region is inside the system memory map. This last operation is performed after the neuron state update for timestep 0, when the *Neuron* cores are usually idle. This guarantees that possible spikes due to above threshold initialisations are generated as soon as possible in the timestep. Furthermore, during the first timestep the *Neuron* cores do not receive any synaptic contributions, therefore it is sufficent to initialise the local regions to 0 values and to skip the read phase for this timestep.

As shown in Figure 5.2, *Synapse* cores receive spikes starting from the first timestep and process them in the same way as described in Chapter 3. This entails unpacking each spike, retrieving the synaptic row address through the Master Population Table, reading the row from the Synaptic matrix in SDRAM and then adding the synaptic weight to the correct contribution slot of the synaptic input buffer (blue and green for *Synapse* core 1 and 2 respectively), according to neuron ID and delay. The difference in the Multi-target approach consists of having longer synaptic rows, which therefore increase the chance of connections in case of sparse networks, and also reduce the number of accesses to shared memory, due to a larger number of synapses read per row at a time. This is however paid at the cost of longer processing time per received spike packet (as shown in Equation 5.6). At the end of the timestep, the synaptic input buffer corresponding to the next simulation timestep is written to shared memory by the *Synapse* cores. Again, similarly to what is described in Chapter 3, a Timer2 event informs the *Synapse* cores to start writing the contributions, and the writing time is calculated according to the *Synapse* cores and target *Neuron* cores allocated for the simulation (according to Equation 5.3). The written contributions contain the synapses for all the target *Neuron* cores. Therefore, as shown in Figure 5.2, if there are 3 target *Neuron* cores, each *Synapse* core will write the synaptic input buffers for all of them in the correct memory (therefore *Synapse* core 1 writes into SDRAM its contributions for *Neuron* core 1, 2 and 3 and *Synapse* core 2 does the same into SysRAM).

Neuron cores, at the beginning of each timestep, send DMA read requests for the contributions subregions from the connected *Synapse* cores. The time required to read the memory regions is a crucial design parameter, because it sets a boundary on when the *Neuron* core can generate the first spike. In fact until all the contributions are read, the *Neuron* cores cannot start processing the neural state updates. This reflects on when postsynaptic *Synapse* cores can start receiving spikes, effectively reducing the spike processing window. It is therefore of paramount importance to reduce this reading interval as much as possible. In order to address this issue, *Neuron* cores are instructed to perform out-of-order read operations of the sub-regions. This means that, based on the *Neuron* core ID, the first read region will be either from SysRAM or SDRAM. This effectively halves the *Neuron* cores accessing the same memory at the same time, by explicitly instructing half of them to first read from SDRAM and the other half from SysRAM. After each read is completed, each *Neuron* core sends the subsequent request to the other memory.

Each *Neuron* core, before starting processing the neural state update, needs to wait for the completion of the read of all its memory subregions. This is performed, similarly to the Heterogeneous model, by having the cores busy waiting on a flag variable. In this case however, the variable is set to True when cumulatively all the reads have completed. Therefore each read completion triggers an event which increments a counter. When this counter reaches the cumulative value of the required reads (corresponding to the number of afferent *Synapse* cores), it sets the flag to True, enabling the *Neuron* core to start performing the state update.

The neural state update is performed sequentially per neuron similarly to the previous implementations, however, to optimise the contribution summation phase for each neuron, the synaptic data on *Neuron* core side is stored into a matrix-based structure, which is accessed as shown in Figure 5.3.

The example shows the case in which 2 postsynaptic *Neuron* cores implementing 8 neurons each receive contributions from 3 excitatory and 2 inhibitory *Synapse* cores. The contribution regions are stored into SDRAM and each *Neuron* core retrieves its subregion. After the data is locally loaded, each *Neuron* core has a local matrix where each row represents the contributions of a specific *Synapse* core, and each column contains the contributions for a postsynaptic neuron. The synaptic summation loop therefore, for each neuron, accesses the corresponding column and sums the contributions as shown in Figure 5.3. Since the number of *Synapse* cores per synapse type can change according to the application, this is provided by the Python toolchain to each core for each synapse type. The

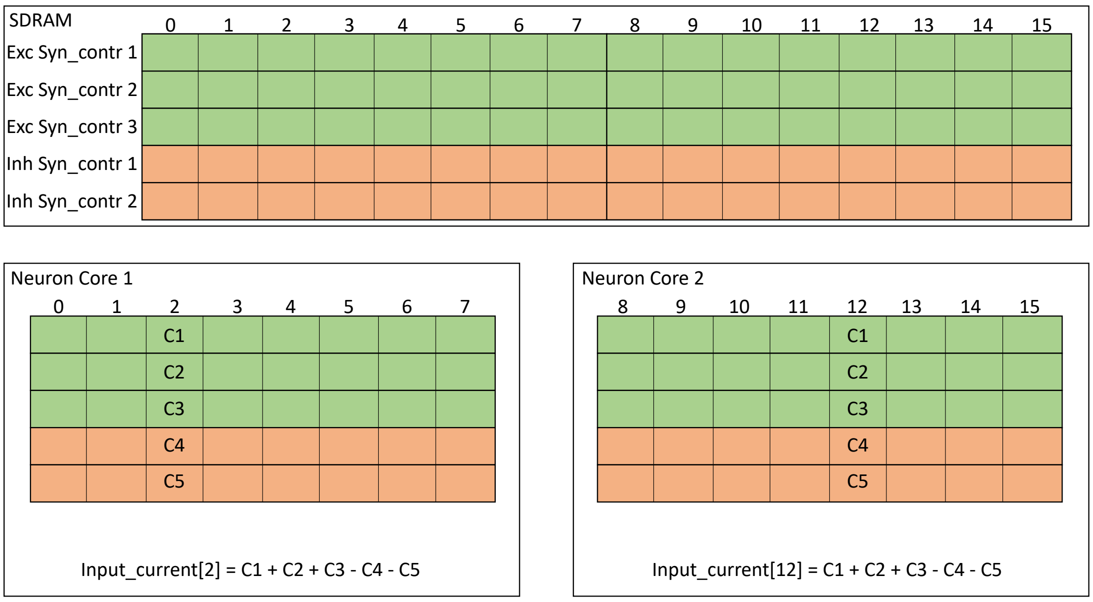


Figure 5.3: Synaptic contribution arrangement and summation for 2 postsynaptic *Neuron* cores receiving from 3 excitatory and 2 inhibitory *Synapse* cores.

offset calculation is therefore performed efficiently. Once the input current is correctly calculated, and all the contributions are added together, the neural state is updated according to the implemented neuron model.

5.3.4 Synaptic Plasticity

The Multi-Target partitioning can be extended to include support to synaptic plasticity. For plastic networks, the time required to process synaptic events is higher compared to the static case, as a weight update phase is needed. Therefore simulations of plastic SNNs would also benefit from reduced processing time per synaptic event. This section therefore shows the changes to the cores interaction in order to perform such a task. A schematic of the updated approach is shown in Figure 5.4.

The presented core allocation is analogous to that shown in Figure 5.2, where two *Synapse* cores target 3 *Neuron* cores each. The addition required by synaptic plasticity is a postsynaptic buffer including spiking information for the postsynaptic neurons. In a spike-based context, *Synapse* cores only need to be informed whether postsynaptic neurons generated a spike or not this timestep. The way this is handled is through an array of flags where each slot corresponds to a postsynaptic neuron. All *Synapse* cores share the same postsynaptic region (red

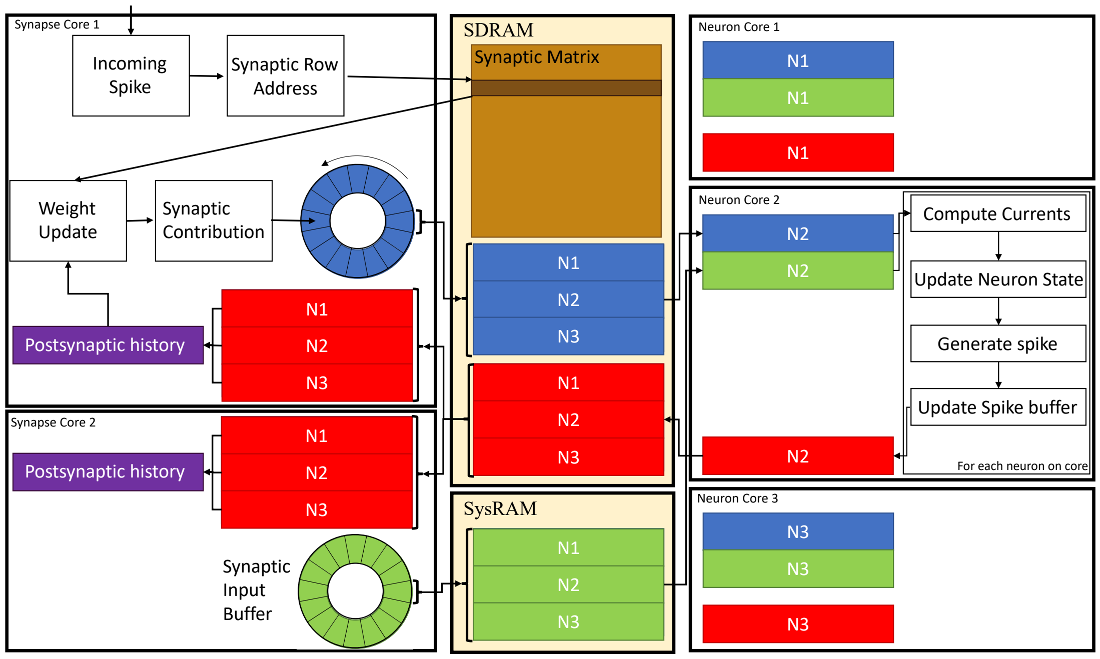


Figure 5.4: Memory usage and data structures for the Multi-Target partitioning approach on SpiNNaker (plastic version), for the case of 2 *Synapse* cores targeting 3 *Neuron* cores each. Both fixed point shared memories are shown with region allocation.

region in Figure 5.4), therefore this area is allocated into SDRAM by the *Synapse* core of the ensemble having the lowest index, and the address is retrieved by all the other *Synapse* cores. The *Neuron* cores get the address in the same way as the synaptic contributions, and will use the same offset to get access to their specific sub-regions. This region is tagged through a unique ID generated by the Python toolchain.

During each timestep, as *Neuron* cores update the neural state, they set each corresponding field of the postsynaptic buffer either to 1 (if the neuron spiked) or 0 (if the neuron did not spike). After all the Neurons have been updated, the postsynaptic buffer is written to SDRAM by each *Neuron* core. Each *Synapse* core can then read the postsynaptic buffer of the connected *Neuron* cores in a single memory transfer. This step is scheduled to happen after a fixed amount of time (for a given configuration), as the *Neuron* cores require a fixed amount of time to update the neural state and write back the postsynaptic buffers. Once the buffer is read, each *Synapse* core processes the postsynaptic events for all the synapses and updates the postsynaptic history trace (purple in Figure 5.4). All the incoming spike packets received up to this point are buffered similarly to what is described in Chapter 4, but the spike processing pipeline is prevented from being started, in order to allow a correct update of the postsynaptic history trace first. The completion of the postsynaptic history trace update triggers the spike processing pipeline, which starts processing spike packets, updating weights and then filling the synaptic input buffers to be transmitted to the *Neuron* cores, following the same approach described in Section 5.3.3.

5.4 Results

This section presents benchmarking of the Multi-target partitioning on SpiNNaker. The performance of the approach is measured according to system memory (in Section 5.4.1), peak synaptic events throughput (in Section 5.4.2) and connection sparsity (in Section 5.4.3).

5.4.1 Memory Benchmarking

This first experiment measures the impact of writing and reading the synaptic contributions between *Synapse* and *Neuron* cores under the new ensembles scheme, showing timings for each possible combination of *Neuron* and *Synapse* cores on a chip. For each arrangement timings are presented for both the SysRAM + SDRAM case, and the SDRAM only case. The results are presented in form of heatmaps, where the horizontal axis shows the number of employed *Synapse* cores, while the vertical axis the *Neuron* cores. All the *Synapse* cores for each case are connected to all the *Neuron* cores of the same case. To avoid interference with external events, and therefore to isolate the transfer times, the values are sampled in the context of a neural simulation in absence of spike packets. Therefore the standard neural state is updated, but the spike processing pipeline and the spike generation phases are turned off. This prevents neural processing from increasing contention, while maintaining the characteristics required by SNN simulations. Each simulation is run for 100 timesteps and read and write times for each core for each timestep are recorded and extracted at the end of the simulation. The presented results are obtained from 10 runs of the same simulation, with different cores placements, in order to ensure consistency.

The reported values show worst, best and average case transfer times obtained by the test. Worst case values are fundamental to estimate the impact of memory access time on the approach, and to correctly allocate timings to allow the processors to initiate DMA transfers in time to maintain real-time performance, adapting them to the application requirements. Average and best cases provide a more detailed analysis on the general performance of the memory operations, showing where optimisations are possible.

5.4.1.1 Reading Times

The results are presented in the form of heatmaps, where the horizontal axis contains the number of *Synapse* cores and the vertical axis the *Neuron* cores. Each square provides the result for an ensemble composed of the number of *Synapse* cores indicated by the column, targeting the number of *Neuron* cores indicated by the row. The maximum number of cores that can be used on a chip for SNN simulations is 15, therefore the cases in purple are configurations not allowed. The case with a single *Synapse* core is omitted, as the transfer terminates before the recording callback returns, therefore it does not affect the processing times. Reading times are measured in μs.

The timings have been extracted in the context of a neural application simulating 64 LIF neurons per Neuron core. Each synaptic weight is stored on a 16 bit (2 B) integer, meaning the contributions of a *Synapse* core targeting a single *Neuron* core amount to 64×2 B $= 128$ B (each DMA read has this fixed length). An increase in the number of cores results in an increase in contention and number of memory transfers.

Worst case reading times are presented in Figure 5.5. The top left plot shows results employing both SysRAM and SDRAM, the top right plot contains results when using only the SDRAM memory, and the bottom plot presents the case by case ratio between SDRAM only and dual memory times. By increasing the number of *Synapse* cores (moving from left to right on the horizontal axis), the number of reads per timestep per *Neuron* core increases. Reads are scheduled by the *Neuron* cores at the beginning of the timestep and performed sequentially, since there is a single DMA engine. As expected, for both the plots, the case with a single *Neuron* core (first line), shows linearly increasing reading times. The use of two separate memories does not influence this aspect, as one read at a time is performed. However it is observed that times in the dual memories plot are slightly lower. This is due to half of the *Synapse* cores contributions being stored into SysRAM which has a lower access time than SDRAM, therefore providing faster access. By increasing the number of *Neuron* cores (from top to bottom on

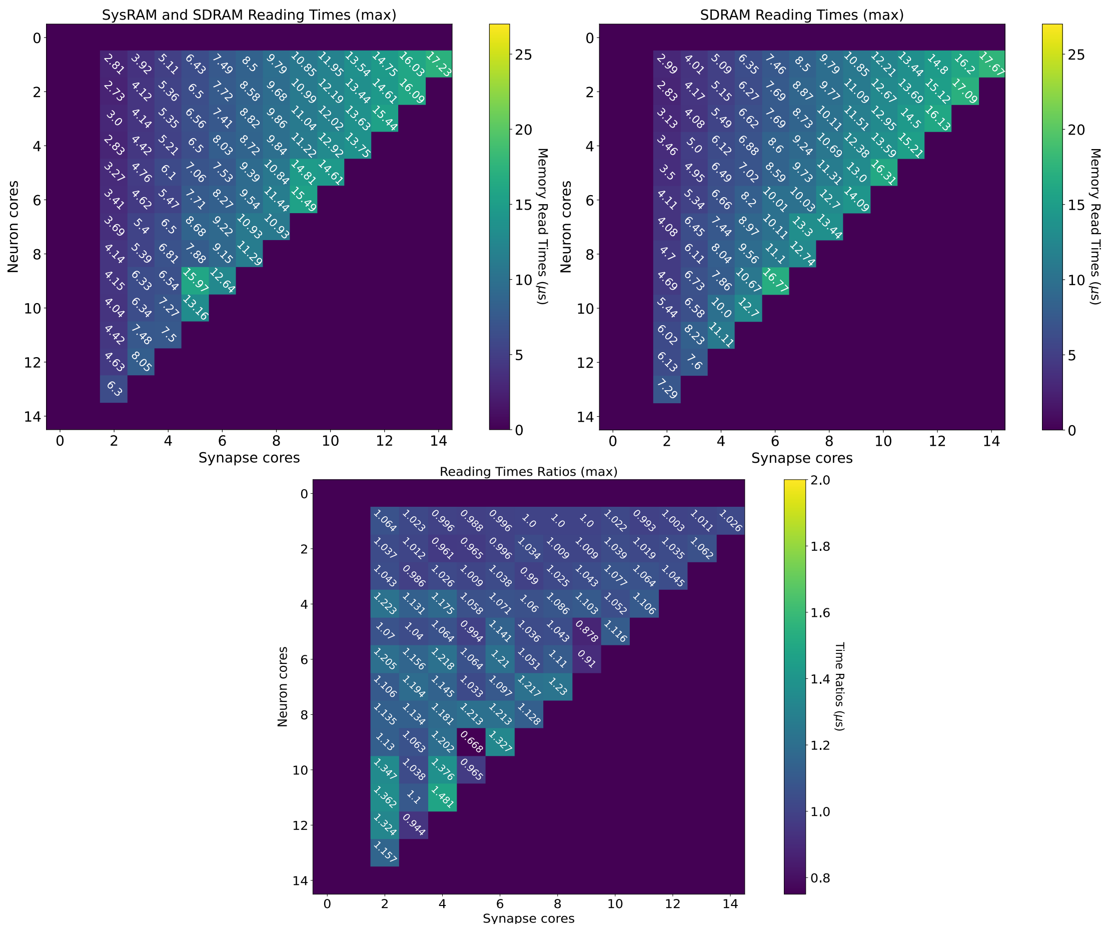


Figure 5.5: Comparison of worst case reading times between dual memory usage (top left) and SDRAM only (top right). Ratio between the two cases is shown as well (bottom). All the allowed configurations of Synapse and *Neuron* cores allowed by a chip are presented.

the vertical axis), the contention increases, as multiple *Neuron* cores try to access shared memory to retrieve their synaptic contributions simultaneously. This case demonstrates the benefits of having two different memories in use with separate access. The SysRAM + SDRAM case indeed performs generally better than the single memory case allowing a gain up to 4 μs. There are, however, some isolated allocations where the single memory case performs better. This is probably due to a bad allocation of the cores on the chip, which results in a slower access to memory. Core allocation affects the memory access time, as to grant fairness, access to memory is regulated by a binary tree with arbiters at every junction point, as detailed in Section 2.5.1. A situation where the allocation of *Synapse* cores is unbalanced can cause higher contention between memory requests, as requests coming from more populated branches of the tree need to be filtered by multiple arbitration steps. Cores are assigned by the SpiNNaker toolchain during the placement phase. The values reported here represent the measured worst case reading times, therefore they are likely to represent the worst allocation of cores.

The worst case for both the experiments happens with 14 *Synapse* cores, which represents the placement with the highest number of sequential reads, performed by a single *Neuron* core. Furthermore, by keeping the number of *Synapse* cores constant, and increasing the *Neuron* cores, the transfer time becomes higher, as the reading contention increases. This reduces the portion of the timestep available for neural processing. It is therefore of paramount importance to understand the requirement of the SNN to be simulated, in order to determine the appropriate number of *Synapse* cores to allocate per *Neuron* core.

The worst case analysis is important from a reading perspective to understand when the *Neuron* cores will start to fire, as the read phase must precede the neural state update and therefore *Neuron* cores must wait until this phase is completed before processing the neuron state update.

Figure 5.6 contains the best case reading times. Comparing the two cases (single vs dual memory), the dual memory allocation always provides better results than the SDRAM only case, especially when the contention increases (from top to bottom) and the number of transfers per core becomes higher (from left to right). It is important to notice the difference between best case and worst case, to understand how much the placement of cores impacts the measured times. Best case reading times show improvements of up to 5 μs, representing a considerable impact on 0.1 ms timesteps simulations (5% of the overall simulation

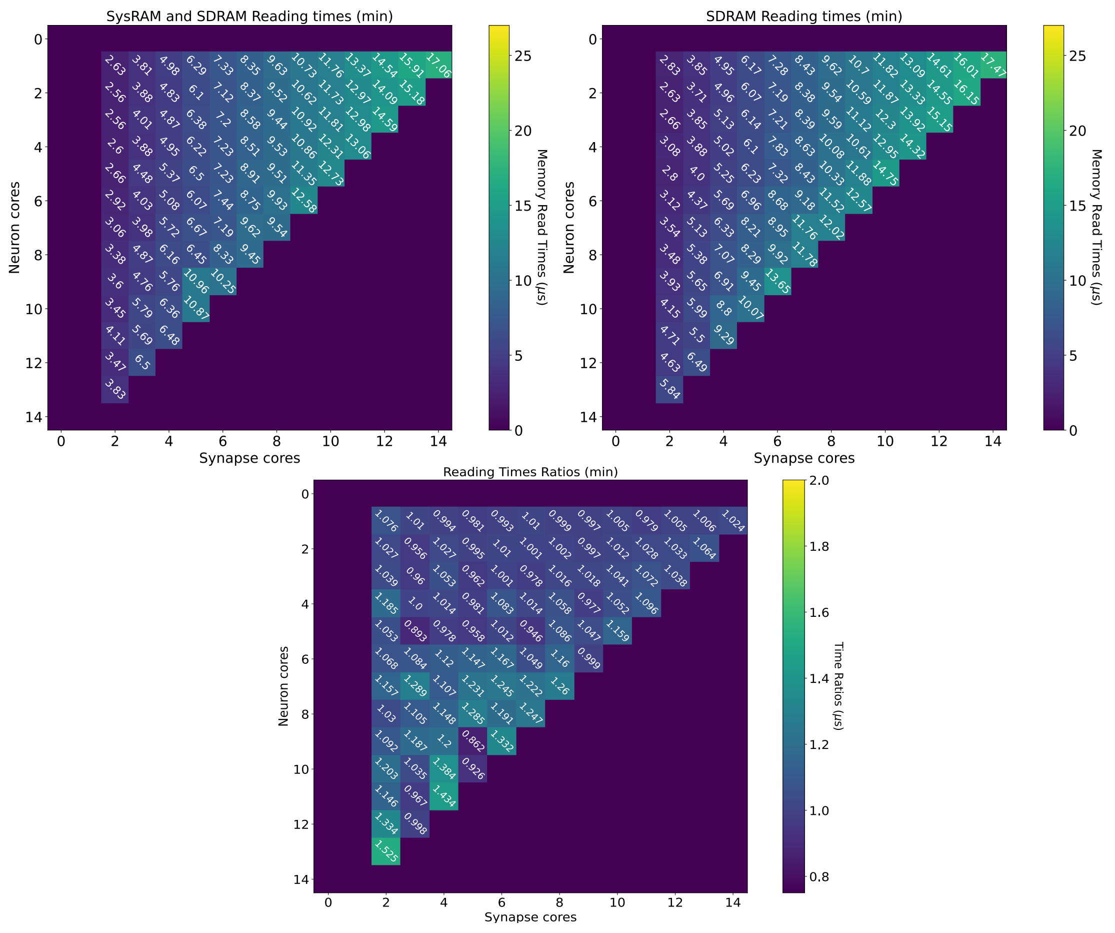


Figure 5.6: Comparison of best case reading times between dual memory usage (left) and SDRAM only (right). Ratio between the two cases is shown as well (bottom). All the allowed configurations of Synapse and *Neuron* cores allowed by a chip are presented.

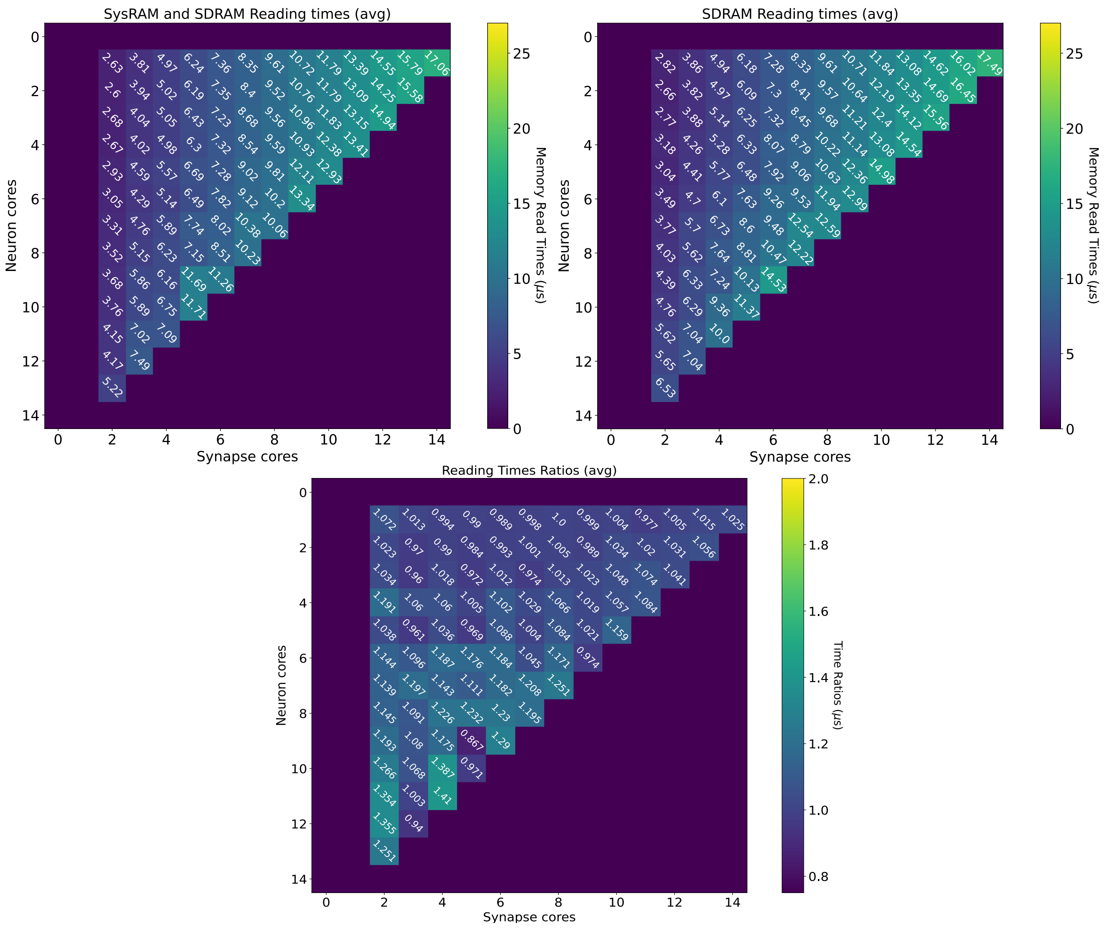


Figure 5.7: Comparison of average reading times between dual memory usage (left) and SDRAM only (right). Ratio between the two cases is shown as well (bottom). All the allowed configurations of Synapse and *Neuron* cores allowed by a chip are presented.

timestep).

Similar considerations can be seen for the average case, shown in Figure 5.7. The average values however are closer to the minimum, for most of the cases, which shows that the majority of the cores act similarly to the best case, and that the worst case tends to be isolated.

The core allocation phase therefore plays a key role in the memory timings. This is confirmed by the presented results, and suggests that a more targeted placement can achieve lower access times. This can be performed by informing the SpiNNaker toolchain of the type of cores (*Synapse* and *Neuron* cores), so that the most critical processors, according to the application requirements, are placed on less contended branches of the memories access trees. This is however a non-trivial operation, as, due to fault tolerance, it is hard to predict which core is available on each chip, and whether all the cores on the chip will be functioning.

5.4.1.2 Writing Times

The measurements for the writing times are shown in Figure 5.8: the top left plot shows the dual memory case, while the top right plot contains the SDRAM only case. The bottom plot presents the case by case ratio, obtained by dividing the single memory times by the dual memory values. Times are measured in μs, and each square represents a single write. Again increasing *Synapse* cores are displayed on the horizontal axis with increasing *Neuron* cores on the vertical axis. By increasing the number of *Synapse* cores (horizontal), the contention grows, as multiple cores attempt to write to shared memory simultaneously. By increasing the number of *Neuron* cores (vertical) however, the size of each write becomes larger. This is because each *Synapse* core performs one single write per timestep. Therefore, by increasing the number of implemented synapses (connected *Neuron* cores), the number of synaptic contributions to be written grows as well. The size of each write is expressed by Equation 5.7, where n is the number of neurons per *Neuron* core (64 in this case), w is the size of a contribution (2 B for standard SNNs) and T is the number of target *Neuron* cores for each *Synapse* core. Therefore, in Figure 5.8, T increases vertically from top to bottom.

$$C = n \times w \times T \tag{5.7}$$

Similarly to the read case, the reported times are the worst case measured writing times, and, for some cases, the access time is worse for the dual memory case. This can be due to several factors, as *Synapse* core contributions are partially located in SysRAM and partially in SDRAM. Although SysRAM provides a faster access, it has a slower transfer rate, therefore, for larger transfers, it can result in having similar or worse performance compared to SDRAM. This, combined with a bad cores placement can result in losing the advantages of using SysRAM, negating the faster memory access while contending the access to the memory controller.

From a writing perspective, the worst case scenario is useful to instruct *Synapse* cores on when to stop processing incoming spikes and start writing the

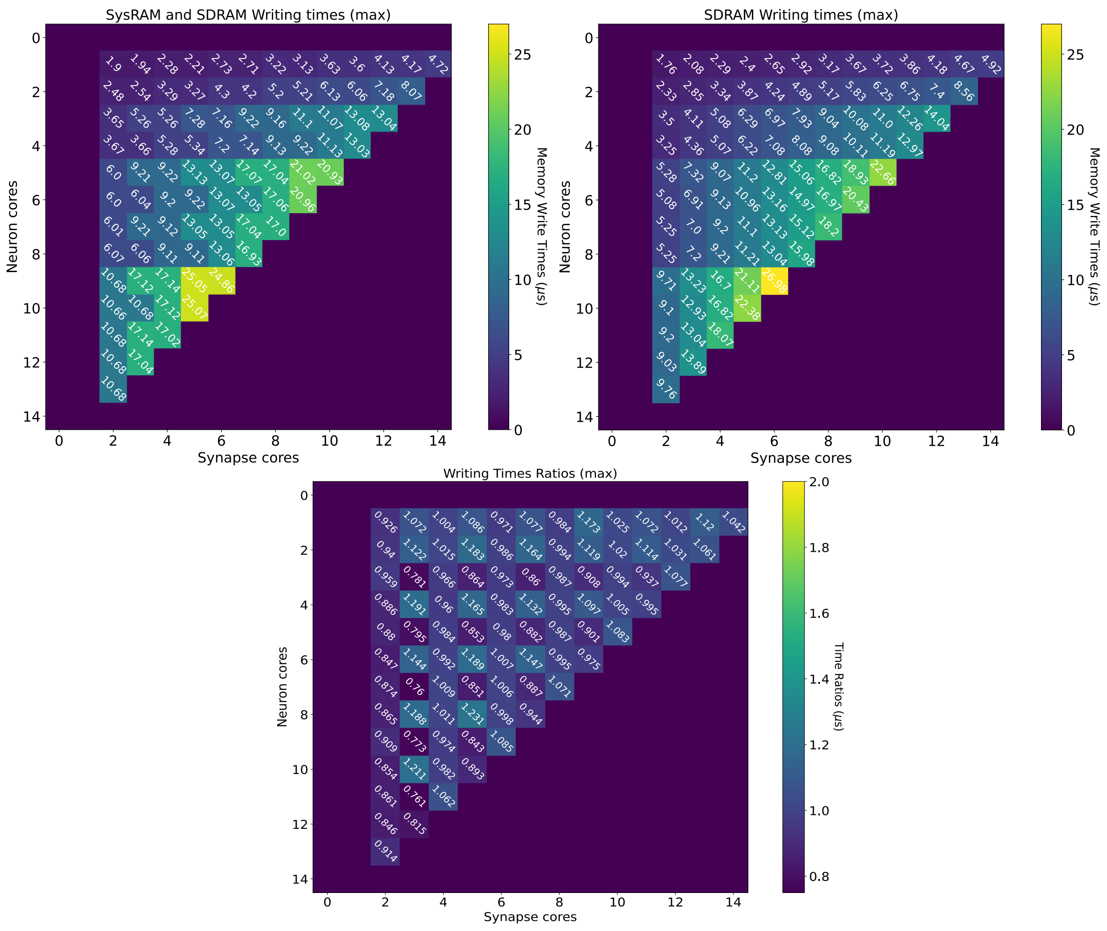


Figure 5.8: Comparison of worst case writing times between dual memory usage (left) and SDRAM only (right). Ratio between the two cases is shown as well (bottom). All the allowed configurations of Synapse and *Neuron* cores allowed by a chip are presented.

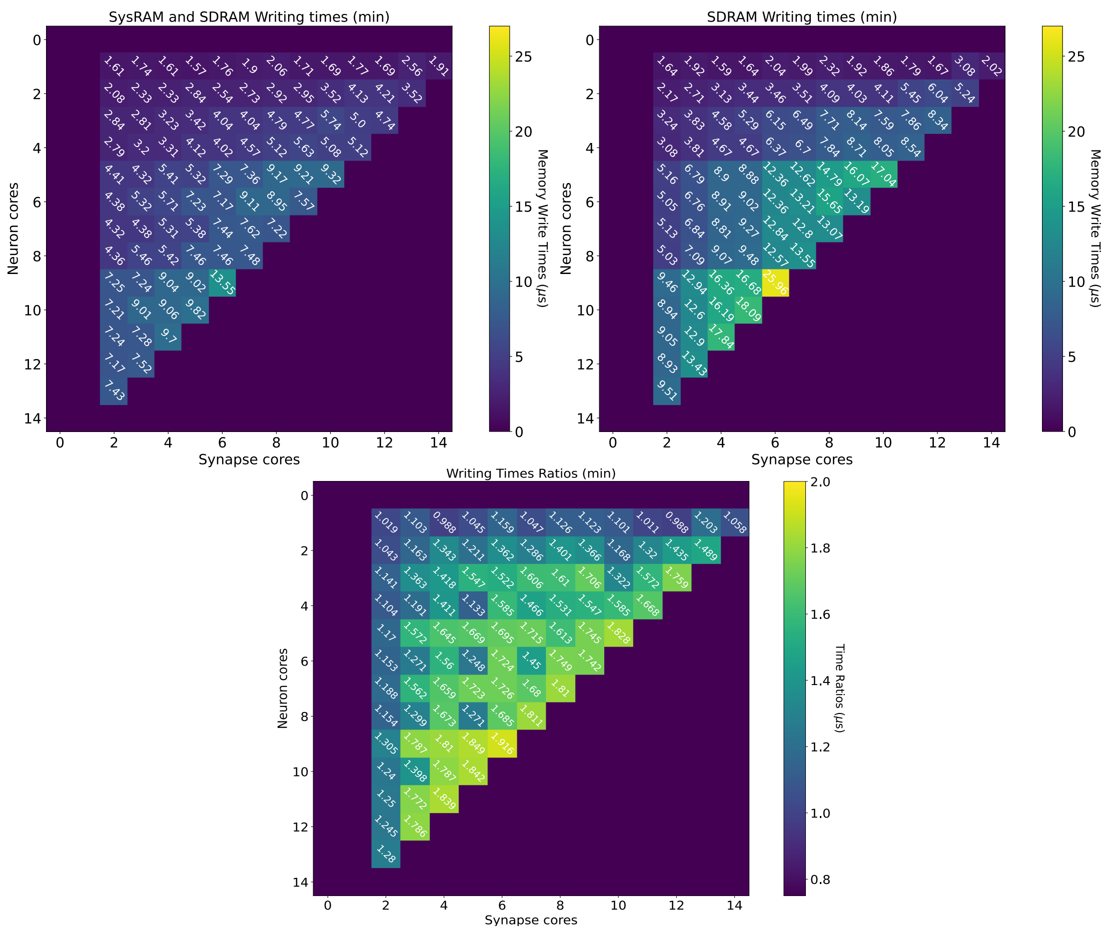


Figure 5.9: Comparison of best case writing times between dual memory usage (left) and SDRAM only (right). Ratio between the two cases is shown as well (bottom). All the allowed configurations of Synapse and *Neuron* cores allowed by a chip are presented.

synaptic contributions to shared memory (in order to meet real-time requirements). The highest recorded writing time is when using SDRAM only with 6 *Synapse* cores targeting 9 *Neuron* cores. This time amounts to 26.98μs. This does not represent an issue in simulations using a timestep resolution of 1 ms, but amounts to more than a quarter of the timestep for real-time simulations with 0.1 ms timesteps.

Figure 5.9 shows the best case writing times. For the writing phase it is more evident how the placement of cores impacts on the transfer time (relative to the reading phase above). Comparing the best with the worst case, significant gains are seen, particularly for the dual memory case. This difference becomes larger when the contending cores increase, and in some cases amounts to 15 μs,

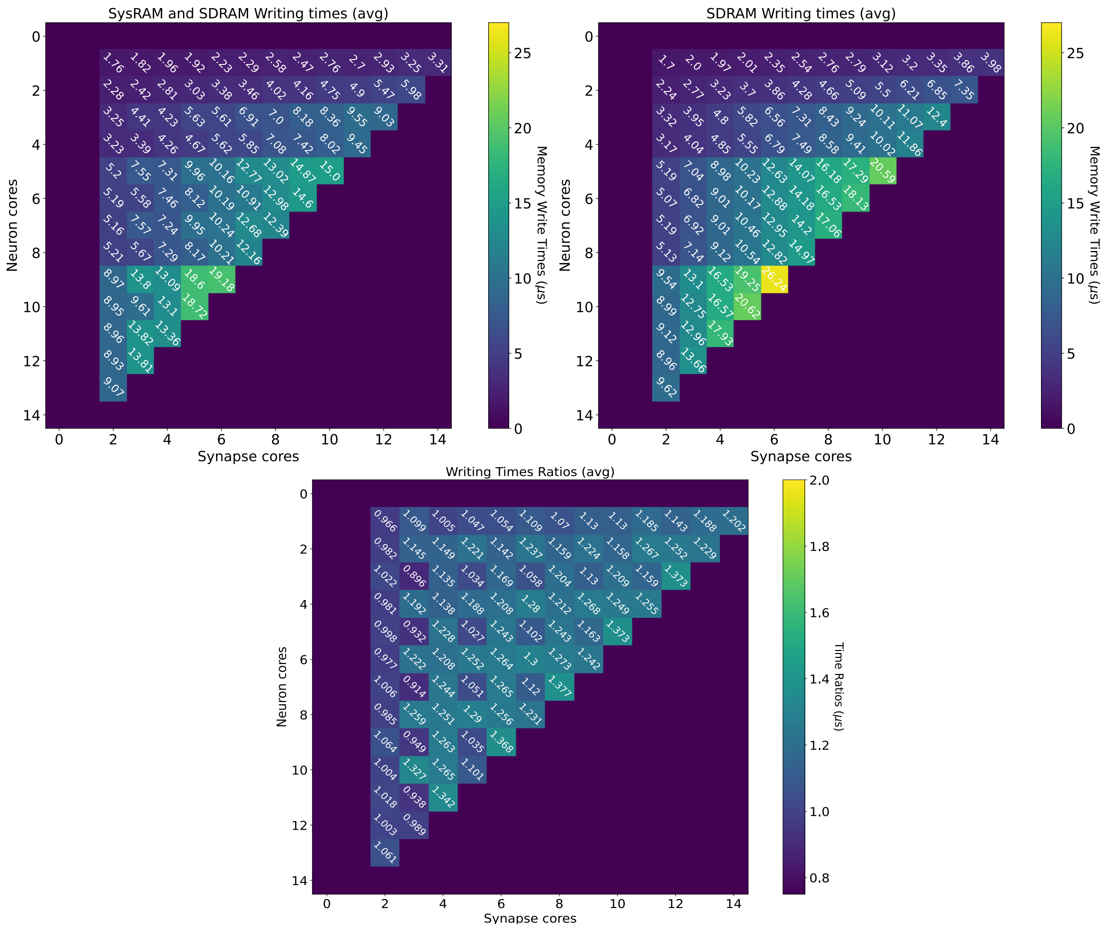


Figure 5.10: Comparison of average writing times between dual memory usage (left) and SDRAM only (right). Ratio between the two cases is shown as well (bottom). All the allowed configurations of Synapse and *Neuron* cores allowed by a chip are presented.

representing a significant impact on 0.1 ms timestep simulations (15% of the overall timestep). A similar comparison is performed between the dual memory and SDRAM only best cases, showing that with higher contention the writing time is halved with the dual memory approach, proving that it is possible to halve the contention by using separate memories with separate accesses.

Similar considerations arise regarding the average case results (shown in Figure 5.10), where access times are up to 6 μs smaller than worst case for dual memories. The SDRAM only case, however, shows numbers closer to worst case, and appears, in some cases, to perform better than the dual memory case with very low contention (2 *Synapse* cores) on the average case analysis. This can be explained by the fact that SysRAM has a slower transfer rate, therefore, when contention is low this has a higher impact on the total time. However, when contention increases, the dual-memory case usage drastically reduces the access time (as only half of the cores are contending the access), therefore reducing the total transfer time, relative to the SDRAM only case.

These results again show the impact of a sub-optimal placement of cores on a chip, and demonstrate that an improved placement strategy, combined with independent memory accesses can have a significant impact on the fraction of the timestep available for neural processing, especially for critical cases such as 0.1 ms timestep simulations.

The worst case writing and reading measurements therefore allow to tailor synaptic contribution writing and reading times to the required number of *Synapse* and *Neuron* cores per ensemble. This avoids overestimations which would further reduce the processing time shown in Equation 5.2. This analysis shows the importance of balancing the number of *Synapse* and *Neuron* cores according to the application requirements, in order to incur minimal memory access penalties. Network sparsity and firing activity also play a key role in the choice of core allocations. Therefore, the next sections focus on these aspects, providing measurements of peak processing of synaptic events per timestep and performance with different sparsity levels.

5.4.2 Peak Throughput Performance

A useful metric to measure the applicability of the Multi-Target partitioning is the peak throughput performance. This consists of measuring the maximum number of synaptic events that can be processed per timestep and comparing these numbers with previous implementations. The number of processed synaptic events per timestep depends on the timestep resolution and on the network connectivity probability. Separate cases are therefore presented here to address different scenarios. This experiment therefore compares the peak throughput performance for the Multi-target partitioning to previous works. To perform a fair comparison, the same SNN is profiled using the different approaches: Multi-target and Heterogeneous models. The same number of cores is allocated for both configurations where possible, but with different internal connections between *Synapse* cores and target *Neuron* cores. A third configuration is also presented, referred to as *single target expanded.* This consists of a standard Heterogeneous partitioning which maintains the same number of *Neuron* cores as the previous two

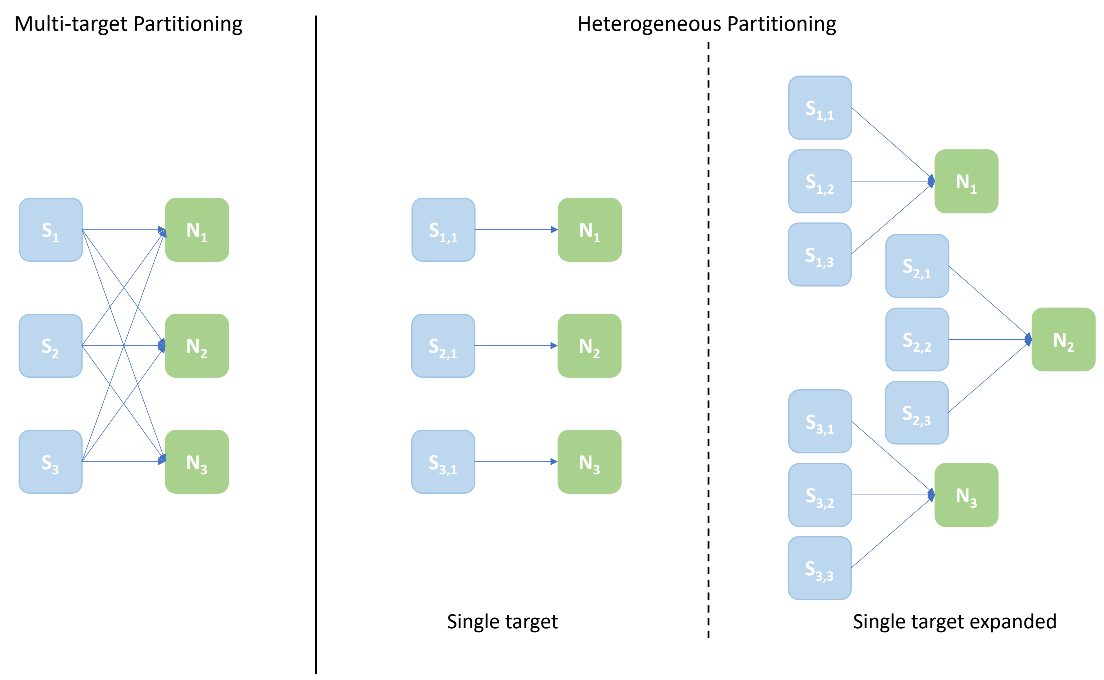


Figure 5.11: Arrangement of *Synapse* and *Neuron* cores under the explored configurations: Multi-target partitioning (left); Heterogeneous partitioning (right). The example shown demonstrates the [3, 3] test case, with 3 *Synapse* cores and 3 *Neuron* cores. For the Multi-target partitioning configuration, each *Synapse* core targets all *Neuron* cores. Comparison to the Heterogeneous approach is provided by: the Single-target partitioning, where the same overall number of cores are used, but connected one *Synapse* core to each *Neuron* core; and the Single-target expanded partitioning, where the same number of *Neuron* cores is maintained, but each with the same number of *Synapse* cores as implemented in the Multi-target approach.

cases, but allocates the same input *Synapse* cores capacity per *Neuron* core as the Multi-target approach. This last configuration provides a useful comparison, as the number of cores required for the single target Heterogeneous partitioning is adjusted to match the input capability of the Multi-target partitioning. The aim of including these cases is, therefore, twofold: first to compare the Multi-target partitioning to its Heterogeneous counterpart employing the same hardware resources, evaluating the performance difference; second, to show that, to achieve the input processing capability of the Multi-target approach, while using the Heterogeneous partitioning, it is necessary to employ a larger number of hardware resources. This is represented by the *single target expanded* case.

A schematic of core allocations for the three approaches is shown in Figure

5.11. The experiments run to evaluate this metric are structured in test cases defined by 2 numbers in the form $[S_c, N_c]$, where S_c is the number of *Synapse* cores and N_c the number of *Neuron* cores – the case shown in Figure 5.11 is $[3, 3]$. The Multi-target partitioning is shown on the left, where all the *Synapse* cores are connected to all the *Neuron* cores. The Heterogeneous partitioning is shown on the right, including the two different mappings explored: *single target* and *single target expanded.* The *single target* Heterogeneous partitioning presents 3 *Neuron* cores receiving input from a single *Synapse* core each, showing an input capacity reduced by a third compared to the Multi-target case. The *single target expanded* in the experiment is therefore comparable with the $[3, 3]$ cases for the two other configurations, however the number of cores allocated is $[9, 3]$. This single-target expanded configuration matches the input capacity per *Neuron* core of the Multi-target partitioning, keeping the same number of neurons and *Neuron* cores (therefore in the presented example each *Neuron* core receives inputs from 3 *Synapse* cores similarly to the Multi-target case, but each *Synapse* core is single target). The intent here is to show that the Multi-target partitioning can reach similar performance compared to this extended configuration, requiring only a fraction of the allocated resources.

The SNN model used for this experiment consists of 2 populations of neurons, configurable with a range of sizes and connectivity. All the presynaptic neurons are LIF [GK02] spiking neurons, with current-based exponentially-decaying synapses. Neurons are initialised with the internal voltage above firing threshold to produce spikes in a controlled manner. This approach is adopted to send spikes, instead of using spike sources, as it better represents the interaction between cores when simulating biologically-representative SNNs. This is because spike sources on SpiNNaker generate and send all spike packets together, causing a high firing activity concentrated at the beginning of the timestep, and then they remain silent. Cores implementing Populations (*Neuron* cores in this case) on the other hand, generate spike packets every time a neuron is updated and the model equations require it to spike, therefore distributing spike packet generation over the timestep.

The size of the presynaptic Population changes according to the number of incoming partitions (number of *Synapse* cores per ensemble) of the postsynaptic Population. These numbers have been obtained experimentally, such that the postsynaptic Population receives more spike packets than it can process. This

Presynaptic Population Size			
	1 ms Timestep(static)	0.1 ms Timestep	1 ms Timestep (plastic)
2,2	476	40	476
3,3	833	63	715
4,2	1000	77	741
4,4	1000	77	770
5,5	1428	100	770
6,3	1667	111	833
6,6	1667	111	909
7,7	2000	125	1000
8,2	2000	143	1428
8,4	2000	143	1428
10,2	2500	N/A	1667
12,2	2500	N/A	2000

Table 5.1: Presynaptic firing neurons for the peak processing benchmark according to the timestep resolution (columns) and allocated postsynaptic resources (rows).

allows saturation of the receivers in order to determine their limits. The number of generated spike packets however needs to be limited, due to limitations set by the SpiNNaker routing infrastructure [MLP$^+$15]. An excessive firing activity would cause higher congestion at the routing level, causing spike packets to be delivered late. This would result in lower processed synaptic events, compared to the real peak throughput, due to late arrivals. The presynaptic Population sizes are summarised in Table 5.1. The values are obtained from a population of 10000 neurons. Multiple simulations have been performed gradually decreasing the presynaptic sizes, while monitoring the state of the SpiNNaker communication network. These values represent the biggest sizes for which all the spike packets were delivered to their destinations without adding network delays or creating network congestion. At the same time, they are high enough to generate more synaptic events than the processing capabilities of the allocated *Synapse* cores for each case. This allowed testing of the limitations of the processing capabilities of the various approaches. For the 0.1 ms timestep resolution, the last two tests have been omitted. For these cases the timestep resolution represents a limiting factor on the number of *Synapse* cores per ensemble, as the *Neuron* cores cannot retrieve the synaptic contributions for more than 8 *Synapse* cores and update the neural state, while honoring the real-time performance. Therefore all configurations with more than 8 *Synapse* cores per ensemble cannot be simulated with 0.1 ms timesteps on SpiNNaker (as detailed in Section 3.2.3).

The postsynaptic Population employs the same type of neurons as the presynaptic Population, and has variable size between 64 to 896 neurons (corresponding to 1 to 14 *Neuron* cores respectively). Different connectivity patterns have been tested in order to demonstrate the robustness of the approach, including 0.1%, 1%, 5% and 10% connectivity. The same experiment was run both with 1 ms and 0.1 ms timesteps. The simulated network was the same in all cases, with the exception of the presynaptic Population size, which is scaled down of a factor $\approx 10\times$. The same experiment has been run both for plastic and static networks and the results are presented separately.

5.4.2.1 Static Results

Results for the static experiment are presented in Figures 5.12, 5.13, 5.14 and 5.15 for 1 ms timestep simulations. Each figure shows results for different connectivity probabilities. The *Multi-target* case is represented by the blue bars, while the *single target* with the same amount of cores by the green bars. The purple bar represents the *single target expanded* case. Finally, the yellow bars (labelled as homogeneous in the plots) provide a baseline, representing the processed synaptic events using the standard SpiNNaker toolchain with the same SNN and neurons per core.

Both the single target cases (green and purple) make use of the Heterogeneous model. The number of employed cores for each test case is indicated on the horizontal axes. The lower axis refers to the *Multi-target* (blue) and the *single target* (green). The upper axis shows values for the *single target expanded* (purple). The chosen configurations of cores allow direct comparison among the approaches. The left number in each tuple represents the *Synapse* cores of that test case, the right number the *Neuron* cores (as shown by the example presented in Figure 5.11). In the case of the Multi-target partitioning, all the *Synapse* cores of the ensemble target all the *Neuron* cores. For the single target cases the number of *Synapse* cores per *Neuron* core is obtained dividing the first number by the second. The blue and green bars are on the same axis because they employ the same number of cores, the difference between these two cases is in the connections between cores. This demonstrates that it is possible to improve the peak processing by rearranging the available units, thanks to the Multi-target

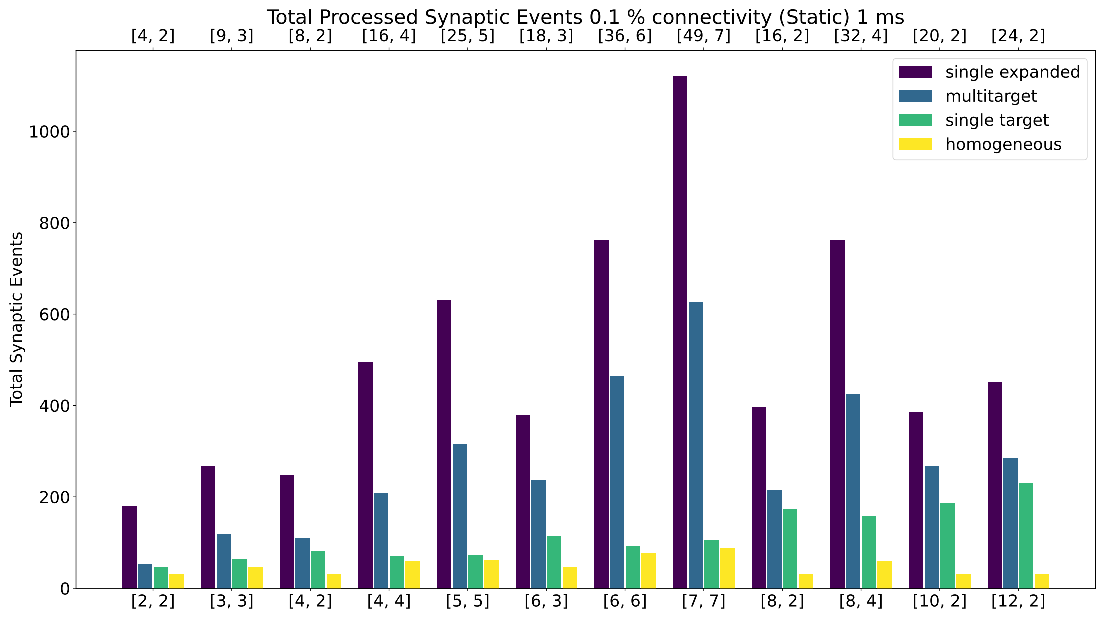


Figure 5.12: Processed synaptic events for the static case with 0.1% connectivity probability, using 1 ms timesteps.

approach. The purple cases use the same number of *Synapse* cores per ensemble (input partition) of the blue tests, however, in this case each *Synapse* core has one single target (therefore there is a single *Neuron* core per ensemble). This replicates the input capabilities of the Multi-target partitioning per ensemble, but it requires a considerably higher amounts of cores compared to the Multi-target case, resulting in the worst case of 56 total cores compared to 14 (8^{th} test case).

In all the cases the *Multi-target* approach (blue) performs better compared to the *single target* model (green). This is because the Multi-target partitioning performs a more efficient use of the available system resources compared to the Heterogeneous partitioning, allocating a higher input processing capacity to each *Neuron* core.

For the 1 ms timestep experiment the highest synaptic event throughput is given by the [7,7] configuration for all the connectivity probabilities under exam. The Multi-target partitioning processes up to $9\times$ the synaptic events processed from the heterogeneous partitioning, while employing the same hardware resources. This happens for the 1% connectivity case (Figure 5.13), however similar results can be observed for all the other cases. The reason why this happens is due to a full exploitation of the source-based partitioning offered by the approach. Each *Synapse* core in the *Multi-target* case receives inputs from one

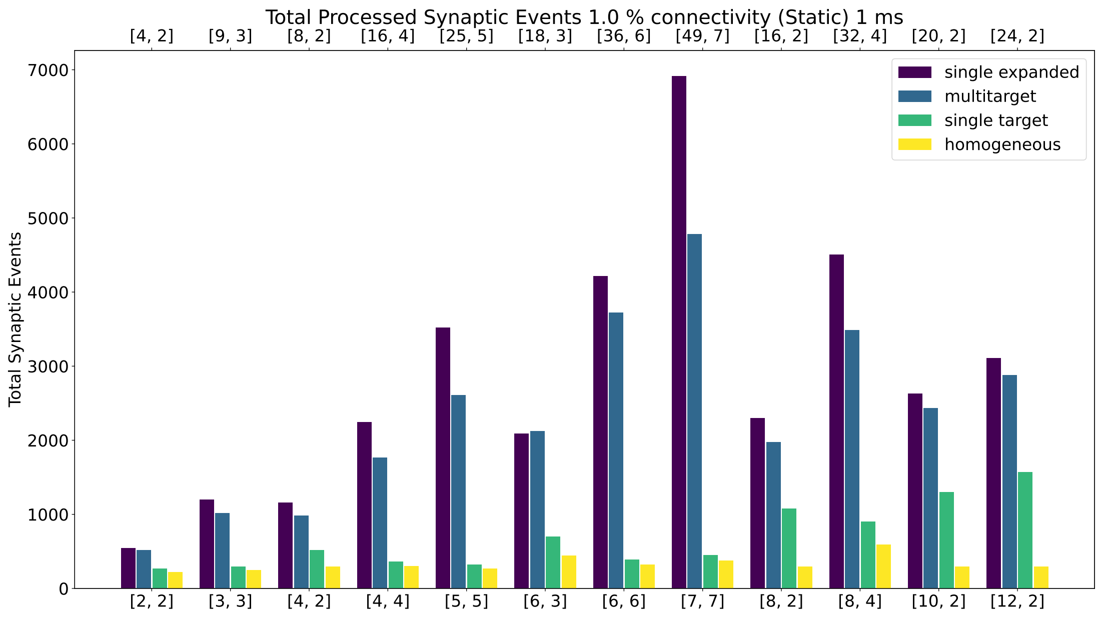


Figure 5.13: Processed synaptic events for the static case with 1% connectivity probability, using 1 ms timesteps.

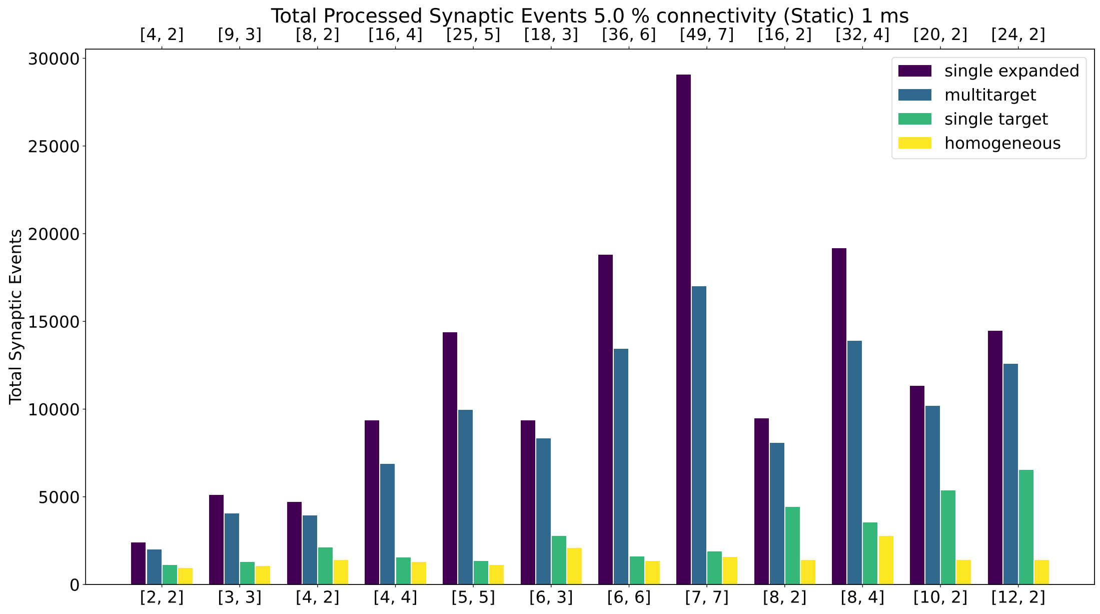


Figure 5.14: Processed synaptic events for the static case with 5% connectivity probability, using 1 ms timesteps.

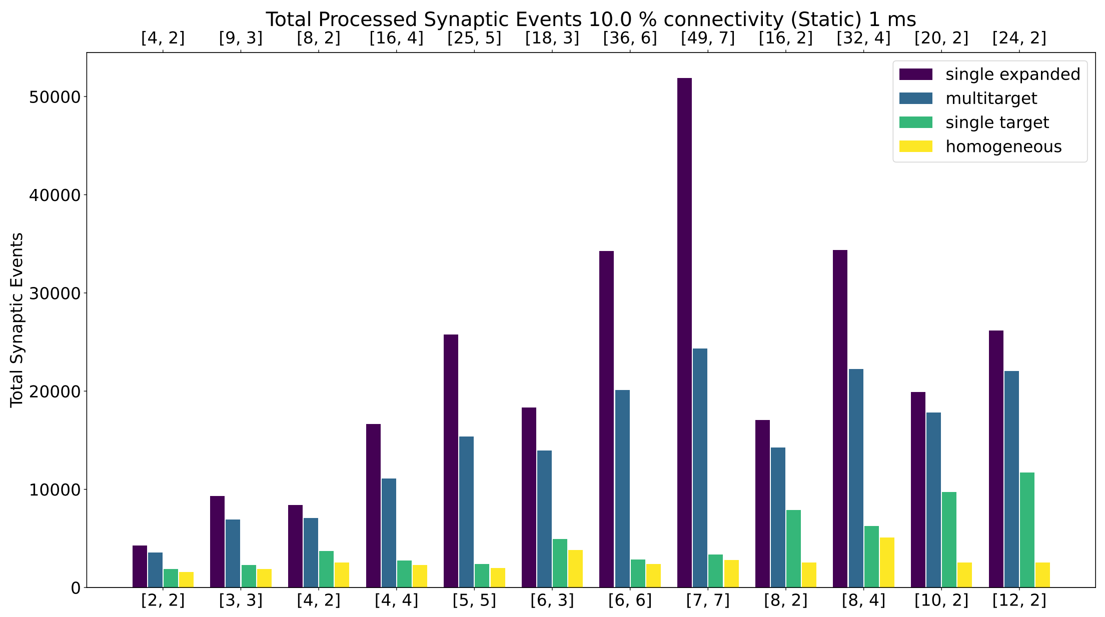


Figure 5.15: Processed synaptic events for the static case with 10% connectivity probability, using 1 ms timesteps.

seventh of the presynaptic neurons and targets all the 448 postsynaptic neurons. The *single target* partitioning on the other hand, has each *Synapse* core receiving inputs from all the presynaptic neurons, but targets only 64 neurons. Because the connectivity is very sparse, a reduced input traffic achieves better results.

The *Multi-target* approach performs well also compared to the *single target expanded* (purple), which represents a remarkable result, since the amount of resources in use is much lower, especially in the [7, 7] case. The single target expanded approach utilises the same number of *Synapse* cores per ensemble as the Multi-target partitioning, but has a single target per ensemble. Therefore, in the [7, 7] case ([49, 7] for the *single target expanded*) each *Synapse* core receives from one seventh of the presynaptic neurons and targets 64 postsynaptic neurons only. Furthermore, the improvements provided by the Multi-target approach compared to the Heterogeneous model do not decrease with increasing connectivity probability. In some cases indeed, they even become more evident (e.g. the [12,2] cases), demonstrating that the Multi-target partitioning also scales well with SNN connection density.

Regarding 0.1 ms timesteps simulations, the results are presented in Figures 5.16, 5.17, 5.18 and 5.19. A similar trend is noted for these results, with the *Multi-target* partitioning performing better than the *single target* case. With

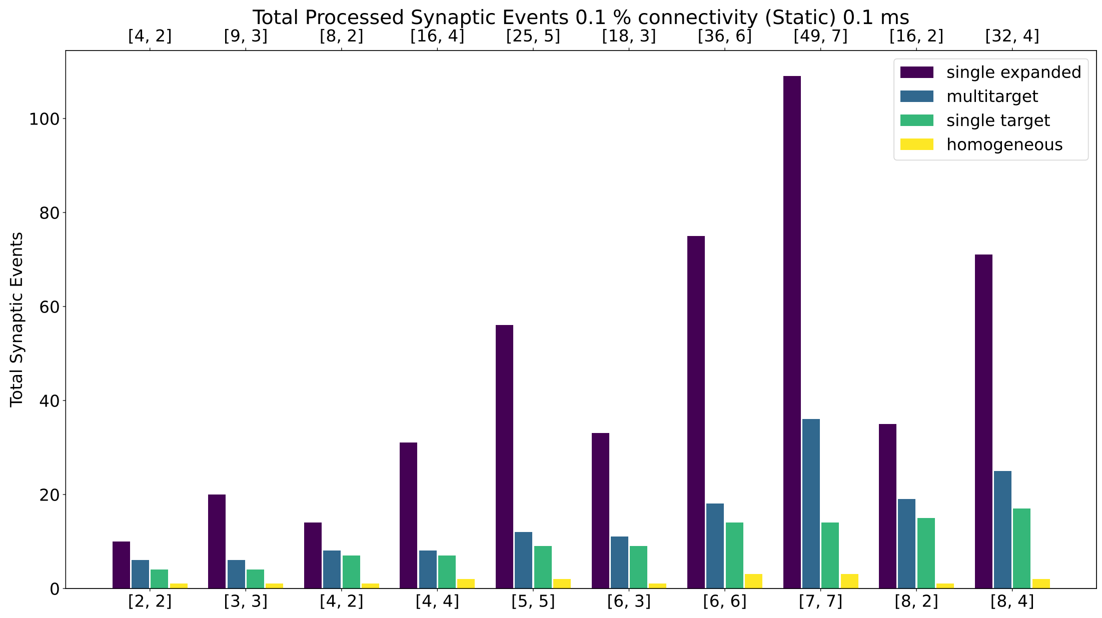


Figure 5.16: Processed synaptic events for the static case with 0.1% connectivity probability, using 0.1 ms timesteps.

higher numbers of *Synapse* cores targeting higher numbers of *Neuron* cores however, performance compared to the *single target expanded* case tends to be lower. This is due to the tight constraints set by the timestep resolution and the fact that memory read and write times for the synaptic contributions do not scale down with the timestep resolution, meaning they take a larger fraction of the timestep. There are therefore some edge cases where the gain is small, particularly with 0.1% connectivity (Figure 5.16). The reason behind this is due to the extremely low probability of connection, which results in zero target packets. The SNN is relatively small in this case, and the total number of synaptic events that are effectively received is low. However the [7, 7] case in Figure 5.16 left, demonstrates again that having longer synaptic rows (the Multi-target approach employs synaptic rows 7 times longer than the Heterogeneous model) enables processing of many more synaptic events per timestep ($\approx 3\times$ here). The reason behind large discrepancies between the extended Heterogeneous approach and the Multi-target with 0.1 ms timesteps, as in the case of the [7, 7] is due to the much higher computational power of the single target extended approach compared to the Multi-target. With 0.1 ms timesteps the reading and writing times have a much greater impact on the available spike processing time. Therefore, in this case where there are $7\times$ more *Synapse* cores, this translates into improved peak

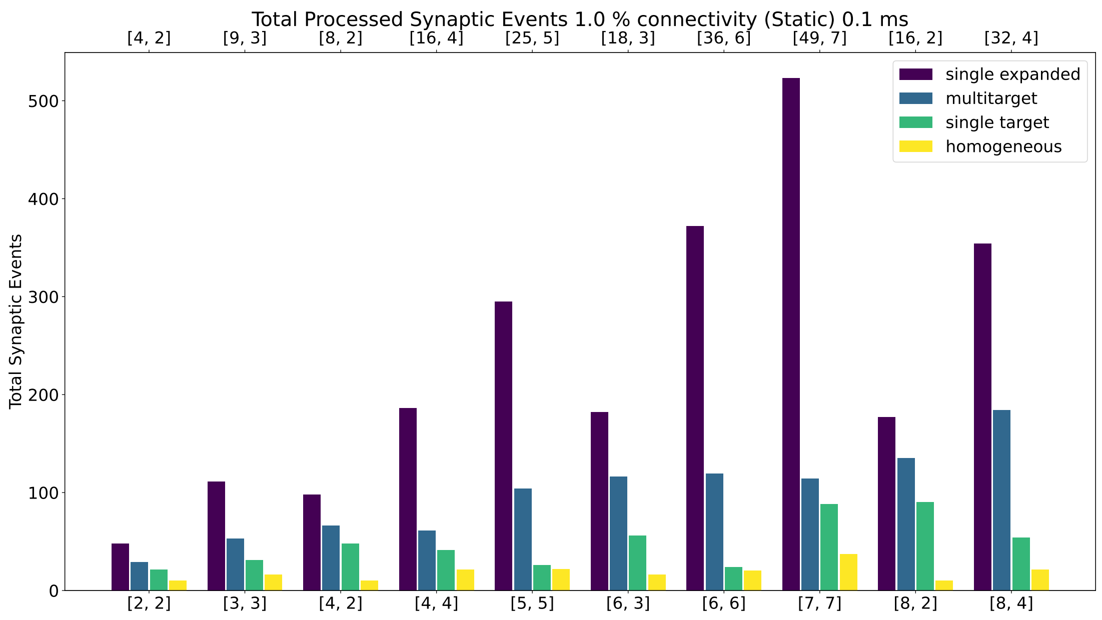


Figure 5.17: Processed synaptic events for the static case with 1% connectivity probability, using 0.1 ms timesteps.

performance.

In accordance to what is specified in Chapter 3, it is not possible to perform simulation with a number of *Synapse* cores per *Neuron* core higher than 8, because of timestep overrun. Therefore 0.1 ms experiments are limited to the first 10 cores configurations.

This experiment shows that, by efficiently using the Multi-target partitioning, it is possible to achieve comparable results to the *single target expanded* case, but with a fraction of the hardware resources (a quarter in the [7, 7] case). Furthermore, with the same amount of resources it is possible to achieve considerably higher synaptic event throughput

The general trend for the three approaches, together with the SpiNNaker toolchain baseline is compared in Figures 5.20, 5.21, 5.22, and 5.23 for 0.1%, 1%, 5% and 10% connectivity respectively, employing 1 ms timesteps. Figures 5.24, 5.25, 5.26, and 5.27 show the results for 0.1 ms timesteps. In these plots the horizontal axis shows the total number of allocated cores, and the vertical axis the processed synaptic events per timestep. The simulations are analogous to those shown in the bar charts. Each point in these scatter plots matches one of the bars (as indicated by the labels). This comparison is useful to evaluate how good a specific approach is compared to the others. Ideally the number of

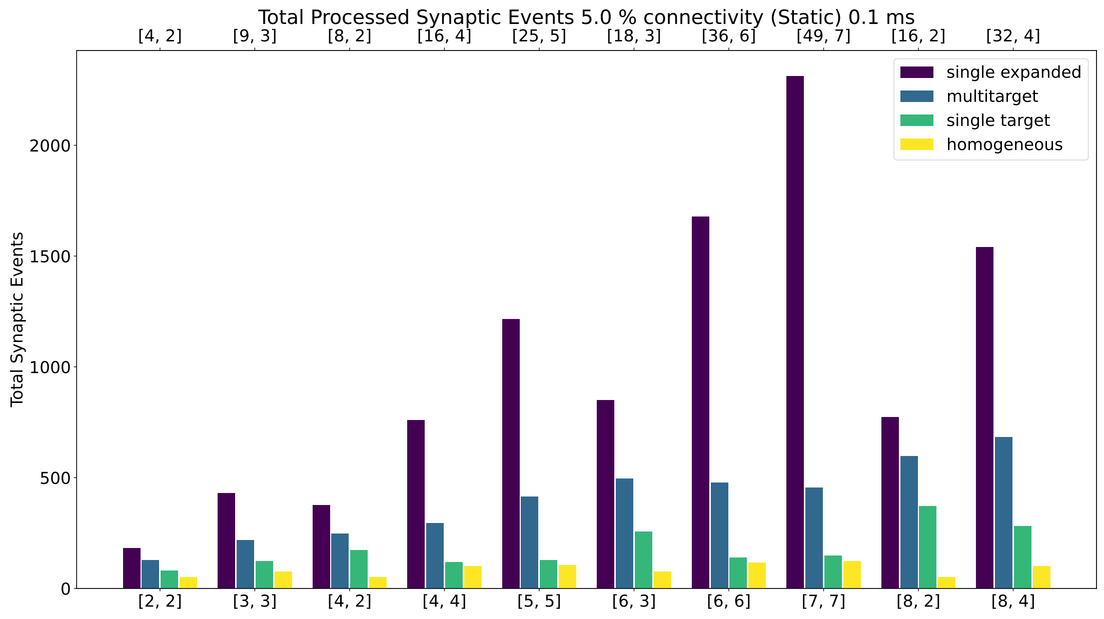


Figure 5.18: Processed synaptic events for the static case with 5% connectivity probability, using 0.1 ms timesteps.

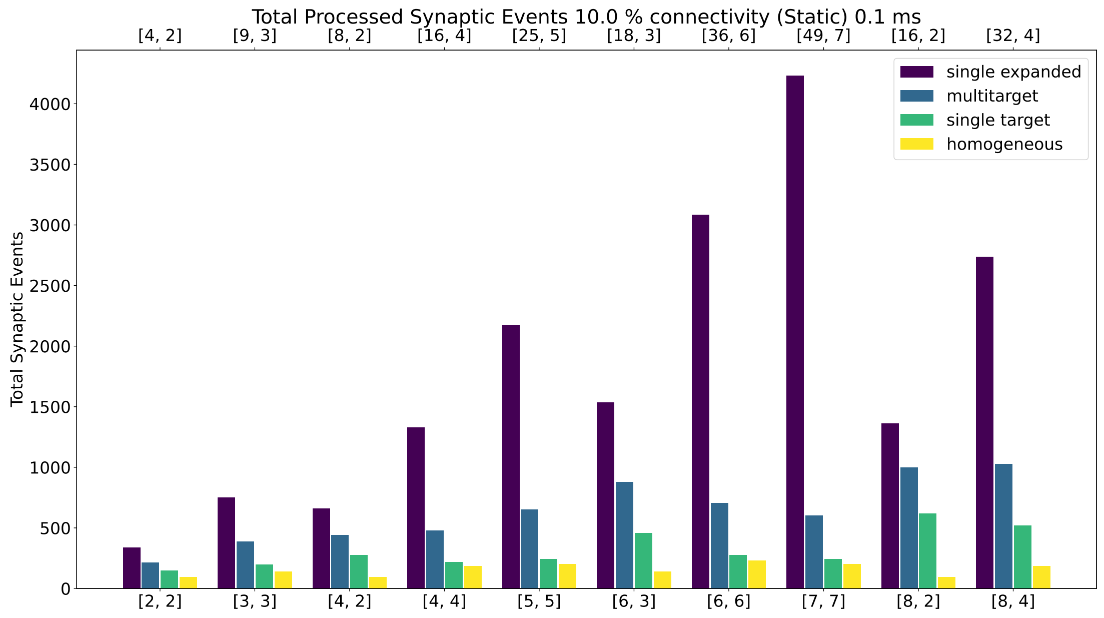


Figure 5.19: Processed synaptic events for the static case with 10% connectivity probability, using 0.1 ms timesteps.

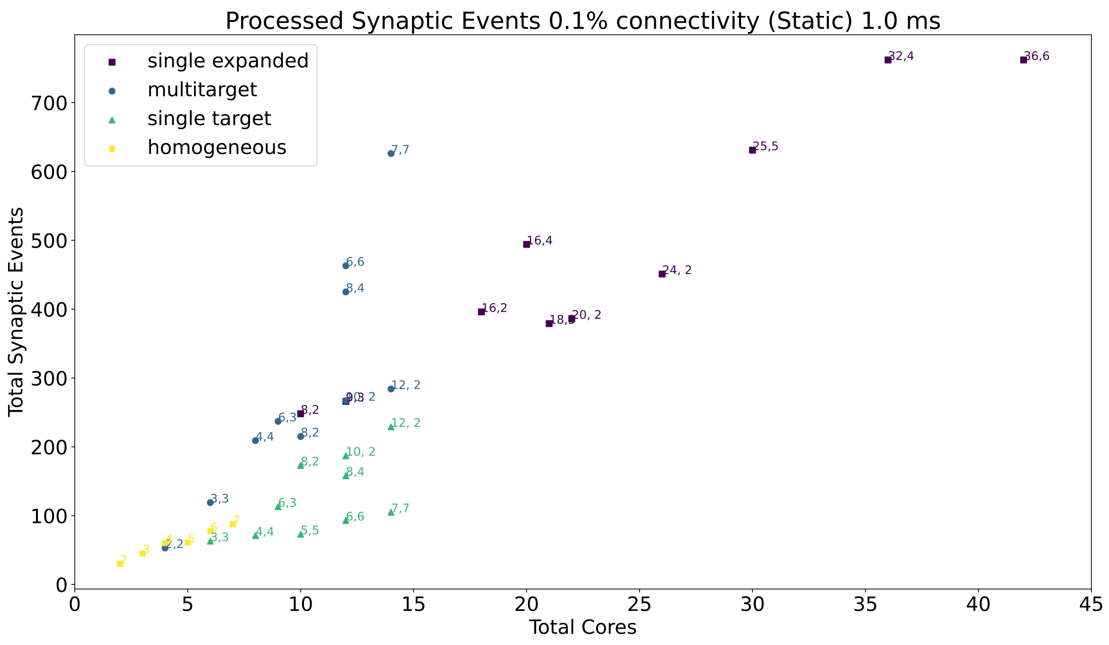


Figure 5.20: Resource allocation vs peak performance. Connectivity probability set to 0.1% and timestep resolution to 1 ms.

processed synaptic events should grow much faster than the required resources. Therefore an optimal solution would result in most of the points in the top left corner of the plot.

For all cases, the single extended case (purple points) processes the highest number of synaptic events, but also requires the largest allocation of hardware resources. The optimal solution remains the Multi-target partitioning (blue points), showing the steepest increases in processing performance with increasing resources. The peak processed synaptic events for the Multi-target partitioning remains close to the single target expanded for all 1 ms simulations, and is still optimal for the 0.1 ms. Discrepancies between the Heterogeneous extended and the Multi-target peak performance are due to timing constraints having a greater effect on neural processing.

Higher connectivity probabilities result in higher numbers of synaptic events processed per timestep, but flatter slopes in these plots. This can be explained by the increasing number of synapses that each spike targets, which results in a higher processing time per spike packet, combined with a higher transfer time required by the longer synaptic rows.

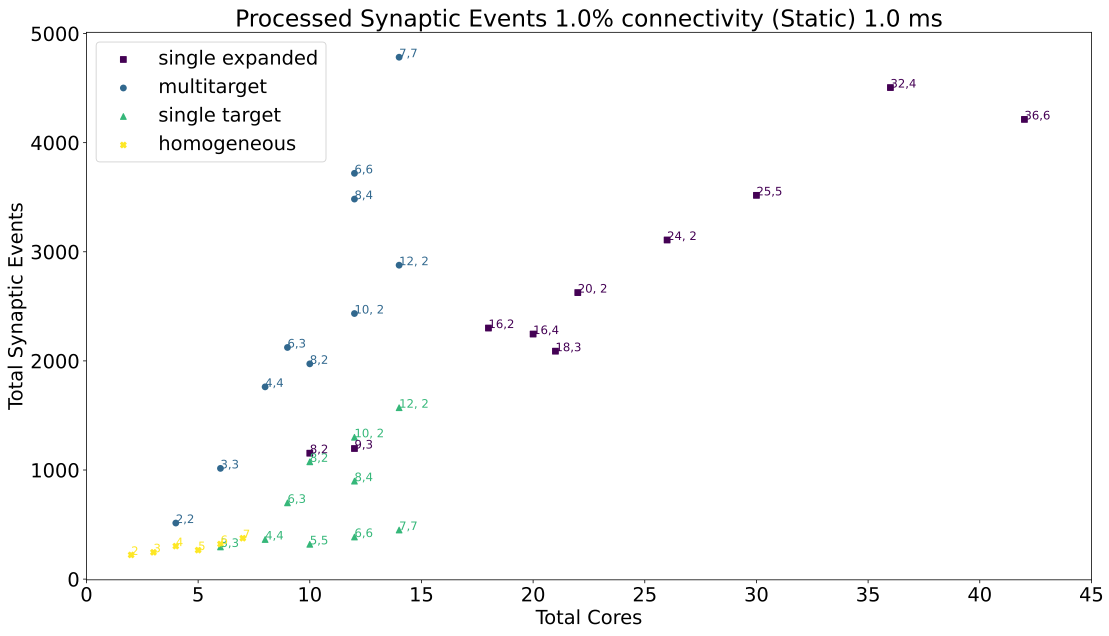


Figure 5.21: Resource allocation vs peak performance. Connectivity probability set to 1% and timestep resolution to 1 ms.

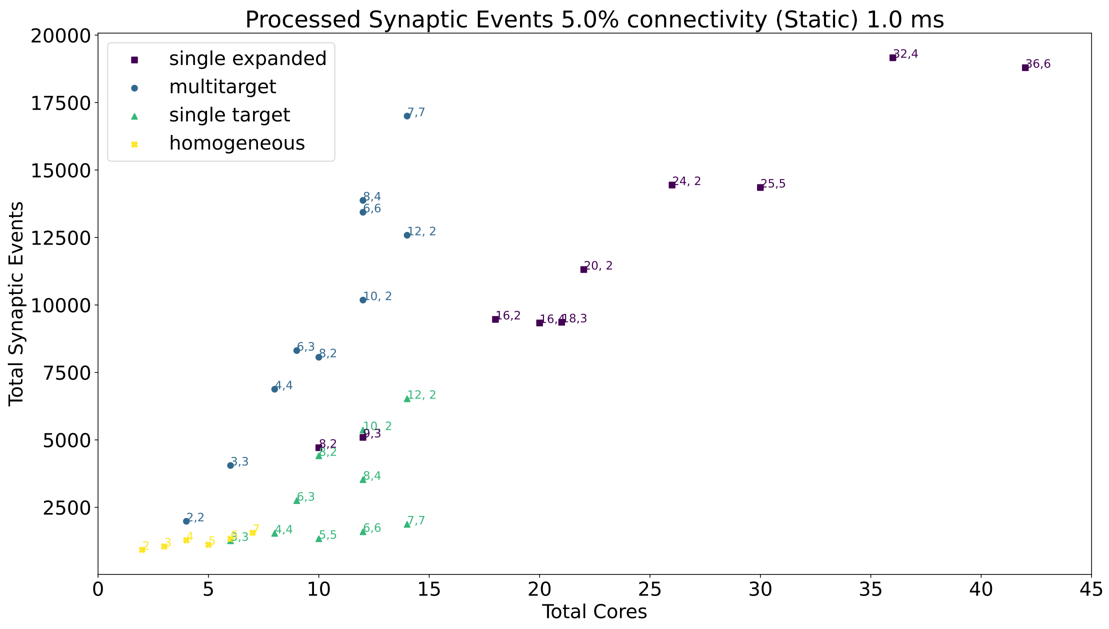


Figure 5.22: Resource allocation vs peak performance. Connectivity probability set to 5% and timestep resolution to 1 ms.

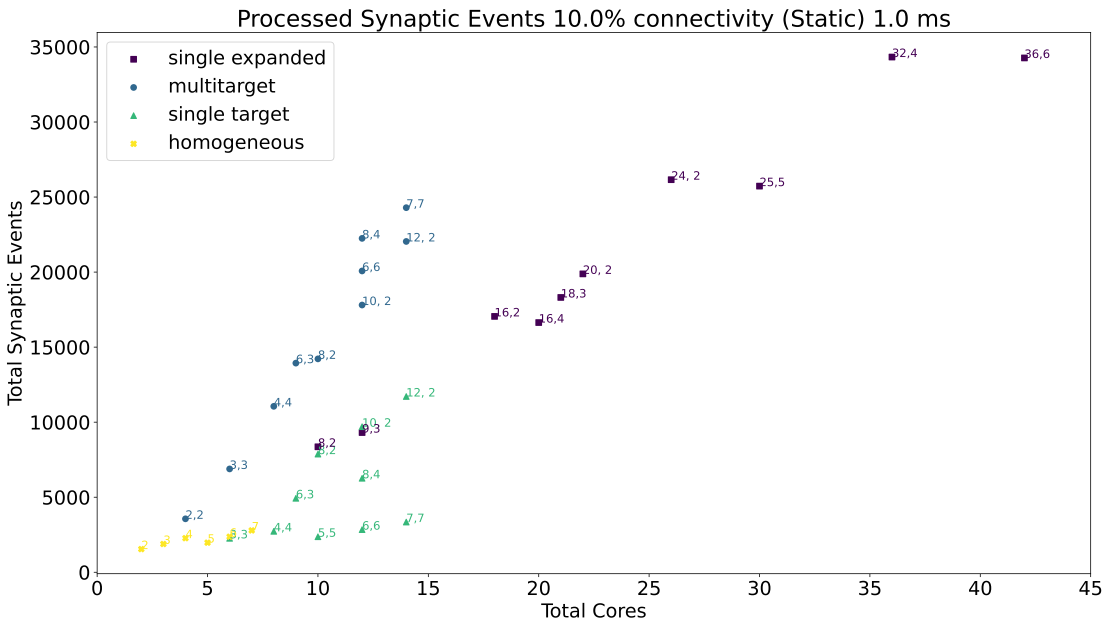


Figure 5.23: Resource allocation vs peak performance. Connectivity probability set to 10% and timestep resolution to 1 ms.

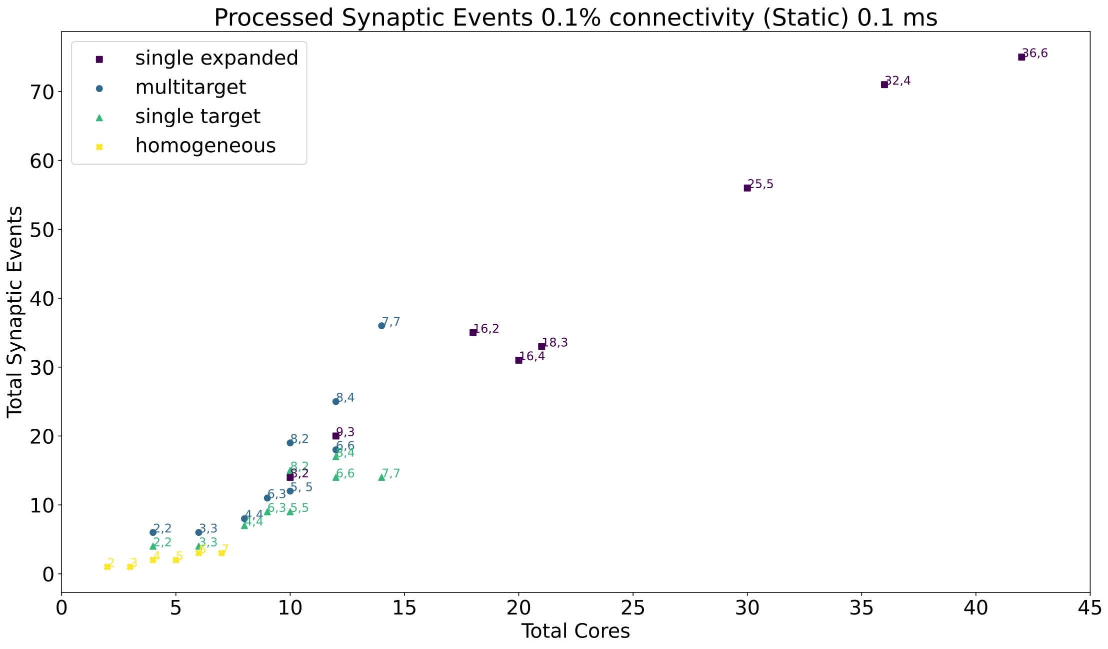


Figure 5.24: Resource allocation vs peak performance. Connectivity probability set to 0.1% and timestep resolution to 0.1 ms.

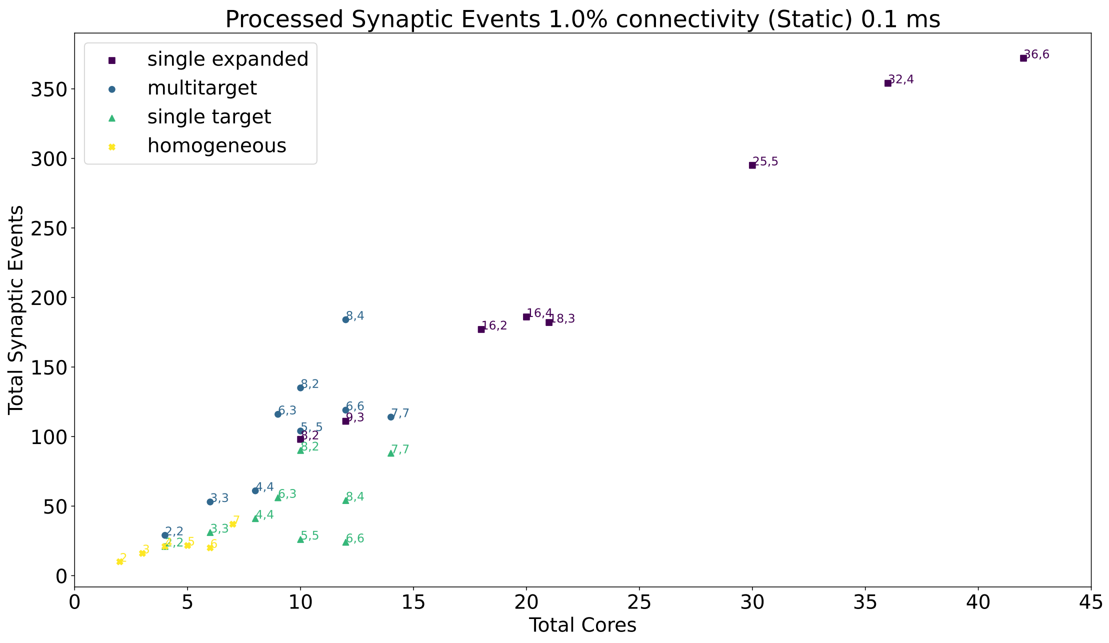


Figure 5.25: Resource allocation vs peak performance. Connectivity probability set to 1% and timestep resolution to 0.1 ms.

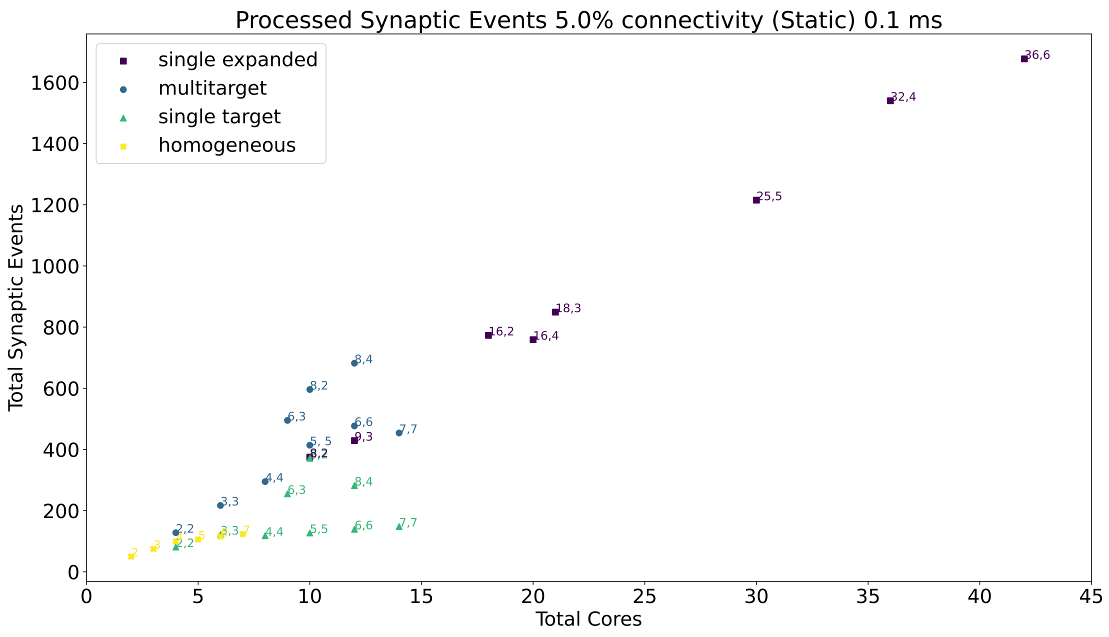


Figure 5.26: Resource allocation vs peak performance. Connectivity probability set to 5% and timestep resolution to 0.1 ms.

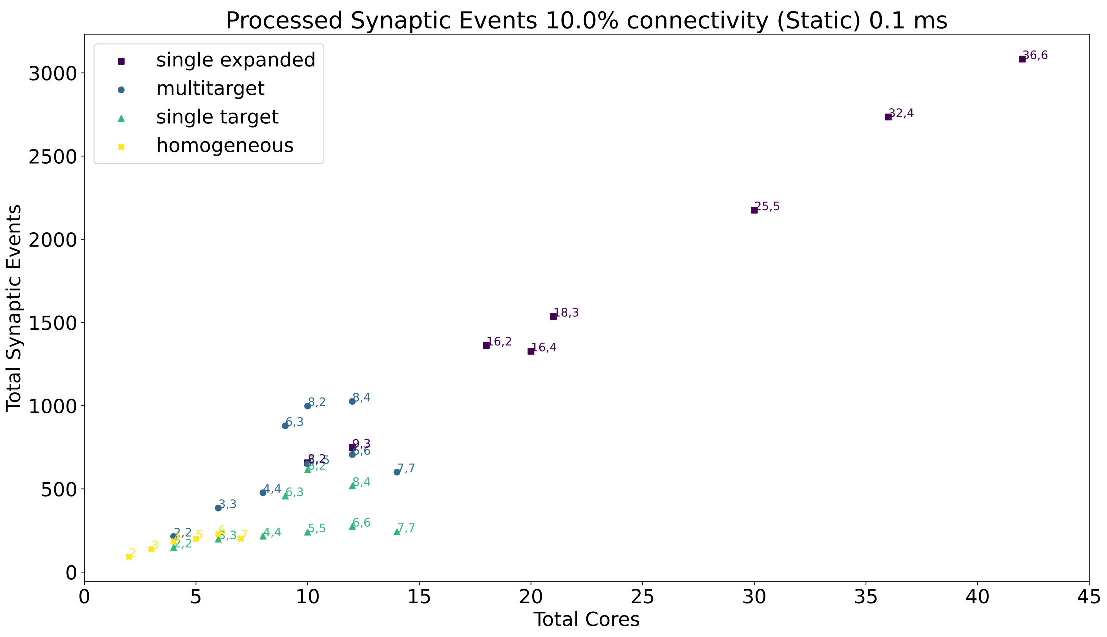


Figure 5.27: Resource allocation vs peak performance. Connectivity probability set to 10% and timestep resolution to 0.1 ms.

5.4.2.2 Plastic Results

In order to test the performance of the plastic implementation of the Multi-Target partitioning, a similar experiment has been designed. The same network used in the static experiment has been employed here, but the connection between the two populations has been defined using STDP, with Spike-Pair rule for timing dependence with Additive Weight dependence. The number of firing neurons has been reduced compared to the static case, as synaptic processing for plastic synapses requires the weight update and the write back phase of the plastic region of the synaptic row. The adopted presynaptic population sizes are shown in Table 5.1. The simulation has been run using 1 ms timestep resolution only. The results are shown in Figures 5.28, 5.29, 5.30 and 5.31. The experiment was replicated with different connectivity probabilities, mimicking the static case. Therefore the simulated connectivity patterns are 0.1%, 1%, 5% and 10%. The used color scheme is the same as the static case, where, for each bar chart the purple bar indicates the *single target expanded* case, the blue bar is for the *Multi-target* case and the green bar is for the *single target*.

For the plastic experiments the benefits of using the Multi-Target partitioning with very sparse networks become more evident, showing cases where the *Multi-Target* approach overcomes even the *single target expanded* in processed

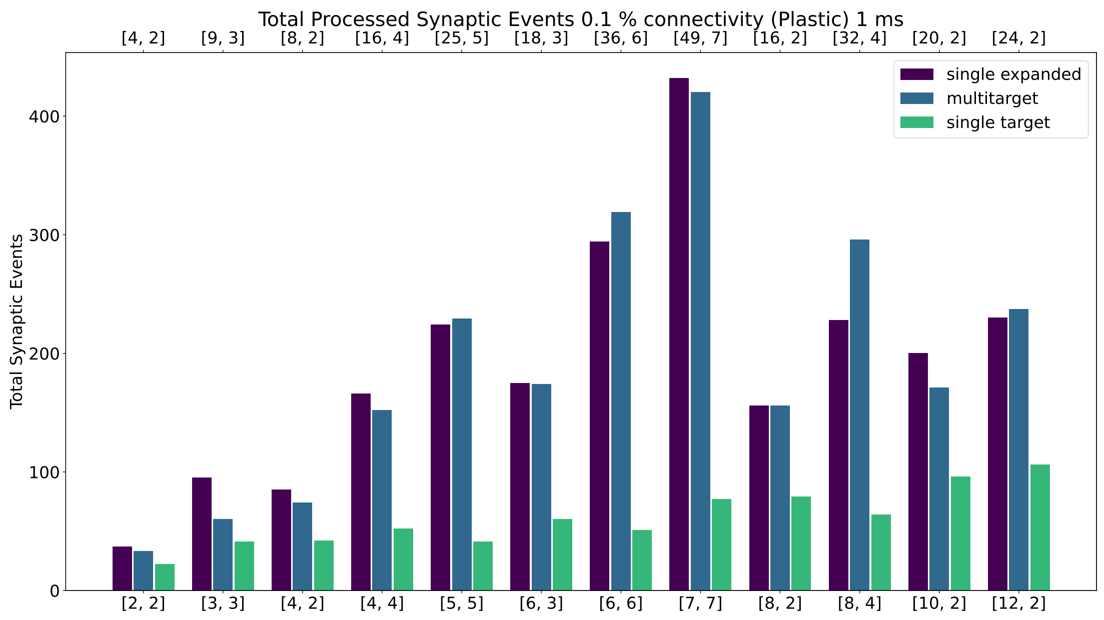


Figure 5.28: Processed synaptic events for the plastic case with 0.1% connectivity probability and 1 ms timestep resolution

synaptic events per timestep. This is due to the differences in processing plastic synapses compared to static synapses. Plasticity, requires the updated weights to be written back to shared memory, therefore doubling the accesses to SDRAM compared to the static case. This operation becomes extremely costly when the number of synapses per row are limited. Therefore, having longer synaptic rows, as in the case of the *Multi-target* approach, allows to amortise these two memory operations over a higher number of synapses, further increasing the number of synaptic events that can be processed per timestep. As expected, this effect decreases with increasing connection density, as rows contain more synapses and therefore the computational power of the extended Heterogeneous approach grants higher processing capability.

In order to evaluate the throughput over the required hardware a scatter representation of the plastic experiment is provided in Figures 5.32, 5.33, 5.34 and 5.35. The color scheme matches the bar charts and each point corresponds to a bar as indicated by the labels. Similarly to the static case, this plots allow to compare how well a specific solution performs compared to the others, with solutions having values close to the top left of the plot considered optimal.

The Multi-target partitioning once again proves to be the optimal solution,

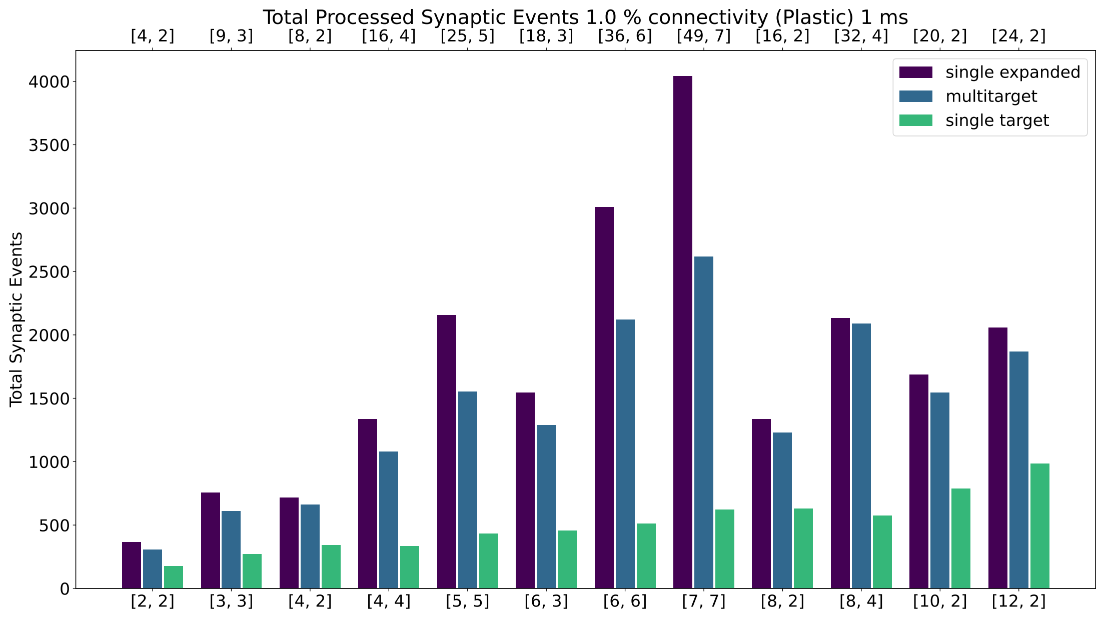


Figure 5.29: Processed synaptic events for the plastic case with 1% connectivity probability and 1 ms timestep resolution

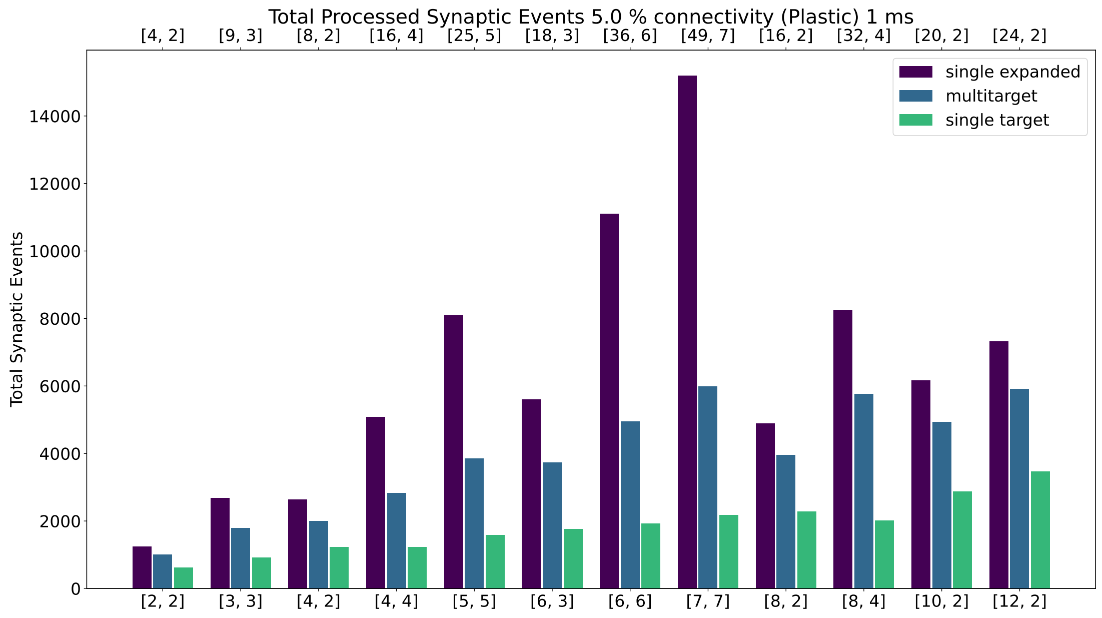


Figure 5.30: Processed synaptic events for the plastic case with 5% connectivity probability and 1 ms timestep resolution

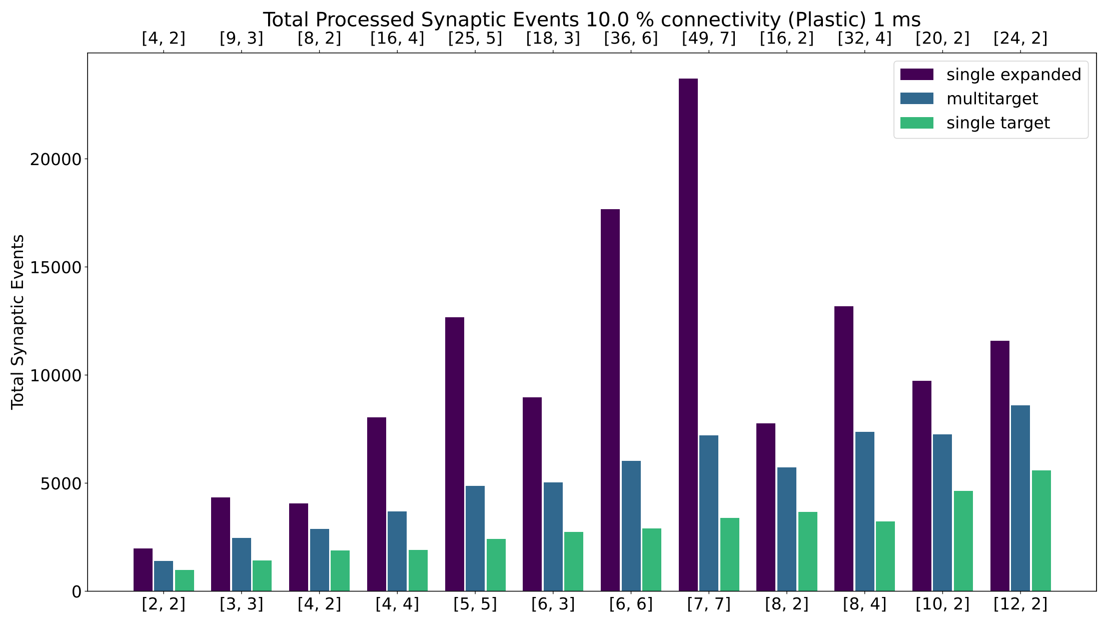


Figure 5.31: Processed synaptic events for the plastic case with 10% connectivity probability and 1 ms timestep resolution

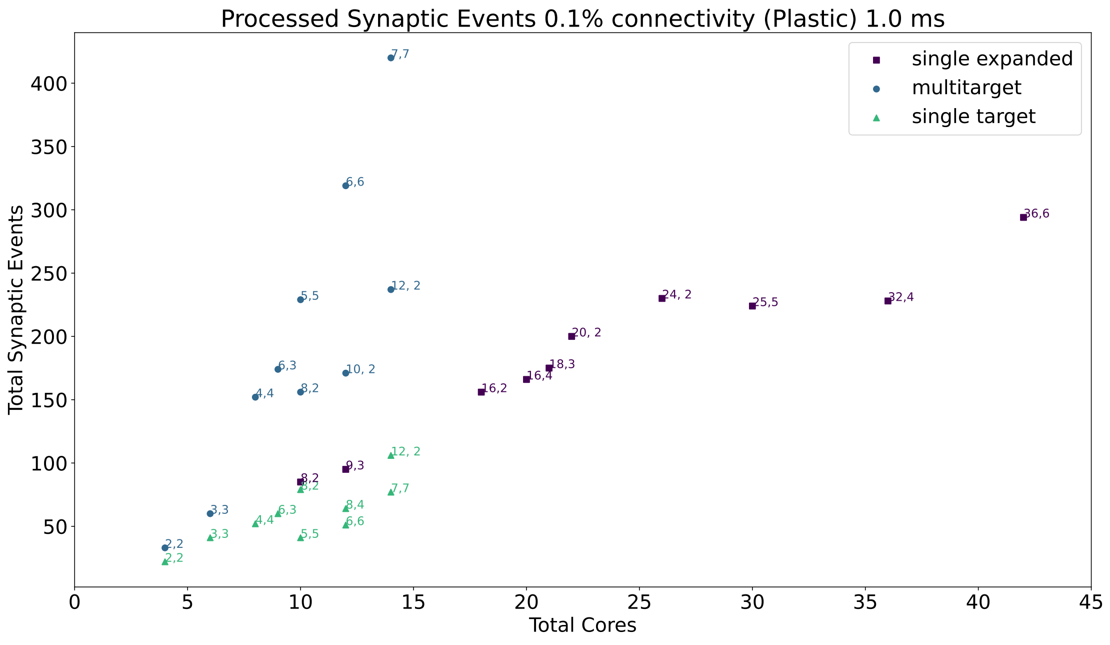


Figure 5.32: Resource allocation vs peak performance. Plastic configuration. Connectivity probability set to 0.1%

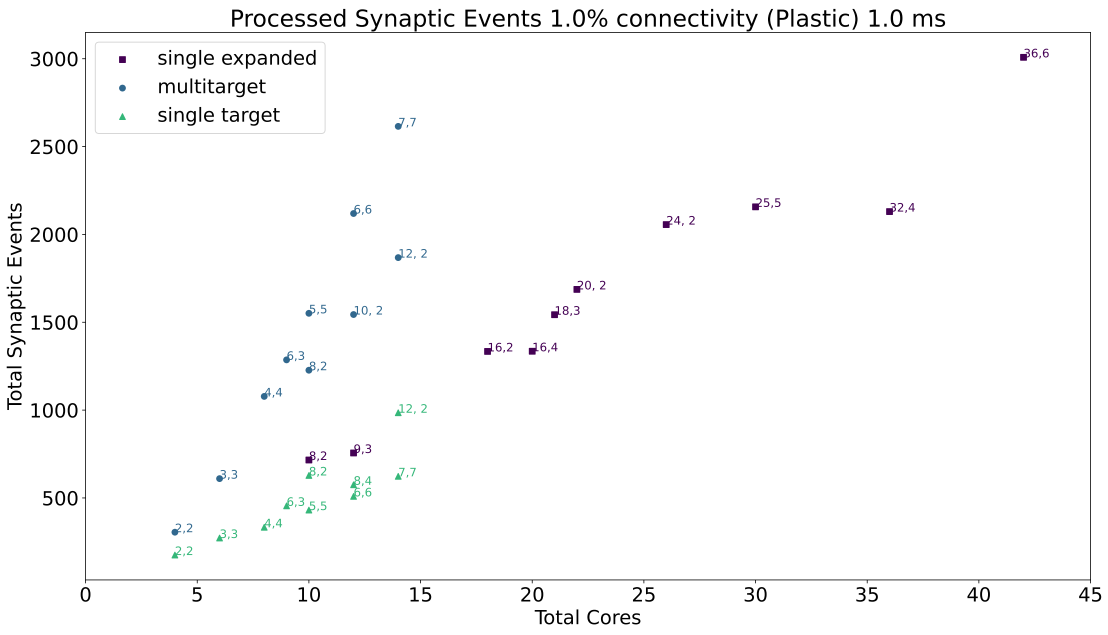


Figure 5.33: Resource allocation vs peak performance. Plastic configuration. Connectivity probability set to 1%

showing the steepest increase of processed synaptic events with hardware resources, dominating for the most sparse connectivity patterns over the other two approaches. With increasing network density the total number of processed synaptic events per timestep becomes larger for the *single target expanded* approach. However, this comes at a much higher cost in required hardware resources, as highlighted for the bar charts. Compared to the *single target* approach the *multi-target* always results in higher processed synaptic events, while employing the same hardware resources.

5.4.3 Sparsity Efficiency

This experiment evaluates the performance of the Multi-Target partitioning comparing different connection sparsity levels together. This experiment shows the variation of the processed synaptic events per timestep with increasing numbers of target *Neuron* cores. The number of *Synapse* cores is kept fixed and the target *Neuron* cores are gradually increased. In order to provide a good balance (and according to the peak performance shown in Section 5.4.2), the chosen number of *Synapse* cores is 7 and the target *Neuron* cores range from 1 to 7, guaranteeing to fit on a single chip. This allocation also allows equal comparison between simulations with 1 ms timestep resolution and 0.1 ms, having set the number of

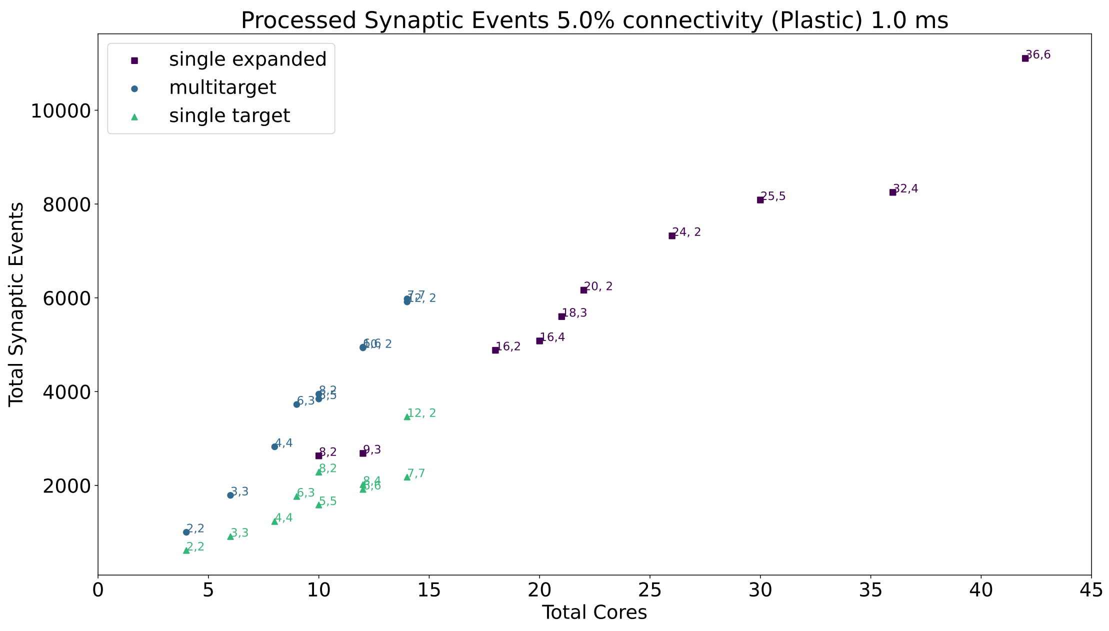


Figure 5.34: Resource allocation vs peak performance. Plastic configuration. Connectivity probability set to 5%

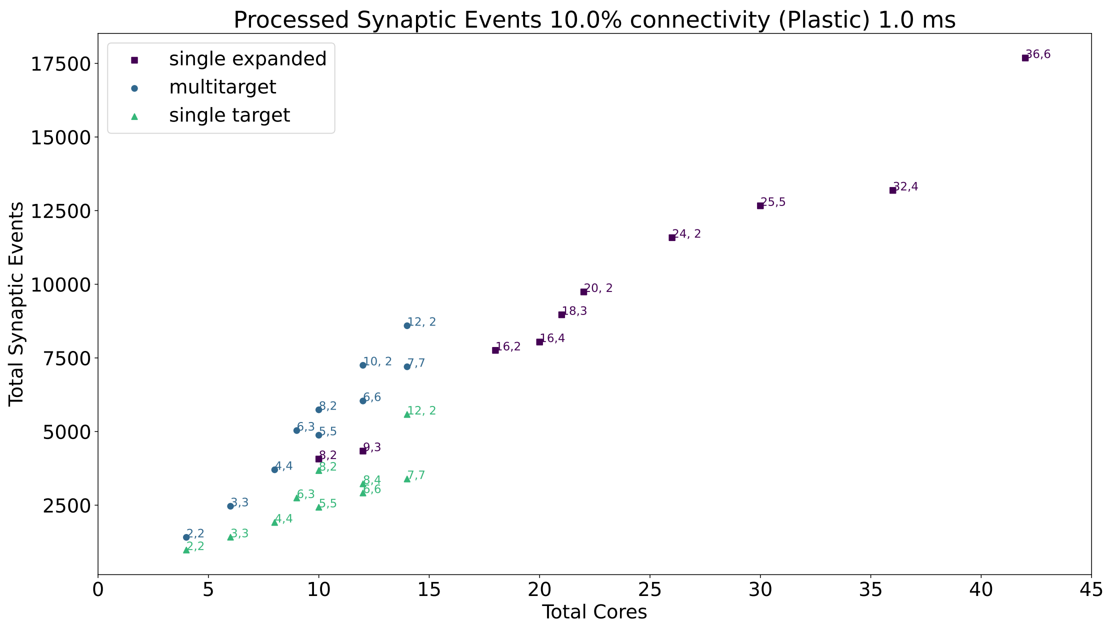


Figure 5.35: Resource allocation vs peak performance. Plastic configuration. Connectivity probability set to 10%

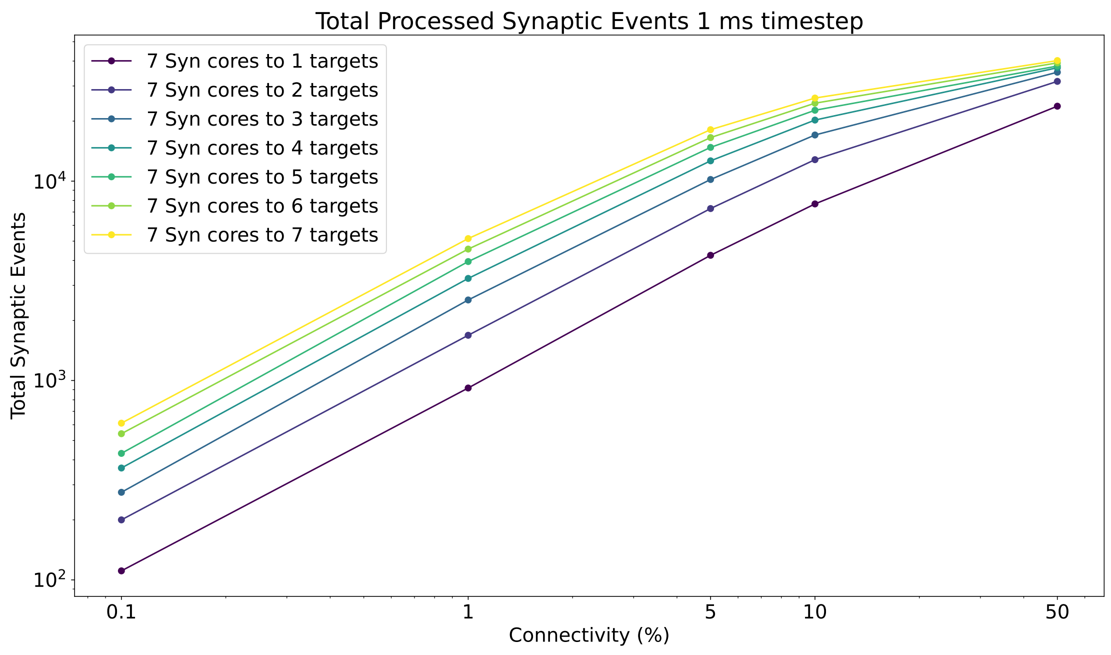


Figure 5.36: Processed synaptic events (1 ms timestep resolution) for different target *Neuron* cores configurations and sparsity levels.

neurons per *Neuron* core in both cases to 64.

Similarly to the peak throughput experiment, the simulation is composed of two populations with variable connectivity probability. The investigated connectivity probabilities for this experiment are 0.1 %, 1%, 5%, 10% and 50%. Denser connectivity patterns are rarely found in biology [HCG+08] and well simulated by traditional hardware such as GPUs, therefore are omitted from this experiment. The results are presented in Figure 5.36 for 1 ms timesteps and Figure 5.37 for 0.1 ms timesteps. On the horizontal axis the connectivity probabilities are shown, the vertical axis displays the processed synaptic events. Each line represents a different configuration of *Synapse* cores to *Neuron* cores, where each *Synapse* core is connected to all the targets of that configuration. The number of synapses per *Synapse* core therefore can be obtained by multiplying the number of *Neuron* cores by 64 (number of neurons per *Neuron* core).

For the 1 ms case (Figure 5.36), as expected, the simulations with higher number of targets process the highest number of synaptic events per timestep. The most evident jump happens between the configurations with 1 and 2 targets respectively, where the synaptic rows double their sizes. This shows that having larger synaptic rows impacts the processing, especially for very sparse networks, by improving the processed synaptic events of $\approx$ 1 order of magnitude for 0.1%

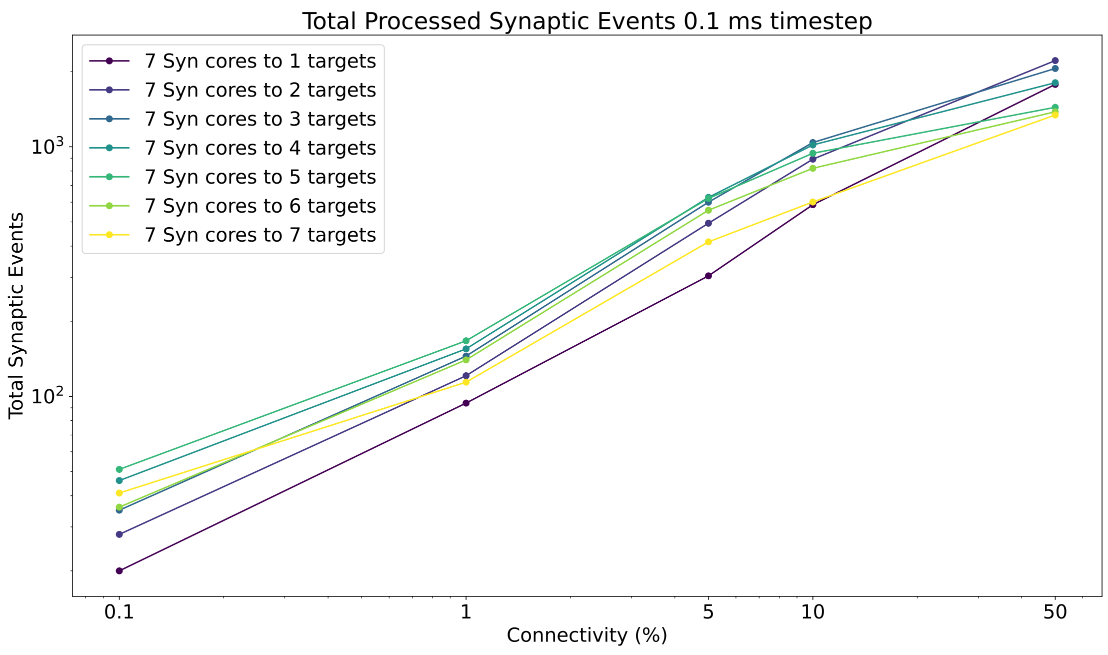


Figure 5.37: Processed synaptic events (0.1 ms timestep resolution) for different target *Neuron* cores configurations and sparsity levels.

connectivity between worst and best case. This gain reduces when the connectivity probability increases, because multiple synaptic events are carried per spike. Therefore the time processing per spike increases as well.

The 0.1 ms case (Figure 5.37) follows a similar trend to the 1 ms case, however the examples with 6 and 7 targets do not give any improvements. The reason for this is due to the time required to perform the transfers between shared and local memories for the synaptic contributions, which have a higher impact on the timestep compared to the 1 ms case. For the sparser simulations (0.1% and 1% connectivity), having multiple target *Neuron* cores gives advantage similarly to the 1 ms case, however, when the network becomes denser the trend starts to invert, having the cost of processing a single packet dominating over the gain introduced by this approach.

5.5 Discussion

The method described in this chapter presents a novel parallelisation approach for neural processing on Neuromorphic hardware, which improves the performance of SNN simulations by acting on the way synaptic matrices are partitioned and processed. The Multi-target partitioning approach provides additional freedom when designing SNN simulations, by allowing to target applications more specifically, according to their requirements. By allowing parameterisation of synaptic and neural processing units, it is possible to allocate the appropriate amount of resources for a given requirement, prioritising the number of *Neuron* processing units for sparser applications and increasing the number of *Synapse* processing units when the fan-in dominates. Thanks to these improvements it is possible to maximise the performance, while using minimal hardware resources and therefore reducing power consumption.

Through a SpiNNaker implementation of the Multi-target partitioning approach, it is possible to improve the peak synaptic processing throughput up to $9\times$ compared to previous results for the same hardware resources. Furthermore, it is possible to obtain comparable processed synaptic events per ms, by reducing the hardware resources to a quarter, resulting in a much smaller machine (and energy consumption) dedicated to the simulation.

The Multi-target partitioning approach additionally enables optimal processing of incoming spike packets, providing a larger pool of target neurons for each spike, hence increasing the length of processed synaptic rows for a given connection density. This greatly reduces the required number of accesses to shared memory per timestep, therefore allowing more efficient processing of sparsely connected networks. This is shown by Equation 5.1 and 5.6, where the number of target neurons for each spike grows according to the number of target *Neuron* cores, expanding the limit beyond a single postsynaptic *Neuron* core. This has the effect of reducing, by a factor N_c the number of destination processors per spike packet, facilitating the routing of spike packets and so reducing the pressure on the communication fabric. Furthermore, this increased number of targets per spike packet, allows to amortise the dominating fixed cost of processing a spike (c_s) [RBB+18] over a higher number of targeted synapses, which can now be larger than that of a single *Neuron* core, overcoming this limitation which is still observed for the Heterogeneous partitioning.

The Multi-target partitioning approach is optimal as it comes with minimal additional costs compared to previous approaches, however the SpiNNaker implementation is limited by the different access patterns to shared memory. The shared memory access time plays a key role in the fraction of the timestep available for spike processing, as shown by Equation 5.2 and by the recorded values presented in Section 5.4.1.1 and 5.4.1.2. The relatively old technology employed by SpiNNaker represents a bottleneck in this context, resulting in both memory contention and transfer size limiting the total system throughput. This causes t_w and t_r (Equations 5.2- 5.4) to increase with the number of cores in the ensemble, consuming approximately half the timestep duration for high timestep resolution simulations such as 0.1 ms. For this reason the need of faster access to shared memory is proven, by showing that there is a large potential gain in having access to multiple separate shared memories, compared to a single shared memory. This consideration opens up to the possibility of using more advanced memory architectures for Neuromorphic hardware, such as multiport memories, since structures like synaptic matrices and synaptic contributions are non-overlapping and therefore would benefit from the capability of separate independent accesses.

The Multi-target partitioning approach also has potential benefits in Neuromorphic systems where all synaptic information is stored locally to the computational units. For these systems the approach would allow synaptic compartments to target multiple neural compartments, improving the handling of sparse connections, and overcoming the limitations set by the fixed coupling between synaptic and neural units. Furthermore the added benefits seen when processing plastic connections offer advantages for online learning applications, particularly in sparsely-connected biologically representative SNNs.

5.6 Summary

This Chapter addresses the third research question presented in Section 1.2, which states: *How can the brain's sparse connectivity and activity be modelled efficiently on Neuromorphic hardware?*

This is addressed through the development of a new parallelisation framework which builds on top of the Heterogeneous model previously proposed. This new method is called Multi-target partitioning and aims at making more efficient use of the system resources by increasing the density of neurons per chip, as well as the size of the synaptic rows processed by each *Synapse* core. The main idea behind this approach is to have *Synapse* cores, a concept introduced with the Heterogeneous Programming model, addressing multiple *Neuron* cores simultaneously. These *Synapse* cores simulate a fraction of the synapses of all the postsynaptic neurons in the group of targeted *Neuron* cores. Under this approach, the neural ensemble described for the Heterogeneous model becomes a group of *Synapse* cores all connected to a group of *Neuron* cores, instead of a group of *Synapse* cores connected to a single *Neuron* core. Together with this approach, a more efficient allocation of memory resources is proposed, highlighting the efficiency of using multiple memories to store synaptic contributions, in order to reduce memory access contention.

The Multi-target partitioning proved to be more efficient than the previous approaches, showing an improvement of up to 9 $\times$ compared to the Heterogeneous model, with the same amount of resources allocated before, and reaching comparable results with previous approaches when only using a quarter of the hardware resources.

Chapter 6

Conclusions

The research presented in this thesis focused on programming techniques for real-time simulations of Spiking Neural Networks on digital Neuromorphic hardware. The SpiNNaker Neuromorphic system was chosen, as the preferred platform, due to its flexibility and scalability to demonstrate efficient approaches to parallelise Spiking Neural Network simulations, in order to achieve real-time simulations of biologically representative networks.

The first research output, namely the Heterogeneous Programming model (presented in Chapter 3), was developed with the aim of improving the placement of Spiking Neural Networks on SpiNNaker, targeting the first research question presented in this manuscript (*How can the process of mapping biologically-representative SNNs be optimised on Neuromorphic Hardware?*). By implementing this approach on the current SpiNNaker software toolchain it was possible to increase the number of processed synaptic events per ms by 12.3× compared to previously published results, while maintaining real-time behaviour. These results suggested to employ this approach to simulate the Cortical Microcircuit network to achieve real-time performance. The application of the Heterogeneous Programming model succeded in simulating the network in real time for the first time ever, achieving a 20× speedup compared to previous simulations. As demonstration of robustness of the approach the simulations were run for 12 hours without interruption. To date this remains the only hard real-time simulation of the full Cortical Microcircuit network on Neuromorphic hardware.

Next, the Heterogeneous Programming model was applied to on-line learning tasks (as described in Chapter 4). This second research effort targeted the second research question (*What are the challenges of implementing on-line learning* *algorithms in real time on Neuromorphic hardware?*), by trying to implement real-time on-line learning algorithms on Neuromorphic hardware. The chosen approach employed multicompartment neuron models from literature to approximate the standard error backpropagation with gradient descent algorithm, but without interrupting the simulation for the backpropagation phase. Thanks to the Heterogeneous model, expanded to include synaptic plasticity, it has been possible to correctly simulate the models and the learning rule on SpiNNaker. However, because of the dense structure of the network and its rate-based nature, which requires constant communication of neurons' firing rates, it was not possible to correctly simulate larger networks, which achieved classification tasks such as image recognition with the MNIST dataset. This exposed limitations of the communication fabric of the chosen platform, which prevented to deliver the correct information to all the destinations in time. The author however does not exclude a future successful implementation, which would require targeted placement strategies which take into account the traffic generated by such a neural network.

Finally, addressing the third research question (*How can the brain's sparse connectivity and activity be modelled efficiently on Neuromorphic hardware?*), the Multi-target partitioning scheme was proposed (presented in Chapter 5), which extends the concept of *Synapse* cores presented by the Heterogeneous Programming model, to address multiple postsynaptic *Neuron* cores. This new approach acts on synaptic matrix partitioning, by allowing postsynaptic processing units to handle rows having configurable sizes (according to the number of target *Neuron* cores). Through this novel strategy it was possible to improve the peak throughput of synaptic events per ms up to 9× compared to the Heterogeneous Programming model and to efficiently simulate sparser networks, by performing a more efficient use of the available hardware resources.

6.1 Future Work

The models presented in this thesis achieved unprecedented throughput of processed synaptic events per timesteps, enabling real-time simulations of biologically-representative SNNs that could not be efficiently simulated before. This section describes future possible extensions to the presented work in order to target further challenges in biologically-representative SNN simulations.

6.1.1 Further Models Extensions

The Multi-target partitioning allows simulations with various ranges of sparsity, by tuning the *Synapse* cores to *Neuron* cores per ensemble ratio. This model is however still limited to the maximum number of cores available per chip, being constrained by shared memory communication. This approach therefore might still present limitations with some edge cases. In simulations involving multiple brain regions (e.g. the Multi-Area model [SBS+18]), the communication among different areas is typically represented by extremely sparse connectivity patterns to model long-range connections. This might result in zero target spike packets even for ensembles with a predominant number of *Neuron* cores, which therefore generate longer synaptic rows. On the other hand, very high fan-in networks (such as cerebellar models [CMM+19]) can result in postsynaptic neurons having an incoming traffic much higher than that allowed by ensembles having a predominance of *Synapse* cores [BMC+21]. For these cases, mechanisms which allow to span beyond the limits of a single chip are needed.

Extreme sparsity could be handled through filtering cores (implementing a mechanism which could be seen as axons in neurons) on external chips. Typically long range connectivity is associated with long delays, therefore intermediate cores which contain additional information on the destination neurons could be used to determine where to send these packets. This would allow reducing the zero target spike packets, having the source cores communicating to a single filtering core, which would then be in charge of forwarding the communication to the destinations.

High fan-in could be handled through a pooling mechanism, therefore it might be possible to instantiate full *Synapse* chips receiving inputs from presynaptic neurons. A potential core per chip could be nominated as *Pooling* core which retrieves the contributions from the *Synapse* cores every timestep similarly to the *Neuron* cores, calculating pooling contributions from the combination of the various synaptic contributions on chip. The *Pooling* cores could then forward their pooling contributions to the postsynaptic *Neuron* cores through packets with payload. This approach would allow scalability, as *Pooling* cores could in principle transmit pooling contributions to other *Pooling* cores that combine them with the local synaptic contributions.

6.1.2 Future Neuromorphic Generations

The applicability of the Heterogeneous and Multi-target partitionings is not limited to the first generation of the SpiNNaker system. The flexibility of the approaches also makes them portable and extendable for the next generation of digital Neuromorphic platforms. The second iteration of SpiNNaker is due to release at the time of writing of this manuscript. This new generation features much more advanced hardware characteristics, where each chip contains a total of 152 ARM Cortex-M4 cores (also referred to as Processing Elements, or PEs) arranged in quartets denominated Quad-Processing Elements (QPEs). Inside a QPE each processor can efficiently read another core's local data memory, through a modified NoC. Each core has a private 128 KB SRAM, and a shared 8 GB DRAM memory is provided per chip [HYD$^+$21, YSC$^+$21] together with 6 bidirectional links managed by an on-chip router, containing 16384 associative routing entries. A Multi-target model implementation for SpiNNaker 2 could therefore map a cluster-based implementation of multiple neural ensembles per chip, where each PE represents either a *Neuron* core or a *Synapse* core. Since each PE has the capability to access the local memory of other PEs on the same QPE efficiently, it is possible to share the synaptic contributions within a QPE, overcoming the contention issue. A step further would include a tree-like structure, where QPEs could implement a group of 4 *Synapse* cores, which generate the synaptic contributions as a single block for the 4 cores, then a single PE per QPE accesses the chip shared memory to communicate with other QPEs implementing blocks of *Neuron* cores. Following the same strategy, a single *Neuron* core per *Neuron* QPE accesses the shared memory to retrieve the contributions. This would expand the ensemble capabilities to a full chip, limiting the memory contention to a quarter of the cores in use. Combined with the much higher memory throughput (6 GB/s vs 1 GB/s for the SpiNNaker SDRAM), this would have a large impact on the synaptic contributions reading and writing times.

6.2 Overall Summary

The presented parallelisation strategies demonstrated unprecedented results when applied to SNNs simulations on digital Neuromorphic hardware. The real-time simulations of the Cortical Microcircuit network demonstrated the capabilities of digital Neuromorphic hardware in simulating complex biologically-representative

SNNs, showing the large impact of a targeted programming approach and proving the capabilities of such architectures in real-time simulations. The flexibility and efficiency of the parallelisation approach have been furthermore demonstrated by employing it in a different context, including dense connectivity patterns and rate encoding, where it demonstrated to handle well such synaptic fan-in. Finally, targeting very sparse SNN simulations, unprecedented results were demonstrated, further enabling a customisation of the network placement phase, to optimise the use of hardware resources, according to the network requirements.

This work aims at providing programming paradigms in order to better exploit this novel type of hardware to bridge the gap in communication between neuroscientists and computer architects. Such programming paradigms can be extended and customised to be applied to the next generation of Neuromorphic hardware, providing further insights on design choices for the future generations of such systems and learning algorithms, targeting sub real-time performance. The design of efficient simulation platforms would in turn enable more efficient learning applications, which combined with low power consumption, could be employed in edge computing scenarios, such as medical devices, space technologies and IoT applications.

Bibliography

[AE05] Peter A. Appleby and Terry Elliott. Synaptic and Temporal Ensemble Interpretation of Spike-Timing-Dependent Plasticity. *Neural Computation*, 17(11):2316–2336, 11 2005.

[AN00] L.F. Abbott and Sacha B. Nelson. Synaptic plasticity: taming the beast. *Nat. Neurosci*, 3:1178–1183, 2000.

[ARM06] ARM. ARM968E-S Technical Reference Manual. *ARM*, 2006. Available at `https://developer.arm.com/documentation/ddi0311/`.

[ARM16a] ARM. ARM Compiler armcc User Guide. *ARM*, 2016. Available at `https://developer.arm.com/documentation/dui0472/`.

[ARM16b] ARM. GNU ARM Embedded toolchain. *ARM*, 2016. Available at `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm`.

[ASC$^+$15] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar, William P. Risk, Bryan Jackson, and Dharmendra S. Modha. TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, 2015.

[BFR$^+$15] Andrew D. Brown, Steve B. Furber, Jeffrey S. Reeve, Jim D. Garside, Kier J. Dugan, Luis A. Plana, and Steve Temple. SpiNNaker—Programming Model. *IEEE Transactions on Computers*, 64(6):1769–1782, 2015.

[BG05] Romain Brette and Wulfram Gerstner. Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity. *Journal of Neurophysiology*, 94(5):3637–3642, 2005. PMID: 16014787.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, Berlin, Heidelberg, 2006.

[BMC+21] Petruţ A. Bogdan, Beatrice Marcinnò, Claudia Casellato, Stefano Casali, Andrew G.D. Rowley, Michael Hopkins, Francesco Leporati, Egidio D'Angelo, and Oliver Rhodes. Towards a Bio-Inspired Real-Time Neuromorphic Cerebellum. *Frontiers in Cellular Neuroscience*, 15:130, 2021.

[Boa08] OpenMP Architecture Review Board. OpenMP Application Program Interface. 2008. Available at `http://www.openmp.org/mp-documents/spec30.pdf`.

[BP98] Guo-qiang Bi and Mu-ming Poo. Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type. *Journal of Neuroscience*, 18(24):10464–10472, 1998.

[BP01] Guo-qiang Bi and Mu-ming Poo. Synaptic Modification by Correlated Activity: Hebb's Postulate Revisited. *Annual Review of Neuroscience*, 24(1):139–166, 2001.

[BSS+20] Guillaume Bellec, Franz Scherr, Anand Subramoney, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications*, 11, 2020.

[CATvS17] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: an extension of MNIST to handwritten letters. *CoRR*, abs/1702.05373, 2017.

[CBM+20] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Muhammad Shafique, Guido Masera, and Maurizio Martina. An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks. *Future Internet*, 12(7), 2020.

[CDLB$^+$22] Dennis V. Christensen, Regina Dittmann, Bernabe Linares-Barranco, Abu Sebastian, Manuel Le Gallo, Andrea Redaelli, Stefan Slesazeck, Thomas Mikolajick, Sabina Spiga, Stephan Menzel, Ilia Valov, Gianluca Milano, Carlo Ricciardi, Shi-Jun Liang, Feng Miao, Mario Lanza, Tyler J. Quill, Scott T. Keene, Alberto Salleo, Julie Grollier, Danijela Markovic, Alice Mizrahi, Peng Yao, J. Joshua Yang, Giacomo Indiveri, John P. Strachan, Suman Datta, Elisa Vianello, Alexandre Valentian, Johannes Feldmann, Xuan Li, Wolfram HP Pernice, Harish Bhaskaran, Steve B. Furber, Emre Neftci, Franz Scherr, Wolfgang Maass, Srikanth Ramaswamy, Jonathan Tapson, Priyadarshini Panda, Youngeun Kim, Gouhei Tanaka, Simon Thorpe, Chiara Bartolozzi, Thomas A Cleland, Christoph Posch, Shih-Chii Liu, Gabriella Panuccio, Mufti Mahmud, Arnab Neelim Mazumder, Morteza Hosseini, Tinoosh Mohsenin, Elisa Donati, Silvia Tolu, Roberto Galeazzi, Martin E. Christensen, Sune Holm, Daniele Ielmini, and Nini Pryds. 2022 roadmap on neuromorphic computing and engineering. *Neuromorphic Computing and Engineering*, 2022.

[CH06] Nicholas T. Carnevale and Michael L. Hines. *The NEURON Book*. Cambridge University Press, 2006.

[CMM$^+$19] Stefano Casali, Elisa Marenzi, Chaitanya Medini, Claudia Casellato, and Egidio D'Angelo. Reconstruction and Simulation of a Scaffold Model of the Cerebellar Network. *Frontiers in Neuroinformatics*, 13:37, 2019.

[CSBI14] Elisabetta Chicca, Fabio Stefanini, Chiara Bartolozzi, and Giacomo Indiveri. Neuromorphic Electronic Circuits for Building Autonomous Cognitive Systems. *Proceedings of the IEEE*, 102(9):1367–1388, 2014.

[Dav12] Sergio Davies. Learning in Spiking Neural Networks. *PhD Thesis*, 2012.

[DBE$^+$09] Andrew Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:11, 2009.

[DC14] Peter U. Diehl and Matthew Cook. Efficient implementation of STDP rules on SpiNNaker neuromorphic hardware. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 4288–4295, 2014.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[DSL+18] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruhuhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro*, 38(1):82–99, 2018.

[FFS+13] Simon Friedmann, Nicolas Frémaux, Johannes Schemmel, Wulfram Gerstner, and Karlheinz Meier. Reward-based learning under hardware constraints—using a RISC processor embedded in a neuromorphic substrate. *Frontiers in Neuroscience*, 7:160, 2013.

[FGTP14] Steve B. Furber, Francesco Galluppi, Steve Temple, and Luis A. Plana. The SpiNNaker Project. *Proceedings of the IEEE*, 102(5):652–665, 2014.

[FLP+13] Steve B. Furber, David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown. Overview of the SpiNNaker System Architecture. *IEEE Transactions on Computers*, 62(12):2454–2467, Dec 2013.

[FMCR19] Carlos Fernandez-Musoles, Daniel Coca, and Paul Richmond. Communication Sparsity in Distributed Spiking Neural Network Simulations to Improve Scalability. *Frontiers in Neuroinformatics*, 13, 2019.

[FOF+20] Edward Paxon Frady, Garrick Orchard, David Florey, Nabil Imam, Ruokun Liu, Joyesh Mishra, Jonathan Tse, Andreas

Wild, Friedrich T. Sommer, and Mike Davies. Neuromorphic Nearest-Neighbor Search Using Intel's Pohoiki Springs. *CoRR*, abs/2004.12691, 2020.

[For09] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.2. 2009. Available at `http://www.mpi-forum.org/docs/`.

[FSG+17] Simon Friedmann, Johannes Schemmel, Andreas Grübl, Andreas Hartel, Matthias Hock, and Karlheinz Meier. Demonstrating Hybrid Learning in a Flexible Neuromorphic Hardware System. *IEEE Transactions on Biomedical Circuits and Systems*, 11:128–142, 2017.

[Fur16] Steve B. Furber. Large-scale neuromorphic computing systems. *Journal of Neural Engineering*, 13(5):051001, aug 2016.

[GB08] Dan Goodman and Romain Brette. Brian: a simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics*, 2:5, 2008.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[GD07] Marc-Oliver Gewaltig and Markus Diesmann. NEST (NEural Simulation Tool. *Scholarpedia 2:1430*, 2007.

[Gen19] Edd Gent. Supercomputer simulates key brain centre in real time. *New Scientist*, 244(3251):7, 2019.

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

[GLR17a] Jordan Guerguiev, Timothy P. Lillicrap, and Blake A. Richards. Towards deep learning with segregated dendrites. *eLife*, 6:e22901, dec 2017.

[GLR17b] Jordan Guerguiev, Timothy P. Lillicrap, and Blake A. Richards. Towards deep learning with segregated dendrites. *eLIFE*, 6, 2017.

[Gra12] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, Berlin, Heidelberg, 2012.

[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 Biology*, 6(7):1–15, 07 2008.

[Heb49] Donald O. Hebb. *The organization of behavior : a neuropsychological theory*. Wiley, 1949.

[HH52] Alan L. Hodgkin and Andrew F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952.

[HH11] Suzana Herculano-Houzel. Scaling of brain metabolism with a fixed energy budget per neuron: implications for neuronal activity, plasticity and evolution. *PLoS One*, 6, 2011.

[HM03] Michael Häusser and Bartlett Mel. Dendrites: bug or feature? *Current Opinion in Neurobiology*, 13(3):372–383, 2003.

[HPT+22] Arne Heittmann, Georgia Psychou, Guido Trensch, Charles E. Cox, Winfried W. Wilcke, Markus Diesmann, and Tobias G. Noll. Simulating the Cortical Microcircuit Significantly Faster Than Real Time on the IBM INC-3000 Neural Supercomputer. *Frontiers in Neuroscience*, 15, 2022.

[Hun] Hunsberger, Eric. *Spiking Deep Neural Networks: Engineered and Biological Approaches to Object Recognition.* PhD thesis.

[HYD+21] Sebastian Höppner, Yexin Yan, Andreas Dixius, Stefan Scholze, Johannes Partzsch, Marco Stolba, Florian Kelber, Bernhard Vogginger, Felix Neumärker, Georg Ellguth, Stephan Hartmann, Stefan Schiefer, Thomas Hocker, Dennis Walter, Genting Liu, Jim Garside, Steve B. Furber, and Christian Mayr. The SpiNNaker 2 Processing Element Architecture for Hybrid Digital Neuromorphic Computing, 2021.

[IEPD17] Tammo Ippen, Jochen M. Eppler, Hans E. Plesser, and Markus Diesmann. Constructing Neuronal Network Models in Massively Parallel Environments. *Frontiers in Neuroinformatics*, 11:30, 2017.

[IGB19] Bernd Illing, Wulfram Gerstner, and Johanni Brea. Biologically plausible deep learning — But how far can we go with shallow networks? *Neural Networks*, 118:90–101, 2019.

[ILBH$^+$11] Giacomo Indiveri, Bernabe Linares-Barranco, Tara Hamilton, André van Schaik, Ralph Etienne-Cummings, Tobi Delbruck, Shih-Chii Liu, Piotr Dudek, Philipp Häfliger, Sylvie Renaud, Johannes Schemmel, Gert Cauwenberghs, John Arthur, Kai Hynna, Fopefolu Folowosele, Sylvain SAÏGHI, Teresa Serrano-Gotarredona, Jayawan Wijekoon, Yingxue Wang, and Kwabena Boahen. Neuromorphic Silicon Neuron Circuits. *Frontiers in Neuroscience*, 5:73, 2011.

[Int21] Intel. Lava- A software framework for neuromorphic computing. *Library Documentation*, 2021. Available at `https://lava-nc.org/`.

[Izh03] Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, 2003.

[Izh04] Eugene M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004.

[Izh07] Eugene M. Izhikevich. Solving the Distal Reward Problem through Linkage of STDP and Dopamine Signaling. *Cerebral Cortex*, 17(10):2443–2452, 01 2007.

[JIH$^+$18] Jakob Jordan, Tammo Ippen, Moritz Helias, Itaru Kitayama, Mitsuhisa Sato, Jun Igarashi, Markus Diesmann, and Susanne Kunkel. Extremely Scalable Spiking Neuronal Network Simulation Code: From Laptops to Exascale Computers. *Frontiers in Neuroinformatics*, 12:2, 2018.

[JRG$^+$10] Xin Jin, Alexander Rast, Francesco Galluppi, Sergio Davies, and Steve B. Furber. Implementing spike-timing-dependent plasticity on SpiNNaker neuromorphic hardware. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2010.

[KF16] James C. Knight and Steve B. Furber. Synapse-Centric Mapping of Cortical Models to the SpiNNaker Neuromorphic Architecture. *Frontiers in Neuroscience*, 10:420, 2016.

[KK01] Konrad P. Körding and Peter König. Supervised and unsupervised learning with two sites of synaptic integration. *Journal of Computational Neuroscience*, 11:207–215, 2001.

[KKN21] James C. Knight, Anton Komissarov, and Thomas Nowotny. PyGeNN: A Python Library for GPU-Enhanced Neural Networks. *Frontiers in Neuroinformatics*, 15:10, 2021.

[KN18] James C. Knight and Thomas Nowotny. GPUs Outperform Current HPC and Neuromorphic Solutions in Terms of Speed and Energy When Simulating a Highly-Connected Cortical Model. *Frontiers in Neuroscience*, 12:941, 2018.

[KN21] James C. Knight and Thomas Nowotny. Larger GPU-accelerated brain simulations with procedural connectivity. *Nature Computational Science*, 1:136–142, 2021.

[KR07] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[KST$^+$21] Anno C. Kurth, Johanna Senk, Dennis Terhorst, Justin Finnerty, and Markus Diesmann. Sub-realtime simulation of a neuronal network of natural density, 2021.

[Lar13] Matthew Larkum. A cellular mechanism for cortical associations: an organizing principle for the cerebral cortex. *Trends in neurosciences*, 36:141–151, 2013.

[LBBH98] Yann Lecun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[LC20] William B. Levy and Victoria G. Calvert. Computation in the human cerebral cortex uses less than 0.2 watts yet this great expense is optimal when considering communication costs. *bioRxiv*, 2020.

[LCTA16a] Timothy P. Lillicrap, Daniel Cownden, Douglas B. Tweed, and Colin J. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7, 2016.

[LCTA16b] Timothy P. Lillicrap, Daniel Cownden, Douglas B. Tweed, and Colin J. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7, 2016.

[LDBK20] Jesus L. Lobo, Javier Del Ser, Albert Bifet, and Nikola Kasabov. Spiking Neural Networks and online learning: An overview and perspectives. *Neural Networks*, 121:88–100, 2020.

[LDC$^+$20] Guoqi Li, Lei Deng, Yansong Chua, Peng Li, Emre O. Neftci, and Haizhou Li. Editorial: Spiking Neural Network Learning, Benchmarking, Programming and Executing. *Frontiers in Neuroscience*, 14, 2020.

[LPD08] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128 $\times$ 128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor. *IEEE Journal of Solid-State Circuits*, 43(2):566–576, 2008.

[LWC$^+$18a] Chit-Kwan Lin, Andreas Wild, Gautham N. Chinya, Yongqiang Cao, Mike Davies, Daniel M. Lavery, and Hong Wang. Programming Spiking Neural Networks on Intel's Loihi. *Computer*, 51(3):52–61, 2018.

[LWC$^+$18b] Chit-Kwan Lin, Andreas Wild, Gautham N. Chinya, Tsung-Han Lin, Mike Davies, and Hong Wang. Mapping Spiking Neural Networks onto a Manycore Neuromorphic Architecture. *SIGPLAN Not.*, 53(4):78–89, jun 2018.

[Maa97] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997.

[MAAI$^+$14] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and

Dharmendra S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[MAD07] Abigail Morrison, Ad Aertsen, and Markus Diesmann. Spike-Timing-Dependent Plasticity in Balanced Random Networks. *Neural Computation*, 19(6):1437–1467, 06 2007.

[MDG08] Abigail Morrison, Markus Diesmann, and Wulfram Gerstner. Phenomenological models of synaptic plasticity based on spike timing. *Biol. Cybern.*, 98:459–478, 2008.

[Mea89] Carver Mead. *Analog VLSI and Neural Systems.* Addison-Wesley Longman Publishing Co., Inc., USA, 1989.

[Mea90] Carver Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636, Oct 1990.

[Mey01] Robert A. Meyers. *Encyclopedia of Physical Science and Technology.* Elsevier, 2001.

[MFM$^+$12] Simon W. Moore, Paul J. Fox, Steven J.T. Marsh, A. Theodore Markettos, and Alan Mujumdar. Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 133–140, 2012.

[MGS12] Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. Spike-Timing-Dependent Plasticity: A Comprehensive Overview. *Frontiers in Synaptic Neuroscience*, 4:2, 2012.

[MLP$^+$15] Javier Mavaridas, Mikel Luján, Luis A. Plana, Steve Temple, and Steve B. Furber. SpiNNaker: Enhanced multicast routing. *Parallel Computing*, 45:49 – 66, 2015. Computing Frontiers 2014: Best Papers.

[MMG$^+$05] Abigail Morrison, Carsten Mehring, Theo Geisel, Ad Aertsen, and Markus Diesmann. Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Computation*, 17:1776–1801, 2005.

[MMG$^+$07] Liam P. Maguire, T. M. McGinnity, Brendan P. Glackin, Arfan Ghani, Ammar Belatreche, and Jim Harkin. Challenges for large-scale implementations of spiking neural networks on fpgas. *Neurocomputing*, 71(1):13–29, 2007. Dedicated Hardware Architectures for Intelligent Systems Advances on Neural Networks for Speech and Audio Processing.

[Moo06] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 11(5):115–133, 1943.

[NCA$^+$20] Pritish Narayanan, Charles E. Cox, Alexis Asseman, Nicolas Antoine, Harald Huels, Winfried W. Wilcke, and Ahmet S. Ozcan. Overview of the IBM Neural Computer Architecture. *CoRR*, abs/2003.11178, 2020.

[NVI] NVIDIA Corporation. NVIDIA Ampere GA102 GPU Architecture. Available at `https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf`.

[OFR$^+$21] Garrick Orchard, E. Paxon Frady, Daniel Ben Dayan Rubin, Sophia Sanborn, Sumit Bam Shrestha, Friedrich T. Sommer, and Mike Davies. Efficient Neuromorphic Signal Processing with Loihi 2. In *2021 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 254–259, 2021.

[PBC$^+$22] Christian Pehle, Sebastian Billaudelle, Benjamin Cramer, Jakob Kaiser, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, Aron Leibfried, Eric Müller, and Johannes Schemmel. The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity, 2022.

[PD12] Tobias C. Potjans and Markus Diesmann. The Cell-Type Specific

Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Model. *Cerebral Cortex*, 24(3):785–806, 12 2012.

[PFT+07] Luis A. Plana, Steve B. Furber, Steve Temple, Mukaram Khan, Yebin Shi, Jian Wu, and Shufan Yang. A GALS Infrastructure for a Massively Parallel Multiprocessor. *IEEE Design Test of Computers*, 24(5):454–463, 2007.

[PG06] Jean-Pascal Pfister and Wulfram Gerstner. Triplets of Spikes in a Model of Spike Timing-Dependent Plasticity. *Journal of Neuroscience*, 26(38):9673–9682, 2006.

[PPG+13] Eustace Painkras, Luis A. Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R. Lester, Andrew D. Brown, and Steve B. Furber. SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation. *IEEE Journal of Solid-State Circuits*, 48(8):1943–1953, Aug 2013.

[PR20] Luca Peres and Oliver Rhodes. Real-time cortical simulation on SpiNNaker. 2020. Available at `https://doi.org/10.6084/m9.figshare.17086667.v1`.

[PR22] Luca Peres and Oliver Rhodes. Parallelization of Neural Processing on Neuromorphic Hardware. *Frontiers in Neuroscience*, 16, 2022.

[QMC+15] Ning Qiao, Hesham Mostafa, Federico Corradi, Marc Osswald, Fabio Stefanini, Dora Sumislawska, and Giacomo Indiveri. A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses. *Frontiers in Neuroscience*, 9:141, 2015.

[RBB+18] Oliver Rhodes, Petruţ A. Bogdan, Christian Brenninkmeijer, Simon Davidson, Donal Fellows, Andrew Gait, David R. Lester, Mantas Mikaitis, Luis A. Plana, Andrew G. D. Rowley, Alan B. Stokes, and Steve B. Furber. sPyNNaker: A Software Package for Running PyNN Simulations on SpiNNaker. *Frontiers in Neuroscience*, 12:816, 2018.

[RBD+19] Andrew G. D. Rowley, Christian Brenninkmeijer, Simon Davidson, Donal Fellows, Andrew Gait, David R. Lester, Luis A. Plana, Oliver

Rhodes, Alan B. Stokes, and Steve B. Furber. SpiNNTools: The Execution Engine for the SpiNNaker Platform. *Frontiers in Neuroscience*, 13:231, 2019.

[RHTF03] Daniel Roggen, Stephane Hofmann, Yann Thoma, and Dario Floreano. Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot. In *NASA/DoD Conference on Evolvable Hardware, 2003. Proceedings.*, pages 189–198, 2003.

[RHW86] David Rumelhart, Geoffrey Hinton, and Ronald Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[RLS01] Jonathan Rubin, Daniel D. Lee, and H. Sompolinsky. Equilibrium Properties of Temporally Asymmetric Hebbian Plasticity. *Phys. Rev. Lett.*, 86:364–367, Jan 2001.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[RPR+19] Oliver Rhodes, Luca Peres, Andrew G. D. Rowley, Andrew Gait, Luis A. Plana, Christian Brenninkmeijer, and Steve B. Furber. Real-time cortical simulation on neuromorphic hardware. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378, Dec 2019.

[SBS+18] Maximilian Schmidt, Rembrandt Bakker, Kelly Shen, Gleb Bezgin, Markus Diesmann, and Sacha J. van Albada. A multi-scale layer-resolved spiking network model of resting-state dynamics in macaque visual cortical areas. *PLoS Comput Biol*, 2018.

[SFM08] Johannes Schemmel, Johannes Fieres, and Karlheinz Meier. Wafer-scale integration of analog neural networks. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 431–438, 2008.

[SH99] Piergiorgio Strata and Robin Harvey. Dale's principle. *Brain Research Bulletin*, 50(5):349–350, 1999.

[SKMM17] Johannes Schemmel, Laura Kriener, Paul Müller, and Karlheinz Meier. An accelerated analog neuromorphic hardware system emulating NMDA- and calcium-based non-linear dendrites. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2217–2226, 2017.

[SKP$^+$22] Catherine D. Schuman, Shruti R. Kulkarni, Maryam Parsa, J. Parker Mitchell, Prasanna Date, and Bill Kay. Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science*, 2:10–19, 2022.

[SMKS00] Jackie Schiller, Guy Major, Helmut J. Koester, and Yitzhak Schiller. NMDA spikes in basal dendrites of cortical pyramidal neurons. *Nature*, 404:285–289, 2000.

[SNSI14] Fabio Stefanini, Emre O. Neftci, Sadique Sheik, and Giacomo Indiveri. PyNCS: a microkernel for high-level definition and configuration of neuromorphic electronic systems. *Frontiers in Neuroinformatics*, 8:73, 2014.

[SPCBS18] João Sacramento, Rui Ponte Costa, Yoshua Bengio, and Walter Senn. Dendritic cortical microcircuits approximate the backpropagation algorithm. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[SPGF11] Thomas Sharp, Luis A. Plana, Francesco Galluppi, and Stephen B. Furber. Event-Driven Simulation of Arbitrary Spiking Neural Networks on SpiNNaker. In *ICONIP*, 2011.

[Spi11a] SpiNNaker Group. SpiNNaker Application Programming Interface (API). *Technical report*, 2011. Available at `http://spinnakermanchester.github.io/docs/SpiNNapi_docV200.pdf`.

[Spi11b] SpiNNaker Group. SpiNNaker Datasheet Version 2.02. *Technical report*, 2011. Available at `http://spinnakermanchester.github.io/docs/SpiNN2DataShtV202.pdf`.

[Spi15] SpiNNaker Group. Rig - libraries for SpiNNaker application support. *Library Documentation*, 2015. Available at `https://rig.readthedocs.io/en/stable/#`.

[Spr08] Nelson Spruston. Pyramidal neurons: dendritic structure and synaptic integration. *Nature Reviews Neuroscience*, 9(3):206–221, 2008.

[SS01] Jackie Schiller and Yitzhak Schiller. NMDA receptor-mediated dendritic spikes and coincident signal amplification. *Current Opinion in Neurobiology*, 11(3):343–348, 2001.

[Tem11] Steve Temple. The APLX File Format. *Technical report*, 2011. Available at `http://spinnakermanchester.github.io/docs/spinn-app-3.pdf`.

[Tem16] Steve Temple. SARK - SpiNNaker Application Runtime Kernel. *Technical report*, 2016.

[TMC$^+$18] Chetan Singh Thakur, Jamal Lottier Molin, Gert Cauwenberghs, Giacomo Indiveri, Kundan Kumar, Ning Qiao, Johannes Schemmel, Runchun Wang, Elisabetta Chicca, Jennifer Olson Hasler, Jae-sun Seo, Shimeng Yu, Yu Cao, André van Schaik, and Ralph Etienne-Cummings. Large-Scale Neuromorphic Spiking Array Processors: A Quest to Mimic the Brain. *Frontiers in Neuroscience*, 12:891, 2018.

[US14] Robert Urbanczik and Walter Senn. Learning by the Dendritic Prediction of Somatic Spiking. *Neuron*, 81(3):521–528, 2014.

[vARS$^+$18] Sacha J. van Albada, Andrew G. Rowley, Johanna Senk, Michael Hopkins, Maximilian Schmidt, Alan B. Stokes, David R. Lester, Markus Diesmann, and Steve B. Furber. Performance Comparison of the Digital Neuromorphic Hardware SpiNNaker and the Neural Network Simulation Software NEST for a Full-Scale Cortical Microcircuit Model. *Frontiers in Neuroscience*, 12:291, 2018.

[WF10] Jian Wu and Steve Furber. A Multicast Routing Scheme for a Universal Spiking Neural Network Architecture. *The Computer Journal*, 53(3):280–288, 2010.

[WFG09] Jian Wu, Steve Furber, and Jim Garside. A Programmable Adaptive Router for a GALS Parallel System. In *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 23–31, 2009.

[YDH$^+$15] Alper Yegenoglu, A Davidson, D Holstein, E Muller, E Torre, and J Sprenger. Elephant – Open-Source Tool for the Analysis of Electrophysiological Data Sets. *Proc. INM Retreat*, 2015.

[YSC$^+$21] Yexin Yan, Terrence C. Stewart, Xuan Choo, Bernhard Vogginger, Johannes Partzsch, Sebastian Höppner, Florian Kelber, Chris Eliasmith, Steve B. Furber, and Christian Mayr. Comparing Loihi with a SpiNNaker 2 prototype on low-latency keyword spotting and adaptive robotic control. *Neuromorphic Computing and Engineering*, 1(1):014002, jul 2021.

[YTN16] Esin Yavuz, James Turner, and Thomas Nowotny. GeNN: a code generation framework for accelerated brain simulations. *Scientific Reports*, 6(1):18854, 2016.