


Doctoral Dissertation
Doctoral Program in Computer Enginering (32^{nd} cycle)

Manycore and Heterogeneous Architectures:

Programming Models and Compilation Toolchains

Francesco Barchi

* * * * * *

Supervisors

Prof. Enrico Macii
Prof. Andrea Acquaviva

Doctoral Examination Committee:
Prof. Tullio Salmon Cinotti, Università di Bologna, Referee
Prof. Nicola Bombieri, Università di Verona, Referee
Prof. Ayse Kivilcim Coskun, Boston University
Prof. Massimo Violante, Politecnico di Torino
Prof. Andrea Bartolini, Università di Bologna

Politecnico di Torino
2 October, 2020

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Francesco Barchi
Turin, 2 October, 2020

Summary

In the last ten years, we have witnessed a revolution in technology. The achievement of the physical limits of silicon lithography has required several architectural innovations, both in the development and integration of dedicated accelerators and in biomimetic systems that seek to change the traditional computational model radically. Nowadays, heterogeneous architectures in a single silicon slice integrate radically different computational models such as SIMD and MIMD, but also programmable logic (FPGA). In the same period Machine learning has developed enormously, and with the advent of Deep Learning the demand for dedicated hardware has led to an explosion of Cambrian accelerators and specialised architectures. In the field of biomimetic systems, the scientific community and companies, like IBM and Intel, have shown interest in neuromorphic systems. A neuromorphic system can emulate at hardware or software level the electrical behaviour of neural networks.

In this context of heterogeneity and new paradigms in computing devices, my research has focused on the programming models of these architectures and the optimisation of their resources. My work has focused on neuromorphic architectures based on manycore architectures and heterogeneous embedded architectures. Neuromorphic architectures offer a new computational paradigm that requires the careful use of communication resources and the development of the entire software stack to provide an appropriate programming model capable of masking the internal architecture complexity to the user. Heterogeneous architectures also require masking their complexity to the user. In this case, I worked on the strengthening of the current compilation chains through the use of deep learning to perform code analysis and to extract information useful to make complex decisions now delegated to the programmer, for example choosing the most suitable calculation unit for code execution.

In the first case, I worked on SpiNNaker within the European Human Brain Project (HBP). SpiNNaker is a manycore neuromorphic architecture for which I developed a network resource optimisation system. Specifically, I developed a toolchain for the

mapping of Spiking Neural Network (SNN) within SpiNNaker measuring its performance based on the reduction of communication on the architecture. By testing my toolchain on a biological network simulation, I experimentally obtained a traffic reduction of 90x compared to performance obtained using the official toolchain. I also developed a communication middleware that exploits the architecture features to implement an MPI-based programming model, providing the user with a library able to abstract the architectural complexity of the architecture and allowing to exploit the architecture easily even outside the neuromorphic simulation context. Specifically, I developed a communication middleware (MCM) that exploits the architecture features to provide broadcast and unicast communication mechanisms and synchronisation primitives between SpiNNaker processors. It was then possible to implement a library (ACF) useful for the rapid reconfiguration of neuronal simulations and to allow rapid exchange of data between the neuromorphic architecture and external architectures. The implementation of MPI relies on these two libraries and allowed to implement two parallelised applications on SpiNNaker. The first application was an N-Body simulation where 2 k particles were simulated on 240 processors with a speed-up of 194 x and efficiency of 80% when compared to the serial version running on a single SpiNNaker core. The second application was a DNA sequence matching algorithm. Results show that the scalability of the SpiNNaker board reaches a 98% efficiency when using more than 100 processors, a 90% efficiency using 600 processors, achieving 88% efficiency when all 767 application processors are used.

In the second case, I worked on source code classification via deep neural networks. In particular, I built a source code classifier in the intermediate representation of LLVM able to discriminate the most suitable calculation unit for fast execution of the analysed code. During this work, I compared two network models, one based on recurrent networks (RNN) and the other on convolutions (CNN). I showed that code analysis at this level is possible by reaching an 85% accuracy in OpenCL kernel classification to run on a multicore CPU or GPU. Besides, the CNN model has proven to be more accurate and easier to train.

Overall this work has allowed exploring the potential of these new generation architectures and will be useful technologies for future developments.

Acknowledgements

These three years of doctorate were made possible thanks to the guidance of Prof. Andrea Acquaviva, the advice of Gianvito Urgese and the support of Prof. Enrico Macii whom I thank from the bottom of my heart. I thank the entire EDA group of the Polytechnic of Turin for the sincere friendship and personal growth, I will always carry you in my heart.

My family has a special place in these thanks, my father Alessandro, my mother Carmela and my two brothers Cristina and Lorenzo are cardinal points of my life to whom I owe what I am and what I aspire to become.

“Grazie di tutto il tuo amore, nostra Signora istruzione”

La bella creola
Murubutu

Contents

List of Tables

List of Figures

Chapter 1

Research context

The research work, carried out during the three years of my PhD, focused on programming models for the new generation of advanced multicore architectures.

In the last decade, we have witnessed a revolution in technology. The approaching physical limits of silicon lithography has made it necessary to explore various architectural innovations, both in the development and integration of dedicated accelerators and in biomimetic systems that are trying to change the current computational model radically. Among these new generation architectures are heterogeneous embedded architectures i.e. composed of several compute units specialised in different tasks for which it becomes increasingly challenging to relieve the programmer from using and knowing specific libraries to exploit the resources offered by the system. Another frontier of manycore architectures is the neuromorphic architectures. They require an entirely new programming model and a software stack that abstracts the functionality to allow the programmer to use them with the same simplicity of traditional architectures. Moreover, in the latter type of architectures, there is a need to optimise the allocation of communication as they are developed to simulate biological neural networks whose computing units interact through an intense exchange of messages.

Mapping tasks on hardware units available on heterogeneous and manycore systems is one of the main challenges for software developers with a significant impact on application reliability, performance, and energy consumption [67, 38, 104]. This problem is common in many fields of applications that go from the mapping of parallel applications on stream-oriented multiprocessor system-on-chip (MPSoC) [78] to the mapping of Spiking Neural Networks on Neuromorphic Platforms [31]. Approaches at compiler, programming model [23, 88, 100, 77, 91], resource allocation and optimisation levels is ongoing to improve the exploitation of such complex architectures.

My thesis work presents different solutions in this field of research. In the case of neuromorphic architectures, I focused on the development of a programming model able to exploit its potential by masking the implementation details, and on the strengthening of current resource management systems, in particular, the communication between computational nodes. In the case of heterogeneous architectures, I focused on the already consolidated and state-of-the-art compilation chain, increasing its capabilities through the use of deep learning techniques to support complex decisions currently taken by the programmer, such as the allocation of tasks on compute units.

1.1 Programming challenges for many core neuromorphic platforms

In the manycore neuromorphic platforms case, I worked on the in the context of the , to study: i) the mapping impact of neural networks on the platform, ii) how to implement a rapid reconfiguration of neural network simulations and iii) how to provide of a more flexible and generic programming model.

The SpiNNaker architecture is a general-purpose, manycore, massively parallel architecture. Since SpiNNaker is inspired by the human brain structure, this architecture tries to exploit the interconnection capabilities among the cores rather than the computational power of the single computational unit. This peculiarity of the SpiNNaker system presents some advantages, such as the low power consumption compared with the classical approach to neural network simulation through the use of standard computer architectures. For this reason, it can be potentially used for a wide range of applications requiring intensive communication between parallel computational elements [31].

Neuromorphic platforms represent an intensive research area because of their capability of efficiently simulating . The simulation of BNN, the structures composing neural tissue, is a promising methodology to gain novel insights into unclear mechanisms underlying brain functions. BNNs are usually represented during the simulations as [55] describing the behaviour of neurons by means of .

Although initially intended for brain simulations, the adoption of emerging neuromorphic architectures is also appealing in fields such as high-performance computing and robotics [52]. It has been proved that neuromorphic platforms provide better scalability than traditional multi-core architectures. Moreover, neuromorphic architectures have a native optimised support [14] so they are well suitable for classes of problems which require massive parallelism as well as the exchange of small messages. However, the tools currently available in this field miss many useful features required to support the spreading of a new neuromorphic-based

computational paradigm.

1.1.1 Configuration Time

SpiNNaker presents challenges in core resources management and external sources communication. Currently, one of the biggest issue for this platform is the time necessary to transfer data from the host server to the SpiNNaker board through a single entry point.

Typically these platforms are connected to an external host computer where configuration information is generated. A host-to-platform communication channel is used to convey this data to an input port of the platform. From here, data should be broadcasted to the computing nodes simulating the neuron models. Due to the high degree of parallelism and the complexity of the interconnect, this communication is in general time consuming and needs to be efficiently managed.

Previous approaches to the generation of the data for neural network simulation have always involved the full generation of data on the host PC and, in a second time, loaded into the simulator memory. From the programmer's perspective, this procedure has the apparent advantage that all the data are available to use, whenever they are needed. However, this also has the disadvantage that repeated patterns are transmitted without exploiting the possibility of compression.

Some advances in parallelise data generation has been made moving with the data specification execution phase on-chip. Nevertheless, in this case, the transmission of the data specifications scripts would still require a considerable amount of time.

This aspect becomes critical not only during initial network configuration time but also when several simulations have to be performed to explore network parameters in brain simulations, where configuration data must be reloaded. In a typical SNN simulation, the configuration might take even more than ten hours against a simulation runtime of few minutes [4]. On top of that, even a single simulation parameter variation would require to restart the whole configuration from scratch. Moreover, in SNN applications, it may be needed to update neuron models. For instance, the host must trigger a switch from a "learning" neuron model used during network training to a "test" neuron model during a classification task execution. This would require a communication protocol supporting a host-controlled runtime network update.

In general, these platforms would benefit from a mechanism supporting self-reconfiguration triggered remotely, to reduce the cost of communication from the external host and to better exploit their inherent parallelism. Also, implementing a cores execution flow control by the host computer, their usage as accelerators would be more effective and flexible.

Point to point communication between processors can cause severe bottlenecks when configuring the board or during the execution of some applications. To overcome the hardware limitation of the architecture, I have developed the capable of diverting point to point broadcast and synchronisation communications on the multicast network. The achievements were: i) Develop a routing key compression system capable of reducing from 1600 to about 50 the rules necessary to obtain point-to-point, broadcast and sync communications on the entire architecture allowing the MCM implementation to be feasible. ii) Develop of the an abstraction layer capable of managing Remote Procedure Calls, Memory Entities (ME) and Virtual Memory Entities (vME): managed memory area, exposing remote procedures to perform operations on them. iii) Develop of the for managing the Host to Board and Board to Board communication in the ACF exploiting the MCM. iv) Enhancement of the configuration phase of SpiNNaker architecture in the Host to Board channel, minimising the communication through channel compression using Multiple Sequence Alignment for SNN simulations, and in the Board to Board channel using ACF and the MCM broadcast communication.

1.1.2 Network Congestion

Given the complexity of the communication activity in simulated SNNs, another significant challenge is to reduce the risk of unpredictable simulation behaviour and failures in the absence of efficient exploitation of platform architectural resources. In particular, how to map neural networks into SpiNNaker computational nodes profoundly impacts the communication activity on the architecture network.

The problem I faced, concerns the mapping of a large number of light parallel tasks with intensive communication into a manycore architecture. A non-efficient communication, in the specific case of SNN execution, may impact real-time capabilities as well as the reliability of the application. Indeed, spikes can be lost due to congestion problems. In general, a possible approach to face the mapping problem is to model the tasks and their communication as a graph to be mapped over the underlying hardware architecture, represented by another graph.

Whereas in this work, I have used the cortical microcircuit application as a test case for demonstrating that an enhanced partitioning and placement system studied for the SNN topology can produce a more reliable and stable configuration for the simulation on the SpiNNaker system [96, 94].

The achievements were: i) Become familiar with the neuromorphic manycore computation paradigm and, in particular, with the SpiNNaker neuromorphic platform. Understanding the challenges and the limits of the architecture in the field of Spiking Neural Network (SNN) simulations. ii) Formalise the SNN mapping problem breaking it into two phases: Partitioning and Placement. For the Partitioning

phase, I used a multilevel k-way graph partitioning strategy capable of generating network-partitions implemented in state-of-art programs for graphs like METIS. The Placement phase, instead, has been formalised as a problem of minimisation of synaptic elongation and solved through Spectral Analysis, Integer Linear Programming, Multilevel Static Mapping (SCOTCH), and Simulated Annealing techniques. iii) I developed the as an interface to the programming environment of SpiNNaker. The tool exploits Spectral Analysis to perform SNN mapping. I made a performance comparison between GHOST and the SpiNNaker toolchain for executing SNN simulations on SpiNNaker.

1.1.3 Program model flexibility

SpiNNaker is under study for accelerating communication-intensive applications involved in computational physics and biology applications for its efficient support for inter-chip communication.

I studied how its programming model limits the usage of SpiNNaker and how to overcome these limitations. These limitations distress both for SNN simulation as well as its usage as an accelerator.

Other works have used this platform to execute general-purpose parallel computation, with positive outcomes both for scaling performances and energy efficiency. In Blin et al. [14], authors have customised the neural model and reproduced the connection graph of a page rank problem as an SNN. They show that the scalability rate of the neuromorphic platform outperforms the general purpose architectures Sugiarto et al. [90] have implemented on SpiNNaker an image processing algorithm using a task graph representation.

However, none of these two approaches has tested synchronous applications, since both of them used an adapted SNN simulated with the standard framework. Since SpiNNaker has mainly been developed to run brain simulations, it does not natively support a general-purpose programming model, like the Message Passing Interface (MPI). In my work, I explore the potential of this type of architectures to accelerate communication-intensive applications by exploiting the MPI library that I developed and optimised from scratch leveraging the SpiNNaker interconnect support. I evaluated the developed library using an N-body simulation kernel, typically used in computational biology tasks such as molecular dynamics. Results suggest that the considered neuromorphic platform with our MPI implementation is promising for tasks where communication is prevalent.

1.2 Programming challenges for heterogeneous platforms

In the context of programming models and compilers for heterogeneous architectures, despite the availability of machine-independent languages, like the *Intermediate Representation* (IR) [50], and interfaces supporting code-fragments (kernels) offloading to hardware accelerators (GPUs, DSPs and FPGA) [23, 88, 100], a consistent amount of research is still in progress for defining automatic mapping techniques aimed at improving the available computational power and avoiding the effort of manual profiling and code manipulations.

While code analysis based on deep learning methods have been developed for high-level languages, such as OpenCL [22], the potential of IR has been exploited to this purpose in recent academic papers [9, 12, 3]. LLVM representation is hardware-independent, and it is a general representation that can be reached from different high-level languages. For this reason, developing a code classifier for IR would be more generally applicable and robust. At this level, source code has undergone a preliminary optimisation pass during high-level code transformation. Various strategies have been applied to improve the classifier performance such as the removal of unnecessary elements [9], contextual flow graph analysis and immediate values usage [12] and code vectorisation [3].

In my work, I addressed this issue, designing a method able to identify, select, and encode the syntactic language elements (tokens) of source code of a kernel compiled in the LLVM - Intermediate Representation (LLVM-IR), I will refer to it as *DeepLLVM*. I used the generated sequence of tokens as input for training a to recognise which is the most appropriate architectural component for each piece of code evaluated.

I compared our approach based on LLVM-IR with the state-of-art solution based on high-level source code (e.g. OpenCL) showing that our solution produces a more accurate mapping, with the advantage of working on a layer decoupled from the code-language.

Our results show that LLVM-IR keeps the informative content needed to perform an effective classification, making possible the application of our classifier to any source code for which an LLVM compiler exists.

Moreover, I present a new method for classification of LLVM source code where I introduce the use of in code classification.

More specifically, *DeepLLVM* integrates convolutional (Conv1D), and layers to extract knowledge from syntactic language elements (tokens) of a kernel compiled in IR. I performed an extensive exploration of the hyper-parameter space composed

of both network and training. The best set was used as a reference for comparison between the CNN model and the solution proposed in [9] and [22] accounting also for the impact of kernel optimisations and token filtering strategies.

Results confirm that IR based classifiers achieve similar or better performance than OpenCL based ones, with the advantage of the generality of the IR representation. Moreover, CNN model outperforms RNN in terms of training time, classification accuracy and overall speedup.

Chapter 2

Background

2.1 Neuromorphic Architectures

Computational neuroscience aims at studying the brain functions in terms of the information processing properties of the structures that make up the nervous system. The nervous system can be studied at different levels of grains and with many methodologies [46]. At fine-grain level, the neurons can be modelled, considering atomic and electrical interactions between the molecules that constitute neurons. On the other hand, at coarse-grain, interactions between brain zones can be modelled through the simulations of neural networks able to reproduce the biological behaviours. These types of networks are called Spiking Neural Networks (SNN) or Neural Networks of Third Generation (3rd Gen. NN) [55].

2.1.1 SNN

Biological neurons collect and transmit action potentials called spikes. A spike is transmitted from a neuron (presynaptic neuron) along a wire, called "axon", to the dendrites of other neurons (postsynaptic neurons). The axon-dendrite contact is called "synapse". A neuron, when receives a spike, detects a change in the electric membrane potential, depending on the weight of that specific synapse. If the membrane voltage reaches a threshold value, the neuron fires a spike. Otherwise, the reached potential will decay over time.

Each synapse is characterized by a specific weight that influences the changes induced by the presynaptic spike in the electric membrane potential of the postsynaptic neuron. On average, the neuron spiking rate (spike/second) ranges from 10 to 100 Hz. The nervous system networks make the importance of every single neuron relatively low. This is due to its very high level of parallelism and its ability to adapt to unknown environments. Remarkable fault tolerance is provided even

after the loss of many neurons.

A neuron can be modelled as a simple processing element that integrates signals coming from its predecessors (pre-synaptic neurons) and transmits the computed value, represented as a spike, to all the neurons connected to the output (post-synaptic neurons). Nowadays, the main used approach is to model the parameters of the neurons such as synaptic currents, firing frequency and membrane potentials. SNNs make use of Ordinary Differential Equations (ODE) to describe the behaviour of biological neurons, therefore introducing the concept of time in the model. Two of the most adopted neuron models are the *Leaky Integrate and Fire (LIF)* [1] and *Izhikevich (IZK)* [40], because they are able to ensure a plausible picture of the biological behaviours with reduced computational costs.

SNN simulations, compared with in-vivo experiments, allow accurate neuron dynamics observation, exploration and validation of plausible theories regarding brain functions. Moreover, SNN simulations can reproduce the experiments with the same conditions. An SNN can be described as a graph where each node, called *Population*, is a homogeneous group of neurons sharing the same model and parameters. Whereas, each edge (*Projection*) represents the rule used to generate synaptic connections between the neurons of two *Populations*.

PyNN [25] and Nengo [11] are the most used APIs to define SNN simulations. Both of them allow the description of many neurons and synapses models (including LIF and IZK neuron models). They can be exploited in a transparent mode on different back-ends such as neuromorphic platforms or software simulators running on general-purpose workstations.

The Neuromorphic engineering aims to mimic biological networks of the nervous system developing neuromorphic simulators.

Neuromorphic simulators allow the study of the working mechanisms acting in the brain and allow the investigation of biological processes underlying neural diseases. Many research groups have developed neuromorphic simulators to develop neuromorphic applications and study brain functions [19]. At the same time, neuromorphic engineers take inspiration from biology to design brain-like systems with brain-specific features. These include extreme parallelism, adaptive responsiveness to unknown environments, fault-tolerance, and very low-power consumption [59].

Simulators can be divided into two main categories: Software Simulators and Hardware Simulators. Hardware simulators can be analogic, digital or implement both technologies. The common aim of these devices is the computation of mathematical models to describe biological neurons and synapses behaviours [55].

Hardware neuromorphic platforms are going to be pervasive in the field of computational neuroscience. This trend is confirmed by the new emerging hardware

architectures that try to emulate the human brain behaviour. These architectures have been recently developed in order to solve complex problems and to overcome the limitations of classical von-Neumann architectures. Some of the most promising examples are represented by the IBM TrueNorth chip [58], the BrainScaleS system [57], and the SpiNNaker machine [31].

There are two main approaches to neuromorphic computing—VLSI architectures: i) where neurons are modelled at transistor-level and communications are handled with connection crossbar array. ii) Custom architectures where general-purpose cores are connected to form a mesh of processors optimised for the transmission of small packets [28, 83, 103]. In the following, I provide a background on neuromorphic representative architectures.

BrainScaleS is a VLSI platform developed at the University of Heidelberg [81]. The main idea behind this project is to use above-threshold analogue circuits to physically model neuronal processes, exploiting analogy between electronic circuits and the ionic circuits in biological neurons. Analogue neurons are delivered using wafer-scale integration.

Dynap-SEL is a VLSI chip called Dynamic Asynchronous Processor Scalable and Learning that is produced with four neural processing cores which implement 256 analogical Adaptive Exponential Integrate and Fire neurons placed in a 16×16 grid with 64 programmable synapses for each neuron. In the Dynap-SEL architecture, it is available also a supplementary core 64 analogical neurons and 8192 plastic synapses with on-chip learning and 4096 programmable synapses [61].

Loihi is a neuromorphic processor produced by Intel [24]. It features a manycore mesh comprising 128 neuromorphic cores, three embedded x 86 processor cores and off-chip communication interfaces that extend the mesh in 4-planar directions to other chips. All logic in the chip is digital and implemented as an asynchronous bundled-data design.

SpiNNaker, the Spiking Neural Network Architecture is a real-time neural network simulator following an event-driven computational approach [31, 32]. This architecture is able to emulate neural populations and to simulate an entire Spiking Neural Network (SNN) in real-time. What sets SpiNNaker apart from all the above platforms is the fact that its architecture does not implement neurons via custom VLSI designed circuits, but it consists of a mesh of general-purpose ARM cores with a neuromorphic connectivity scheme. While the platform is designed to run SNN simulations and a software stack is provided to facilitate this purpose, in principle, the general-purpose cores can run any C program compiled for ARM.

The communication inside the neural networks is usually represented using an asynchronous event-driven model. The Address Event Representation (AER) protocol

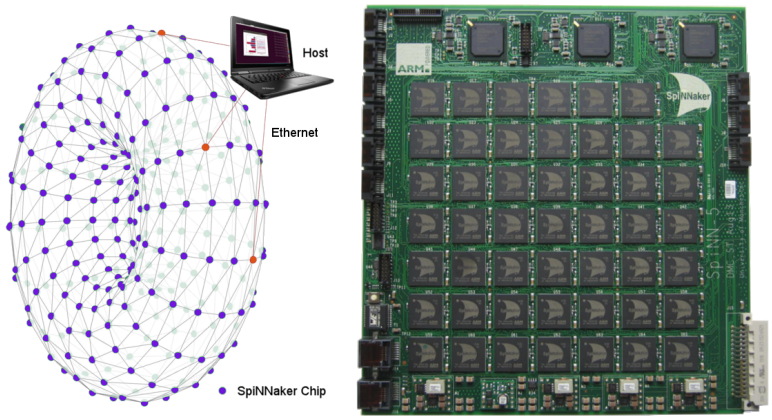


Figure 2.1: SpiNNaker board with 48 multi-core chips connected in a toroidal-shaped triangular mesh.

is implemented to distribute spikes across the system [15] to reproduce this communication procedure in neuromorphic platforms,

AER defines the transmission rules across the network: i) The packet containing the address of the neuron that generated the spike is sent through the communication links without information regarding target neurons; ii) If a post-synaptic neuron recognizes a specific address (belonging to one of the pre-synaptic neurons), it takes in charge the packet and starts its elaboration. This is the most adopted method in state-of-the-art technologies [62].

Using this SNN description system, Van Albada et al. [97] designed an SNN implementing the cell-type specific cortical microcircuit (CM) model created by Potjans et al. [70]. Then they simulated this SNN on a neuromorphic multi-chip manycore platform called SpiNNaker [32] using the standard application partitioning and placement system for setting up the simulation on the board.

2.1.2 SpiNNaker

The Spiking Neural Network Architecture (SpiNNaker) has been developed in 2006 by Furber et al. [31] at The University of Manchester in the context of The Human Brain Project (HBP). The idea was to create a massively parallel manycore system Globally Asynchronous Locally Synchronous (GALS) inspired by the mammalian brain, trying to emulate its connectivity and simulating a large scale SNNs in real-time with a low power impact [89]. The whole system is composed of 1 036 800 general-purpose CPU and 7 TBs of RAM distributed throughout 1 200 SpiNNaker boards each one containing 864 ARM processors [32].

This system mimics the features of a biological neural network through the implementation of several features:

- Native parallelism: Each biological neuron is a fundamental computational element within a massively parallel system. Likewise, SpiNNaker uses parallel computation.
- Spiking communications: In biology, neurons communicate through spikes. The SpiNNaker architecture uses source-based Address Event Representation (AER) packets to transmit the equivalent of neural signals (i.e. action potentials) [71]. Each AER packet identifies the event source through an addressing scheme.
- Event-driven behaviour: Neurons are very power efficient and consume much less power than other modern hardware. The hardware is put into "idle" state until an interrupt event does not trigger an action to reduce power consumption, [42].
- Distributed memory: In biology, neurons use only local information to process incoming stimuli. The SpiNNaker architecture features a hierarchy of memories: memory local to each of the cores and an SDRAM local to each chip.

The spinnaker connectivity is built such a two-dimensional toroidal shaped triangular mesh of SpiNNaker chips Figure 2.1. Each chip is made of general-purpose ARM cores. These processors are flexible and capable of aiding the rapid evolution of neuroscience research. Each core represents the processing node, where neurons activities are simulated. The populations of neurons are described in software, and the spikes are represented as packets. These packets are propagated through the on-chip and inter-chip communication links via routers.

Platform configuration requires several software modules for converting the SNNs simulations, designed by neuroscientists, into executable and configuration files to set-up the board [76, 43].

Further details about the SpiNNaker architecture can be found in [31, 74]. Supporting tools are described in [17] while Rast et al. in [72, 73] describe spinnaker communication protocols and systems.

In the following sections you will find detailed information about the hardware and software structure required to present the contributions of next chapters.

2.1.3 Hardware

The basic building block of the SpiNNaker system is the Spin5, a Printed Circuit Board (PCB) that hosts 48 SpiNNaker chips. The SpiNNaker chips are placed to form a toroidal shaped triangular mesh where each chip is connected to six nearby chips Figure 2.2. Each Spin5 has a Board Management Processor (BMP), two

100 Mbit/s Ethernet interfaces for linking the board with external devices and three FPGA for linking the board with others six Spin5 PCBs. The first ethernet interface is directly connected to a chip (the Root chip), the second ethernet interface is used to communicate with the BMP.

Chip

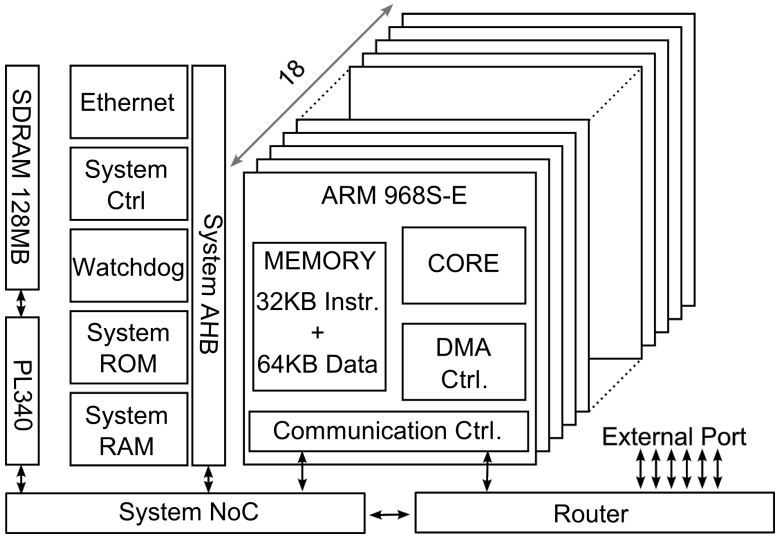


Figure 2.2: The SpiNNaker chip architecture.

The SpiNNaker chip is a System-in-Package (SiP). The package is composed of two VLSI die, the first one host a Low-Power Double-Data-Rate Synchronous Dynamic Random-Access Memory (LP-DDR-SDRAM, aka. SDRAM) with 128 MiB and the second one a System-on-Chip (SoC) [30].

The SoC is physically connected to the SDRAM via a PL340 interface and hosts [72]:

- 32 KiB Static RAM (aka. System RAM or SysRAM)
- 32 KiB ROM (which contains the software necessary for the machine boot-strap, aka. System ROM or SysROM)
- 18 ARM processors
- Ethernet interface
- Router, custom designed for manage six external links to other SpiNNaker chips (inter-chip communication) and eighteen internal links to the hosted processors (intra-chip communication)

In the SpiNNaker SoC, the eighteen processors are connected through the System NoC to the custom router and to the others resources: System ROM, System Controller, System RAM, SDRAM and the ethernet physical interface (PHY) (Figure 2.2).

At the board start-up, each SpiNNaker chip runs a hardware check at the end of which, if some component is not responding, it is disabled. After the check-phase, each SpiNNaker chip selects a processor to be used as for managing the entire node. All the other seventeen cores are available for the execution of user-defined applications [43]. This last class of cores are called .

More schematically in a SpiNNaker chip, the cores are organised as follows: i) One Monitor Processor (MP), it executes the *SC&MP* program, which is a sort of monitor which performs whole chip management tasks, it performs operations of memory management and acting as a packet manager, able to receive and transmit packet traffic to the other cores. ii) One Spare Processor (SP), reserved for manufacturing yield-enhancement purposes. iii) Sixteen Application Processors (APs), used for application processing.

During the boot procedure, called by the host computer, all chips set theirs coordinates respect to the ethernet enabled chip (0,0), initialise the router, and selects the core for the monitor processor role [84]. The ethernet enabled chip (Root Chip) is the only one that is physically connected to the 100 Mbit Ethernet interface of the PCB. During this phase, SC&MP is loaded on all MPs, and the is loaded into all others APs. The user applications can be executed on top of SARK [84]. The host computer can now communicate with all processors using the Monitor Processor of the Root Chip.

Core

Each processor of the eighteen available in a SpiNNaker chip, is an ARM 968 a 32-bit RISC ARM processor with ARMv5TE microarchitecture. The operative clock frequency of the processor is 200 MHz.

The processor has its own DMA controller and two private tightly-coupled memories (TCM) one for instructions (ITCM) and one for data (DTCM). In this way the processor can access four memory areas: i) a 96 KiB . It is divided into ITCM containing instructions (32 KiB) and DTCM containing application data (64 KiB). ii) The 32 KiB System RAM integrated into the chip and shared between all the cores. iii) The 32 KiB System ROM shared between all processors that contains the bootstrap software. iv) The 128 MiB SDRAM shared between all cores.

The processor lack floating-point unit, since the differential equations of the neural models, are in the domains of real numbers a programmer needs to use a mechanism of value rescaling that allows working with only fixed-point numbers.

Communications

The SpiNNaker chips have six bidirectional links that allow connections to form a triangular lattice which is folded onto the surface of a toroid. A custom Router incorporated in each chip manages inter-chip and intra-chip communication [31]. This structure of link allows each router to communicate with its own six neighbours chips [53].

Three different packet routing mechanisms can be identified:

- The communication among SpiNNaker chips inside the same PCB, mediated only by the routers Figure 2.3.c
- The communication among SpiNNaker chips inside different PCB through the six SATA-like links, mediated by routers and three FPGAs mounted on each PCB. In this way, a PCB can be connected with other six PCB.
- The communication among SpiNNaker chips and Host Computer via Ethernet interface, mediated by the routers, MPs and Root Chip of the involved PCB. The off-board communication between the Host Computer and the SpiNNaker cores is supported using a 100 Mbit Ethernet interface physically connected to the Root Chip. The Root Chip that receives UDP packets from the Host is in charge of forwarding the UDP payload within the network of SpiNNaker chips.

The kernel of the interconnection among all cores of all SpiNNaker chips is the Router (Figure 2.3), specifically designed to deliver packets as fast as possible (0.1 µs per hop) [17].

The particular design of the Router, despite limitations on the synchronous transmission of packets [94, 96] allows dispatching the incoming packets, coming both from external links and from internal cores, to multiple outputs [30].

The length of these packets can be up to 72bit. The packet is divided into 8bit of packet header 32bit of protocol header and 32bit of optional payload. The Router operates with four types of packets:

- are used for reaching many cores across the board. They are widely used during neural simulations for spreading neural potentials to multiple destinations (emulating synapses potential transmission). These packets are routed using a routing table of 1024 entries, stored in a ternary CAM with three values per entry: routing key, routing mask and routing rule. The protocol header of an incoming multicast packet is compared with all table entries. The matching operation is performed concurrently to improve performances the packet routing key is filtered with the routing mask and then compared with the routing key of each entry in the routing table. In case of a match,

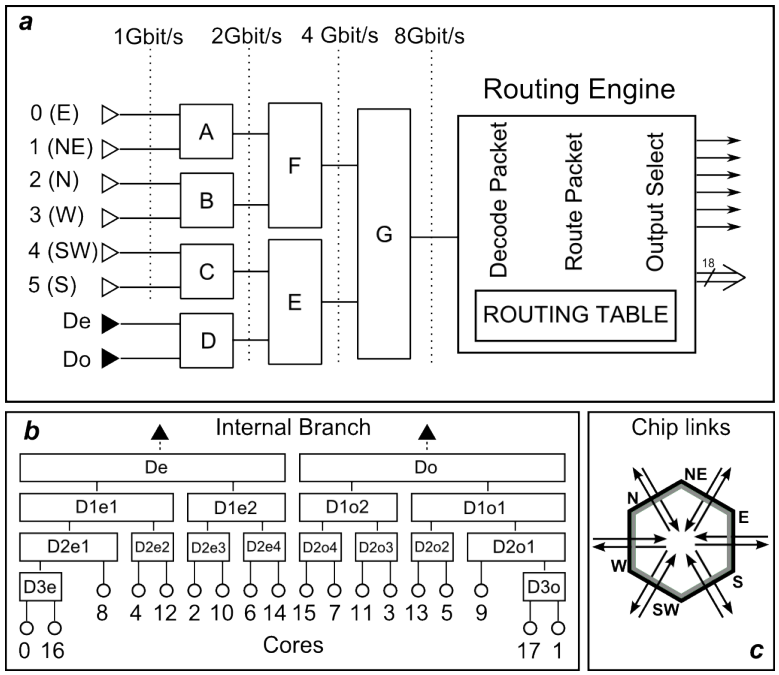


Figure 2.3: SpiNNAker Router details: a) the principal branch that merge the incoming packets (external and internal) to be provided on the routing engine input; b) the multiplexer tree that connect the internal cores to the principal branch; c) the six external link of the chip.

the Router sends the packet to the destinations specified in the associated routing rule. This type of packet can transport a payload of 32bits.

- packets are used for reaching an exact core of the board uniquely identified by the coordinates of the belonging chip and its virtual processor ID (a number from 0 to 17). These packets are routed using a dedicate routing table. For each possible destination coordinates, there is a 3-bit entry which is decoded and used to recognise if the packet should be sent over an external link. If the destination is within the local chip, the packet is always delivered to the monitor processor. It is in charge to forward the payload to the destination core. This type of packet transports a payload of 32bits.
- packets are used for initialising the board and for implementing a keep-alive mechanism useful for understanding if there are broken links and, in this case, calculate a different path. This packet can only be managed by MPs and can be routed only to MPs of neighbours chips. Moreover, during the board configuration phase, they are used for loading the applications in the cores using a flood-fill mechanism. This type of packet can transport a payload of 32bits.

- packets are used for reaching a fixed destination, by default, the chip attached to the ethernet controller. The advantage of this type of packet is that it provides 64 bits of payload.

An emergency routing procedure is available to restore the connection when an output link is stuck due to congestion or hardware failure and increase the router reliability. Moreover, when an MC packet is not recognized in the internal routing table, it is routed by default on a predetermined output link to optimise the multicast propagation

Payloads higher than 32 bit need the usage of SC&MP API. The MPs provide a higher level of abstraction that simplifies the usage of chip interconnection. The can be used to manage communication between processors up to 256 Bytes [32]. The Monitor Processors act as a middleware between the SDP protocol and the on-board network. A Monitor Processor that receives an SDP packet splits the whole frame into 32-bit fragments to be delivered in the internal network through the PP packets.

2.1.4 Software

SpiNNaker usage is made possible thanks to a software library. The software library can be split into two main groups:

- PyNN, a Python library used to describe and manage SNN simulations without any particular knowledge of the board
- Host-side software, high level software written in Python
- Board-side software, low level software written in C and Assembly

The SNN simulation is managed in the host computer, where SNNs are described using PyNN [25]. PyNN is a domain-specific Python library developed to define SNN simulations and allows to use many neural and synapses models. It can be used, in a transparent mode, to manage simulations on different backends such as neuromorphic platforms or general-purpose workstations. The SNN simulation is configured on the SpiNNaker system using a PyNN backend called *sPyNNaker* [76]. The sPyNNaker library translates the SNN description into configuration files to be sent to the SpiNNaker board [96, 10]. During the execution of SNN simulations on the SpiNNaker Machine, spikes are represented as MC packets and transmitted through the network using the routers of the SpiNNaker chips [62].

Generally, to execute an application on SpiNNaker, like the Spiking Neural Network (SNN) simulations, a programmer needs to use a set of high-level software modules running on the host computer [32, 96].

Host-side software

The host-side software, which is composed of several modules, is used to start simulations and to map them on the SpiNNaker system. The modules used for SNN simulations are:

- sPyNNaker: it contains the front-end specifications and implementation for the PyNN API. This module is a wrapper of the PyNN simulator and is responsible for translating the high-level description into populations of neurons to be loaded onto the SpiNNaker system. This module is independent of the PyNN version, there are two additional modules called sPyNNaker7 and sPyNNaker8 that provide support to specific versions of the PyNN tool (0.7 and 0.8+ respectively)
- SpiNNMachine: it is a Python abstraction of a SpiNNaker machine. Its functionality is to create a representation of the current state of the allocated SpiNNaker machine in terms of chips, cores, routable links, available routing entries and available SDRAM
- SpiNNMan: this module is used to communicate with a SpiNNaker board, sending and receiving packets (using the UDP protocol) through the Transceiver class which is its main component. The SpiNNMan module allows to get the state of the machine, boot it with a specific version of the software, load application binaries and access the SDRAMs of the single chips
- PACMAN: it is in charge of the *Partitioning and Placement* operation which is performed by creating a graph representing the Application that will be executed (called Application Graph) and to partition it in order to fit the SpiNNaker system (performing a correct distribution on the available resources and generating the routing information to implement the communication between vertices, obtaining the so-called Machine Graph). This module keeps track of the size of the available system and of the usable memory into each chip.
- Data Specification: this module is used to generate memory images, containing the Neuron Model Implementation configuration data needed during a simulation, for each SpiNNaker core involved in the simulation. The Data Specification tool is composed of two main parts: the Data Specification Generator (which is in charge of generating the Data Specification Language files, a set of text files containing the required instructions for generating the memory images) and the Data Specification Executor (which is in charge of executing the instructions contained in the generated files and of creating the memory images)
- SpiNNFrontEndCommon: this module provides common functionalities to all

others libraries

Steps in between the PyNN SNN description and its execution on the SpiNNaker board are handled by a Python package called *PArtition and Configuration MANager (PACMAN)*. This package provide utilities for SNN *Partitioning*, *Placement* and *Routing* [33].

PACMAN uses the PyNN representation of SNN composed by Populations and Projections to build the *Population graph*. This graph is elaborated following three main phases.

- During the ***Partitioning*** phase, each neuron population is divided in portions called *part-population* in order to satisfy the core constraint of maximum number-of-neurons per core. This division is made by selecting a subset of neurons without any consideration about the neuron connectivity.
- In the ***Placement*** phase, each *part-population* is assigned to a different core by means of a simple algorithm performing the sequential positioning. Once all the cores of a chip are filled, *PACMAN* starts to fill the cores of the next chip following a radial order.
- During the ***Routing*** phase the *part-populations* disposition over the board is evaluated in order to identify the best routing paths between chips. Once the best paths are identified, the generation of routing tables is performed for each chip involved in the simulation.

The partitioned and placed SNN is passed to the configuration pipeline in charge to configure the SpiNNaker board with the files generated in the host. Finally, in the *Board configuration* phase, configuration files and routing tables are sent through the Ethernet connection to each core and chip that execute the simulation.

The phase of simulation is one of the most critical in terms of execution time as well as resources management. The aim of this phase is to fill the APs memory with the configurations data needed during a simulation. In order to spread these data to each core involved in the simulation, the data specification phase is divided into two steps. The first is the that uses a set of specific DS-Commands for the generation of the configuration data that represent the SNN in the cores. The second step is the , where DS-Commands are executed, and the computed values are stored in the cores memories.

The DS-Commands allow defining a meta-language that let to perform some operations in SpiNNaker Chip memory. The Data Specification Execution can be performed on the host computer (DSE-OnHost) or directly on SpiNNaker Cores (DSE-OnBoard). In the first DSE version, it is necessary to load the generated simulation image to the SDRAM of each node. Whereas using the DSE-OnBoard, the DS-Commands are sent to the board that will interpret and execute them.

DSE-OnHost makes the host computer execute the DS-Commands for the on-host configuration of each core. The generated configurations of memory dump of each SpiNNaker Core are then saved in configuration files. Then, the files are read and sent through the Ethernet to the SpiNNaker board in the form of packets. These packets are saved in the chip SDRAM. With this DSE implementation, the computational effort of both phases of Data Specification (DSG and DSE) are performed on the host side, and the intrinsic high-parallelism and low-power consumption of the SpiNNaker system are not exploited at all. Indeed, the DSE phase is highly parallelizable, but in the DSE-OnHost the implementation is very sequential by causing a very slow configuration for the simulation.

DSE-OnBoard is a new implementation has been proposed in order to overcome some of the limitations of the DSE-OnHost. The DSE-OnBoard allows sending SCP packets to the SpiNNaker Chip in order to write the DS-Commands directly in memory. Then, in each AP a special program is loaded. When executed, this program starts to run the DS-Commands that it finds in the SDRAM and generates the data structures itself directly on-board.

This new implementation allowed to move the computational effort necessary for the creation of data structures from the Host Computer to the SpiNNaker Board. However, the serial implementation and the sequential writing procedure of the memory chunks to each SpiNNaker Chip remains a lack in terms of flexibility of the configuration mechanism.

Board-side software

The board side software is built using a cross-compiler and converted into a binary format ready to be uploaded on SpiNNaker (APLX). The Monitor Processor loads the APLX into an area of chip shared memory where the application processors can reach. The APLX is read by the APs and used to initialise the content of private processors memory, DTCM with data and ITCM with instructions.

The board-side software includes C and ARM-Assembly libraries:

- SpiNNaker Control and Monitor Program (SC&MP), executed by all Monitor Processors, manage the whole chip, the communications between application processors via the SpiNNaker Data Protocol (SDP) and the communication with external devices via the SpiNNaker Command Protocol (SCP).
- SpiNNaker Application Runtime Kernel (SARK), a minimal runtime kernel executed by all Application Processors to support the execution of the code, to manage the packet transmission and provides low-level functions to expose hardware functionalities of the SpiNNaker chip to the application level.
- SpiNNaker API (Spin1) is developed on top of SARK and implements the

Event-Driven Programming (EDP used by applications to build efficient code [17].

Regarding the monitor processors, during the bootstrap, a software called SC&MP (SpiNNaker Control & Monitor Program) is loaded. Its tasks are:

- Supervise the chip components
- Load application on APs
- Manage high-level communication protocols
- Manage communications with external devices

All the application processors, at the lowest level, run the SpiNNaker Application Runtime Kernel (SARK), which performs three main functions

- to provide a library for memory management, interrupt control and other low-level operations available to the application
- to provide mechanisms for the monitor core to communicate with application processors and for external devices to control the application and access to memory.
- to initialise the core by setting the stack and some peripherals and then to call the main procedure of the application causing its start

The applications run on top of SARK, for instance, the Neuron Model Implementation (NMI) used in SNN simulations, are composed of a set of callback functions registered on some events in order to react to software or hardware interrupt signals. Spin1 library provides a software scheduler to implement event-driven programming on SpiNNaker.

This programming model allows achieving low power consumptions, it maintains the core in *idle* state and sensitive to only interrupts, until an event (associated to a software or hardware interrupt) triggers a callback function execution. As soon as an event is presented to the processor, it wake-up, performs the associated task and then returns in *idle* state. It is possible to configure different priorities for different events in order to set a callback hierarchy and to be sure to serve the most important requests first. Depending on the role of the core there can be different types of software running on it, the most common are timer tick, packet reception, DMA transfer completion and a configurable user event (via software interrupt).

The callbacks can have different priority levels, a priority level of negative one registers the callback on the Fast Interrupt (FIQ). A priority level of zero registers the callback to a normal priority interrupt (IRQ) and priority level greater or equal to zero indicate a queueable callback.

2.1.5 Communication

The communications with external devices (like Host Computer) are made possible through the Spinnaker Datagram Protocol (SDP) and the Ethernet interface connected to the Board Root Chip (BRC), that redirects all the received SDP packets, encapsulated in UDP/IP packets, to its Monitor Processors (BRC-MP) [92].

Spinnaker Datagram Protocol

Spinnaker Datagram Protocol (SDP) is a high-level protocol provided by the SpiNNaker APIs at the user level. This protocol can be used for both internal (core-to-core) and external (core-to-device and device-to-core) communications. The current implementation of the SDP protocol is based on packets that can contain up to 256 bytes of payload.

Each packet is composed of a header field used for addressing and control purposes and a data field which contains the payload. The length of the payload must be implicit to avoid the addition of a length field.

The SDP Header has eight fields (one byte per field):

- The Flag field is used to indicate if a response is expected or not.
- The IPTag field is used for external communication purposes.
- Source Chip X, contains the X coordinate of the sender
- Source Chip Y, contains the Y coordinate of the sender
- Destination Chip X, contains the X coordinate of the receiver
- Destination Chip Y, contains the Y coordinate of the receiver
- Source Processor / Communication Port, contains the Virtual CPU identifier (5bit) and the communication port (3bit) of the sender
- Destination Processor / Communication Port, contains the Virtual CPU identifier (5bit) and the communication port (3bit) of the receiver

When SDP is used for internal communications, the Source and Destination fields in the header indicate the X and Y coordinates of the involved chips (range 0-255), the CPU fields indicate the desired core (valid range 0-17) for the specified chip, and the Port field (range 0-7) can be used to address specific functions and communication flows.

Monitor processors manage the forwarding of SDP packets to the SpiNNaker Chip Network splitting a whole SDP packet (header and payload) in fragments of 32 bits to be sent as PP packets to the desired core. For reliability purposes, SC&MP implements an acknowledgement system. The sender MP after sending sixteen

packets enters in a wait state until the receiver MP sends an acknowledgement. When all the fragments of the whole SDP packet are received, the receiver MP copies the reconstructed SDP packet into the Message Box, a portion of System RAM, and triggers an interrupt to the target core. Then, the target core can react to the interrupt and read the SDP from the Message Box. When an AP receives an SDP on communication port zero it is caught and processed by SARK, for all others communication ports is necessary to register a function callback to manage the SDP payload.

SDP can also be used for external communications. In this context the SpiNNaker Datagram Protocol is used to establish communication over the 100 Mb/s Ethernet link that connects external devices with the SpiNNaker board. To indicate that an external device is involved in the transmission and that the packet will be routed outside the SpiNNaker network, the Processor/CommunicationPort field present in the SDP Header must be set to 0xFFFF in order to invalidate the Destination or the Source fields.

When the destination field is invalidated, all packets are automatically sent to the BRC-MP that start the procedure to send the whole SDP outside the board. In this case, an SDP packet needs to be embedded into the data field of a UDP datagram. This field contains 2 bytes of padding used to align the start of the SDP packet to a 4-byte boundary, simplifying the packet manipulations inside the SpiNNaker system.

The BRC-MP use the IPTag field, a 8-bit number, as an index for a table available only in BRC, in which for each IPTag corresponds an IP address and UDP port. This mechanism is used to avoid to store the complete IP address in each SDP packet. IPTags can be permanent or transient. A permanent IPTag is created and removed manually. A transient IPTag is created when an SDP packet (with a reply-expected flag set) is received and removed when the SDP reply packet is sent back over the Ethernet interface.

The SDP bandwidth from Host to Root Processor is ~10 MiB/s and drops to 2 MiB/s from the Root Processor to another SpiNNaker Core.

Spinnaker Command Protocol

The data field of an SDP packet could be formatted to follow the specifications of the SpiNNaker Command Protocol (SCP) [93]. This protocol is used for low-level interactions with the SpiNNaker system for debugging purposes and program loading. Commonly it is used to send a command to a specific processor and to convey the response from that processor. The packet format consists of the SDP header (8 Byte) followed by a fixed-length header of 16 Byte plus a variably sized data field up to 256 Bytes. The first two bytes of an SCP header indicate the

conveyed command in case of a command packet or, otherwise, the return code after the execution of that command in case of a response packet. The following two bytes are used to detect lost packets (indicating the sequence number of the command). The three *Arg* fields can be used as 32-bit arguments or return values. The remaining bytes (up to 256) are used for data.

This protocol is used to initially control the SpiNNaker system from the host computer by sending commands to the kernel running on every active core (SC&MP on Monitor Processors and SARK on Application processors). These commands are typically used to download application programs and perform low-level functions such as getting kernel version running on the SpiNNaker processors, reading/writing the memory locations, and triggering the execution of programs.

The SCP provides four low-level instructions for accessing chip resources and extracting debugging information, such as the working state of the APs or the number of packets processed by the Router. Furthermore, SCP provides signals for controlling the application execution state and for modifying the AP memory at low-level.

The implementation of this protocol is embedded in the kernel (SARK) that must be kept as light as possible because of the limited memory resources and small computational power of the cores. Working at the kernel level and missing a direct interface to applications, SCP cannot be used effectively on the application layer.

2.2 Heterogeneus Architecture and Code Analysis

Programming models like LLVM, GLOW [77] and XLA [91] are emerging as they are suitable to support dedicated HW accelerators in heterogeneous platforms such as Apple A12, Nvidia Tegra and Xilinx Zynq embedding specialised compute-units such as ARM big.LITTLE, GPU, FPGA.

Besides the offloading mechanisms, techniques for automatic source code analysis to decide how to map computational kernels on the available accelerators are emerging.

In recent years the scientific community has tried to answer the question *"can we analyse the code like text?"* exploiting natural language translations, classifications and code modelling.

As shown in the survey of Allamanis [5], the source code maintains some properties of natural languages (it can be considered, like text, a human communication form) but it has profound differences. Some of the code properties like executability, formality and structure make it more complex to analyse than text.

Compiler designers started considering the adoption of machine learning techniques

to obtain heuristic compilers capable of learning from the data [99, 7].

Several techniques have been proposed in the literature to represent programs using a set of quantifiable properties or features compatible with the inputs of the learning module [60]. Standard machine learning algorithms typically work on fixed-length inputs, so the selected properties shall be transformed into a fixed-length vector of features (boolean, integer, or real values). Compiler researchers have designed, during the years, various forms of program features for their machine learning algorithms. These include static code structures extracted from the source code or the compiler intermediate representation [41] and dynamic profiling information obtained through run-time profiling of the program execution [20]. Compiler optimisation methods based on supervised learning have been proposed using Bayesian Networks [8], Support Vector Machines [87, 66], Decision Trees [60, 26] and Graph Kernels [65].

In 2013, D. Grewe [36] develop a workflow to translate an OpenMP program in OpenCL and to decide for each generated OpenCL kernel the most suitable compute unit between CPU and GPU. Usually, an OpenCL developer defines a kernel when it knows that the code fragment is better to be accelerated in an OpenCL device (usually a GPU). However, when the kernels are built automatically, a decision process is necessary. In [36], the authors define metrics manually to extract from the code (like the number of calculation operations or local and global memory access) to make decisions based on a probabilistic method (C4.5 decision tree classifier).

In Cummins et al. [22], the decision tree classifier had been replaced with a deep learning model based on a RNN. Using deep learning is no more necessary to extract the features manually since they are inferred automatically during the training phase. The authors show an improvement in the classification accuracy compared to the previous work presented in [36].

In my works described in the last chapter, the methodologies were developed and customised for analysing kernels implemented in OpenCL. I adopted the intermediate representation (IR) level of the LLVM compiler. LLVM is increasingly adopted in the embedded system world, because it is capable of decoupling the front-end compiler from the target architecture, in this way many optimisation steps can be performed at the IR level before generating the binary machine code. Source code features, at this intermediate level, can be exploited to perform complex compilation decisions, including allocating code fragments to architecture devices. Machine learning techniques can be applied to learn these characteristics by creating a learning model based on training code fragments.

One of the challenging issues in this field is the problem of projecting the source code in a continuous metric space. In my work [9] I use a simple method to introduce the code stream directly into the network, after a filtering phase, and let

the Embedding Layer learn the best token projection. A couple of other teams have proposed two alternative solutions. In Ben-Nun et al. [12], the authors propose Inst2Vec a system to pre-train the embedding layer analysing the Contextual Flow Graph (XFG). In Aggarwal et al. [3], they propose IR2Vec, a procedure to project an IR in a continuous metric space directly.

In this work, I explore the usage of two well known DNN structures used in the state-of-the-art natural language processing methodologies: CNN and RNN [102, 35].

RNNs are a particular type of model developed to process temporal sequences [79]. An RNN maintains an internal state, acting as a memory, that summarises the information extracted from the input sequence. In literature, a standard RNN implementation is the Long Short-Term Memory (LSTM), a network able to learn when to memorise or forgot information of the input sequence and correlate together elements at different times. RNN are used in state-of-art papers [22, 12, 3].

Convolutional Neural Networks are successfully used in the context of image recognition [51]. Behind the success of this type of network, there is the assumption of information locality in the input data. All data inside a region called "kernel" are considered correlated, and this correlation is weighed by a filter, identical for any region considered in the input. For image classification, the kernel shape has two dimensions, but this technique can also be used in temporal signals using one-dimensional kernels. The convolution operation performs an element-wise multiplication between the input data in the kernel region and the filter and accumulates the results in a single scalar. The filter moves along all input dimensions by a fixed step called stride. A convolution layer uses multiple filters to explore a different type of kernel relationship. Each filter contributes to building a channel of the output tensor. In this work CNN is for the first time introduced in a code classifier method, fully characterising its performance with an extensive exploration and comparison with LSTM. To make this comparison fair, we implemented the LSTM model in our architecture.

Chapter 3

Programming tools and middleware for manycore neuromorphic platforms

3.1 Architecture Profiling

In this chapter, I describe a top-down profiling approach that evaluates the performance of the neuromorphic platform during SNN simulations. For this purpose, several SNN configurations were designed and executed on a SpiNNaker board (Spin5).Each configuration was customized in terms of:

- SNN topology;
- Neuron model type;
- Neuron parameters;
- Neuron placement on the SpiNNaker cores.

The profiling procedure can exploit the SNN configurations to evaluate the complex problems revealed during SNN simulations. This goal is to investigate how to decrease the packets traffic circulating over the inter-chip network to improve simulation reliability and communication efficiency. The research developed over two main phases: A top-down profiling analysis to detect bottlenecks in the SpiNNaker communication system, followed by the development of an SNN Mapping algorithm inspired by the analysis performed in the first phase.

3.1.1 Profiling data

I monitored three different traffic counters available at the chip level and collected warnings at the core level to extract useful data for board profiling:

- Router Multicast Local counter traces the number of packets sent by the internal cores to the router and correctly propagated;
- Router Multicast External counter considers the packets incoming to the router from the external port and correctly retransmitted;
- Router Multicast Dumped counter is incremented when a processed packet, ready to be transmitted, is dropped for some reasons. Usually, the principal cause of this event is a busy state of receiver chip.
- Warning Output Queue Full (WOQF) indicates the amount of internally generated packets that cannot be transmitted to the router. These events can increment the computational loading of those cores that have to retransmit packets, sometimes leading to the saturation of queues and consequent packet missing or packet delay;
- Physical to Virtual core ID conversion list allows identifying the physical core position into the chip where the simulated populations are executed. This information becomes essential, during the simulations, to investigate the core/router transmission channel limits and the related maximum core load.

In order to evaluate SpiNNaker performances on real SNNs I executed the simulation of the *Cortical Microcircuit (CM)* proposed by Potjans et al. [70]. This network represents the four layers constituting $1mm^2$ of human brain cortex (L23, L4, L5, and L6). Each layer consists of *inhibitory* and *excitatory* neuron populations modelled through the setting of specific parameters in the IF neuron model. Excitatory populations have positive synaptic weight while inhibitory neuron synapses are negative. The network represented in Figure 3.1 is described in PyNN [13]. It is composed by 77 k neurons, grouped in eight populations, and about $3*10^8$ synapses. Special source populations (*SRCPops*) are used to generate spikes with a Poisson probabilistic process. These *SRCPops* are connected to each IF population of the *CM* to simulate the background activity of adjacent areas.

During the analysis, the network has been reduced in terms of neurons and synapses number to satisfy execution time and resources availability constraints. Adopted scaling factors are in the range from 1% to 20% both for neurons and synapses.

The simulation is monitored and the spikes are detected using the standard method, that consists of the interrogation of each core running a part-population.

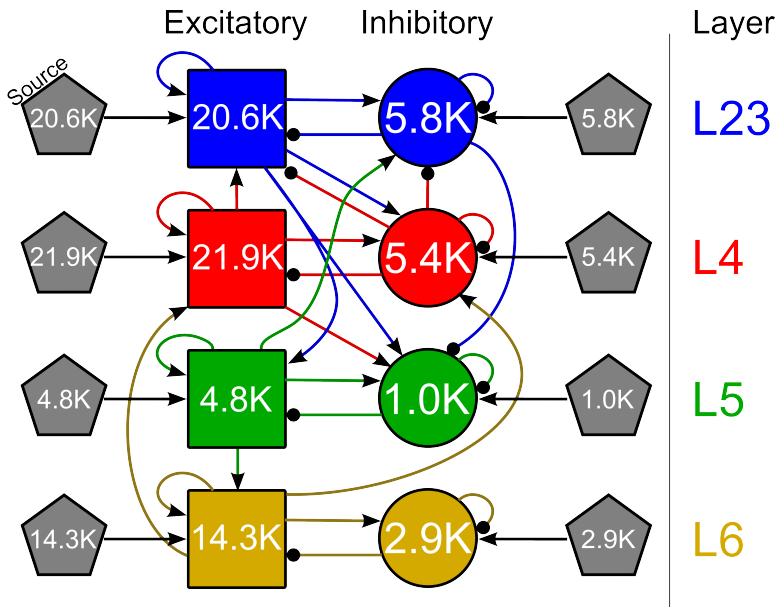


Figure 3.1: Graph representation of the SNN populations used to simulate the Cortical Microcircuit behaviour. The four layers are represented with different colors, the square represents Excitatory populations while Inhibitory are drawn as circle. The hexagon in grey stand for spike sources.

Three unexpected behaviours were pointed out during the CM simulations on the SpiNNaker board for some network scaling factors:

- The simulation does not begin if the PACMAN generated part-populations do not respect the physical core constraints (DTCM saturation).
- The simulation starts but some cores enter in Error state, and consequently the simulation is aborted (Network congestion and communication buffer saturation)
- The produced results are biologically plausible, but some spikes are missed (Network congestion)

The simulated network is evolving based on the circulating spikes. Being the impact of the missed packets not predictable a-priori it is fundamental to avoid these events that can lead to reliability problems.

3.1.2 Method

A significant amount of research has been done to highlight the capability of simulating large SNNs on neuromorphic platforms such as SpiNNaker [75, 85]. However the behaviour of the network is generally evaluated from a biological point of view

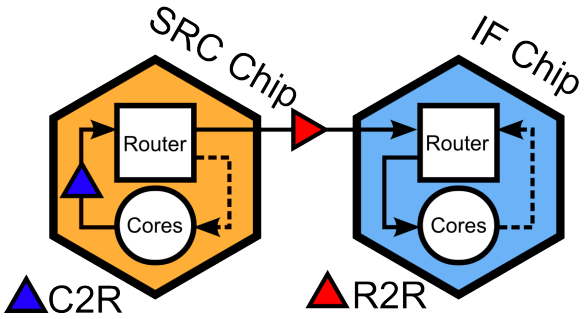


Figure 3.2: The Base Configuration Network (BC): in orange are described the *SRC Chip* while in blue the *IF Chip*. The white circle represents the cores of chip.

without considering hardware faults or packets missing. Indeed, is well accepted that the SNN simulations are relatively uninfluenced by system variations and imperfections [63].

In order to execute an accurate profiling, I designed a customised SNN able to stimulate the critical behaviours of SpiNNaker. This customized SNN is flexible enough to be used as the basic component for the design of complex use cases. One of those behaviours arises when a large amount of packets are transmitted in a single link at the same time in both directions with the consequent loss of packets.

The Base Configuration is build using two populations placed on two different chips (Figure 3.2). The first population called *Spike Source* (*SRCPop*) is used to send spikes to a connected target population following a predefined time vector. The second population, *IFPop*, is composed by *Integrate and Fire* neurons and connected one-to-one to the *SRCPop*.

The behaviour of both populations is deterministic, since the *IFPop* parameters have been set to generate a new spike when a spike from the *SRCPop* is received. During the simulation, spikes generated by the *IFPop* are stored and counted in order to be compared with the number of packets generated in those cores running the *SRCPop*.

At the end of the simulation, all cores are queried to provide the information described in subsection 3.1.1 (i.e. number of warnings and dropped packets).

The Base Configuration can be parametrised at three levels: i) The population size, responsible for the modulation of the number-of-cores used in the analysis, and the consequent number-of-packets circulating on the segments of network under test; ii) The max number-of-neurons that can be simulated over a single core; iii) The exact location of chips and cores running the two populations.

Using these parameters, several customised configurations can be build to force the overload of specific communication segments providing useful information about

the traffic sustainability. The Base Configuration allowed us to highlight a critical problem represented by the loss of packets. This occurs when all the neurons of *SRCPop* fire together and generate a huge amount of traffic over the lines of the router under investigation.

In particular, I investigated two classes of traffics: The *Core to Router* traffic generated when packets come from the Cores to the Router of a chip, and the *Router to Router* traffic generated when packets are transmitted from a Router to another Router through one or more chips.

Core to Router traffic analysis

The *Core to Router (C2R)* traffic analysis is exploited to study the behaviour of the Router overloading its first multiplexer branch with packets coming from internal cores (Figure 2.3.b).

The *SRCPop* are manually placed in selected cores and connected to a *IFPop* located in another chip. Many placement configurations have been tested in order to identify those able to avoid packet conflicts when the maximum traffic is reached.

In order to simulate a huge number of packets accessing the first internal layer of the router I adopted the Basic Configuration (in Figure 3.2) letting the neurons belonging to *SRCPop* to fire all together at the same time.

Moreover, in order to show a counter example where an additional software component can solve the detected problem, the same analysis has been performed using two *SRCPop* models. The first model exploits a software buffer to store the untransmitted packets that have to be re-introduced into the router, whereas in the second model this buffer is not used.

In Figure 3.3 are reported four configurations (from E1 to E4) designed to show the C2R traffic response of the board with different load scenarios.

In the experiment *E 1* the two populations of 4096 neurons grouped in 16 part-population are placed on two different chips. The *SRC* neurons models without retransmission buffer are used. The neurons of *SRCPop* fire at the simulation time of 100 ms and send spikes to the connected *IF* neurons. When the *IF* neurons receive a spike they generate a new packet and store the event. At the end of the simulation these events are counted and their occurrence compared with the number of packets conflicts collected in the *SRCPop* cores.

The *SRCPop* modelled without SW buffer lead to a correlation one-to-one between the number of packets conflicts and the number of packets missed in the *IFPops*. Missing packets are dropped by the *SRCPop* and do not produce an increment of the *Router Multicast Dumped* since the packets do not reach the router.

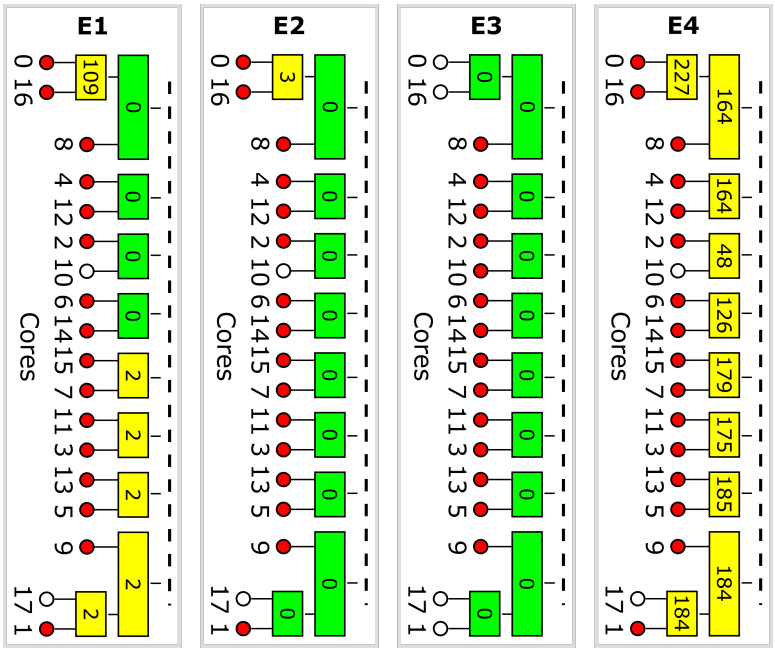


Figure 3.3: The four experiments are represented in the router internal branch of chip executing the SRCPop. Red circles identify cores running simulation while in the connection nodes (in yellow or green) are reported the number of conflicts detected. In the experiment E 1 256 SRC neurons without retransmission buffer are placed in each of the 16 cores. Experiment E 2 is executed using only 50 neurons per core. Experiment E 3 run 256 neurons per core on 12 cores. Experiment E 4 run 256 neurons per core with retransmission buffer on 16 cores.

In Figure 3.3.E1 are reported the configuration and the results of experiment E1. It can be observed that cores connected to the same first layer of the router internal branch (yellow and green rectangles) miss an equal amount of packet between each other; 109 packets for cores 0 and 16, while 2 packets for all the odd cores.

The first hypothesis assumes that the observed relationship is caused by bandwidth limitations on the internal router tree (Figure 2.3.b) even if the bandwidth reported on the data-sheet (1250 packets/ms per core) is sufficient to support a communication rate of 256 packets/ms per core [29], that produce an overall traffic on the router of 4096 packets/ms.

Two configurations are then executed to evaluate the reasonableness of this bandwidth limitation hypothesis.

The configuration E2 (Figure 3.3.E2) reduces the number of neurons per core from 256 to 50 in order to decrease the C2R traffic on the router to 800 packets/ms.

The configuration E3 (Figure 3.3.E3) avoids concurrent packets in the first layer

of router (where the majority of packet conflicts are detected), applying a delay of 1 ms to the cores 0-16 and 1-17. This configuration generates a C2R traffic on the router equal to 3584 packets/ms.

Results in Figure 3.3.E2 show that even in the configuration E2 some packets are dropped in the first router layer shared by cores 0 and 16. Instead, in configuration E3 all the 3584 packets were simultaneously transmitted without losses. These results highlight that conflicts are generated in the first internal router layer and are related to the *SRCPops* placement on the cores, disproving the hypothesis of router bandwidth limitation.

Finally, as last analysis, the first configuration is re-executed using the *SRC* neuron model with retransmission buffer. SRC models that make use of this buffer are able to store the conflicting packets and re-inject them as soon as possible. This re-injection system allow the correct transmission of all the 4096 generated packets.

However, during this experiment I detected an higher number of conflicts with respect to the unbuffered solution (Figure 3.3.E4). An average of 151 conflicts per core for the buffered *SRCPops* versus 13 conflicts per core for the unbuffered version. All neuron models implements this technique and for this motivation it is difficult to lose internally generated packets. However, in case of congested configurations or for highly synchronous applications this solution can be time spending. Indeed, supplementary computational load is required from cores to sustain the re-transmission operations. Moreover, the adoption of these neuron models can cause premature termination of simulations due to the accumulation of delays by cores that are busy to retransmit packets.

Router to Router traffic analysis

The *Router to Router (R2R)* traffic analysis is designed to investigate traffic configurations that cause dropped packets in the inter-chip network.

The identification of such configurations is fundamental to define reliable rules about traffic fluxes that can be used by the SNN-PP software to avoid the creation of hot spots.

I designed several configurations, three of which proposed in Figure 3.4, using multiple instances of the *Basic Configuration* (Figure 3.2) to simulate traffic peaks on routers and links. In these configurations, the *SRCPops* are placed on different chips, connected to *IFpops* and configured to fire all together at the same time.

In accordance with the schemes proposed in Figure 3.4:

- The *SRCPops* and *IFpops* chips are represented as orange and blue hexagon.

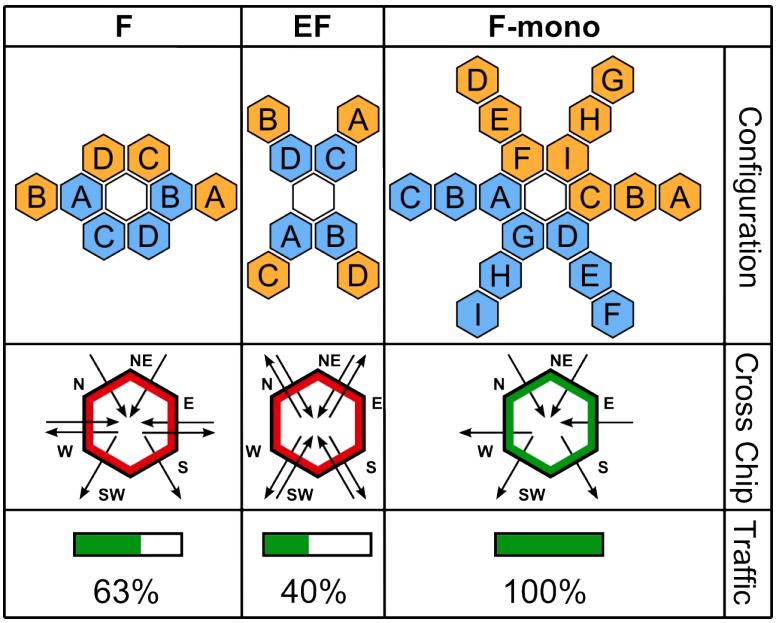


Figure 3.4: Three configurations investigated for the R2R traffic analysis. Configurations have three main characteristics: i) the orange and blue hexagon represent the *SRCPops* and *IFpops*; ii) *SRCPops* and the connected *IFpops* are placed symmetrically in relation to the chip under investigation and marked with the same letter; iii) the hot-spot chip is represented as the white central hexagon. In the traffic section is reported the percentage of packets reaching the destinations.

- *SRCPops* and the connected *IFpops* are placed symmetrically in relation to the chip under investigation, *Cross Chip*, and marked with the same letter
- The *Cross Chip* is represented as the white central hexagon;

The three selected configurations allow us to make some considerations about traffic fluxes responsible for the reduction in system reliability during the simulations.

In the first configuration called *F* (Figure 3.4.F) the *SRCPops* are placed to transmit packets through 4 ports of the *Cross Chip*.

Two out of these four ports are used both as inputs and outputs: The port East(0) of the chip under investigation gets input traffic from *SRCPops A* and outputs traffic from the *SRCPops B*. Similarly the West(3) port sustains the traffic of the same populations with reversed order. Moreover, only for the East-West traffic, the path is designed to pass the packets through a middle chip before to be introduced in the *Cross Chip*. The other two ports North-Est(1) and North(2) are used in one direction only to pass the spike generated from *SRCPops* D and C to the *IFpops* connected on the port Sud-West(4) and Sud(5).

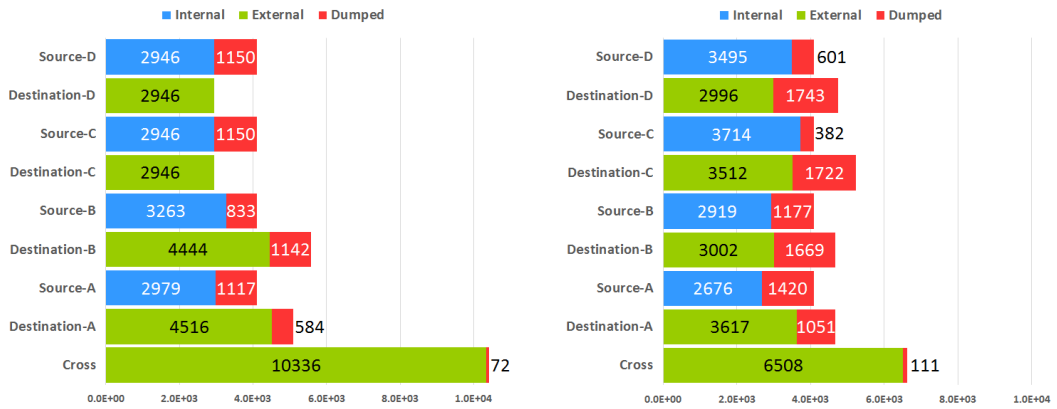


Figure 3.5: Router counter values in configuration F and EF

During this analysis an unexpected number of *Router Multicast Dumped* events with relative packets loss has been detected in all the chips involved in the traffic with the exception of destinations C and D. Figure 3.5 reports on the number of Internal, External and Dumped packets of each chip involved in the simulation. A considerable amount of packets is lost in all the routers that try to send packets through the chip under investigation Indeed even if 16 384 spikes are generated by the *SRCPops* (4 *SRCPops* * 16 cores * 256 neurons per core) the router of *Cross Chip* processed only 10 336 packets. The 6 048 lost packets can be due to the simultaneous use of East-West communication links in both directions that generate deadlock conditions in the routers involved in the transmission path.

The configuration *EF* (Figure 3.4.EF) is designed to investigate the hypothesis that simultaneous bidirectional transmission from the same port can be the cause of critical traffic situations. In this configuration all the four involved ports of the *Cross Chip* are used as input/output at the same time.

Results are reported in Figure 3.5. This configuration accounts for an higher number of lost packets with respect the F case. Indeed, only 6 508 spikes are process by *Cross* chip router, instead of the expected 16 384 spikes or the 10 336 packet processed by *F* configuration. Whereas, the majority of packets are dumped in the neighbour chips.

A punctual comparison between the amount of spikes that reached all the four chips running the *IFpops* is provided for both the discussed configuration in Figure 3.6.

In both configurations all the routers of chips running *SRCPops* get all the packets from their cores. Indeed in both figures (Figure 3.5 and Figure 3.5) the total amount of packets introduced in the routers of the chips running *SpikeSourcePops* is 4096 (Internal + Dropped).

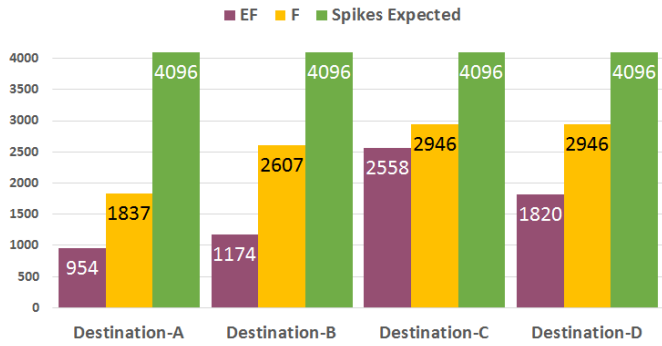


Figure 3.6: Number of spikes detected in the Destination *IFpops* for configurations *F* and *EF*.

However, because the *Cross* chip is in a busy state, that increase when the ports are used in both directions at the same time, a deadlock chain effect is backwards propagated from the busy router to the chips involved in the communication path with relative loss of packets.

A third configuration called *F-mono* has been designed (Figure 3.4.F-mono) in order to validate the hypothesis that a deadlock is more likely to occur if the links are used at the same time in both directions, and to confirm that the packets loss is not due to bandwidth problems.

In this configuration the traffic flows through the Cross chip in one direction only. Three *SRCPops* send packets through the three inputs ports of the Cross chip (Est(0), North-Est(1) and North(2)). These packets are then redirected respectively to West(3), South-West(4) and South(5) where nine *IFpops* are connected. With this configuration 36 864 spike packets (9 *SpikeSourcePops* * 16 cores * 256 neurons per core) are sent through the *Cross* chip without any loss of packets.

The use of retransmission buffer for the simulation of configurations *F* and *EF* determined a huge amount of conflicts, on average 6 conflicts per packet are generated. Indeed the transmission of 4096 packets in the first internal branch generated about 30k conflict warnings as reported in Figure 3.7.

Furthermore, the number of links simultaneously accessed as input/output is related to the number of detected *WOQF*, as reported in Figure 3.7 where more WOQF are detected when four bidirectional links are used. Indeed, 108 k conflicts were detected for the four bidirectional links configuration versus 94.5 k conflicts detected in the *F* configuration.

These results demonstrate that the simultaneous communication involving opposite router links give disadvantages even for the balancing of load per core.

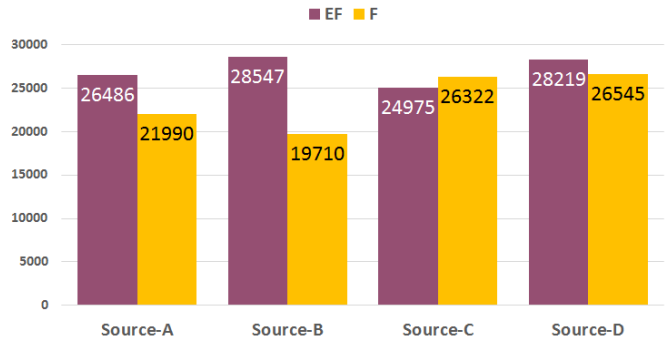


Figure 3.7: Output queue warning number detected in cores running *SRCPops* for the configurations *F* and *EF*.

3.1.3 Results

In light of considerations made in previous section, I re-executed the Cortical Microcircuit (CM) simulation by adopting customized partition and mapping methods designed to relax the requirements in order to match the board constraints. The CM simulation is executed by imposing following parameters:

- 5% of neurons (N05)
- 20% of synapses (K20)
- 100 maximum allowed neurons-per-core

This configuration was chosen because N05 produces 3 854 *IF* neurons, the same amount of *SRC* neurons and special populations used to extend the synaptic delay called *DelayExtension*. These 11 562 neurons can be simulated over 144 cores in 10 chips with reasonable configuration and simulation time.

Maximum allowed neurons-per-core is an important parameter for a reliable simulation (low number, or no packet loss). If few neurons-per-core are set, to many cores are used and a general traffic increasing is detected in the R2R link levels. While an high number of neurons-per-core on one hand can impact on the R2R traffic reduction, since less cores are used, but on the other hand cores may not be able to update the neuron dynamic state in time and the C2R traffic is increased.

The first simulation is done using the standard partition and mapping procedure applied by PACMAN while the second simulation exploits a customized partitioning and mapping algorithm, MANUAL.

In Figure 3.8 are reported the populations arrangement in the SpiNNaker board when PACMAN and the MANUAL procedures are adopted. The chips are coloured with the same colors used to represent the populations in Figure 3.1.

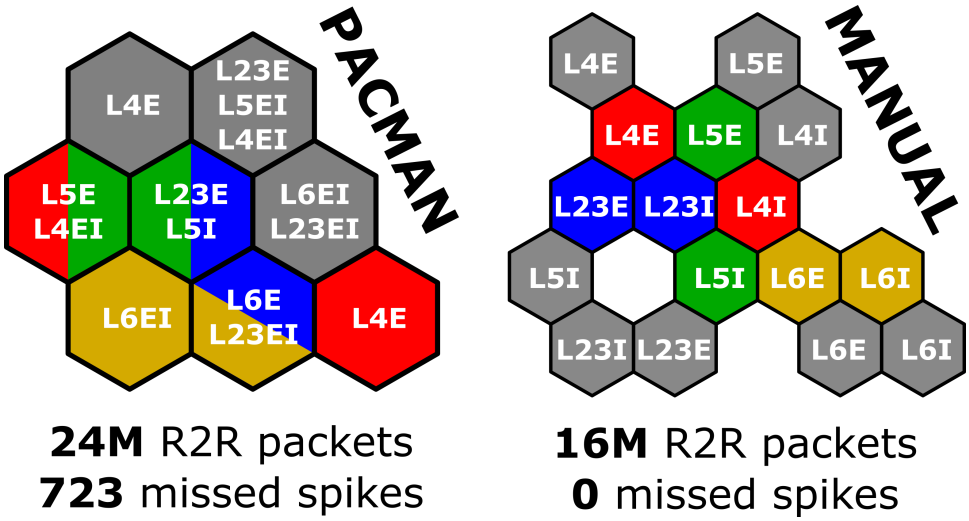


Figure 3.8: Placement of CM network: each hexagon represents a SpiNNaker chip. The colors represent the *CM* populations executed in each chip, in gray chips that contains the *SRCPops*.

Table 3.1: External, Internal and Dumped counters for the CM network with 5% of size placed with *PACMAN*.

Chip	Drop	External	Internal	Neurons	Description
0,0	276	3 773 911	4 882	847	2 IFpops
0,1	206	2 630 277	12 729	815	3 IFpops
1,0	114	5 587 441	3 334	710	3 IFpops
1,1	116	4 424 623	2 412	687	2 IFpops
1,2	0	0	2 672 252	795	1 SpikeSourcePops
2,0	0	2 692 613	9 471	795	1 IFpops
2,1	11	2 696 375	5 559 023	1 557	4 SpikeSourcePops
2,2	0	2 672 252	4 397 319	1 502	5 SpikeSourcePops
Total	723	24 477 492	12 661 422	7 708	

Moreover, the overall data traffic collected by the router counters, during one second of simulation, is reported below the configurations, together with the missed spikes and in Table 3.1.

In the first *CM* simulation, when PACMAN is adopted, a total of 723 packets have been dropped. Whereas, the R2R packets characterizing the on board traffic are about 24M. Even if the number of dropped packets can be considered very low with respect to the overall number of circulating packets, it is a good practice to ensure the correct transmission in order to prevent unreliable simulation results.

In the second *CM* simulation, the populations are mapped on the board applying the MANUAL algorithm. On the right side of Figure 3.8 it is shown this positioning where a full chip is used to run a single population. In order to optimise the packets flow among the chips, *SRCPops* are executed in the perimeter chips while other populations are placed in the middle. For populations L23E, L23I and L5I the respective *SRCPops* are distributed according to the scheme identified in the F-mono configuration with respect to the chip (2,2) avoiding bidirectional transmission on the links.

The use of MANUAL mapping procedure, that prevented sub-optimal configurations, is useful to reduce the number of R2R packets and to eliminate the dropping events. Indeed this customized procedure produces a reduction of 33% of the number of R2R packets, from 24 M to 16 M.

3.1.4 Final Remarks

In this chapter I proposed a methodology for profiling densely interconnected neuromorphic multi-chip manycore platforms for real-time SNN simulations. The methodology has been used to characterise reliability issues in the SpiNNaker platform, taken as case study. More specifically, the methodology has been used to highlight the impact of neuron population mapping on the optimal platforms resource exploitation.

A simulation of a cortical microcircuit network has been evaluated with different scale factors to characterise its unreliable behaviour such as packet losses in communication links and simulation failure. The complexity of the network makes it impossible to investigate these problems using the original biological network, therefore I designed a methodology based on the usage of custom SNN configurations to unveil both local and external network traffic issues.

Exploiting the local traffic analysis I proven that one of the causes of unreliability was due to packet conflicts in the internal router tree related to traffic congestion caused by unbalanced population placement.

Exploiting the external traffic analysis, I found out that unreliability is due to simultaneous occupation of communication links by packet flows in both directions when the traffic flows through a hot-spot router of the network.

Finally, a customized mapping procedure, based on the performed characterisation, was applied to the cortical microcircuit SNN simulation on the SpiNNaker board. Results show that, thanks to an effective neuron population placement, it is possible to improve simulation reliability by decreasing the total number of packets exchanged between chips and removing the packet losses. This last experiment demonstrated that the HW limits can be in part exceeded if smart hardware aware

Table 3.2: External, Internal and Dumped counters for the CM network with 5% of size placed with *CUSTOM* method.

Chip	Drop	External	Internal	Neurons	Description
0,0	0	3 456	4 351		DelayExtension
0,1	0	3 146	0		Not Used
1,0	0	8 232	9 129		DelayExtension
1,1	0	17 740	704 983	291	Source_L23I
1,2	0	18 001	161 580	53	Source_L5I
2,0	0	2 092	0		Not Used
2,1	0	15 040	2 647 850	1 034	Source_L23E
2,2	0	3 532 713	0		CROSS_CHIP
2,3	0	2 675 945	878	1 034	L23E
3,0	0	1 264	0		Not Used
3,1	0	828	0		Not Used
3,2	0	191 753	490	53	L5I
3,3	0	727 444	1 047	291	L23I
3,4	0	3 706 900	4 260	1 095	L4E
3,5	0	0	3 681 697	1 095	Source_L4E
4,1	0	1 264	3 333 150	719	Source_L6E
4,2	0	3 364 249	828	719	L6E
4.3	0	856 929	1 919	273	L4I
4,4	0	800 847	2 040	242	L5E
5,1	0	0	493 013	147	Source_L6I
5,2	0	520 765	1 264	147	L6I
5,3	0	10 144	0		Not Used
5,4	0	6 300	830 338	273	Source_L4I
5,5	0	0	773 424	242	Source_L5E
Total	0	16 465 052	12 652 241	7 708	

configuration tools are used.

This profiling approach can be generalized in order to be used on different neuromorphic architectures to discover which are the best traffic fluxes and the optimal configurations to be applied. In the next chapter the results of these analysis will be the basis for the development of a tool for optimal partitioning and placement of neuron populations.

3.2 SNN Mapping

In this chapter, I present a methodology for mapping a task graph representing the SNN computation on a multi-chip manycore architecture with communication awareness. To achieve this target, I designed a task mapping framework capable of analysing the network of neurons to find a configuration with the goal of reducing the communication between computational nodes. The neuron-to-core mapping problem has been formalised as a problem of minimisation of synaptic elongation. Intuitively, this metric represents the cumulative distance that spikes generated by neurons running on a specific core have to travel to reach their destination core.

The framework starts by extracting a graph of independent processes from a neural network description. In the case of SNN, the direction of a communication path is also to be represented using a directed graph. On the platform side, the interconnect structure is described as a graph where nodes represent on-chip cores while edges represent physical communication links between them. In this way, I formalised a neuron-to-core mapping as a graph-matching problem solvable through the exploitation of various algorithms available in the literature. The specific formulation I devised for SNN mapping takes into account the typical organisation of these type of neural networks into neuron populations, sharing similar characteristics as well as the neuron model.

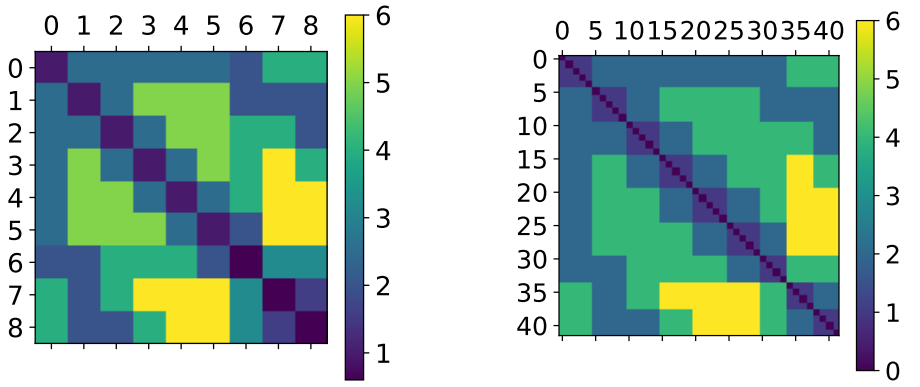
The results obtained by comparing four mapping algorithms points out and quantify the relevance of the communication direction information to achieve a better mapping if compared with non-directional algorithms.

3.2.1 Method

The SNN placement into the neuromorphic architecture can be view as an optimisation problem that involves two graphs: $\mathcal{G}_{\mathcal{N}}$ and $\mathcal{G}_{\text{CPU}}$.

A graph $\mathcal{G} = (V, E, \mathcal{W})$ is a mathematical representation for describing a set of elements V and a set of relations $E \subseteq \{(v_i, v_j) : v_i, v_j \in V\}$ among them. The elements are called *nodes* of the graph and the relations are called *edges* of the graph. An edge $e_{ij} \in E$ binds two nodes $v_i, v_j \in V$ to each other. A graph can have a $\mathcal{W} : E \to W$ function that associates an edge $e_{ij} \in E$ to a value $w_{ij} \in W$. The value $w_{ij} = \mathcal{W}(e_{ij})$ is called edge weight. A graph can be categorised according to two properties: i) If the nodes on edges form unordered pairs $e_{ij} : \{v_i, v_j\}$ the graph is said *undirected* otherwise it is said *directed* and the nodes on edges form ordered pairs $e_{ij} : (v_i, v_j)$. ii) If the weight set W is empty the graph is said *unweighted*, otherwise it is said *weighed*.

A Spiking Neural Network (SNN) can be represented using a directed and weighted graph called *neuron graph* $\mathcal{G}_{\mathcal{N}}$. In $\mathcal{G}_{\mathcal{N}}$ the nodes are the SNN neurons and the



(a) Distance matrix obtained using a coarse-grain target graph

(b) Distance matrix obtained using a fine-grain target graph

Figure 3.9: The distance matrix of a placement area using a coarse-grain and a fine-grain target graph.

edges are the SNN synapses. Taking into account a synapse $e_{ij} : (v_i, v_j)$, the neuron v_i is called pre-synaptic neuron and the neuron v_j is called post-synaptic neuron. The edge weight w_{ij} (called synaptic weight) represents the synapse contribution to injected current into the post-synaptic neuron after a stimulus received by the pre-synaptic neuron.

The neuromorphic architecture can be represented using an undirected and weighed graph, called *target graph* $\mathcal{G}_\mathcal{T}$.

The *target graph* can be more or less detailed. If the nodes of the graph are the SpiNNaker Chips, I define $\mathcal{G}_\mathcal{T}$ as *coarse-grain*. If the nodes of the graph are the SpiNNaker Processors, I define $\mathcal{G}_\mathcal{T}$ as *fine-grain*.

If the *target graph* is coarse-grain, all edges represent the inter-chip communication links. If the *target graph* is fine-grain, all edges between two processors located on the same chip have a weight of one, while all edges between two processors belonging to adjacent chips have a weight of two.

This choice is determined by the structure of the arbiter which feeds the SpiNNaker Routers. A SpiNNaker Router has two branches for introducing packets according to their origin: the 18 internal processors and the six nearby chips. It has been demonstrated in [94] that the arbiter does not correctly manage some traffic configurations coming from the external links. It was therefore decided to disadvantage all inter-chip communications with twice the weight of intra-chip communications. The Figure 3.9 shows the differences between the coarse-grain model and the fine-grain model through the distance matrices obtained from the graphs of the target nodes.

We can define the placement problem $\Pi : \mathcal{G}_{\mathcal{N}} \rightarrow \mathcal{G}_{\mathcal{T}}$ as a minimization problem (Equation 3.1).

$$\underset{f(\pi)}{\text{minimize}} \quad f : \sum_{e_{ij} \in E_{\mathcal{N}}} d(\pi(v_i), \pi(v_j)) \tag{3.1a}$$

$$\text{subject to} \quad \pi(i) = \pi(j) \rightarrow \mathcal{M}(i) = \mathcal{M}(j),\ i, j \in V_{\mathcal{N}} \tag{3.1b}$$

$$|\pi(i) = p| \leq \mathcal{S}(\mathcal{M}(i)),\ i \in V_{\mathcal{N}}, p \in V_{\mathcal{T}} \tag{3.1c}$$

Where $\pi : V_{\mathcal{N}} \rightarrow V_{\mathcal{T}}$ is the placement rule, $\mathcal{M} : V_{\mathcal{N}} \rightarrow M$ is the neuron-model association rule and, $\mathcal{S} : M \rightarrow \mathbb{N}$ is the association rule between a neuron model and the maximum number of neurons per node. The goal of the placement procedure is to minimise the *overall synaptic stretching* (Equation 3.1a) to reduce the communication along the network nodes. The *synaptic stretching* is the distance between the nodes where two adjacent neurons are placed. Two constraints affecting the placement problem: i) All neurons mapped into a target node must be of the same model (Equation 3.1b). ii) Each node can simulate only a certain number of neurons, and the quantity depends on the complexity of the neuron model (Equation 3.1c).

Problem Relaxation

A SNN is almost never described in $\mathcal{G}_{\mathcal{N}}$ form, due the high complexity in manage all neurons and synapses, but it is normally described in terms of Populations and Projections. A Population $\mathcal{P}$ is a set of neurons that share the same model and the same properties. A Projection between two Population $\mathcal{P}^{(a)}$ and $\mathcal{P}^{(b)}$ defines the rule in charge to generate a set of synapses where the pre-synaptic neurons are in $\mathcal{P}^{(a)}$ and the post-synaptic neurons are in $\mathcal{P}^{(b)}$. I will refer to the *Population-Projection graph* using the notation $\mathrm{G}_{\mathcal{P}}$.

We can eliminate the two constrains (Equation 3.1b, Equation 3.1c) redefining the problem Π working with $\mathcal{G}_{\mathcal{P}}$ and splitting each population $\mathcal{P}^{(i)}$ into a set of partial populations $\left\{\mathcal{P}_1^{(i)}, \mathcal{P}_2^{(i)}, \ldots, \mathcal{P}_z^{(i)}\right\}$. All partial populations must contains at most a number of neurons equal to the maximum number of neurons allowed to be simulated in a target node: $|\mathcal{P}_j^{(i)}| \leq n^{(i)}\ \forall j = 1, \ldots, z$, with $n^{(i)} = \mathcal{S}(\mathcal{M}(\mathcal{P}^{(i)}))$.

In this way we obtain the *partial population graph* $\mathcal{G}_{\text{pp}}$. The edges of the partial population graph are weighed and ordered. Given an edge $e_{ij} \in E_{\text{pp}}$ between two partial population, its weight w_{ij} is equal to the number of synapses shared between the neurons belonging to the two partial populations.

We can redefine (Equation 3.1) using the placement rule $\pi : V_{\text{pp}} \rightarrow V_{\mathcal{T}}$ that map a partial population into a processor (Equation 3.2).

$$\underset{f(\pi)}{\text{minimize}} \quad \sum_{e_{ij} \in E_{\text{pp}}} d(\pi(v_i), \pi(v_j)) * w_{ij} \tag{3.2a}$$

$$\text{subject to} \quad |\pi(i) = p| \leq 1, \ i \in V_{\text{pp}}, p \in V_{\mathcal{T}} \tag{3.2b}$$

In (Equation 3.2a) we modify the cost function to take into account the number of synapses shared between the target nodes. The rule in (Equation 3.2b) describes the single constraint of the problem: a target node can only contain one partial population.

Graph Partitioning

The partition problem of $\mathcal{G}_{\mathcal{P}}$ can be solved in different ways. In [96] it was treated as a problem of clustering. The provided solution was divided into three step:

- Graph expansion: $\mathcal{G}_{\mathcal{P}} \rightarrow \mathcal{G}_{\mathcal{N}}$
- Spectral embedding: $\mathcal{G}_{\mathcal{N}} \rightarrow \mathbf{L}$
- Clustering and legalisation: $\mathbf{L} \rightarrow \mathcal{G}_{\text{pp}}$.

In the first step the neuron graph $\mathcal{G}_{\mathcal{N}}$ is created by applying the synaptic generation rules defined in $\mathcal{G}_{\mathcal{P}}$ (the populations graph). In the second step, a spectral embedding procedure is applied to the neuron graph.

The spectral embedding involves the eigendecomposition of a representative matrix of the graph. In the case of $\mathcal{G}_{\mathcal{N}}$ (a directed graph) it was used a Laplacian Matrix (Equation 3.3) obtained throught a transition matrix induced by a random walk [21].

$$L = I - \frac{(\Phi^{\frac{1}{2}} P \Phi^{-\frac{1}{2}} + \Phi^{-\frac{1}{2}} P^T \Phi^{\frac{1}{2}})}{2} \tag{3.3}$$

The results of the spectral embeddings is the rapresentation of $\mathcal{G}_{\mathcal{N}}$ in the eigenspace $\mathbf{L}$, a space belonging to $\mathbb{R}^{|V_{\mathcal{N}}|}$. The neurons can be clustered in the eigenspace using the *KMeans* algorithm. After the clustering, a legalisation phase gathers in groups all neurons belonging to the same cluster and the same population. Finally, a second legalisation phase, called *fusion*, builds the partial populations putting together the nearby groups of neurons until reach the maximum number of neurons that a processor can simulate.

Other techniques of graph clustering are *Multilevel Graph Partitioning* and *Markov Cluster Algorithm* [45, 98]. These techniques, like the Spectral Clustering, was born for undirected graph and their usage should be analysed using different symmetrisation techniques if applied to a directed graph.

3.2.2 Mapping

As seen in subsection 3.2.1 our goal is placing $\mathcal{G}_\mathcal{N}$ into a set of nodes $\mathcal{G}_\mathcal{T}$. In section 3.2.1 we have relaxed the constraints of the problem separating it into two subproblems: i) Clustering $\mathcal{G}_\mathcal{N}$ (or partitioning if consider $\mathcal{G}_\mathcal{P}$ as a starting point) into the partial population graph. ii) Placement of $\mathcal{G}_{\text{pp}}$ into $\mathcal{G}_\mathcal{T}$. I have briefly described the clustering (or partitioning problem) in the section 3.2.1. In this section, I independently explore the placement problem by comparing different techniques: Naïve, Spectral Embedding, Scotch and Simulated Annealing.

Naïve Placement

The Naïve approach is the standard mapping procedure adopted in the SpiNNaker toolchain for assigning populations of neurons to be simulated on the cores available in the SpiNNaker Platform. It is a computationally light method to perform the graph placement without taking into account neither the connectivity of the source graph and the connectivity of the target graph.

The target graph was ordered following a polar coordinate system (ρ, φ) starting from a chip of choice. The radius $\rho = \max(|x|, |y|, |x-y|)$ has been calculated using the hexagonal distance. The angle $\varphi \in [0, 2\pi)$ is expressed in radians. The procedure starts to place a partial population into each processor and change the chip when all processors inside a chip are used. As the ρ increases, the sub-populations will be distributed along chips separated by a greater and greater distance.

Spectral Embedding

The procedure involves the spectral analysis of the graph and a dimension reduction procedure to obtain a planar representation of it. By doing so, the target graph can be directly superimposed on the graph of the partial populations through an *Integer Linear Programming* (ILP) problem.

This procedure requires the extraction of the first five eigenvalues, and the relative eigenvectors, from the matrix $\mathbf{L}$. The eigenvectors form a matrix $\mathbf{\Lambda}$ that represents the partial populations in a $\mathbb{R}^5$ space. We apply a non-linear dimension reduction procedure using *Sammon Mapping* obtaining a space in $\mathbb{R}^2$.

The Sammon Mapping algorithm minimise the error function in (Equation 3.4) where d_{ij} is the distance in the high-dimensional space (eigenspace) and d^*_{ij} is the distance in the low-dimensional space (placement space) [80].

$$E = \frac{1}{\sum_{i<j} d_{ij}} \sum_{i<j} \frac{(d_{ij} - d^*_{ij})^2}{d_{ij}} \tag{3.4}$$

Each chip is represented as a point (x, y) using an axial coordinate system on an hexagonal grid (the chip mesh). We superimpose the graph $\mathcal{G}_{\mathcal{T}}$ on $\mathcal{G}_{\text{pp}}$ projecting the chip mesh in the placement space (Equation 3.5).

$$\begin{pmatrix} x^* \\ y^* \end{pmatrix} = \sqrt{\frac{2A_h}{3\sqrt{3}}} \begin{pmatrix} \sqrt{3} & -\frac{\sqrt{3}}{2} \\ 0 & \frac{3}{2} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.5}$$

Where (x, y) is the chip coordinate in the hexagonal grid, and (x^*, y^*) is the chip coordinate in the placement space. The side length of the hex is used as a normalising factor and calculated using the area $A_h = \frac{A}{m}$ occupied by each chip (with m the number of chips). The normalising factor allows scaling the chip mesh making it compatible with the area A occupied by the partial populations.

In the case where the target graph is fine-grain, we need to introduce the processors in the placement space. To ensuring spatial coherence, it was decided to place them equidistant along a circumference centred on the chip coordinate. The radius is chosen in such a way that it is smaller than the distance between two processors belonging to different chips.

After projecting the points into the placement space, they are translated to centre them on the median of the points representing the partial populations. Now we can describe the placement problem using the ILP formulation (Equation 3.6).

$$\begin{aligned}
&\underset{f(\mathbf{X})}{\text{minimize}} && f : \sum_{i=1}^{n}\sum_{j=1}^{m} x_{i,j} d_{i,j} && \text{(3.6a)} \\
&\text{subject to} && \sum_{i=1}^{n} x_{i,j} \leq k \quad \forall j \in \{1, \ldots, m\} && \text{(3.6b)} \\
& && \sum_{j=1}^{m} x_{i,j} = 1 \quad \forall i \in \{1, \ldots, n\} && \text{(3.6c)}
\end{aligned}$$

Where the matrix $\mathbf{X} = (x_{ij})$, $x_{ij} \in \{0,1\}$ is the *placement matrix*. An entry $x_{ij} = 1$ means that a partial population i is mapped on a target node j. Two constraints affecting this ILP formulation: i) Each target node can host at most k partial populations (Equation 3.6b). ii) Each partial population can be associated to only one target node (Equation 3.6c).

The ILP problem was modelled using PuLP Python library and solved with *COIN-OR branch and cut* (CBC) solver.

Scotch

The Scotch mapping procedure makes use of the programs available in the homonym software suite (SCOTCH). The Dual Recursive Bipartitioning (DRB) is the primary procedure used by this tool [69]. The DRB can use a plethora of other bi-partitioning methods according to a strategy defined by the user or deducted by graph properties. The main available methods are: Gibbs-Poole-Stockmeyer [34], Fiduccia-Mattheyses [27], Greedy Graph Growing [45] and Diffusion [68].

The mapping workflow with SCOTCH plans to pre-partition the target graph through the *amk_grf* program. The *amk_grf* program take in input a graph (*grf* format) and create a target file (*tgt* format) which contains a decomposition-defined target architecture of same topology as the input graph.

Once a decomposition of the target graph has been obtained, the graph of the partial populations is placed on the target graph using the *gmap* program. The program *gmap* takes in input the partial population graph in *grf* format and the target graph in *tgt* format and performs the DRB procedure minimising the communication cost function[1]. The *gmap* output file is a mapping file (*map* format) that contains the association between the Source node and the Target node.

I had developed a Python module able to exporting a NetworkX graph to a file according to the *grf* format used by SCOTCH and capable of automating the procedures described above.

Simulated Annealing

The Simulated Annealing is a well know procedure used to find a good solution to an optimisation problem [49]. Given the problem in (Equation 3.2a), it is convenient to express the overall synaptic stretching in a matrix form and define a cost function to minimise. Given the partial population graph $\mathcal{G}_{\text{pp}}$ we build its Adjacency matrix $\mathbf{A} = (a_{ij})$ as described in (Equation 3.7).

$$a_{ij} = \begin{cases} w_{ij} & \text{if } \exists (v_i, v_j) \in E_{\text{pp}} \\ 0 & \text{otherwise} \end{cases} \qquad \forall i, j \in \{1, \dots, n\} \tag{3.7}$$

Given the target graph $\mathcal{G}_{\mathcal{T}}$ we build its distance matrix $\mathbf{D} = (d_{ij})$ where each entry d_{ij} is the lenght of the mimimum path between two target nodes cpu_i and cpu_j. The distance matrix can be build using the Floyd–Warshall algorithms or repeating Dijkstra's algorithms if $|E_{\mathcal{T}}| \ll |V_{\mathcal{T}}|^2$.

[1] The SCOTCH cost function is similar to our Synaptic Stretching

Assuming to have as many partial populations as target nodes and a placement rule $\Pi : \{v_1, \ldots, v_n\} \rightarrow \{\text{cpu}_1, \ldots, \text{cpu}_n\}$ we construct the *permutation vector* $\pi : (\Pi(v_1), \ldots, \Pi(v_n))$ and the *permutation matrix* $\mathbf{P}_\pi = (p_{ij})$ in row form (Equation 3.8).

$$p_{ij} = \begin{cases} 1 & \text{if } i = \pi_j \\ 0 & \text{otherwise} \end{cases} \quad \forall i, j \in \{1, \ldots, n\} \tag{3.8}$$

The permutation matrix is applied to $\mathbf{D}$ to permutate its rows and columns. We obtain the matrix $\mathbf{D}_\pi = \mathbf{P}_\pi \mathbf{D} \mathbf{P}_\pi$. The overall synaptic stretching can be expressed in a matrix form and used as the cost function for the simulated annealing algorithm (Equation 3.9).

$$f : \mathbf{e}^T (\mathbf{A} \odot \mathbf{D}_\pi) \mathbf{e} = \sum_{i,j} a_{ij} * d_{ij}^{(\pi)} \tag{3.9}$$

Where $\odot$ is an element-wise multiplication and $\mathbf{e}$ is a column vector whose all elements are equal to one. In the case of a fine-grain $\mathcal{G}_\mathcal{T}$, before perform the synaptic stretching evaluation, the matrix $\mathbf{A} \odot \mathbf{D}_\pi$ should be collapsed in order to aggregate the processors belonging to the same chip. I used the Simulated Annealing implementation provided in the SciPy ecosystem using the *temperature* to decide how many elements of the permutation vector π are to be swapped.

3.2.3 Results

In this section, I present the exploration experiments using the methods described in subsection 3.2.1.

I used the Cortical Microcircuit (CM) as benchmark network [70]. This network represents the connectivity of neurons inside a slice of the cerebral cortex with an area of $1\,mm^2$. The CM has been chosen because it is a rapresentative biological model with a relativly high global connectivity (5%) and natural clusters defined by the four cerebral cortex layers $\{L_{23}, L_4, L_5, L_6\}$. The CM is described in terms of Populations and Projections with two populations for each layer, for a total of 8 Populations and 64 Projections.

The network is composed of *Leaky Integrate-and-Fire* (LIF) and *Spike Source* (SRC) neuron models. The LIF neurons are models that mimic the biological neurons behaviour. The SRC neurons are simple programmable applications for outputting signals when desired. In this network, the SRC neurons are used to simulate the background activity of cortical neurons not in the model. Each SRC neuron is

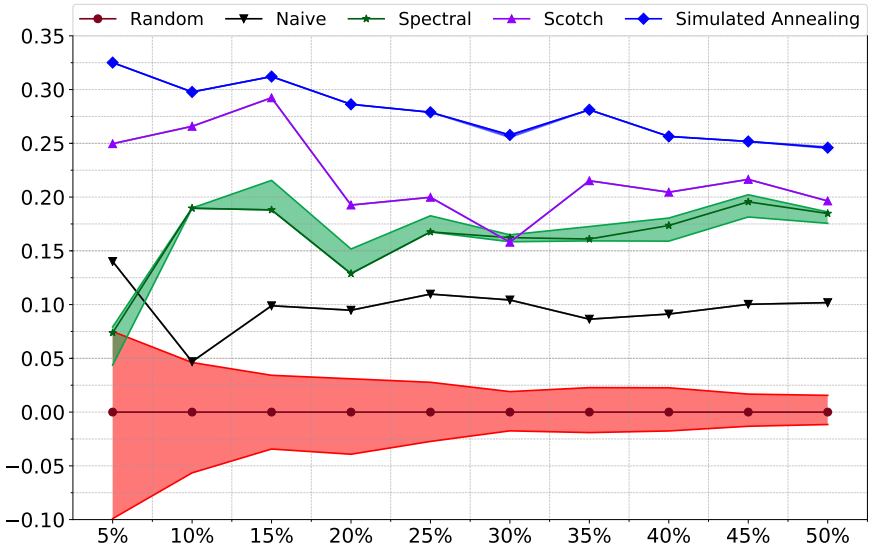


Figure 3.10: The graph represents the improvement of a mapping technique compared to the median of the results obtained with a random placement using a constraint of 1000 neurons per chip. The x-axis shows the CM scale factor. The areas represent the first and third quartile of the results obtained on 100 samples.

connected to only one LIF neuron, so they can be excluded by the $\mathcal{G}_N$ provided that processors are reserved for their execution.

The CM model has 7.72e+4 LIF neurons and 2.99e+8 synapses. A network CM_p is a down-scaled CM to a percentage p:

- $\text{CM}_{5\%}$ has 3.86e+3 neurons and 7.47e+5 synapses.
- $\text{CM}_{10\%}$ has 7.72e+3 neurons and 2.99e+6 synapses.
- $\text{CM}_{50\%}$ has 3.86e+4 neurons and 7.47e+7 synapses.

For each processor in charge of simulating a LIF partial population, we must reserve two further processors. A processor is reserved for the simulation of SRC neurons and a further processor is reserved to host a special application necessary to manage synapses with delays greater than 10 ms, as described in [94]. Taking into account a set of 16 processors belonging to the same chip, we can place five partial populations per chip for a total of a thousand neurons per chip.

For simplifying the problem we perform a sequential slicing of each population in order to obtain partial populations with at most 1 000 neurons. In this way, we can use a coarse-grain target graph where the nodes are the spinnaker chips (each chip with 5 processors and 200 LIF neurons per processor).

The experiment environment is composed of four different mapping procedures:

Naïve, Spectral, Scotch and Simulated Annealing. I had generated 5 CM networks for 10 different scale factors, from 5% to 50%, for a total of 50 networks. For each network, I applied all mapping procedures 20 times. I evaluated the performance of each mapping procedure for each scale factor, using the fitness function (Equation 3.9). As a result, I obtained a distribution of 100 different placement results concerning overall synaptic stretching.

The performance of mapping procedures are compared to the performance of random placement. The median value of the results obtained with the Random procedure is used to compute the percentage improvement of the results obtained with other techniques.

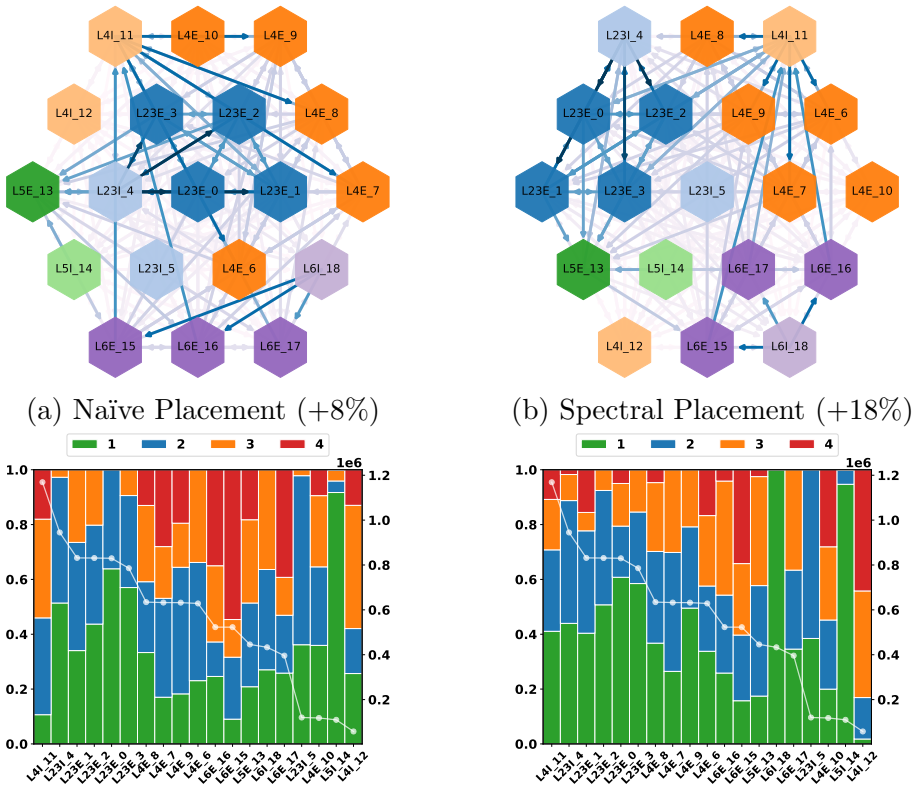
In Figure 3.10 is depicted a chart that summarize all the experiments. On the x-axis, there are the network scale factors, on the y-axis the percentage placement improvements versus random. The data series are represented by polylines of different colours representing the medians of the results set. Each polyline is drawn within an area whose extremes delimit the first and third quartile of the results set.

In Figure 3.11 and in Figure 3.12 are depicted the mapping results of a $\text{CM}_{20\%}$ into a target graph of 19 chip using the four placement techniques. Each hex represents a SpiNNaker chip connected with six neighbours. The colour of the hex area points out the belonging of the neurons, mapped on the chip, to one of the eight populations of the CM. The number of synapses shared between two partial populations is highlighted with the colour intensity of the edge that connects them. The different concentration of the connections with more synapses can be appreciated qualitatively from the Figure 3.11a to Figure 3.12b and quantitatively from the Figure 3.11c to Figure 3.12d. In Figure 3.11a can be seen how the Näive method does not consider the connectivity but place each partial population sequentially following the polar ordering of the chip. Indeed there are many connections with a large number of synapses directed towards distant chips.

This not happens in Figure 3.12b where the Simulated Annealing can localise in a defined area all partial populations with a high number of shared synapses. In Figure 3.11c and Figure 3.12d the same information can be appreciated quantitatively. The chart has a bar for each partial population. Each bar represents the overall outgoing synapses of a partial population and shows the percentage of synapses at different levels of elongation. The white line depicts the number of synapses belonging to each partial population. The partial populations are sorted in descending order according to the total number of synapses.

We can see how better methods improve the percentage of synapses at a distance of one chip (Green) and decrease the percentage of synapses at a distance of four chips (Red).

While the results of the coarse-grain model were obtained by imposing a maximum



(a) Naïve Placement (+8%)

(b) Spectral Placement (+18%)

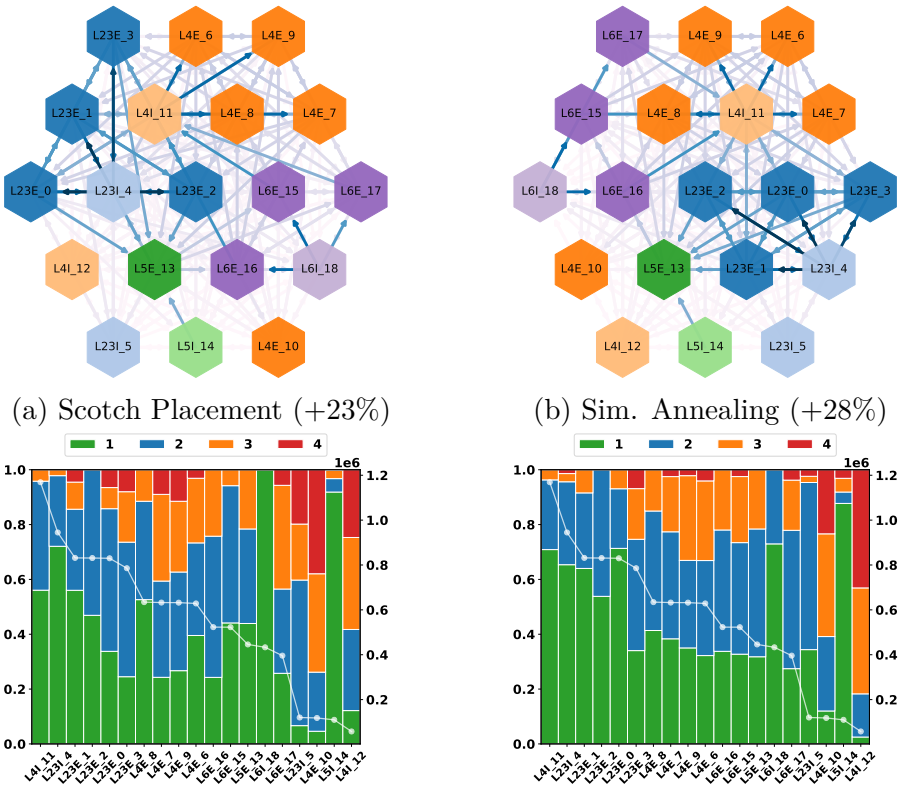
(c) Naïve Placement

(d) Spectral Placement

Figure 3.11: The figures in the first row represent the placement of the partial population graph build from a $CM_{20\%}$ with 1000 neurons per chip on 19 chip (5 processors per chip). The figures in the second-row represent for each partial population the number of synapses (white line) and the percentage of synapse stretching.

of 1000 neurons per chip belonging to the same population, the results obtained with the fine-grain model were evaluated using three different values that limit the neurons per processor: 100, 150, 200. The results obtained using 200 neurons per processor are shown in Figure 3.13a. In the Table 3.3, the results are shown in terms of processors involved. Where possible, a maximum of 5 processors per chip was used, because the network CM, in addition to the *lif* neurons considered there, makes use of other applications including a manager for synapses for high delays and a manager of external stimuli [src]. For each processor that simulates *lif* neurons, two other processors are required for a total of 15 processors per chip. In any case, in this exploration some configurations required more processors than theoretically available, so I ignored this constraint where necessary. Each chip, therefore, hosts from 500 to 1000 neurons belonging to different populations.

As the Figure 3.13 shows, the results show a profile similar to the one obtained with the coarse-grain model. However, it is noted, mainly for problems with many



(a) Scotch Placement (+23%)

(b) Sim. Annealing (+28%)

(c) Scotch Placement

(d) Sim. Annealing

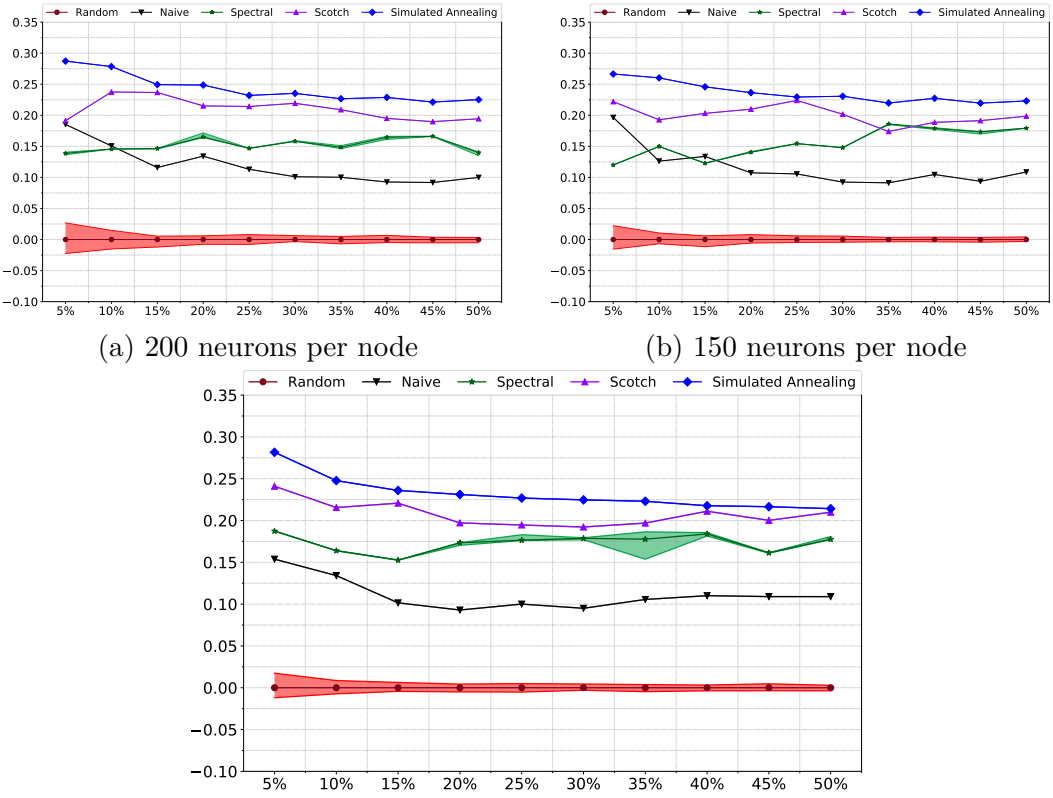
Figure 3.12: The figures in the first row represent the placement of the partial population graph build from a $CM_{20\%}$ with 1000 neurons per chip on 19 chip (5 processors per chip). The figures in the second-row represent for each partial population the number of synapses (white line) and the percentage of synapse stretching.

processors are involved, that the use of the methodology based on SCOTCH obtains results slightly inferior to Simulated Annealing. Considering the high efficiency of the solution offered by the SCOTCH suite and the simple $A + A_T$ symmetrisation necessary to use the tool, it is possible to renounce to the 2% of improvement but obtain a fast and acceptable solution.

3.2.4 Implementation

I implemented the GrapH Optimiser Spinnaker Tool (GHOST), a Python library capable of interposing between PyNN and sPyNNaker to intercept the neuron graph structure and provide an optimised SNN configuration to sPyNNaker to reduce R2R packet traffic. This software layer expose a PyNN front-end and implements an SNN mapping method based on Spectral graph analysis.

As described in section 3.2.1 SNNs can be represented through three graph layers,



(a) 200 neurons per node

(b) 150 neurons per node

(c) 100 neurons per node

Figure 3.13: The three graphs represent the improvement of a mapping technique compared to the median of the results obtained with a random placement using three different constraints for the number of neurons in a processor and fine-grain target graph. The x-axis shows the CM scale factor. The areas represent the first and third quartile of the results.

each of them useful for the execution of specific operations: i) *Population Graph* $G_{\mathcal{P}}$ is the representation where each vertex is a PyNN Population and each edge is a Projection with a connector for the synapses generation; ii) *Neuron Graph* $\mathcal{G}_{\mathcal{N}}$ is a SNN representation where each vertex is a neuron and each edge is a synapse; iii) *Part-population Graph* $\mathcal{G}_{\text{pp}}$ generated from the Neuron Graph through a clustering procedure. Each edge is a set of synapses and each vertex is a part-population with a number of neurons that satisfies the neuron per core constraint.

The main processing flow, shown in Figure 3.14, takes as input the *Population Graph* removes the *SRCPops* that will be partitioned and placed at the end of procedure, and expands $G_{\mathcal{P}}$ in order to get the *Neuron Graph*.

During the *Partitioning* phase,$\mathcal{G}_{\mathcal{N}}$ is analysed using spectral clustering techniques.

Table 3.3: The size of the fine-grain target graph in terms of number of cores and number of chips. We have always tried to keep five processors per chip using different constraints for the number of neurons per chip. Configurations marked with (*) can not be mapped on five processors per chip.

CM	Neurons per Node								
	200			150			100		
	Core	Chips	Core/Chips	Core	Chips	Core/Chips	Core	Chips	Core/Chips
5%	24	5	4.8	28	6	4.7	42	9	4.7
10%	42	9	4.7	54	11	4.9	80	16	5.0
15%	62	13	4.8	80	16	5.0	120	24	5.0
20%	80	16	5.0	107	22	4.9	157	32	4.9
25%	100	20	5.0	132	27	4.9	196	40	4.9
30%	120	24	5.0	157	32	4.9	236	48	4.9
35%	140	28	5.0	184	37	5.0	274	46	(*)6.0
40%	157	32	4.9	209	42	5.0	312	45	(*)6.9
45%	178	36	4.9	236	48	4.9	351	44	(*)8.0
50%	196	40	4.9	261	44	(*)5.9	390	44	(*)8.9

The generated clusters, of predefined neuron size, are then used to create the *Part-population Graph* $\mathcal{G}_{\text{pp}}$. The spectral analysis applied to the *Neuron Graph* allows to label each neuron with a n-dimensional coordinates, in this way neurons can be managed like points where the distance between two of them represent their connectivity. Applying the clustering algorithm is then possible to isolate sets of neuron highly connected and to map them together. Thus, the partitioning problem can be solved iteratively through the *Sub-Clustering* phases, where neurons from the same population and cluster are grouped in sub-clusters matching the neurons per core constraint.

Vertexes of *Part-population Graph* $\mathcal{G}_{\text{pp}}$ are generated using the centroids of sub-clusters. Thus, each vertex of this graph represent a sub-cluster. Moreover, in order to prevent the generation of small part-populations, that lead to unoptimised use of cores, a *Fusion* phase is executed where sub-clusters are analysed and in some cases manipulated. If neurons belonging to the same population are split in more than one clusters and one of them has less than 20% of neurons that a core can simulate, these neurons are moved into the nearest cluster. In Figure 3.15 a simple example is shown, where a neuron belonging to the green population is

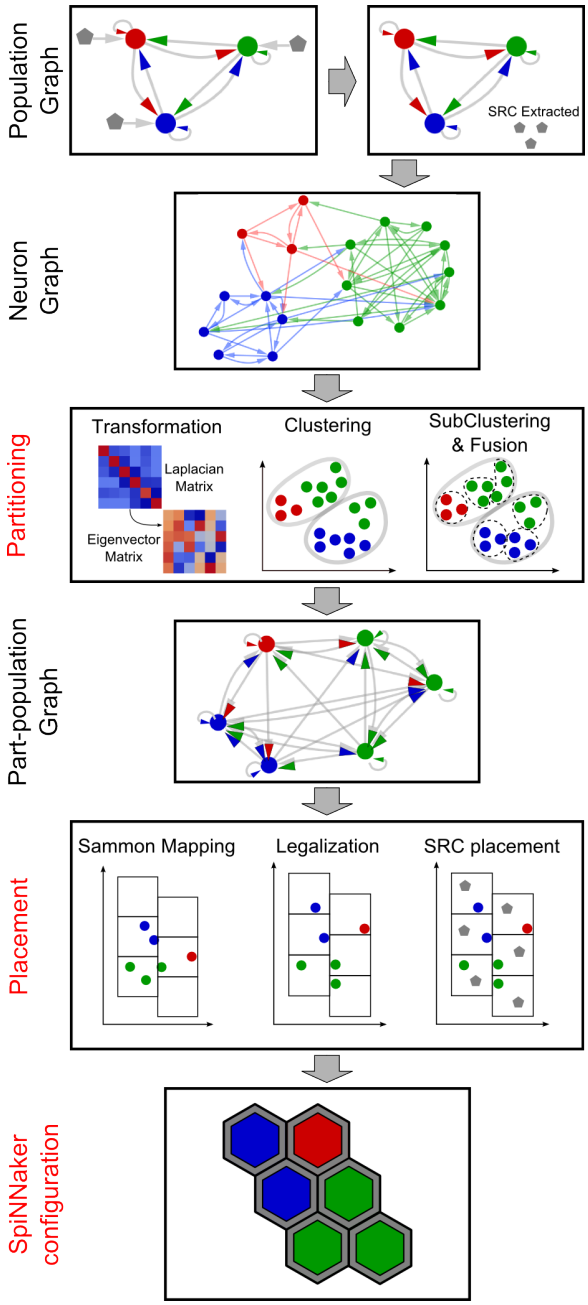


Figure 3.14: Flowchart of SNN Spectral Analysis based Partitioning and Placement algorithm. From SNN Population Graph to the chip placement of neurons.

clustered with neurons of the blue population. The fusion procedure recognise the green neuron and reassign it to one of the green sub-clusters, avoiding the creation of a supplementary vertex into the *Part-population Graph*.

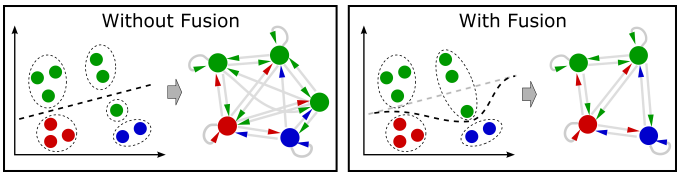


Figure 3.15: Fusion step. This procedure allows to merge uncompleted clusters in order to optimize the resources use.

For the *Placement* phase the part-populations graph is elaborated using the *Sammon Mapping* multidimensional scaling algorithm. This scaling procedure is applied in order to adapt the graph multidimensionality in a bi-dimensional space. This representation can be used for a direct placement process. During the *Sammon Mapping* part-populations that fall in an area are mapped in the free cores of the chip. If the number of part-populations in a single chip/square exceed the number of available cores a *Legalization* procedure is applied. In the example reported in *Figure 3.14* the constraint is one part-population for each chip, whereas in real cases up to sixteen part-populations can be placed. At last step, the *SRCPops* isolated in the early phase are directly placed on the reserved cores. The spike source neuron models (SRC) are used in SNNs to start or maintain special regimes of activity. For example, in the Cortical Microcircuit , the SRC neurons are used to simulate the background activity of adjacent cortical areas. SRC neuron is usually connected to a single target neuron and configured to simulate an high level of activity generating an intense traffic of packets. For this motivations placing SRCPops on the same chips that host the target neurons is a good practice to reduce the R2R traffic. Space can be reserved in each chip for this type of population accordingly to their particular connectivity. At last step, the configuration files are generated and sent to the SpiNNaker board.

Three Partitioning and Placement variants are adopted for this purpose. i) The *No-Fusion* make use of the GHOST procedure described in section 3.2.2 with the exception of the fusion step that is not implemented. ii) The *Fusion* uses the full procedure shown in Figure 3.14 to transform the Population graph into an optimised SpiNNaker configuration. iii) The *Random* is used as a reference to validate the improvement obtained by the other two variants. It makes a random division of *Neuron graph* considering only the number-of-neurons per part-population that must be kept homogeneous. Whereas, it apply a radial placement of the IF part-populations keeping in the same chip the associated SRCPops and DelayExtension.

The Cortical Microcircuit (CM) has been used to demonstrate the effectiveness of mapping method implemented in GHOST and to compare the achieved results with those obtained by the use of the mapping method implemented in PACMAN. In order to be compliant with the experiment previously proposed the scaling factor

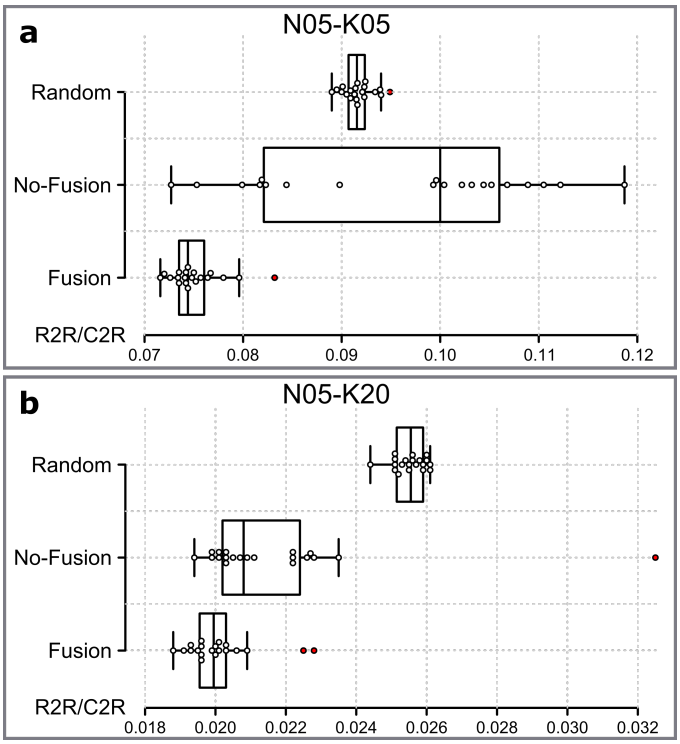


Figure 3.16: Distribution of R2R/C2R ratio computed on a two configurations N05-K05 and N05-K20. Twenty simulations have been performed for the three SNN-P&P techniques.

N05-K20 has been adopted. Moreover, in order to observe the system response when synapses number decrease from 3 M (20%) to 750 k (5%) I used a second scaling factor equal to N05-K05.

Six rounds of simulations were executed to extract the ratio R2R/C2R and the overall R2R packets circulating in the network. In each round, 20 simulation were performed using one of the three mapping procedure variants (Fusion, No-Fusion or Random) to set-up the SpiNNaker board with one of the two scaled CM. R2R/C2R ratio represents the traffic circulating in the inter-chip network versus the traffic generated into the chips. This ratio is used to compare the performances of the three investigated variants. Lower values of this rate correspond to the capacity of network to keep local the communications, reducing the number of packet circulating on the inter-chip links.

As can be seen in Table 3.4, the N05-K05 SNN placed with the SNN-PP *Random* variant generate an average of 292 k R2R packets. Instead, the network scaled at N05-K20 generate an average of 323 k R2R packets. In Figure 3.16 it is shown as CM configurations generated using the SNN-PP *Random* variant produce small fluctuations on the R2R/C2R ratio, with all the values concentrated near 9.10E-2

Table 3.4: Cortical microcircuit R2R traffic detected for two configurations N05-K05 and N05-K20 when three Partitioning and Placement variants of proposed technique are applied: Random, No-Fusion and Fusion. R2R packets percentage is computed comparing Spectral and Fusion variant with the Random configuration.

CM config.	Mapping	Packets R2R [10^3]					
		Worst		Average		Best	
N05-K05	Random	302		292		283	
	No-Fusion	378	+25%	318	+9%	231	-18%
	Fusion	265	-12%	237	-19%	228	-19%
N05-K20	Random	331		323		309	
	No-Fusion	411	+24%	264	-18%	245	-21%
	Fusion	289	-13%	252	-22%	238	-23%

for the N05-K05 case and 2.55E-2 for the N05-K20.

The executions of N05-K05 configured with *No-Fusion* variant produced a larger range of R2R/C2R ratio (Figure 3.16.a). Indeed, if compared with the *Random* variant, only part of the experiments come out with more balanced configurations. This is due to the generation of small part-populations (less than 10 neurons) that lead to the use of supplementary cores, thus increasing the R2R traffic.

To prevent this unwanted behaviour, in the SNN-PP has been added the Fusion step that is able to solve this problem. Experiments performed using the *Fusion* variant demonstrate that the R2R/C2R rate is always lower than the rate produced by the *Random* variant. Considering the N05-K20 we note that even the R2R/C2R rate of *No-Fusion* variant is always better than the rate of *Random*. This is principally due to the higher number of connections that produce more distant points for the clustering algorithm that can better balance the clusters and make rare the generation of small clusters.

The average of R2R packets produced by N05-K20 placed on SpiNNaker using the SNN-PP *No-Fusion* is reduced of 60 k packets with respect to those placed with the *Random* variant (Table 3.4). This is not true for N05-K05 where the influence of small part-population affects the average. Indeed, an increase of 26 k R2R packets is detected when the *No-fusion* variant is used. CM configurations generated with *Fusion* variant give always better balanced traffic and less R2R packets than the *Random* and *No-fusion* variants. The *Fusion* worst case with 265 k and 289 k packets is better than *Random* best case of 20 k R2R packets (-6% of R2R packet). In average the *Fusion* variant decreases the R2R packets of 55 k in N05-K05 and

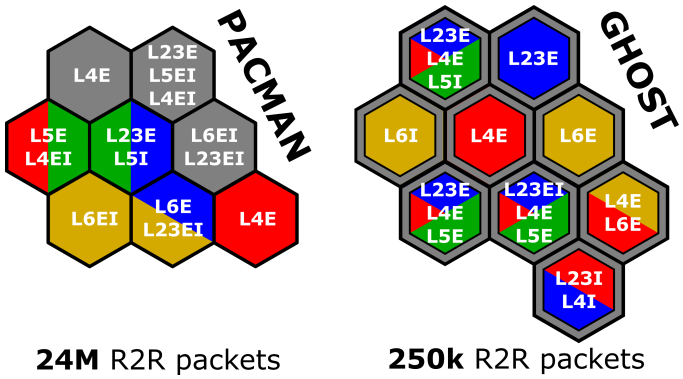


Figure 3.17: The placing of CM network: each hexagon represent a SpiNNaker chip. The color represent the CM population inside the chip, in gray the chip that runs the SRCPops.

70 k in CM N05-K20 (20% of R2R packet less than *Random* variant).

First and third quartiles of Fusion box plots in Figure 3.16 confirm that the spectral analysis is a suitable technique that applied to the mapping problem within SpiNNaker can produce good results in reducing the inter-chip traffic. Effect is enforced if associated with a fusion system capable to avoid the generation of little Part-populations.

At last analysis step, I propose the comparison between CM Partitioned and Placed using PACMAN and the configuration produced by our mapping algorithm implemented in GHOST. As it is possible to note in Figure 3.17 the 24 M of R2R packets generated by CM placed with PACMAN are reduced to 250 k if CM is configured with GHOST allowing a 96x reduction of R2R traffic.

3.2.5 Final Remarks

In this chapter, I described a mapping problem that involves a complex directed graph to be placed in a mesh of processors. I have modelled the mapping problem of an SNN into the SpiNNaker processors-mesh splitting the problem into 3 phases: population graph expansion, neuron graph clustering, and partial populations graph mapping. Focusing on the mapping phase, I have identified and tested 4 methodologies to solve the problem. The *Naïve* method (a simple heuristics) tries to maintain the location of the populations by placing them according to the order of creation without taking into account the real connectivity of the network. The *Spectral* method uses the graph eigendecomposition to obtain a planar representation of the SNN graph and performs the node association with the chip mesh through an ILP formulation. The *Scotch* method uses the *Dual Recursive Bipartitioning* heuristic for fast mapping of a source graph into a target graph. The

Simulated Annealing method uses the well-known *SA* procedure to minimise a cost function, the synaptic elongation.

I have redefined the cost function of the placement problem (the synaptic elongation) bringing it into matrix form as a function of a permutation vector. I have chosen the cortical microcircuit at different scale factors as our benchmark network, preferring it for its high connectivity and the presence of clusters. After performing several tests on the chosen benchmark network, the results highlight the superiority of the Simulated Annealing method that works natively on direct graphs. Using a fine-grain model, the gap between the SA and SCOTCH based method has narrowed, especially when dealing with large graphs. In these cases, the SCOTCH-based method has the advantage of providing an acceptable solution in a shorter time.

This modelling system for SNN placement problems can be adapted to other architectures such as Intel Loihi and SpiNNaker 2 for investigating new mapping techniques to be adopted for improving the usability of these emerging architectures.

The Spectral method was implemented in GHOST, a Python module compliant with the sPyNNaker tool-chain. A Cortical Microcircuit was simulated again with two scale factors in order to demonstrate the effectiveness of the developed mapping approach with respect to random neuron placement. Finally comparisons were made between configurations produced by PACMAN and GHOST. From these simulations was evident that GHOST is capable to reduce the number of used cores, results in lower R2R traffic, 96X when GHOST is adopted.

3.3 Communication Middleware and Message Passing Interface

In this chapter I describe the PoliTO SpiNNaker Software Stack (P3S). It is composed of three principal software layers developed during the second and the third year of my PhD with the aim of implement a message passing interface for SpiNNaker.

The first P3S module is the Multicast Communication Middleware (MCM) for the operating system (SARK) of the SpiNNaker system. The middleware aims to overcome the hardware limitation of the architecture when it is required a point to point communication between processors. All point-to-point communications must, therefore, be processed by the Monitor Processor. The middleware is capable of diverting unicast, broadcast and synchronisation communications on the multicast network of SpiNNaker. Considering that for obtaining unicast, broadcast and synchronisation communication are necessary about 1600 rules, but the routers have only 1024 available routing rules entries, I have defined a routing key

compression system capable of reducing to about 50 the rules necessary to obtain communications on the entire SpiNNaker PCB. MCM exploits the low-level test-and-set ARM capability to implement variable-polling in lock-sync primitives. This feature not only reduced the time required for synchronise all of 864 processors (time reduction between 30% and 40% based on the number of processors and chips involved) but also impacted the performance of the high-level software layers when lock-sync primitives are required. Results from experiments in host-to-board unicast communication demonstrated three times better performance than the built-in communication system.

The second P3S module is the Application Command Framework (ACF) an abstraction layer capable of managing Remote Procedure Calls (RPC). The ACF uses the Application Command Protocol (ACP) for managing the Host to Board and Board to Board communication. Moreover, the ACF manages the Memory Entities (MEs), a managed memory area, exposing functions to perform "Create, Read, Update and Delete" (CRUD) operations on them. ACF provides an application-level synchronisation and the concept of allowing the API to be flexible enough to avoid waste of memory space when used to implement buffers in communication protocols.

The third P3S module is the MPI implementation. Exploiting the Virtual Memory Entities, the application level synchronisation and the improved Host to Board interface in the ACP, I drastically enhanced the MPI implementation performance allowing to i) create a simulation context covering the whole board (all of 864 SpiNNaker processors) ii) drastically reducing memory impact on processors DTCM and iii) drastically reducing the time required for configuring the MPI runtime on the board side.

3.3.1 Multicast Communication Middleware

The data transmission within the architecture is currently performed using the Spinnaker Data Protocol (SDP) on top of UDP in external devices communications and point to point network (PP) in internal communications between SpiNNaker Chips. The use of the PP network limits the system as it requires the Monitor Processors to fragment the data to be sent and to recompose them once the target chip is reached, 3.18. Furthermore, this type of communication is sequential and does not exploit the full capabilities of the system network parallelism. Point to Point protocol is slow for in-board communications since it needs the intermediation of the Monitor Processors in addition to chips involved in the communication. As explained in 2.1.5 when sending a SDP packet, the core saves the packet inside the SysRAM, the MP of the chip reads the content of SysRAM, splits the content in 32bit fragments and forwards them to the Monitor Core of the destination chip. The latter copies the message received into the system memory and informs the

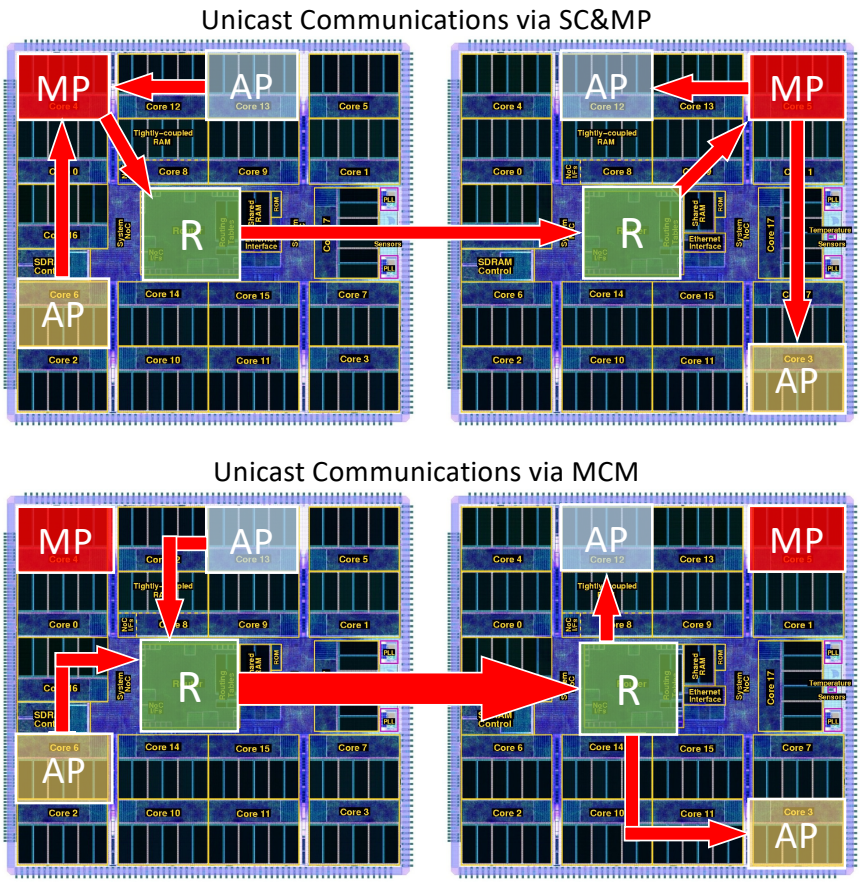


Figure 3.18: MCM Scenario

recipient Application Processor that the packet has been received.

The multicast SpiNNaker Network allows to implement a direct communication between processors. With efficient use of the header of the MC packets, it is possible to create a Broadcast, Unicast and Multicast communication system and also synchronisation protocols. Each routing key can either contain fields for indicating the coordinates of the destination core in case of a unicast communication or the sender information in case of a broadcast one. My idea has been to develop a Middleware able to exploit the multicast protocol features to speed-up the system communications.

In this section the features of Multicast Communication Middleware (MCM) will be presented. It will also describe the routing key compression rules needed to make the middleware feasible.

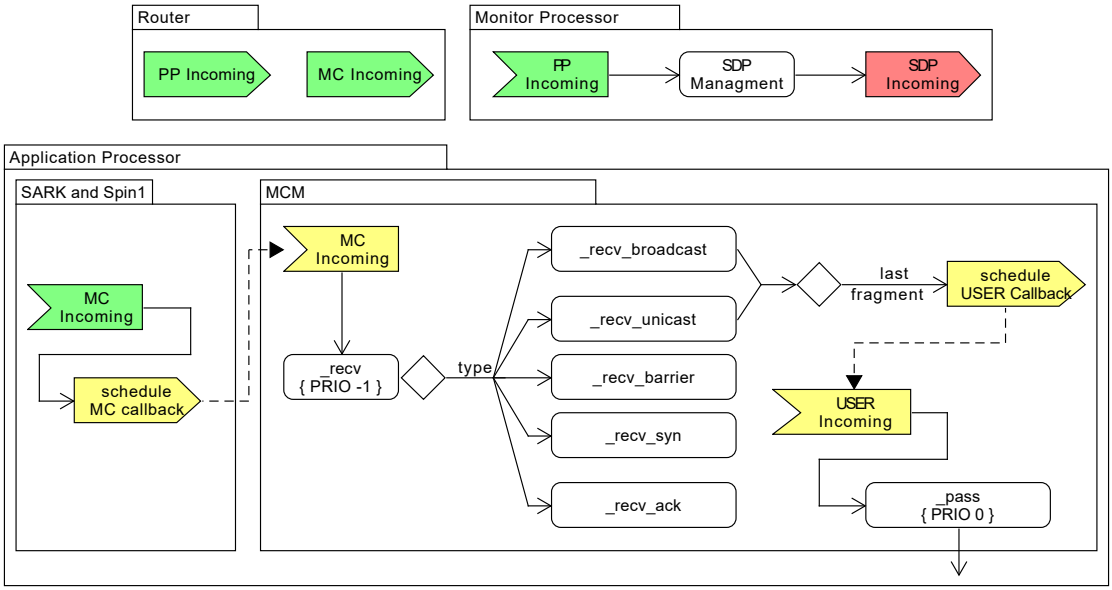


Figure 3.19: MCM operating flow after receiving a Multicast packet

MCM primitives

MCM implements several features, including an ADT to manage hash tables and all APIs to manage communications via SpiNNaker multicast network. MCM allows sending frames of up to 272 Bytes, and each MC packet transmits fragments of four or six bytes depending on the type of communication. It also allows to manage unicast synchronization through two Sync and Ack packets and broadcast synchronization (a.k.a. Global Barrier or Barrier) through a system of multi-level synchronization packets to do not overload the network.

MCM exposes the main functions in a public header `mcm.h`:

- `mcm_send(...)` This is the main function of the MCM library. It accepts three byte-streams and their size. In this way, it is possible to manage the header, body and tail of a message independently. It allows choosing the communication channel: Broadcast or Unicast. When using Unicast channel, an additional field is required to indicate the destination of the message.
- `mcm_callback_register(mcm_callback_t c)` With this function we can register a callback to be used when receiving a byte stream via MCM.
- `mcm_barrier()` This function starts a broadcast synchronization procedure. It allows to create a barrier over the whole execution context.

- `mcm_syn(mcm_core_t *destination)` This function sends a unicast synchronization packet via MCM.
- `mcm_wfs(spin2_core_t *source)` This function waits for the reception of a unicast synchronization message via MCM.

MCM also has a set of private functions not visible outside the library that are used internally to manage the reception of multicast packets as in Figure 3.19:

- `_send(...)` Primitive communication function to send a fragment, takes as arguments the stream of bytes that make up the fragment and its size, a flag to indicate if the fragment is the last of the stream, a flag to indicate if the fragment must be sent with extended_payload (6 Byte instead of 4 Byte), the coordinates of the destination pivot_core. It is called by the function `mcm_send(...)`.
- `_recv(uint32_t key, uint32_t payload)` Primitive function to receive a multicast packet, it is the entrypoint of all functions `_recv_*(...)` described below. It is called with maximum priority (PRIO -1) from the SARK/Spin1 scheduler when a multicast packet is received.
- `_recv_broadcast(...)` manages the reception of a data packet (fragment) on the broadcast channel, the source of the packet is used to identify the reconstruction buffer. In case the last fragment flag is active, schedule a USER event to call the `_pass(...)` function.
- `_recv_unicast(...)` manages the reception of a data packet (fragment) on the unicast channel, the source of the packet is used to identify the reconstruction buffer. In case the last fragment flag is active, schedule a USER event to call the `_pass(...)` function.
- `_recv_syn(...)` manages the reception of a Syncronization packet. Sets a sync flag relative to the source of the packet.
- `_recv_ack(...)` handles the reception of an Acknowlegment packet. Sets the ack flag (only one exists).
- `_recv_barrier(...)` handles the reception of a Barrier packet. Increases multi-level synchronization counters.
- `_pass(...)` calls the function registered with `mcm_callback_register(...)` and at the end of the execution releases the reconstruction buffer. It is called with high priority (PRIO 0) from the SARK/Spin1 scheduler when receiving a USER event.

The MCM (Figure 3.19) workflow starts when a Multicast packet is received from the Router containing a data fragment or synchronization signal. The Router

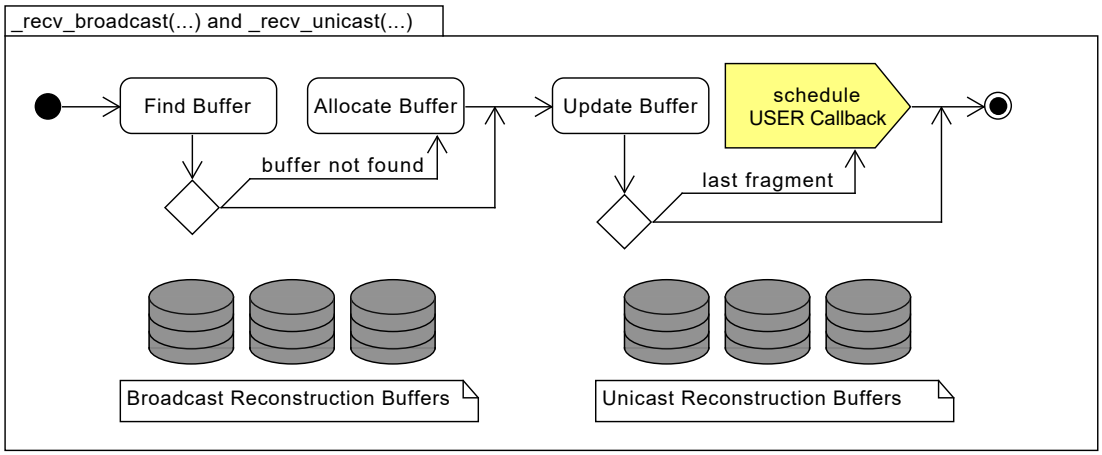


Figure 3.20: The MCM receive fragment workflow

on receiving a packet triggers a hardware interrupt on the core intercepted by SARK/Spin1, which reacts by performing the procedure associated with the interrupt. In the case of MCM, we register a maximum priority callback (PRIO -1) that allows the atomic execution of the operations necessary to process the content of the MC packet as fast as possible.

If the package contains a data fragment depending on the source channel (Unicast or Broadcast), the package will be processed by two different functions. Both share the same workflow as shown in Figure 3.20. MCM has four reconstruction buffers per channel, each buffer has a size of 272 bytes, for a total of 2 176 Bytes in DTCM (reconstruction buffers can, if desired, be allocated in SDRAM but this would compromise the reception performance of a fragment). MCM can, therefore, virtually handle four communications at the same time. When the first data fragment is received, a reconstruction buffer is associated with the packet source. The following fragments of data will then be inserted into the correct reconstruction buffer until the last fragment is received. At this point, the execution of the `_pass()` function is scheduled, which will call the function in charge of managing the received data and finally free the reconstruction buffer.

Syncronization

MCM provides synchronization capabilities both between processor pairs (unicast synchronization) and across the whole execution context (broadcast synchronization or barrier).

In the first case we use the functions `mcm_syn(mcm_core_t *destination)` to start

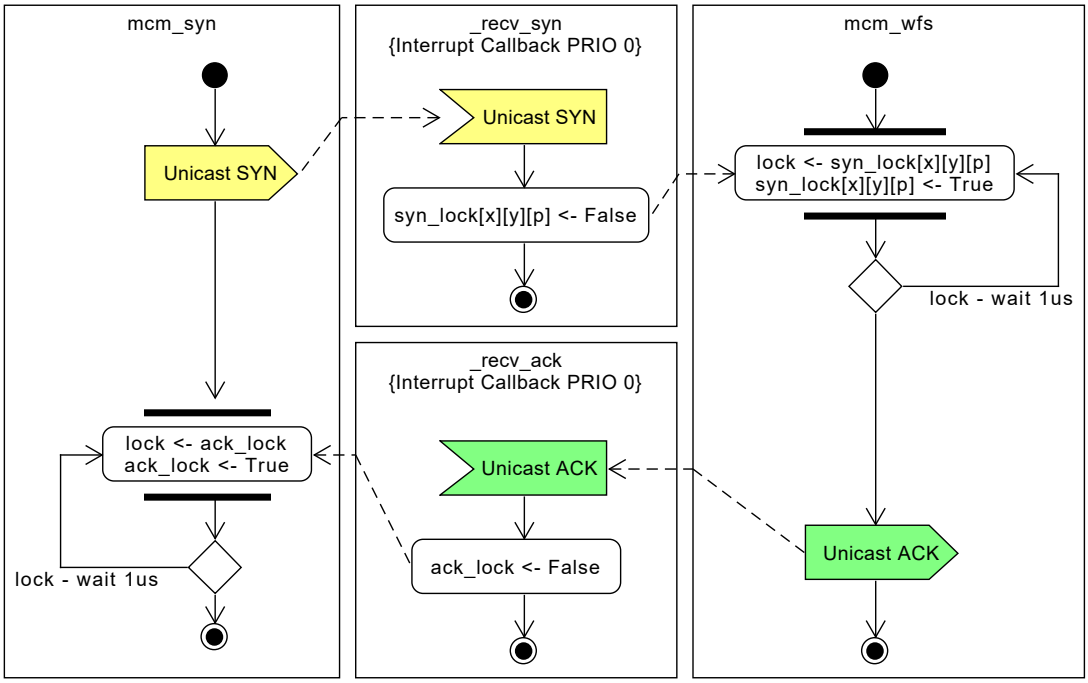


Figure 3.21: The Unicast synchronisation workflow in a SpiNN-5 Board.

synchronization and `mcm_wfs(spin2_core_t *source)` to wait for synchronization. Both functions are blocking, and involve interaction between processors as described in Figure 3.21. The modality is therefore asymmetric, with one processor in the active role (`A`) and one in the passive role (`B`).

Each processor contains an array of `sync_lock[8][8][16]` flags of the weight of 1 KiB used to store the received synchronization signals, and a single `ack_lock` flag to store the reception of an acknowlegment signal.

In case the `A` processor starts the active synchronization before the `B` processor the latter will set `sync_lock[Ax][Ay][Ap] <- false` and `A` will start a busy-loop polling on `ack_lock != true`. When the `B` processor is ready to passively synchronize on `A` it will start a busy-loop polling on `sync_lock[Ax][Ay][Ap] != true`. At the end it will send an ACK message to the `A` processor.

This asymmetric synchronization mechanism makes it possible to implement higher level communication protocols that include, for example, operations to prepare on-the-fly reception buffers (such as ACF virtual memory entities).

The implementation of broadcast synchronization, on the other hand, requires All-to-All communication from all processors to signal to the entire execution context that it has reached the barrier before continuing with the execution of the code. In

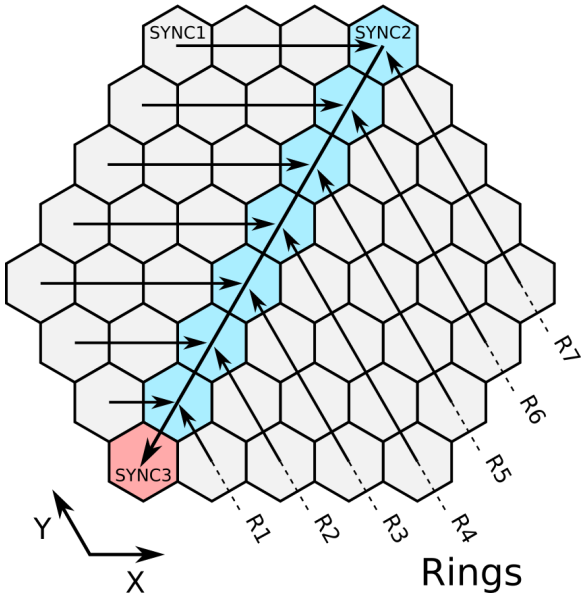


Figure 3.22: The Broadcast Syncronization Regions in a SpiNN-5 Board.

order to make this functionality feasible, I have reduced the problem to a multi-level synchronization. I have identified three levels of synchronization, the first identifies all the processors belonging to a chip, the second identifies all the chips that share a particular position within the board, and the third level identifies the entire board.

All processors with vID equal to one are level one synchronizers. Depending on the chip position, they can also act as level two or level three synchronizers. The level 1 synchronization island includes the processors in use inside a SpiNNaker chip and is called Chip-XY.

As shown in Figure 3.22 level 2 synchronizers have $X = Y$ coordinates and are therefore arranged along the diagonal. The level two synchronization island is called Ring-K and includes all level 1 synchronizers belonging to chips with $\max(X, Y) = K$ coordinates. Their synchronizer is the chip with coordinate $X = Y = K$. This arrangement avoids overloading routers as shown in section 3.1 and minimizes packets circulating on the network. The packets will flow inside the ring without creating complex propagation flows in the board.

The level 3 synchronizer is one and belongs to the X=0 Y=0 chip. The level 3 synchronizer island therefore includes all level 2 synchronizers and is called Board.

The synchronization of the Chip-XY islands takes place via a semaphore in shared memory within the individual chips. The level 1 synchronizer performs a busy-loop polling on the semaphore until it reaches the desired number of synchronized processors.

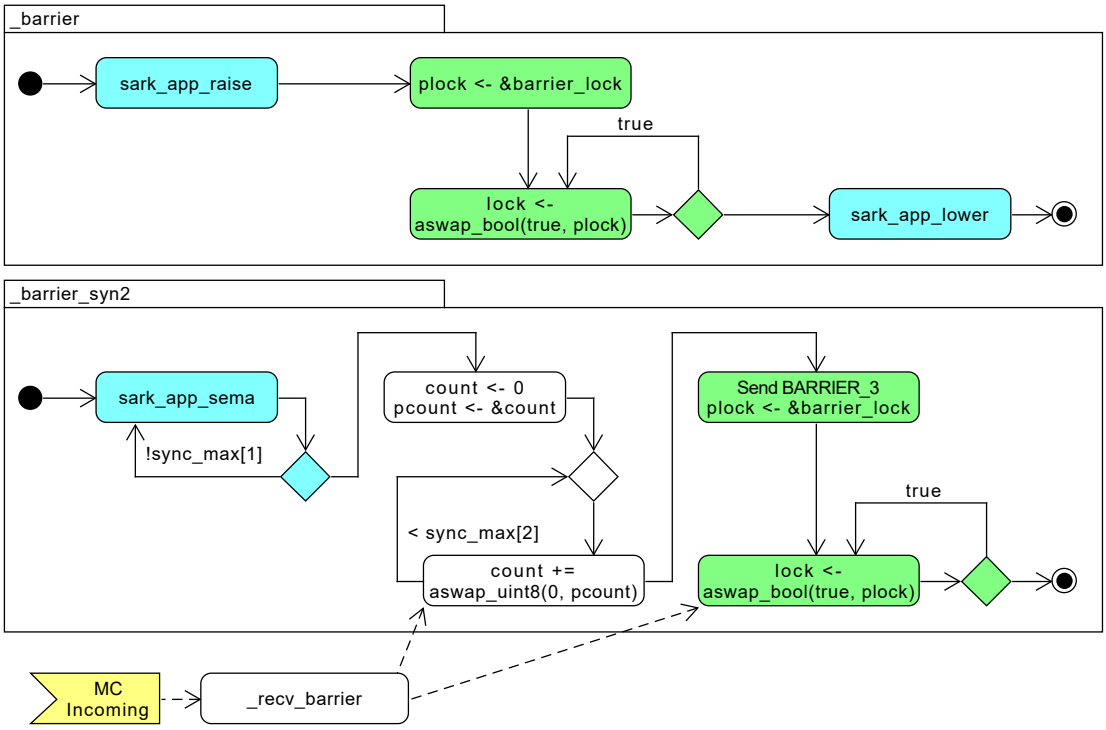


Figure 3.23: The Broadcast Syncronization workflow in a SpiNN-5 Board.

The synchronisation of the Ring-K islands takes place through the exchange of Barrier-Sync2 messages issued by the level one synchronisers and received by the level 2 synchroniser which performs a busy-loop polling on a counter from which it will exit once the desired number of synchronised chips belonging to the ring has been reached.

The board synchronization takes place through the exchange of Barrier-Sync3 messages issued by the level two synchronizers and received by the level 3 synchronizer that performs a busy-loop polling on a counter from which it will exit once the desired number of synchronized rings belonging to the board has been reached. Finally, it will send a BARRIER-END broadcast package that will end the synchronization.

In this way I have exploited the parallelism of the board and I synchronized the entire architecture in just four steps.

The call to the `mcm_barrier()` function will then be specialised according to the role of the processor in the following functions:

- `_barrier()` Executed by non-synchronizing cores, it manages the execution of a multilevel synchronisation (barrier). It emits a signal on the synchronization

semaphore shared at chip level and waits for a BARRIER-END packet.

- `_barrier_syn1()` Executed by a level 1 synchronizer (Chip Level), it manages the execution of a multilevel synchronisation (barrier). It waits for N signals on the shared synchronization semaphore at chip level, emits a BARRIER-SYNC2 packet to the level 2 synchronizer and waits for a BARRIER-END packet.
- `_barrier_syn2()` Executed by a level 2 synchronizer (Ring Level), it manages the execution of a multilevel synchronisation (barrier). It waits for N signals on the shared synchronization semaphore at chip level, M packets BARRIER-SYNC2 and emits a BARRIER-SYNC3 packet to the level 3 synchronizer and waits for a BARRIER-END packet.
- `_barrier_sync3()` Executed by a level 3 synchronizer (Board Level), it manages the execution of a multilevel synchronisation (barrier). It waits for N signals on the synchronization semaphore shared at chip level, K packets BARRIER-SYNC3 and emits a BARRIER-END packet in broadcast.

Packet Headers

Multicast packets have 32bit routing key and 32bit payload. The 32bit routing key are used by routers to forward packets but not all bits of the routing key need to be used.

The router, upon receiving a multicast packet, looks for a match between the routing key of the packet and an entry of the routing table. The entries of the routing table are composed of triples: routing key, routing mask, routing rule. The goal is to identify the correct routing rule to forward the packet. The packet routing key is simultaneously compared with all rows of the routing table. The comparison is done first by masking the packet routing key with the routing mask and then comparing the result with the routing key from the routing table. If the two keys are equal then the routing rule will be active. Finally the first active routing rule of the routing table will be used.

This procedure provides a high degree of flexibility in handling the routing key fields, allowing the possibility to specify fields that do not contribute to the packet routing but are used by the protocol for other purposes.

In the case of unicast package the routing key is divided as follows:

- Bit 30-31, MCM Type field, set to 0 to identify a unicast package
- Bit 27-29, DST-X field, Indicates the X coordinate of the receiving chip, valid value range 0-7

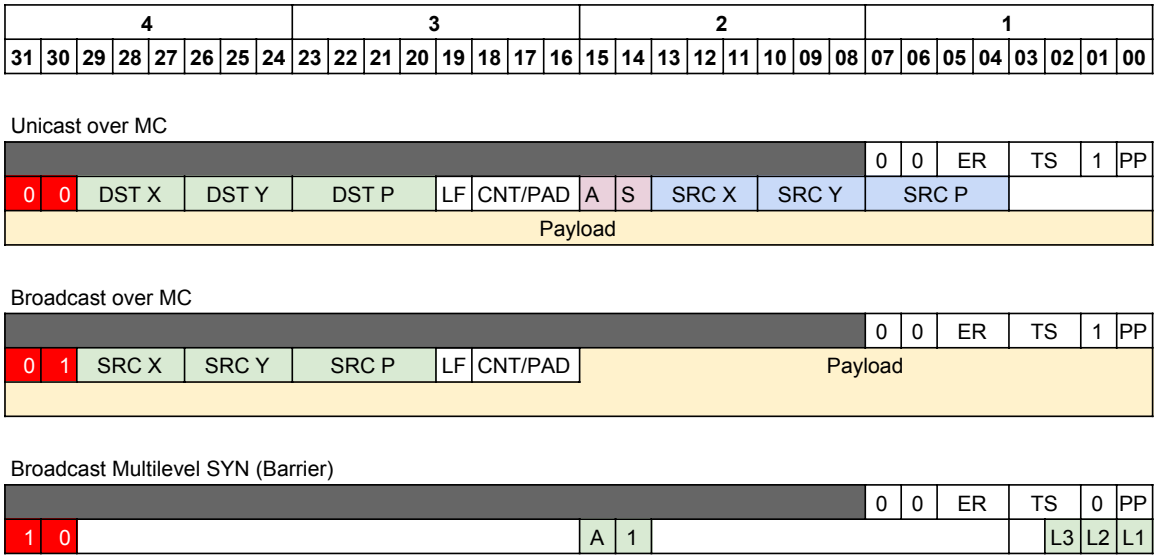


Figure 3.24: MCM Header for Unicast, Broadcast and Multicast communication packets

- Bit 24-26, DST-Y field, Indicates the Y coordinate of the receiving chip, valid value range 0-7
- Bit 20-23, DST-P field, Indicates the vID of the receiving processor, valid value range 0-15 with implicit offset of one.
- Bit 19, LF filed, Last Fragment flag
- Bit 16-18, CNT field if LF is false, fragment counter with a window of 7 fragments. PAD field if LF is true, indicates padding bytes in the Payload field, range of valid values 0-3
- Bit 15, ACK field, Indicates whether the package is an Acknowlegment package
- Bit 14, SYN field, Indicates whether the package is a Synchronisation package
- Bit 11-13, SRC-X field, Indicates the X coordinate of the source chip, valid value range 0-7
- Bit 8-10, SRC-Y field, Indicates the Y coordinate of the source chip, valid value range 0-7
- Bit 4-7, SRC-P range, Indicates the vID of the source processor, valid value range 0-15 with implied offset of one.

The following four bytes contain the payload of the package.

In the case of a broadcast package, the routing key is divided as follows:

- Bit 30-31, MCM Type field, set to 1 to identify a broadcast package
- Bit 27-29, SRC-X field, Indicates the X coordinate of the source chip, valid value range 0-7
- Bit 24-26, SRC-Y field, Indicates the Y coordinate of the source chip, valid value range 0-7
- Bit 20-23, SRC-P field, Indicates the vID of the source processor, valid value range 0-15 with an implicit offset of one.
- Bit 19, LF Field, Last Fragment flag
- Bit 16-18, CNT field if LF is false, fragment counter with a window of 7 fragments. PAD field if LF is true, indicates padding bytes in the Payload field, range of valid values 0-5
- Bit 0-15, 2 Payload Bytes

The following four bytes contain the payload of the package. Since we do not need to specify the package destination and synchronization flags, I use two bytes of the header to insert 2 additional payload bytes, bringing the total payload to 6 bytes.

In the case of Broadcast Multilevel SYN (Barrier) package the routing key is divided as follows:

- Bit 0, Level 1 barrier, packets that will flow to the Chip Synchroniser. Not used, shared memory semaphores are used.
- Bit 1, Level 2 barrier, packets that will flow into the Ring Synchroniser
- Bit 2, Level 3 barrier, packets that will flow to the Board Synchroniser
- Bit 15, Barrier Ack, broadcast from chip x=0 y=0

The routing rules for this type of packets are dependent on the location of the router to be configured and have a `0xFFFFFFFF` mask.

For unicast and broadcast packet types the only bits used by routers are the ten bits highlighted in green in Figure 3.24. The routing masks can then be set to `0xFFF00000`. In total, 2^{11} 2048 routing rules are required to handle unicast and broadcast packets, plus rules for barrier synchronization. The routing tables contain only 1024 lines, so we need to implement a system to compress the use of the rules. A first attempt could be to use `0xFF000000` masks considering only the X and Y fields for all the rules that do not refer to the position of the router to configure and `0xFFF00000` masks for the rules that refer to the position of the router to configure.

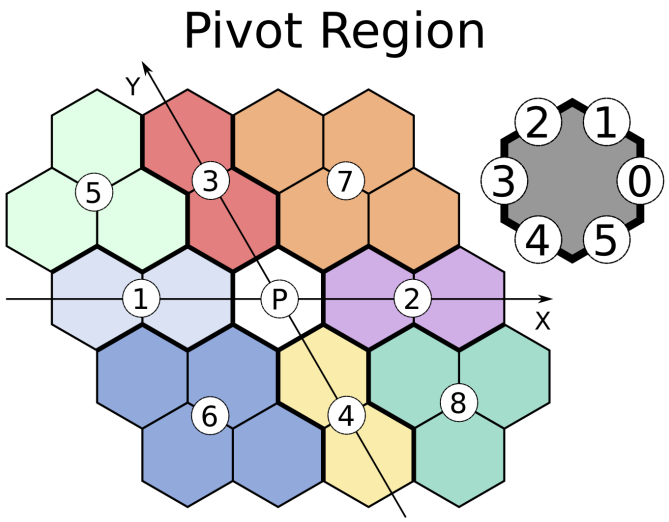


Figure 3.25: Pivot Chip and regions for the SpiNNaker chips

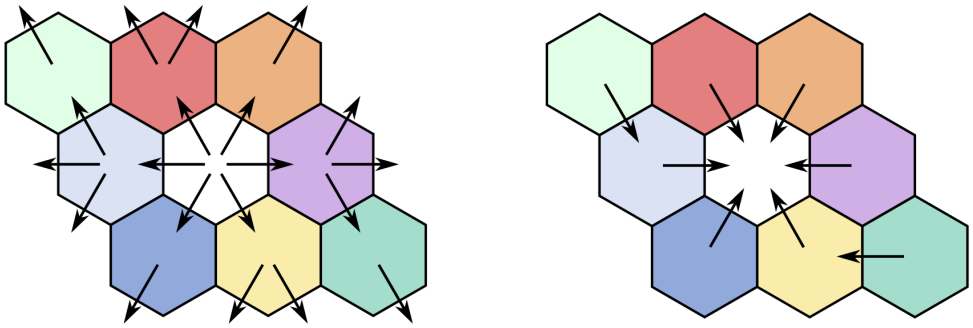


Figure 3.26: Broadcast and Unicast routing rules

This would only use 2^7+32 rules for a total of 160 rules plus the rules for barrier synchronization but it is possible to further reduce the use of the routing table.

Routing Rules

I designed an algorithm to generate compressed rules that make possible the efficient usage of routing tables.

The transmission on the board can be either seen as a packet coming from a source having multiple destinations in multicast communications or as a packet directed to a specific target in unicast communications. I define the concept of Pivot Chip as the source chip in broadcast communications and the target chip in unicast communications.

Table 3.5: Routing rules to apply in Unicast e Broadcast MCM communication channels. The first column define the chip region relative to Pivot Chip and the condition based on chip coordinates

Region	Condition	External Chip Ports in Routing Rule											
		Broadcast						Unicast					
		0	1	2	3	4	5	0	1	2	3	4	5
P	$X = X_P\ Y = Y_P$	X	X	X	X	X	X						
1	$X < X_P\ Y = Y_P$			X	X	X		X					
2	$X > X_P\ Y = Y_P$	X	X				X				X		
3	$X = X_P\ Y > Y_P$		X	X									X
4	$X = X_P\ Y < Y_P$					X	X			X			
5	$X < X_P\ Y > Y_P$			X									X
6	$X < X_P\ Y < Y_P$					X			X				
7	$X > X_P\ Y > Y_P$		X									X	
8	$X > X_P\ Y < Y_P$						X			X			

The routing rules are built upon the assumption that a SpiNNaker board is divided into eight regions relative to a specific Pivot Chip that is the centre of the packet propagation. These regions are shown in Figure 3.25.

Each six external-links of a SpiNNaker chip has a port-number and a label. The port number grows anti-clockwise from 0 to 5. A routing rule can activate one or more external link ports.

In Figure 3.26, I show how a packet must be routed following the Broadcast rule or the Unicast rule. In order to build the routing rules the first step is to obtain the region to which the chip X, Y, that we want configure, belongs compared to the pivot chip X_P, Y_P. Now, depending on the region the chip belongs to, we can assign the rule to apply as shown in Table 3.5 and in Figure 3.26.

We get 80 addressing rules (2^6+16, 16 are used to manage processor rules in case of region P) for each communication channel. Leveraging the behavior of the router, which accepts multiple matches within the ruouting table using the first entry that matches, and the binary overlay in the six bits that compose the coordinate of the pivot chip, we can reduce the rules by a factor of 3x obtaining only 24 (8+16).

To achieve this we proceed with the following algorithm.

- Grouping: we group all routing keys associated with a region into a Region Key Group. For each key we consider the 6 bits of the Pivot-X and Pivot-Y fields
- Consensus Regions: We identify the positions where the bits of all words in the Region Key Group have the same value.
- Fusion: we generate a fusion key, a 6 trit word (trinary digit, 0, 1, X). Each word will assume values 0 and 1 for all positions belonging to the consensus regions, depending on the value of the consensus regions, otherwise it will assume value X.
- Analysis: For each fusion-key an entry-prototype is created, a triple of integer values (X-number,X-position,K,M) which contains, in order:
  - X-number the number of trit to X
  - X-position the position of the first trit to X
  - K a 6bit word in which each trit of the fusion-key is reported by replacing the trit X with bit 0
  - M a 6bit word in which is associated for each trit different from X a bit to 1, and for each trit to X a bit to 0.
- Sorting: the triple is sorted by values (X-number,X-position,K)
- Routing Key: The K field of the triple is used as the routing key and the M field as the routing mask.

To summarize, the routing key are first grouped according to the routing rule, Region Key Group, and summarized in a single fusion-key. A fusion-key contains the binary entry of the Region Key Group consensus regions and a X trit in the discordant positions. The fusion-key are then sorted according to the number of X trit, the position of the first X trit, and finally by the fusion-key routing itself (considering as a binary word with the X trit sobstituted with a bit 0).

We then obtain an ordered list of eight pairs (routing key, routing mask) that allow to route all possible packets coming from or directed to a pivot chip. It is guaranteed that even in case of multiple matches the correct routing rule will be chosen because the rules are sorted by generality and the first rule that matches is always the correct one.

Spinlock and Atomic Swap

The MCM library needs a system to manage portions of code that require to test particular conditions on global variables before continuing with the subsequent code execution. Usually, this is required while waiting for a lock-flag to be released

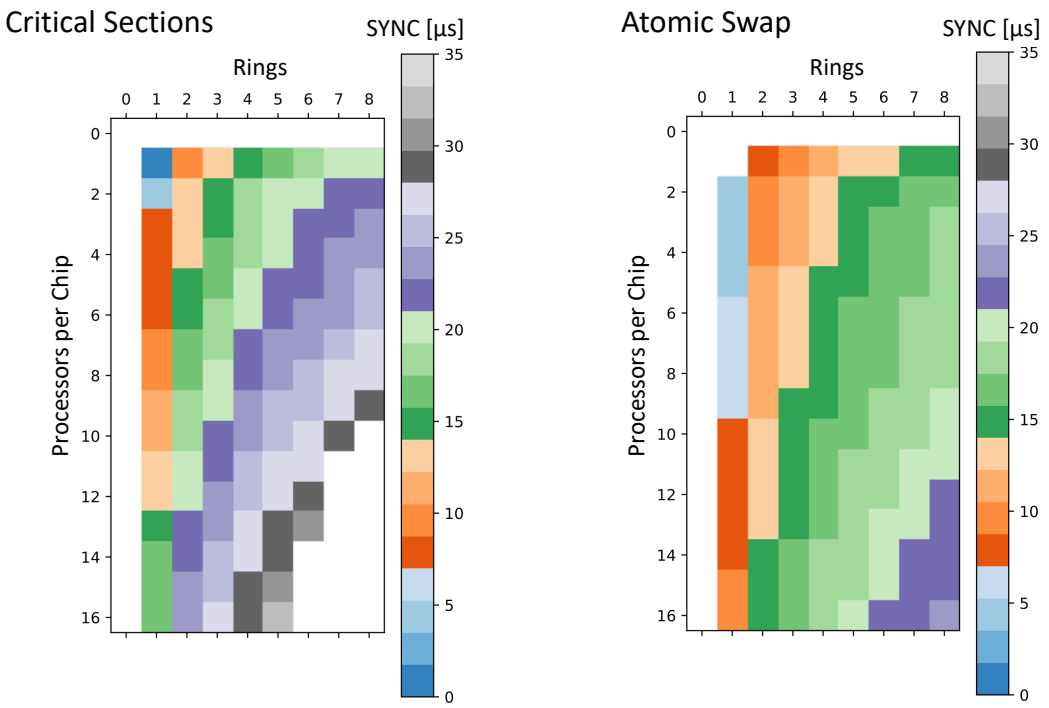


Figure 3.27: Impact of atomic swap with respect to critical sections during an MCM Barrier synchronization. Considering for example the configuration of 5 rings and 16 processors we get a 37% improvement in the time needed for synchronization, from 32 µs to 20 µs.

or for a counter variable to reach a specific desired value during synchronization primitives.

These variables, like lock-flag and event-counters, are modified by high priority interrupts. The lock-flag function, on the other hand, is executed at a lower priority and must perform a test operation on the variable and, in some cases, a reset operation to return it to its original value.

This functionality can be implemented in two ways. A first solution can be the scheduling of an event, to be triggered when the lock is released, and then by switching to sleep-mode. A second solution is the implementation of busy-loop polling on the lock variable (spinlock). In the first case, we introduce the overhead of the Spin1 scheduler and, in some situations, the simultaneous sleep-mode output of many processors causes current peaks on the architecture. Considering also the average waiting time for synchronization messages of less than 100 µs, I opted for the implementation of a spinlock.

In this case, we need to manage the polling on the lock variable in atomic mode. The current SARK library allows creating critical sections by disabling interrupts.

```
1    static inline uint8_t mcm_swap_uint8(uint8_t set, uint8_t *test) {
2        uint8_t check;
3        __asm__ __volatile__ ("swpb %[check], %[set], [%[test]]"
4            : [check] "=&r" (check) // & force to use different registers
5            : [set] "r" (set),
6              [test] "r" (test) );
7        return check;
8    }
```

Figure 3.28: Atomic swap implemented in MCM

Within the critical section, it is, therefore, possible to test the lock flag in an atomic mode without a higher priority interrupt changing its value. The ARM processor that composes the SpiNNaker chips, however, allows performing an atomic swap operation between DTCM elements. I have therefore implemented Critical Section and Atomic Swap spinlock models to measure their performance during SpiNNaker chip synchronization.

The implementation of the spinlock using the instruction `swpb`, adequately integrated with the library through inline assembly, allows a considerable reduction of the time needed to run a barrier on the whole board, as we can see in Figure 3.27. In particular, for a high number of processors involved, a speedup of 37% is obtained.

3.3.2 The Application Command Framework

In this section, I describe the *Application Command Framework (ACF)*.

The Application Command Framework works at the application level and extends the state-of-the-art software for this platform increasing its flexibility and efficiency of reconfiguration. In particular, it allows the transmission and interpretation of high-level op-codes defined by the users and embedded in the distributed applications (i.e. *Remote Procedure Calls - RPCs*). Indeed, cores can thus execute commands transmitted by external devices. The protocol supports both host-to-platform and core-to-core communication. Through this mechanism, the protocol also implements the possibility to manage the cores memory (i.e. triggering read-/write operations) by abstracting physical memory addresses using virtual IDs defined as *Memory Entities*. As a result, cores can communicate, trigger operations and synchronise their execution.

More generally ACF enables the embedding of alternative computational flows in the applications running on the board allowing the host to control their behaviour at runtime through RPCs and manage their memory using Memory Entities. Exploiting these features, ACF also provides core-to-core communication and synchronisation support.

I implemented ACF as an application library that can be used by the applications.

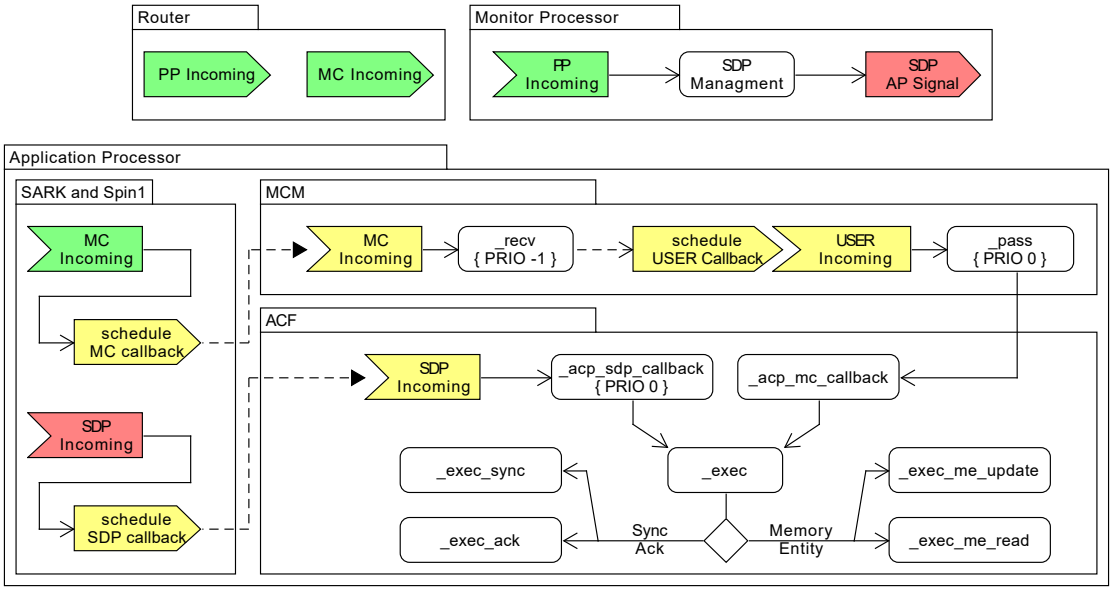


Figure 3.29: ACF workflow in receiving messages.

To support the communication between processors in distributed applications, I defined the Application Command Protocol (ACP). The header of an ACP packet consists of 4 Bytes containing the *Packet ID* and the *Command ID*. The following bytes are used for the header of the command (from 4 to 12 Bytes) depending on which *command ID* is in the packet. Likewise, the format of the *Command Payload* depends on the type of command, it has a maximum limit of 264 Bytes length.

An ACP message can be transmitted over the board using two communication channels. The *ACP over SDP* uses the SDP protocol implemented in the native SpiNNaker software stack. The Monitor Processor of the sender chip breaks the SDP packet into PP packets and sends them to the Monitor Processor of the receiving chip. The *ACP over MCM* uses the multicast (MC) channel and the MCM library. The sender Application Processor breaks each ACP packet into a set of MC packets, each one is a 32 bits fragment, and transmits them using the Multicast Communication Middleware.

ACP workflow

The Application Command Framework is implemented in two libraries, *SpynnakerACF* and *ACF*. The *SpynnakerACF* is implemented as a Python package and is organised as a collection of classes and utility methods used in the host computer to create, send and receive commands to SpiNNaker chips through the Ethernet connection. The library provides a framework to define the functions to be implemented

when a SpiNNaker core receives a command. The framework is customisable, as users with particular needs can extend the default set of commands for supporting new functions that will help the design of flexible applications.

The *ACF* library is built on top of SARK, the event-driven programming model provided by the *Spin1* library and MCM. This module provides three main functionalities: i) At network level, it implements three channel of communications: `Channel Core`, `Channel Broadcast`, `Channel Host`. ACF use the unicast communications provided by MCM for the `Channel Core` and the broadcast communications provided by MCM for the `Channel Broadcast` (*ACP over MCM*). The `Channel Host` instead use *ACP over SDP*. ii) It provides a customizable framework for supporting command management (*Remote Procedure Call*). iii) It implements an abstraction level of the memory blocks through the definition of *Memory Entities*.

In Figure 3.29 I show the workflow designed to manage the reception of an ACP packet. The Monitor Processor mediates the *ACP over SDP* implementation using point to point connections. Whereas the *ACP over MCM* implementation is directly managed by the Application Processors.

The main functions provided by the library to manage RPC and CRUD operations on Memory Entities are:

- `acf_cmd_create(...)` To register a function locally on a Command ID. It accepts as parameters a 16-bit integer to represent the Command ID and a function pointer.
- `acf_cmd_delete(...)` To remove a function from a Command ID. It accepts as a parameter a 16-bit integer to represent the Command ID.
- `acf_cmd_run(...)` To execute the function associated with a Command ID. It accepts as parameters a 16 bit integer to represent the Command ID, a byte buffer and its size containing the payload to pass as argument to the function, the communication channel, the destination (only for `Channel Core`) and a synchronization flag to make the execution blocking.
- `acf_me_create(...)` To create a memory entity locally. Accept as arguments a Variable ID, the size of the memory entity, and two function pointers for write and read callbacks.
- `acf_me_read(...)` To read the contents of a memory entity, locally. It accepts as parameters a Variable ID, the byte buffer to write the content of the memory entity, the length in bytes you want to read, the communication channel on which to synchronize, the remote source (only for `Channel Core`) and a synchronization flag to make the execution blocking.

- `acf_me_update(...)` To update the content of a memory entity, locally.
- `acf_me_update_remote(...)` To update the content of a memory entity remotely. Accepts as parameters a Variable ID, the byte buffer you want to write in the memory entity, the length in bytes you want to write, the communication channel, the destination (only for `Channel Core`) and a synchronization flag to make the execution blocking.
- `acf_me_delete(...)` To remove a memory entity locally.

The library has a private header containing all the communication primitives used internally to manage ACP, and memory entities. The main functions are:

- `_exec()` interprets and executes an ACP packet received over the network.
- `_me_create()` manages the creation of a memory entity
- `_me_delete()` removes a memory entity
- `_me_read()` reads the content from the buffer of a non-virtual memory entity.
- `_me_update()` writes to the buffer of a memory entity, if virtual requires the previous use of `_me_set_buffer()`
- `_me_get()` gets the handler of a memory entity
- `_me_is_lock()` controlla se la memory entity ha un read-after-write lock
- `_me_is_virtual()` check if the memory entity is virtual
- `_me_set_lock()` set a read-after-write lock
- `_me_set_buffer()` set a buffer for a virtual memory entity.

Command Management

The command management ACF functionality (RPC) can be recreated in the SCP by modifying the SARK kernel in order to support new commands[2]. However, extending the available commands requires extensive kernel modifications, with additional efforts devoted to maintaining the kernel light, stable, and safe.

Instead, ACF only requires to create a new callback in the commands list, allowing the user to easily define custom commands. More specifically, the user registers the function implementing the command interpreter as a callback function on the ACF library. This function is associated with an identifier, *Command ID*, and stored in a hash-table. This solution provides much better flexibility than a static vector.

[2]Currently each Application Processor can execute four SCP commands: Get Kernel Version, Memory Read, Memory Write and Application Run.

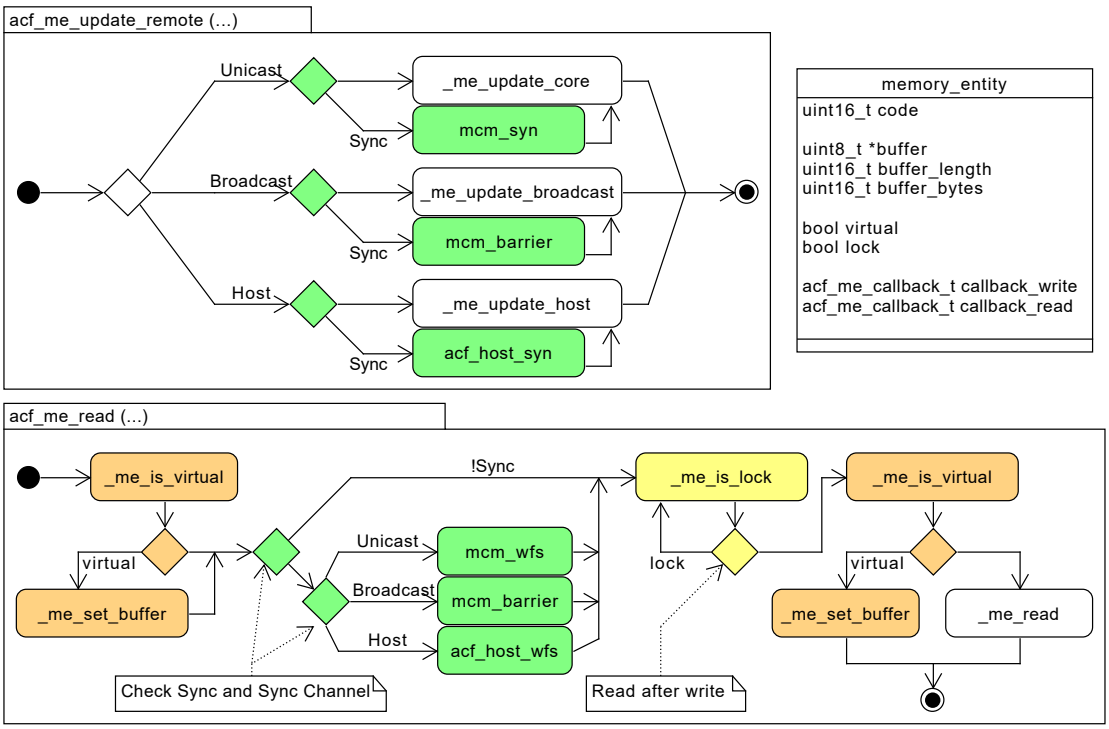


Figure 3.30: ACF - Memory entity read and remote update

When the ACP packet (Figure 3.29) is processed, the following steps are involved:

1. The ACP header is read, and the Command ID is extracted.
2. The ACF library searches the hash table for the callback function coupled with the Command ID.
3. If the callback does not exist the command is ignored, otherwise the function pointer is extracted.
4. The selected callback reads the command header section and executes the command.
5. If required, the selected callback reads the command payload section.
6. If the command requires a reply, an ACP packet is created and appended to the outcome packet queue.

Memory entities

The Application Command Framework provides an abstraction of regions of memory called . A memory entity is an ADT that represents a memory area of maximum

Table 3.6: This table shows the operations on memory entities that can be used through ACP communication channels. The table shows the functions for both on-board ACF runtime and on-host ACF runtime. (1) Supports read-after-write synchronization and virtual memory entities with remote writes on all ACP channels (2) Not yet implemented, requires Host side Memory Entites support. (3) Work in Progress (4) Gather operation, under study

Runtime	Channels	ME Operations			
		Create	Read	Update	Delete
Board	Local	`create`	`read`[1]	`update`	`delete`
	Core	-[3]	-[3]	`update_remote`	-[3]
	Broadcast	-[3]	-[4]	`update_remote`	-[3]
	Host	Requires a specific Host ACF Runtime support			
Host	Core	-	RemoteME::read	RemoteME::update	-
	Broadcast	Requires a specific Board ACF Runtime support			

256 Byte on which it is possible to perform CRUD (Create, Read, Update, Delete) operations. The Update operation can also be performed on Memory Entities belonging to other Application Processors.

Currently, ACP implements the following functionalities on Memory Entities:

As shown in Figure 3.30, the read function of a memory entity allows to set the synchronization on a network channel before reading the data. In this case, there is a synchronization between the core that wants to perform a remote update operation and the core that waits for the data on the memory entity. This mode, also called binding memory entity, allows creating synchronized communication buffers between cores. It is the responsibility of the user to create applications that do not enter a deadlock.

Synchronizations are implemented differently depending on the type of communication channel. In the case of `ACP_CHANNEL_CORE` MCM, Unicast synchronization is used. In the case of `ACP_CHANNEL_BROADCAST` MCM Barrier synchronization is used. In the case of `ACP_CHANNEL_HOST`, an ACF implemented synchronization similar to MCM Unicast synchronization but usable to external devices via ACPoverSDP is used.

Once the synchronization phase is completed, the reading procedure of the memory entity enters the Read after Write lock phase. The processor waits for the remote processor to finish sending data and write it correctly inside the ME buffer.

This double synchronization, Remote Channel Sync and Read After Write Lock,

supports the Virtual Memory Entity feature. Some applications do not require data to remain on the ME, so allocating a buffer to the ME would be a waste of memory and time. It would be necessary to copy twice the buffer during the reading phase, the first time copying the ACP packet coming from outside on the ME buffer and the second time copying the buffer of the ME in the buffer provided during the reading phase.

A virtual Memory Entity is not provided with a physical buffer but requires to associate a buffer every time an attempt is made to read it. In this case, before the sender starts sending the Memory Entity Update command, the receiver must be ready and associate a buffer for storing the data.

In this case, the sender sends an MCM Sync Message, an ACP Syn Message or starts an MCM Sync Message before beginning the transmission according to the chosen communication channel. Only at the end of synchronization will send the memory entity update command.

This feature is useful to create synchronized communication points between processors and will be used by the MPI implementation to implement communication buffers between MPI nodes.

3.3.3 Message Passing Interface

To implement a high-level message passing interface we need some low-level functionalities:

i) An interface for handling the SpiNNaker native multicast communication. ii) A synchronisation system between all computing nodes. iii) A high-level interface to read/write data between nodes.

In order to implement the MPI library for SpiNNaker PCB (SpinMPI) these requirements are satified by two auxiliary libraries, MCM and ACF. As explained in subsection 3.3.1 MCM is an extension of Spin1, the standard application library for Spinnaker. It provides an interface to use the multicast message system in a broadcast way and implements a synchronisation system. Whereas, ACF implements Remote procedure calls, exploiting the Application Command Protocol (ACP) to send and to receive commands between SpiNNaker nodes and from/to External Devices. ACF provides some facilities and built-in commands for sharing memory objects (referred as *Memory Entities*).

The SpinMPI is built over these two layers, in particular, I implemented the receiving buffer as an *ACF Virtual Memory Entity*. Both ACF and SpinMPI have an on-host runtime written in Python (ACF-Runtime and MPI-Runtime).

The goal of this section is to describe the implementation of the MPI programming

model that exploits the SpiNNaker event-driven programming model and the on-board interconnection structure using the MCM and ACF libraries.

The MPI reference implementation, OpenMPI, provides two components: i) `mpicc` is a wrapper of a C compiler that provides the environment variables to include the library files and to link the object files. ii) `mpirun` is the OpenMPI runtime environment that launch and manages the execution of the application over multiple nodes.

Differently, a SpinMPI application is compiled with `spinnaker_tools`, the compiler toolchain provided with the SpiNNaker board. The launcher is the Python package `MPI-Runtime` that loads the compiled application on a set of processors called `MPI-Context`. Moreover, it initializes the application with the context info such as MPI Rank and MPI Communicator via the ACP.

The `MPI-Context` is identified by the number of rings involved and by the number of processors used for each chip: `Context` : ($\texttt{RingID}_{\texttt{MAX}}$, $\texttt{VID}_{\texttt{MAX}}$). Each Ring contains a variable number of chips. For example, Ring 0 contains the chip (0,0), Ring 1 the chips (0,1)-(1,1)-(1,0), Ring 2 the chips (0,2)-(1,2)-(2,2)-(2,1)-(2,0), and so on, as shown in Figure 3.22. Each processor is identified by a vID[3]. For example, for parallelizing a program on 32 processors we can choose a context of $C(1,8)$, $C(3,2)$ or $C(5,1)$, each of them describes a set of 32 processors. When the context is defined, the application is loaded on SpiNNaker.

An MPI application starts with the `MPI_Init(...)` function, whereas on SpiNNaker we need to call the `MPI_Spinn(...)` that initialize the libraries to implement the event driven programming and insert in the scheduler queue the function that contain the MPI application (`mpi_main`). In this way we detach the MPI application from the standard entry point for SpiNNaker application, `c_main()` and include it in a standalone function. A code template is reported in Figure 3.31.

The `MPI_Spinn(...)` is performed in three steps: i) In the callback registration step, the MCM and ACF[4] libraries register several callbacks to manage the incoming packets, the SDP packets coming from the Monitor Processors and the MC packets from the router. In this way, only ACPoverSDP (SDP on port 7) and ACPoverMCM packets are dispatched to the ACF. ii) The second step is to initialise the support for multicast connections. MCM register the routing rules for each possible source and ACF register an internal callback that will be used for execute the `ACPoverMCM` messages. iii) In the last step, the handler that contains the MPI application code

[3]Each processor has a physical ID (location on the die) and a virtual ID assigned when the machine is powered up.

[4]MCM callbacks are registered directly on the Spin1 events, while the ACF callbacks are registered on MCM

```
1  #include "mpi.h"
2
3  ...
4
5  void c_main() {
6    MPI_Spinn(mpi_main);
7  }
8
9  void mpi_main(uint arg1, uint arg2) {
10   MPI_Init(NULL, NULL);
11
12   ...
13
14   MPI_Finalize();
15 }
```

Figure 3.31: SpinMPI code fragment to run the MPI runtime on-board

is scheduled on a low priority event queue before to leave the control to the event scheduler.

Inside the `mpi_main`, the MPI library is initialised through the `MPI_Init(...)` function. The initialization procedure is divided into three phases: i) Initialize ACF `MemoryEntities` that are recorded and will be used to receive from the `MPI-Runtime` informations like the processor Rank and the coordinates (x, y, vID) of all processors involved in the `MPI-Context`. ii) Receive the context information from the runtime and configuration variables for the MCM synchronisation feature. iii) Finally, each processor waits for a signal from the `MPI-Runtime`. Once the `MPI-Runtime` has verified that all the processors have been properly configured, it sends the signal and the `MPI_Init(...)` function ends.

MPI offers two types of communications: Point to Point (`1to1`) and collective (`1to*`, `*to1`, `*to*`). The 1to1 communications have three properties:

- Blocking/Non-Blocking, a blocking function is released only when the data buffer to be sent can be modified, otherwise non-blocking functions are released immediately and the submission is postponed or delegated to a competing thread.
- Synchronous/Asynchronous, synchronous functions (send only) require a receiving acknowledgement from the receiver before considering the communication successful.
- Buffered/Unbuffered, the message before being sent and/or received is copied into a system buffer

The MPI specification defines MPI_Send and MPI_Recv as blocking functions and MPI_Isend and MPI_Irecv as non-blocking functions. These functions can be also Synchronous or Buffered. at the discretion of interface implementer. The user can

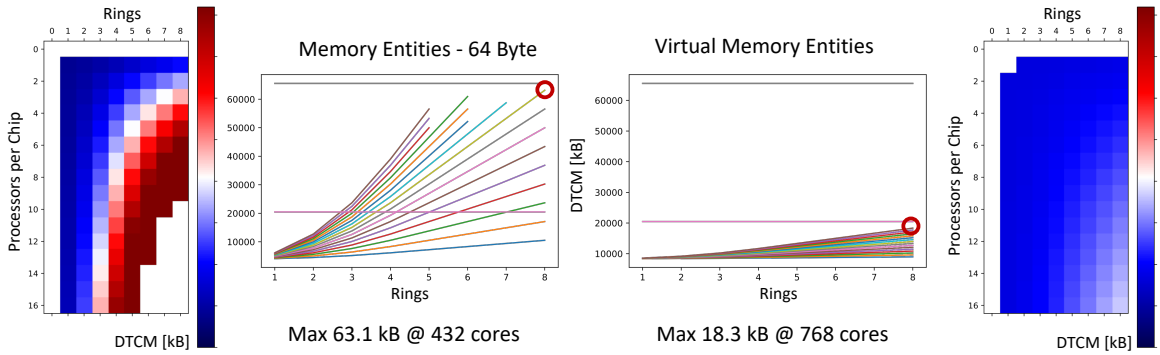


Figure 3.32: Impact of virtual memory entities with respect to 64 Byte memory entities in MPI implementation.

force the use of synchronous functions via MPI_Ssend and MPI_Issend.

On SpiNNaker, the implementation of MPI_Send is blocking, synchronous, and unbuffered, while MPI_Recv is blocking, and unbuffered.

MPI unicast communication functions use MCM for sending the write command and modifying the content of `Virtual Memory Entity` used as communication buffer in the target processor. Specifically, the MPI_Recv buffer is an ACP `Virtual Memory Entity` with a maximum content of 256 Bytes. For sending more than 256 Byte it is necessary to fragment the data and sends fragments individually.

The use of virtual Memory Entites guarantees the possibility to use all 768 available processors in a Spin5. Considering for example the maximum feasible configuration, the virtual memory entities allows the whole board usage with an impact of the whole software stack in DTCM of 18 KiB. Instead allocating 64 Byte for each communication buffer allow a maximum feasible configuration of 432 core, with an impact of the whole software stack in DTCM of 63 KiB, saturating all the memory available for the application Figure 3.32.

The collective communications differ depending on the type of operation and the multiplicity of sender and receiver nodes. In particular, there are three types of operations: synchronism (described in section 3.3.1), data reduction (to be implemented), and data movement (described in the following).

The collective communication functions implemented are as follows:

- `MPI_Bcast` Implements the broadcasting of a message. The entire message is sent simultaneously to all nodes of the communicator. It is equivalent to N `MPI_Recv`.
- `MPI_Scatter` Implements the broadcasting of N messages. Each message is addressed to a different destination. If no aggregated data propagation

mechanism that take advantage of the architecture parallelism is available, this function is equivalent to N `MPI_Send`.

- `MPI_Gather` Implements the reception of N messages. Each message is sent by a different processor. If no aggregated data propagation mechanism to exploit the parallelism of the architecture is available, the function is equivalent to N `MPI_Recv`.
- `MPI_Allgather` Implements an All to All communication, all processors send a message to all other processors. It is equivalent to N `MPI_Gather` or N `MPI_Bcast`.
- `MPI_Barrier` Implements a barrier on which all nodes in the context must synchronize.

The SpinMPI library provide an efficient implementation of MPI_Bcast and MPI_-Allgather that are functions for replicating data on the nodes. This type of functions are implemented using `acp_update_remote(...)` on Broadcast ACP Channel exploiting the MCM broadcast communication protocol.

The MPI_Allgather is implemented as a simple linear cycle where each processor, in turns, perform an MPI_Bcast. This is possible only if all involved processors are synchronised. The MCM library provide automatically syncronization capabilities.

The same MCM syncronization capabilities are also used to implement the `MPI_-Barrier()` function.

3.3.4 ACF - Case Studies

In order to demonstrate the advantages of ACF, I exploited the library to enhance existing applications in the development environment for SpiNNaker (Figure 3.33). Within such applications, I assessed the added value provided by ACP in terms of additional features compared to the existing SCP support.

The first application taken into consideration is the program used to configure the cores before a simulation. I have introduced ACF in the configurator in order to overcome some limitations imposed by the current SCP-based system. The configurator consists of an interpreter of commands executed on each processor of the architecture, that needs a set of op-codes preloaded in memory. In the current system, the entire op-code sequence is introduced into memory using SCP.

With ACF, a configurator is now capable of receiving op-codes at runtime, with a two-fold advantage: i) a reduced use of memory (it is not necessary to store an entire list of op-codes but only a portion) and ii) the parallelisation of the procedure (while sending op-codes to processor B, processor A starts to process its set of op-codes). This application has also been used to profile the performance of the framework.

The second application taken into account is the neuronal model used in SNN simulations. More specifically, I introduced ACF in the implementation of the neuronal model, in order to allow the reconfiguration of some operating parameters of the synapses during the simulation. This was exploited within two different applications: an SNN classifier, and an SNN simulation where I tune the synapse delay parameter.

The classifier SNN is made of two phases, learning and testing. The current SCP-based pipeline requires running a complete simulation in both phases, due to the necessity of using two different neural models, one for the learning and the other for the testing. The introduction of ACF avoids this overhead, allowing a reconfiguration of the neuron's behaviour that makes it usable in both phases.

The SNN simulation consists of a linear sequence of neurons that stimulate each other. Within this application, I evaluated the possibility of either tuning SNN parameters or introducing complex behaviours to the simulation in real-time. More specifically, I introduced ACF in the neural model to allow the modification of the delay of the synapses during the simulation.

ACF for Interactive Data Loading

The benchmark application described in this section is a program used to configure SNN simulations. Our modified configurator runs on the host computer and sends commands to the SpiNNaker Board.

When a simulation runs, one of the very first steps involving the nodes of the board is the *data specification* (DS) phase. DS is one of the most critical phases concerning the time of execution as well as resources management. This phase aims to fill the memory of Application Processors (APs) with the configuration data needed for running a simulation.

Each NMI is equipped with its *data specification generator* (DS-G). The DS-G produces a sequence of commands (op-codes) that together make up the *data specification program* (DS-P). The DS-P is executed by a virtual machine called *data specification executor* (DS-E) which, processing each op-code, configures the memory of the application processor. The DS-E can be performed on the host computer (*DS-E on-host*) or directly on SpiNNaker Board (*DS-E on-board*).

The DS-E on-host version produces a memory image for each AP. All data are sent to the SpiNNaker Board and written in the memory of each involved core. As the full memory image is transmitted in a serial way core-by-core, this implementation does not fully exploit the intrinsic high-parallelism and low-power consumption of the SpiNNaker system. The computational effort of DS-G and DS-E phases remains on the host side.

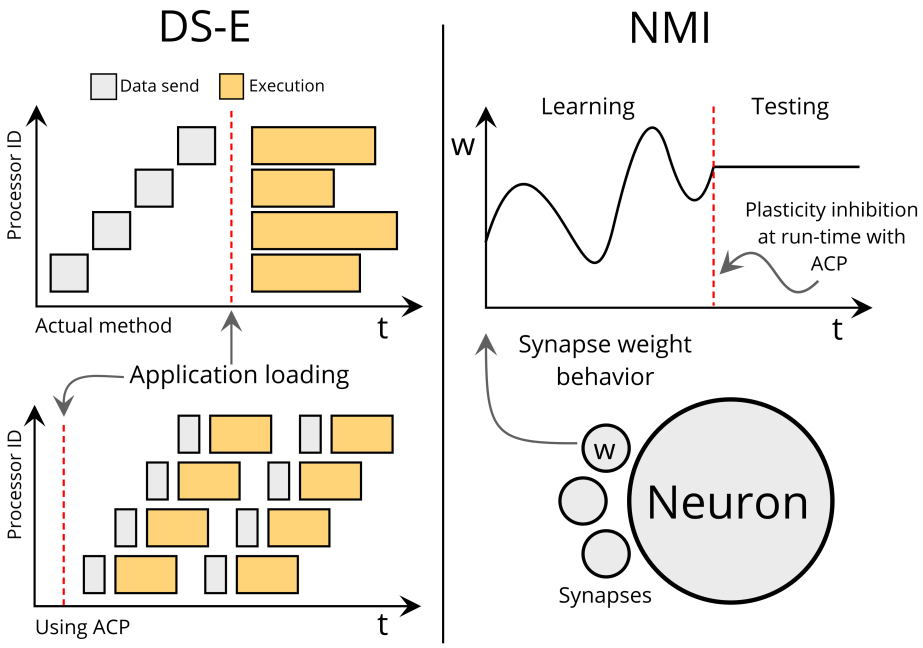


Figure 3.33: Two application improved by the usage of ACF. On the left, we used ACF for implementing a parallel transmission system of op-codes that configure the many processors of the board. On the right, we used ACP for triggering two different synaptic behaviours of a neuron model during an SNN simulation.

A DS-E on-board version is also available and can be executed by SpiNNaker application processors and deployed into each core involved in the simulation. To use this version, the host must first transfer DS-P to the cores. Then, the on-board DS-E will generate the data structures directly on the SpiNNaker memory.

While this implementation avoids the need to transfer core application memory images from the host, still it requires to upload the full DS-P to the core memories. With ACF, we can overcome this limitation, and we obtain two advantages: i) The DS-P can be executed on-the-fly through RPCs sent by the host (*Interactive DS-E On-board*), without requiring its complete transfer, thus saving memory; ii) It allows to exploit platform parallelism, as configuration commands are spread to the cores that can generate their data structures in parallel (*Interleaved Transmission*).

The *interactive DS-E on-board* makes use of the ACP for the transmission of the *data specification program*. I use this application to evaluate the performance and the reliability of the ACP over SDP implementation counting the number of packets lost as a function of the *Packet Delivery Delay Time* (t_{pdd}), that is the time elapsed between the transmission of two packets. This delay between packets avoids overloading the monitor processors involved in the transmission, thus improving the reliability of host-to-board data transmission.

I used as *testcase* the configuration of a biologically inspired SNN designed by Potjans et al. [70] and implementing the four layers constituting the human brain Cortical Microcircuit (CM). I scaled-down the number of neurons and synapses to 10% of the original network (CM_{10}) to satisfy time and resources constraints for fitting the SNN simulation in a single SpiNN-5. The SpiNNaker software maps the CM_{10} SNN on 240 cores distributed among 15 chips of a single SpiNNaker Board.

The Figure 3.34 reports the per-processor distribution of memory requirements of this application when SCP is used to load the whole data specification programs. As it can be easily gathered from the plot, the distribution of DS-P size is very heterogeneous, and a significant amount of the overall data transmission have very large memory requirements (>1 MB). On the other hand, using ACP for the same application, I obtained data transmission always happening in chunks of small fixed size (1 kB). Hence, ACP reduces the memory footprint by 90% at least. This has a positive impact on the memory access time, as it allows to leverage the fast DTCM memory (64 kB upper-bound). On top of that using the ACP the overall CM_{10} configuration time is reduced from 213 to 190 seconds. As described in [4] a CM_{100} could require even ten hours of configuration times against few minutes of simulation runtime, hence, using ACP has a significant impact on the overall simulation efficiency.

The ACP packets encapsulate the DS op-codes inside an ACP over SDP packet and are transmitted using two techniques: *Serial* and *Interleaved*. In the *Serial* transmission I consecutively send all packets directed to a processor before changing the destination. Whereas, in the *Interleaved* transmission I change the destination for each packet so that I never send two consecutive packets to the same chip. I run both *Serial* and *Interleaved* transmission 20 times, one per each t_{pdd} value, in a 50 µs to 500 µs range using a 50 µs step.

During these runs, I counted how many APs completed the configuration and how many packets were lost. The loss of a packet is confirmed by the processor when the application receives a packet with an unexpected sequence number. On the Monitor Processor side, I counted the occurrences of . An SWE is the error triggered when the Root Processor saturates its message queue. If this event occurs during reception by a local AP, the local AP must handle the exception. If the event occurs while receiving a packet from a remote entity, the packet is lost. When too many SWEs occur, the Root Processor enters in the status and it stops, making the SpiNNaker Board unreachable by the host computer. During our evaluation of the results, I considered the *(RTE)* a critical failure for the test. Conversely, I considered successful those tests terminated with SWE counter equal to zero.

The test environment consists of a host computer, equipped with an Intel Core i5-4670 @ 3.40GHz, 4 GB DDR3-RAM and running GNU/Linux Debian 8 (Jessie) distribution, a Gigabit Ethernet Switch and a SpiNN-5.

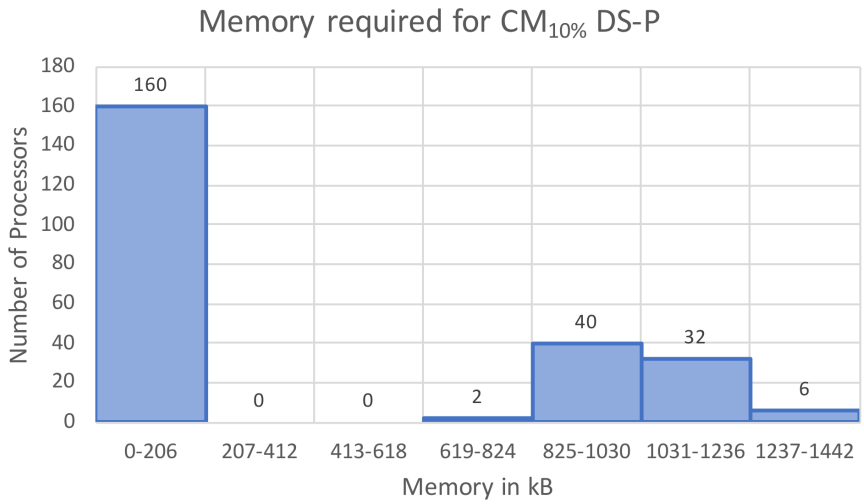


Figure 3.34: DS-P Memory Footprint. Per-processor memory usage of Data Specification Program leveraging SCP.

SERIAL

t_{pdd} [µs]	Configuration Packets		Root Processor		APs	
	Missed / Total	Class of 9s	SWE	RTE	FINISH	WAIT
50	*	*	*	100%	*	*
100	20 661 / 9 561 640	2	39 186	0%	63.9%	36.1%
150	1 570 / 9 561 640	3	28 273	0%	91.2%	8.8%
200	122 / 7 171 230	4	91	25%	96.7%	3.3%
250	71 / 9 083 558	5	96	5%	98.8%	1.2%
300	22 / 9 561 640	5	0	0%	99.5%	0.5%
350	13 / 9 561 640	5	0	0%	99.7%	0.3%
400	15 / 9 561 640	5	0	0%	99.6%	0.4%
450	9 / 9 561 640	6	0	0%	99.7%	0.3%
500	17 / 9 561 640	5	0	0%	99.6%	0.4%

INTERLEAVED

t_{pdd} [µs]	Configuration Packets		Root Processor		APs	
	Missed / Total	Class of 9s	SWE	RTE	FINISH	WAIT
50	136 089 / 956 164	0	131 070	90%	43.0%	57.0%
100	6 447 / 9 561 640	3	6 100	0%	57.1%	42.9%
150	314 / 9 561 640	4	179	0%	93.4%	6.6%
200	121 / 9 083 558	4	29	5%	97.3%	2.7%
250	19 / 9 561 640	5	1	0%	99.6%	0.4%
300	12 / 9 561 640	5	0	0%	99.7%	0.3%
350	26 / 9 561 640	5	0	0%	99.4%	0.6%
400	26 / 9 561 640	5	0	0%	99.4%	0.6%
450	9 / 9 561 640	6	0	0%	99.8%	0.2%
500	11 / 9 561 640	5	0	0%	99.8%	0.2%

Table 3.7: Test results At variation of t_{pdd} these tables describe: i) The number of ACP packets lost and the relative class of 9s ($number\ of\ 9s\ in\ 1 - missed/total$). ii) The status of Root Monitor Processors in terms of Software error and Runtime exceptions. iii) The status of Application processors in term of percentage of them that receive all own packets.

In the first set of tests, I implemented a serial transmission for stressing the Application Processors and the destination Monitor Processor detecting the minimum t_{pdd} that guarantees a reliable transmission without any loss of packets. ACP over SDP packets are transmitted sequentially from the Root Node to the Application Processor until the configuration packets are consumed.

Table 3.7.SERIAL summarises the results of these stress tests. The table refers to all 20 simulations for each t_{pdd} chosen. I detected critical failure conditions (RTE > 0) when t_{pdd} is in the range of 50-250 µs. In the case of t_{pdd} time equal to 50 µs, it

was not possible to obtain any data because all the twenty simulations failed. This critical failure is due to a chain of events starting with the saturation of the shared message box between MP and AP that creates a deadlock condition overloading the MP in charge of dispatching the ACP packets. Considering t_{pdd} values above 250 µs the system can configure the 99.999% of the APs (reliability class of 9s equal to 5). From the results, I obtained a t_{pdd} of 300 µs to avoiding the saturation of buffers in Application Processors.

I designed this benchmark to find the minimum threshold of t_{pdd} time that guarantees the correct transmission of all configuration packets from the Root Processor.

In this scenario I avoided the overloading of the target MP, stressing only the Root Processor. The host computer forwards configuration packets so that two consecutive packets are never sent to the same chip. For example, if I want to configure the AP-1 of the Chip-0-0, the AP-2 of the Chip-0-1 and the AP-3 of the Chip-1-0 respectively, the procedure works as follows:

1. ACP over SDP Packet-1-1 is sent to the MP of the Chip-0-0 and forwarded to the AP-1.
2. After a waiting time equal to t_{pdd}, the Packet-2-1 is sent from the Root Processor to the MP of the Chip-0-1 and forwarded to the AP-2.
3. The Packet-3-1 reaches the MP of the Chip-1-0 and is forwarded to the AP-3.
4. The Packet-1-2 is sent to the MP of the Chip-0-0 for the AP-1.

This mechanism of interleaved transmission continues until the configuration is complete. In the final phase of the transmission, if the cores to be configured lie in a single chip the packet delivering delay is increased using a safety threshold of 1 ms, in order to avoid saturation of the target MP involved in the configuration of the last cores. Using this technique, I prevent the burst transmission of packets to the same SpiNNaker Chip, giving to the MP and to the APs a sufficient amount of time to reconstruct and process ACP packets.

The use of interleaved sending of configuration packets made it possible to analyse the limits of the Root Processor previously masked by the errors generated by the MPs of the target chips.

In Table 3.7.INTERLEAVED I reported the results of this test. I identified the critical failure condition (for all the 20 repetitions) that occurred when I imposed a delay time t_{pdd} of 50 µs and a single critical event for t_{pdd} equal to 200 µs. In the t_{pdd} range of 100 µs to 250 µs I detected several SWEs that indicate the lower bound imposed by the limits of the hardware component involved in the delivering process of packets inside the SpiNNaker Board. I identify the cause of this issue to a limitation of the Root Processor used to fragment the SDP packets into PP

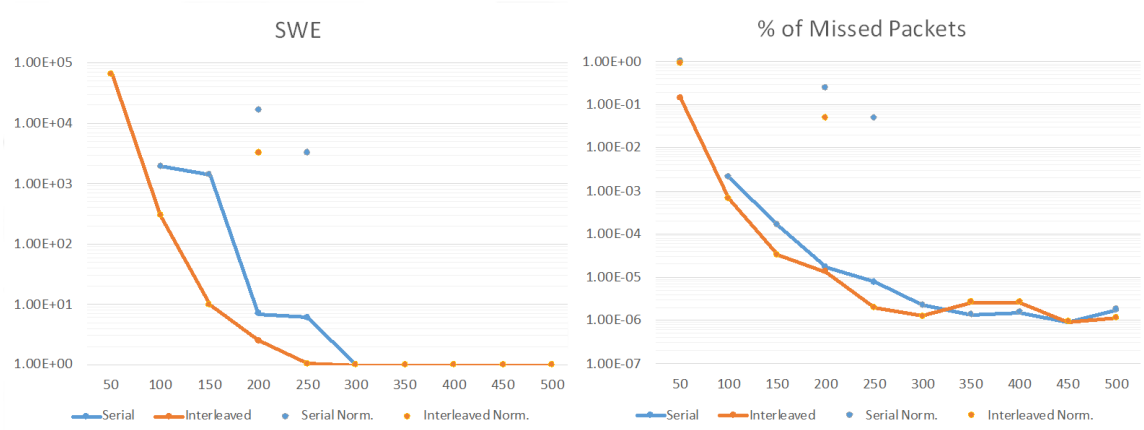


Figure 3.35: Software Errors and percentage of missed packets for serial and interleaved transmission. The plot on the left shows SWE at increasing values of t_{pdd} (µs). The lone points are values inserted to take RTE into account (the y-axis is in logarithmic scale, all values are incremented by 1 to avoid zeros). The plot on the right shows the number of missed packets at increasing values of t_{pdd} (µs). The lone points are values inserted to take RTE into account (the y-axis is in logarithmic scale and represents a percentage).

packets. The system can configure the 99.999% of APs with a reliability class of 9s equal to 5 using values of t_{pdd} higher than 200 µs.

The plots in Figure 3.35 show the occurrence of SWE when the t_{pdd} is increasing. We note a higher number of SWE when ACP over SDP packets are transmitted with the Serial method, thus validating our hypothesis that the interleaved transmission is a valid solution to the issues related to HW and time limits of target chips having to reconstruct and interpret incoming ACP packets. In the interleaved transmission the Root Node is responsible for the generation of SWEs.

I consider as successful transmissions those without errors. A transmission of this type can be detected for $t_{pdd} \geq 250$ µs for interleaved and $t_{pdd} \geq 300$ µs for serial loading. I highlighted, in the chart of Figure 3.35, the presence of RTE using outliers points. This condition happens at 200 µs for the Interleaved transmission and at 250 µs for the Serial transmission.

The missing packets counted at the target have a trend similar to SWE (see Figure 3.35). The number of missing packets reduce to a value near to 1.0E-06 for $t_{pdd} \geq 250$ µs in the Interleaved transmission and for $t_{pdd} \geq 300$ µs in the Serial transmission. Even in this situation, we can observe the same crash events represented as single points in the figure.

In summary, we identify an ideal delay time of 300 µs for the Serial transmission

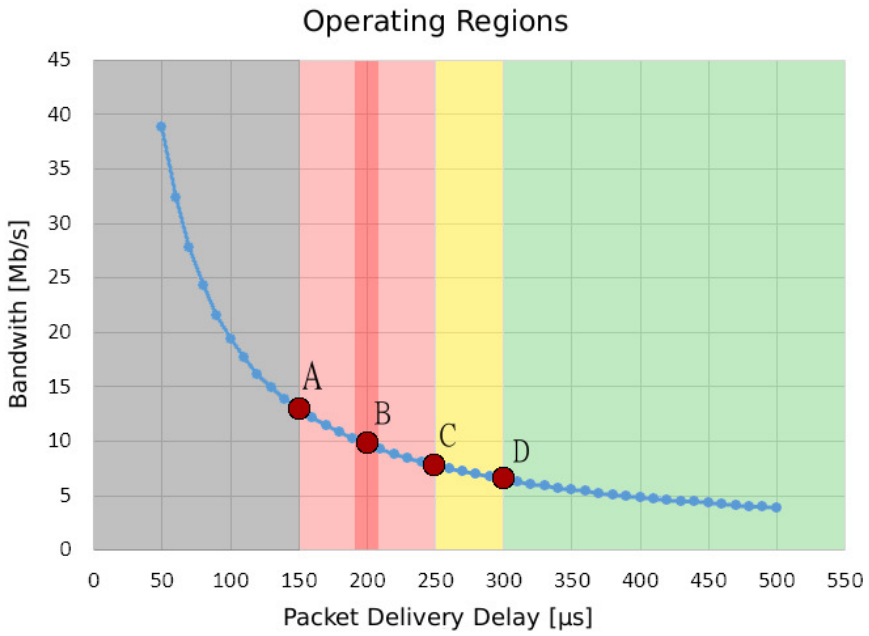


Figure 3.36: This plot depicts trend of communication bandwidth when t_{pdd} (µs) is increasing. The different operating zones are represented with different colours. Red points represent values of four different boundaries. A is the last recommended value with the interleaved sender: 13 Mbit/s. C is the last recommended value with the serial sender: 7.5 Mbit/s. D is a secure value that works for all senders: 6.5 Mbit/s. In B we get an unusual amount of RTE.

and of 250 µs for the Interleaved transmission.

In Figure 3.36 we can identify four t_{pdd} operating zones: i) Green area: no problems encountered, all packets are correctly transmitted; ii) Yellow area: only Interleaved transmission can terminate without error; iii) Red area: detected some issues, acceptable only when using Interleaved loading; iv) Below 150 µs (grey area) the system is unstable. The bandwidth profile shows the throughput for each operating zone: For yellow and green operating zones we reach values between 6 and 8 Mbit per second. Whereas, the red operating zone allows a bandwidth between 8 and 13 Mbit per second.

In order to load the DS-P opcodes and exploit the multicast network with MCM (ACP over MCM) I have identified two possible approaches. Both involve the use of a functionality to be developed in the application in order to transform ACP over SDP communications into ACP over MCM operations.

The first approach involves the use of the `ACP_CHANNEL_BROADCAST` communication channel and an on-host op-code compression system through a sequence alignment

algorithm (Multiple Sequence Alignment, MSA). The idea is to identify for each DS-P groups of opcodes in common, in order to generate a single opcode stream to broadcast on the architecture. The consensus zones between the sequences define the match groups. Finally, a single sequence is created to which each opcode is associated with a number that identifies the match group. When a processor receives an opcode, it checks if it belongs to the specified match group. If so, it will accept the opcode, otherwise it will ignore it. To convert the ACP over SDP channel to ACP over MCM I have set the sequence sending from the host computer to the SpiNNaker processor x=0 y=0 p=1, which will forward it immediately via a Memory Entity Update command on the Broadcast channel when it receives the ACP packet.

The second approach does not involve channel compression via Multiple Sequence Allignment but takes advantage of the parallelism of the sixteen Application Processors of the Root Chip (Chip x=0 y=0). All sixteen processors (which we will call Configurators) are programmed to receive an ACP over SDP and redirect it to the `ACP_CHANNEL_CORE` communication channel (unicast MCM channel) through a Memory Entity Update. Each processor can forward the packet to a subset of other processors, so there is no chance to create hotspots on the Router network. The host computer will forward the packets in Interleaved mode, these will reach the Configurators that will forward them in parallel on the multicast network.

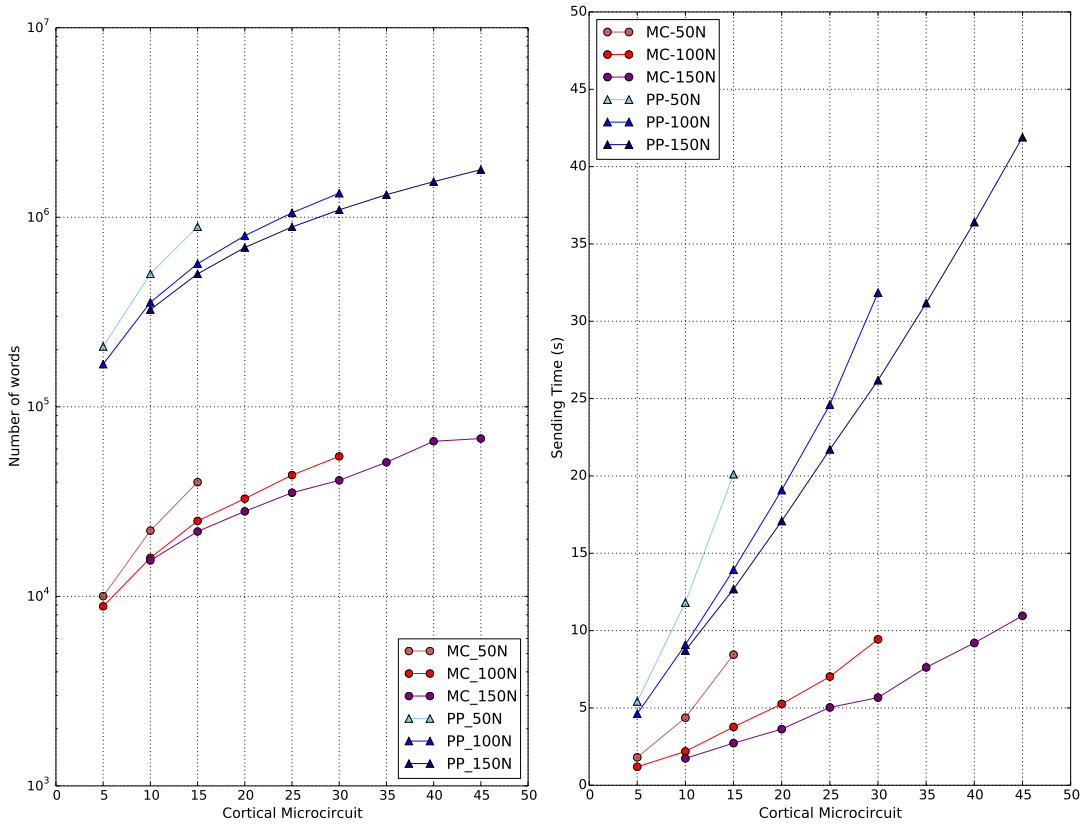
The benchmarks for the two phases of the first version of the new Data Upload protocol (the MSA and the Broadcast sending phase) are shown in Figure 3.37.

The two images show the results of multiple executions of the Cortical Microcircuit network using different scaling factors. The distributions of neurons per core tested are 50, 100 and 150 respectively. Different scalings for the network have been tested (from 5% up to 45%). All the simulations performed fit inside a SpiNN-5 board.

In the two pictures are presented both the situation where the MC Data Upload protocol is used and the one where the normal ACP over SDP data flow has been chosen.

The plot in Figure 3.37a presents the total number of words of the stream to be sent to the SpiNNaker system both with and without the MSA compression step. The y-axis is in logarithmic scale. It is possible to notice that the number of generated words differs of more than one order of magnitude between the two cases for all the three scaling.

By increasing the simulated percentage (x-axis), it is possible to see that the gap between these two approaches grows. This is visible especially for the case with 150 neurons per core (dark blue and purple lines in the charts). In this case, indeed, the number of packets generated without the MSA for a simulation at 10% is 13 times greater than the one generated if MSA is used. By scaling this network up to



(a) (b)

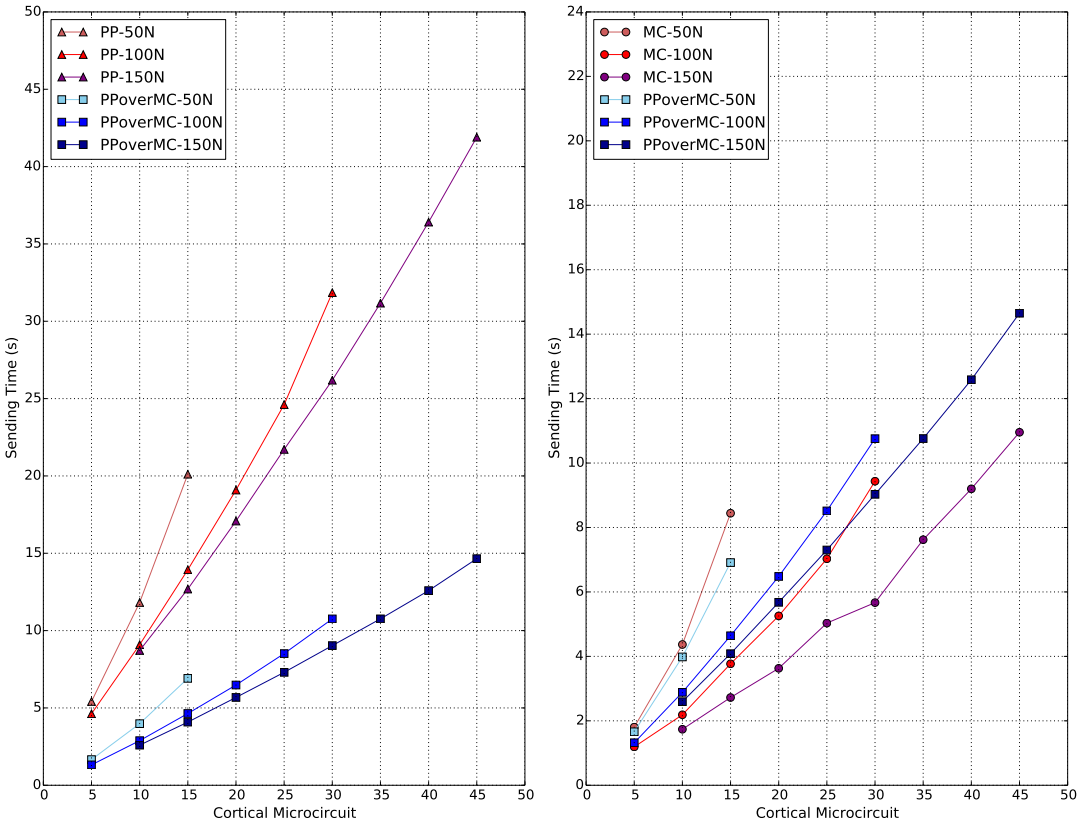
Figure 3.37: Figure 3.37a Comparison between the number of words before and after the MSA compression Figure 3.37b Comparison between sending times using the ACP over SDP protocol and the ACP over MCM

the 45%(maximum percentage fitting inside a SpiNN-5), the stream without MSA becomes 30 times greater than the one after the Alignment operation. Similar values are generated for the other values of neurons per core (from 20 to 23 times is the gain with 50 neurons and from 17 to 30 times with 100).

With reference to Figure 3.37a, it is possible to say that the MSA step brought, as foreseen, good results in this type of data compression.

The results of the data upload phase are shown in Figure 3.37b. Both the axes are in linear scale. In this case, the y-axis represents the sending times measured in seconds. The color scheme is identical to the one used in Figure 3.37a and the simulations performed are the same.

The Point-to-Point sending times grow exponentially with the percentage of the Cortical Microcircuit simulated. The same behaviour can be noticed for all the three different distributions of neurons per core. This is due to the fact that,



(a) (b)

Figure 3.38: Figure 3.38a Comparison between sending times using the ACP over SDP and ACP over MCM-Unicast protocols. Figure 3.38b Comparison between sending times using the ACP over MCM-Broadcast and ACP over MCM-Unicast protocols.

increasing the scaling, the simulated network grows and, for this reason, the stream generated for each core becomes bigger.

On the other hand, by using the Multicast protocol for sending compressed data, only a linear increase in sending times is obtained. The most significant gain can be seen for the simulation at 45% using 150 neurons per core, in which, the time required for sending all the stream in Point-to-Point is 61 seconds while, using the Multicast approach, all the board is filled in only 12 seconds.

However, there is a drawback in this Data Upload protocol: the alignment times are not negligible and they negatively impact the performances of this approach highly increasing the total transmission time.

The obtained results are nevertheless significant because they show that improvements are possible if the MSA tool is correctly addressed for example, to launch

the same simulation several times on the board, or as we will see later allow you to modify some parameters of the simulation at runtime..

In order to prove the efficiency of the new data configuration protocol, the same tests executed for the previus approach have been run for the Unicast version as well. The results are shown in Figure 3.38.

Figure 3.38a represents a comparison between the configuration phase performed using the ACP over MCM protocol and the standard ACP over SDP version, while Figure 3.38b shows its performances with respect to the transmission of the aligned commands in broadcast.

In the first case, the gain is evident. It is worth noticing that, besides having lower values, the curve indicating the upload times of the ACP over MCM-Unicast has a linear behaviour instead of the exponential one that can be seen for the ACP over SDP approach. These improvements are due to the fact that the transmission is now implemented on the Multicast network, although the packets have a single destination. This allows not to use the Monitor processors as intermediaries and to communicate directly with the destinations.

Another feature that increases the speed of the data transfer is the use of the whole Ethernet chip as configurator. This approach grants that the load is distributed over a whole chip instead on a single core, giving the chance to send packets to other configurators while one is receiving and manipulating.

During the performed tests, as can be seen in Figure 3.38a, this new protocol allowed to send data up to 3 times faster than the standard ACP over SDP version. Again, by increasing the scaling of the network, the gain grows as well. To prove this point, with a 5% Cortical Microcircuit the sending time is 2.5 times lower than the standard PP, while when using a 45% scaling factor it becomes 3 times lower. This can be generalised for all the three distributions of neurons simulated as shown by the graph.

This new protocol is moreover able to reach performances really close to the Multicast one, as shown in Figure 3.38b.

The case in which the Cortical Microcircuit simulated had 50 neurons per core is slightly different from the other two. This is because the length of the streams of data are quite small and the groups sending phase has a higher impact on the total time of the transmission.

By increasing the number of neurons per core and the scaling it can be noticed that both the MCM-Broadcast and the MCM-Unicast transmission times grow linearly (and in the worst case scenario the MCM-Unicast is only 4 seconds slower).

The reasons why the new protocol is so close to the Multicast version in terms of

performances are that it does not need the group definition phase which is computationally expensive and, besides, packets are sent by interleaving the cores on the Ethernet chip.

The assigning phase between configurators and Application processors in MCM-Unicast is performed through a single ACP packet, while for sending the groups in the MC approach, it is necessary to fragment the information in several datagrams and to repeat this operation for all the cores, instead of setting only the Ethernet chip.

Furthermore, the MCM-Broadcast version needs to send SDP packets containing the stream to the Monitor processor of the Ethernet chip that will forward the ACP datagrams to the destination groups. In order not to lose packets, it is necessary to insert a *Throttling time* between two SDPs in such a way this Monitor core has the time to receive and forward the data. A similar delay is present for MCM-Unicast approach as well, but it is lower because, while a configurator is receiving and forwarding, the host software is sending packets to another one.

What makes this results so interesting is that the total time to upload the board for the MCM-Broadcast version is given by the sum of the Alignment time and the data upload phase, while for the MCM-Unicast version no additional steps are required, resulting in a final time lower than the Multicast version.

ACP for SNN Applications Reconfiguration

In this section, I describe a test designed to highlight the capability of the ACP when used for reconfiguring the application parameters at runtime during the simulation. For this test, I selected two different networks, where the neuron model was modified to support the features defined in the ACP.

The first network is a bio-inspired SNN for multivariate classification designed by Schmuker et al. [82]. This SNN is inspired by the chemical sense of insects evolved to encode and classify odorants in the natural environment. Schmuker et al. [82], based on these insights, developed a computational method to encode, process, and classify handwritten numbers.

The second network is an SNN composed of a chain of neurons that stimulate each other (Synfire Chain) [2]. The speed of propagation of neuronal stimuli (spikes) depends on the synaptic delay of the individual synapses that connect them.

The SNN-Classifier has three functional layers. In the first layer, the original stimulus space is sampled by Virtual Receptors (VRs) which respond proportionally to the data input proximity, thus encoding the stimulus using cone-shaped radial basis functions with large overlapping receptive fields. The centroids of the basis

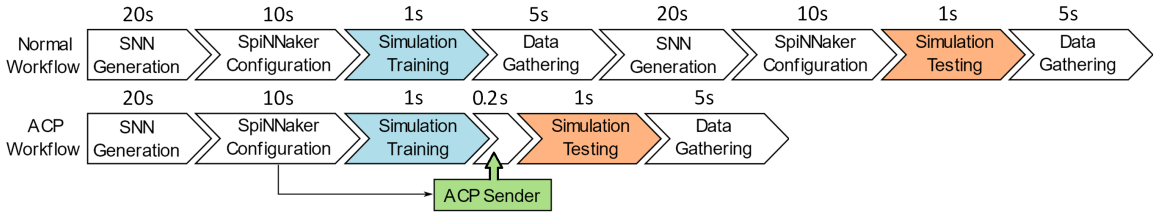


Figure 3.39: Two different workflows to perform training and testing phases with the SNN classifier. Without the ACP the network uses the workflow to the top, the training and testing phases are two different simulations. With the ACP usage, the network uses the workflow to the bottom, the training and testing phases are in the same simulations.

functions (the VR points) were placed using the neural gas algorithm [56], a self-organising process to map the feature space described by a picture data set. In the second layer, the lateral inhibition decorrelated the signals from the VRs. Signals from Virtual Receptors (in the form of firing rates) reach the Receptor Neurons (RNs) modelled as Integrate and Fire (IF) neurons.

Each population of RNs excites a population of *Projection Neurons (PNs)*, which in turn send their spikes to one population of *Local Inhibitory neurons (LINs)*. Each LIN population sends inhibitory projections to all other PN populations in the second layer, exerting lateral inhibition, reducing the correlation between VR channels and scattering the representation of the multi-dimensional pattern. Signal decorrelation during the second layer significantly increases the classification accuracy. Finally, in the third layer, olfactory scent perception is modelled by a machine learning classifier able to classify the input data linearly.

The synapses with plasticity model are situated between the second and the third layer and are connected with a set of neurons that have the functionality of trainers since their signal selectively stimulates the synapses associated with the class to learn at a given instant.

The execution of the SNN classifier is divided into two execution phases: the training phase and the testing phase. During the training phase, the network is built with plastic synapses, and the trainer neurons are configured to emit a spike so that the submitted sample (the samples are presented for 200 ms) is coupled with the desired class. With this implementation, in the absence of the trainer neurons spikes, the plastic synapses deviate from the learned weights, causing the incorrect behaviour of the network. Hence, it is not possible to perform the test phase in the same simulation of the training.

Another network configuration is required to solve this issue. At the end of the Training phase, all the learned weights are downloaded from the board and used to

rebuild the classifier using static synapses neuron. By re-running this new network, it is possible to evaluate the classification performance of the network.

I show the workflow on the top of Figure 3.39, where all the operations to simulate an SNN classifier, require about 72 s. This time is the overall period required to perform two generation phases (20 s each), two board configuration phases (10 s each), two simulation phases (1 s each), a download phase for recovering trained synapses weights (5s), and a download phase for collecting classification results (5s).

I implemented the ACP inside the neuron application to improve the overall simulation time. The application uses an ad-hoc command to change the synaptic plasticity behaviour. On the host side, an application (the Sender) making use of the SpynnakerACF library, is in charge of sending all processors the command for disabling the learning capabilities of plastic synapses. The Sender is executed when the simulation is ready to be run on the board.

In this way, it is possible to perform both Training and Testing phases with a single simulation and the usage of the ACP Sender application configured to inhibit the plastic synapses after 2 s from the start of the simulation. By doing so, both the phases of the classifier are performed in just about 38 s (2.2 s of simulation). I added 200 ms between learning and testing phases to give time to the ACP Sender to propagate packets to all the cores. This workflow is depicted in Figure 3.39.

ACP Sender transmits with a Packet Delivery Delay time, t_{pdd}, set to 200 µs, allowing a safe transmission of the switching packet to all 864 cores of a SpiNNaker Board in about 170 ms. During the tests, I stressed the system running 12 instances of classification network at the same time. As a result, I correctly sent all the ACPoverSDP packets to all 864 cores involved in the simulation. The router overload resulted in some missed MC packets, representing the spikes during the simulation, but this did not impact on the performance of the network and the classification results.

The significant advantage provided by the ACP embedded in the classifier SNN is the possibility to run both training and test phases without reconfiguring the board.

The second application based on SNN is the Synfire Chain. The network is composed of a long sequence of neurons linked together by a single synapse. Currently, the simulation once configured and started does not allow to modify any parameter. Unless making usage of synapse models whose weights vary autonomously, the only way to modify the parameters of the neuronal model such as weight and synaptic delay is to know the position in RAM of the data regions related to each neuron and through SCP commands directly modify the content of the memory.

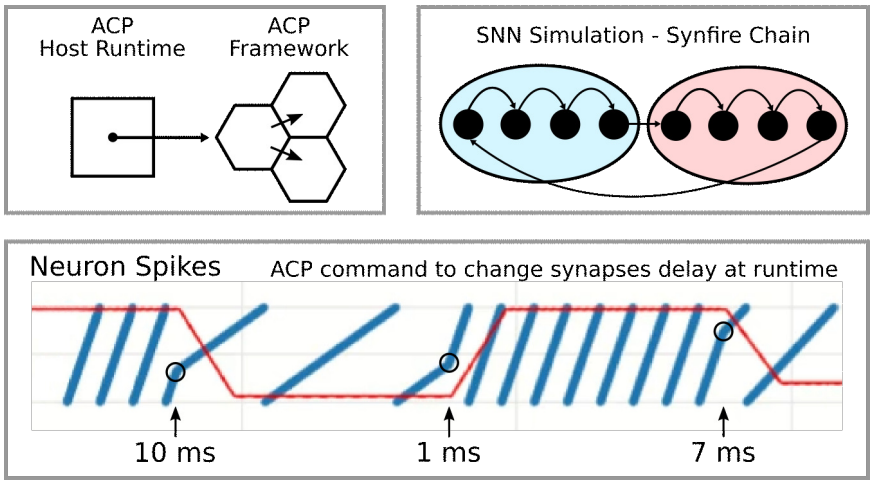


Figure 3.40: ACP reconfiguration of neuronal parameters. Above: the test configuration and the benchmark network used. Below: the graph of the spikes emitted by the neurons (blue), mean network activity (red). The series of spikes changes slope after receiving the synaptic delays reconfiguration commands at 10 ms, 1 ms and 7 ms.

I used the memory entities provided by the ACP framework to manage a set of parameters as modifiers of the actual model parameters, without modifying the whole configuration plan of the neural application. By doing so, I were able to modify the delay of the synapses in real-time, during the simulation of the synfire chain. Figure 3.40 depicts a scheme of SNN, the ACP components involved and the timeline of an SNN simulation. The user can dynamically change the corresponding parameters (e.g. synapse delay) using the ACF runtime library to send commands to the SpiNNaker cores to modify the memory content exploiting Memory Entities support.

Figure 3.40 shows the trend of the network spike series (blue lines) and the average synaptic activity of the whole network (red line). Circles are used to highlight the points at which the commands are triggered to modify the memory entities, which affects the synaptic delay between the neurons. The slope of the spike series increases with a minimum delay (1 ms) and decreases when the synaptic delay is set to a higher level (10 and 7 ms).

This case study demonstrates that ACP framework, by allowing runtime reconfigurations, can be used for effective host-controlled SNN parameters exploration.

3.3.5 MPI - Case Studies

To test the implementation of MPI on SpiNNaker, developed using the functionality provided by ACF and MCM, I used two different applications that use MPI to parallelize some of their functions. The first is a simple implementation of a Verlet integration (in particular Velocity Verlet) to solve the dynamics of an N-Body problem. The second is the distribution of genetic queries to implement a simple genetic aligner distributed on SpiNNaker processors. Using these two benchmarks I evaluated the scalability and performance of the library. In addition, I demonstrated that SpiNNaker can be easily used using a parallel programming model outside the neuromorphic context.

N-Body Simulation

The NBody simulation consists of N particles each with a position in a D-dimensional space $\vec{x}^p \in \mathbb{R}^D$ and with a mass m_p. Integration of motion equations with Velocity Verlet is discretized by step equal to $\tau = \Delta t$ and consists of three equations to be solved for each particle p: i) Equation 3.10: position update, ii) Equation 3.11: calculation of forces due to gravitational interaction iii) Equation 3.12: velocity update.

$$\vec{x}^p_{t+1} = \vec{x}^p_t + \vec{v}^p_t \tau + \frac{1}{2}\vec{a}^p_t \tau^2 \tag{3.10}$$

$$\begin{aligned} \vec{a}^p_{t+1} &= \frac{1}{m_p}\vec{F}^p_{t+1} = \frac{1}{m_p}\sum_i \vec{F}^{i,p}_{t+1} \\ &= \frac{1}{m_p}\sum_i -G\frac{m_i m_p}{|\vec{x}^p_{t+1} - \vec{x}^i_{t+1}|^3}(\vec{x}^p_{t+1} - \vec{x}^i_{t+1}) \end{aligned} \tag{3.11}$$

$$\vec{v}^p_{t+1} = \vec{v}^p_t + \frac{1}{2}(\vec{a}^p_t + \vec{a}^p_{t+1})\tau \tag{3.12}$$

The NBody simulation was parallelized with MPI by distributing the particles, equally, on each computation node. Each node will have to update only the positions and speed of its particles, bringing complexity from $O(N^2)$ to $O(\frac{N}{P}N)$ where P is the number of processors.

To compute the force exerted on a particle p each node must know the position of each particle of the system. At each iteration a particle position update step is then performed by the function `MPI_Allgather(···)`. All calculations were executed in fixed points.

I implemented `MPI_Allgather(···)` with a broadcast transfer mediated by MC packets. The asymptotic complexity of communication goes from $O(P^2)$ to $O(P)$

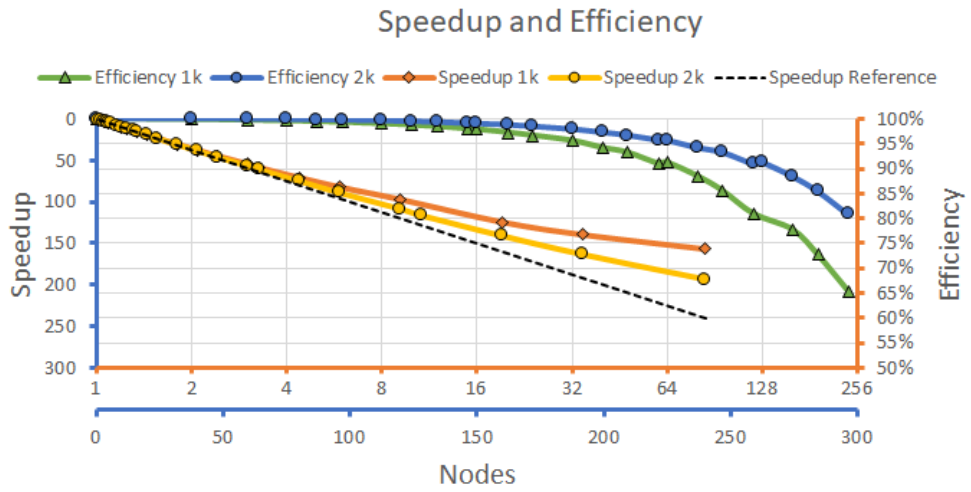


Figure 3.41: Speedup on blue axes and efficiency on orange axes measured for two simulation sizes, 1 k and 2 k particles.

as data replication is done in parallel by architecture routers. We evaluate the performances of the implementation in terms of speed-up χ_n and efficiency η_n as the number of used computational nodes increases.

$$\chi_n = \frac{\Delta T_1}{\Delta T_n} \tag{3.13}$$

$$\eta_n = \frac{\chi_n}{n} \tag{3.14}$$

I performed two series of simulations, with 1 k and 2 k particles, in order of analysing the impact on the efficiency when the number of particles to be calculated for each node is increasing. Moreover, I increased the number of processors for each simulation series from 1 to 240, in order of evaluating the scalability of the MPI implementation.

As shown in Figure 3.41 the results show good scalability performances, comparable with state-of-the-art implementations on parallel computing platforms [18]. The speed-up is directly proportional to the number of cores until 100 nodes, and reach 194 x when 240 nodes are used to simulate 2k particles (156 x for 1 k particles).

The efficiency stays above 90% with 64 processors for the 1 k simulation and up to 128 processors for the 2 k simulation. With 240 processors, I obtained an efficiency of 65% for the simulation with 1 k particles and more than 80% for the simulation with 2 k particles. With this results we can speculate and hypothesize that is convenient to distribute the problem on additional processors.

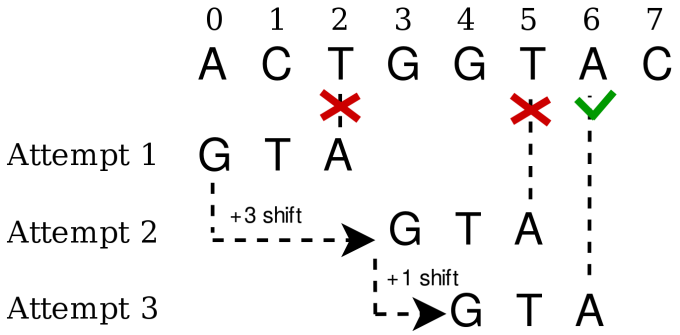


Figure 3.42: Intuition of the Boyer-Moore search procedure.

Results show that the considered neuromorphic architecture with the proposed MPI library is a promising solution for accelerating communication intensive applications.

The DNA Pattern Matching Algorithm

One of the most recurrent and widely studied problems in computer science is pattern matching. This problem has several real-world applications such as fast sub-string searching for network intrusion detection, mail spam filters, protein motif search and DNA/RNA sequence alignments [86]. Given a text string T of length n and a pattern string P of length $m \leq n$, the pattern matching problem can be stated as retrieving all positions i where pattern P occurs in text T, such that $0 \leq i \leq n - m$.

A straightforward solution for the pattern matching problem consists of looking for the pattern sequence in the text position by position until every occurrence is found. Unfortunately, such approach leads to a $O(m \cdot n)$ asymptotic complexity, which is not acceptable for large sets of data.

Given the practical relevance of this problem, many approaches were proposed in the literature for improving the naïve way. One of these is the *Boyer-Moore* algorithm [16, 39], which trades space usage for time efficiency, defining rules for pruning the search space avoiding the exploration of all text positions.

Figure 3.42 provides an intuition for this approach; given the text in the picture, the first attempt looks for pattern "GTA" in position ⓪, which is not correct.

The naïve approach would perform the next search from position ①, but this is not ideal since the first instance of the letter "G" in the pattern occurs at position ③ in the text, meaning that searching any position in the middle is useless. Implementing this optimisation requires pre-processing of the pattern to be matched; a *shift table* is computed, storing the number of text positions that can be safely skipped for

each symbol in the target alphabet. Whenever a mismatch is found, given the next symbol to be searched, the *shift table* is accessed and the next position to be considered is computed.

I used a refined version of the *Boyer-Moore* algorithm, also known as *Fast string matching method for Encoded DNA sequences (*FED*)* [48], which takes advantage of the low-cardinality of the DNA alphabet. In the *FED* version, each of the four symbols composing the DNA alphabet is assigned a unique 2-bit code, packing four elements into a single byte, padding last bits with zeros in the case of sequences where the length is not a multiple of 4. Additionally, a bit-mask is used to distinguish valid bits from padding in the last encoded byte.

The procedure consists of two successive steps:

- *Pre-processing*, where texts and patterns are encoded and a *shift table* is computed for every pattern to be matched.
- *Matching*, where the actual search procedure is performed, is implemented as a byte-by-byte comparison between the text and pattern encoded sequences. If every byte of the pattern is sequentially found in the text, then the current position is registered as a match. Otherwise, the *shift table* is accessed to compute how many positions the pattern is allowed to skip before performing the next check.

Figure 3.43 summarizes the string matching procedure flow. As long as the customised *Boyer-Moore* procedure can perform a matching operation on encoded sequences, the encoding step can be considered not part of the algorithm as it can be done offline by storing the encoded sequences in custom binary files which constitute the actual source of data for the pattern matching engine.

Pattern matching over DNA sequences can be considered an embarrassingly parallel application, because the average use case consists in matching millions of patterns against multiple text sequences, independently [101].

The inputs for the benchmark application are two binary files, storing the encoded texts and patterns to be analyzed. From an algorithmic point of view, running *FED* on already encoded sequences is equivalent to loading plain sequences and encoding them online. For the sake of benchmarking the communication effort in the target platforms, I decided to encode sequences off-line. Moreover, I split text sequences into a set of chunks with a given fixed size. This step is required because in bioinformatics applications, generally, the text represents one or more genomes and its size is not suitable to be sent in a single shot as it is.

Our parallel implementation of the search algorithm identifies two main roles among the MPI processes—the *MPI control process*, which is the role adopted by the MPI

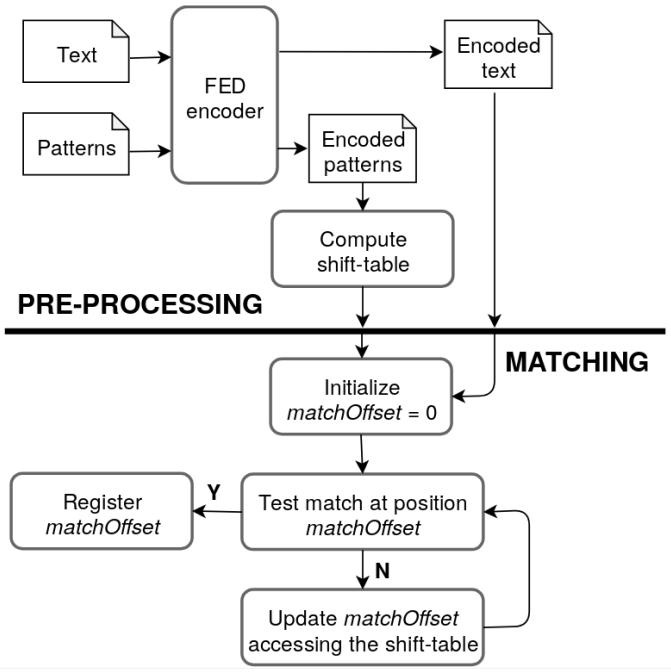


Figure 3.43: Flowchart of the string matching algorithm.

process with rank 0, and the *MPI worker*, which is the role adopted by all remaining MPI processes.

The algorithm works in two distinct steps, outlined in Figure 3.44: configuration (A) and match (B). During configuration step (A), the *MPI control process* accesses the file system, loads the *FED* encoded patterns and distributes them among the *MPI workers* so that each working process handles approximately the same workload. Pattern distribution is implemented as a set of point-to-point communications, using MPI_Send/MPI_Recv primitives. Once an *MPI worker* receives its patterns it computes the *shift table* for them, completing the pre-processing phase shown in Figure 3.44. This strategy allows both to reduce the amount of data sent over the communication network, as the patterns are already encoded and to distribute the pre-processing efforts equally among all available working nodes, as long as any *MPI worker* finalizes the pre-processing step on its patterns only.

During the matching step (B), the *MPI control process* loads the encoded chunks of text and broadcasts them one at a time to all the *MPI workers*, which are in charge of performing the actual pattern matching procedure by calling the search primitives. As shown in Figure 3.44, once a match is found, it is saved into a buffer local to the MPI instance that discovered it. Once every chunk has been analysed, all the MPI instances synchronize to produce two report files containing information about the matches found and the run-time needed for accomplishing

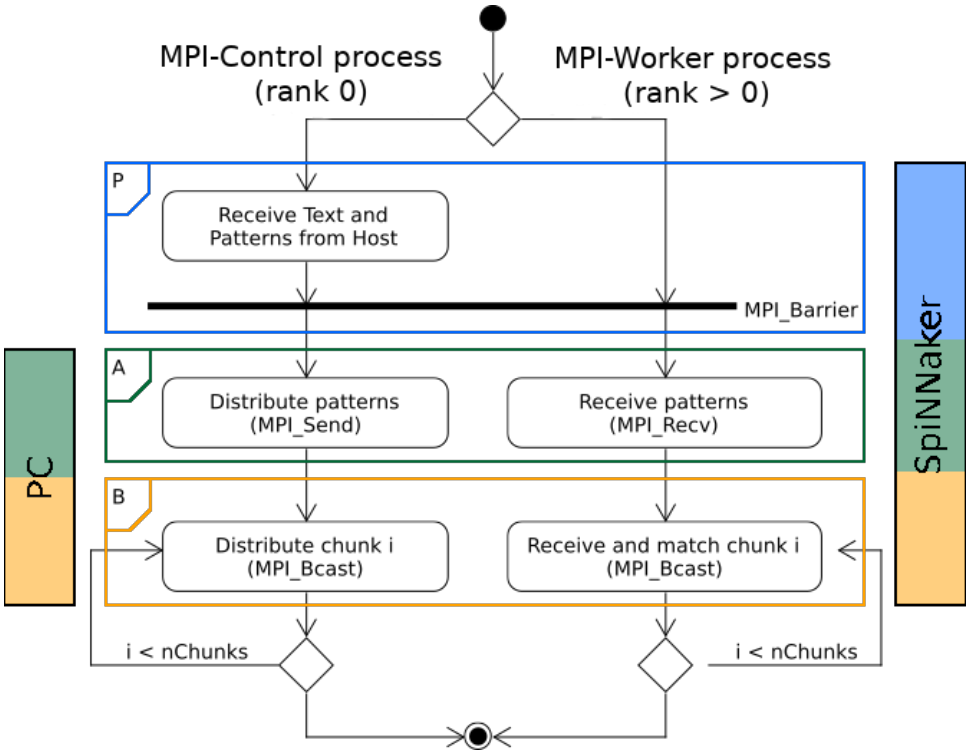


Figure 3.44: Flowchart of the implementation of MPI-FED on a general purpose architecture and on SpiNNaker. The step A performs the configuration, the step B execute the matching, whereas during the step P our implementation implement a preliminary phase for transferring the data to the SpiNNaker board.

their tasks.

The implementation of *FED* with MPI for SpiNNaker retains the configuration (A) and match (B) phases from the previous section, as depicted in Figure 3.44. However, an additional preliminary phase (P) is required in order to transfer the problem data to the board. The configuration step (A) will then be performed by one of the SpiNNaker cores, taking up the role of *MPI control process*.

Using the SpinMPI Python library, the host launches the *MPI Runtime* and creates an *MPI Context* declaring the number of chips and cores that will be used by the application on the Spin5 board. The *MPI Runtime* is also in charge of loading and starting the application binary on the board.

In the preliminary phase (P), the communication between the computer host and the on-board application is performed through the use of ACP memory entities (MEs). First, the binary files containing the genome and the search patterns are read by the *MPI Runtime*. In this phase, the host will write into a ME belonging to processor (0, 0, 1) (the *MPI control process*) two integers indicating the number of

chunks (`nchunks`) and patterns (`npatterns`) which will be loaded into SpiNNaker. The *MPI control process* allocates in SDRAM the memory necessary to contain all chunks and patterns. After allocation is performed, the addresses of these memory blocks are read by the *MPI Runtime*, again using ACP. The *MPI Runtime* can proceed to fill the *MPI control process* memory with the genome and the search patterns previously read.

An MPI Barrier forces all *MPI workers* to wait until the *MPI control process* has received all data from the *MPI Runtime*. Once the problem data has been transferred (phase Ⓟ), phase Ⓐ can begin. The *MPI control process* distributes the patterns among all worker cores through MPI_Send/MPI_Recv primitives and the *MPI workers* store the pattern data in their DTCM and compute the *shift tables*.

The phase Ⓑ begins after all patterns have been distributed. The *MPI control process* sends a text chunk to all *MPI workers* executing a broadcast communication. The SpiNNaker implementation of the *MPI_Bcast* function is a blocking call, as the memory limitations of the platform do not allow for large communication buffers; hence, the *MPI control process* will proceed to send the next chunk only after all workers have processed the current chunk. On the worker side, only one text buffer is allocated into DTCM, since the text chunks will be processed sequentially and a chunk can be replaced whenever a new one is obtained. When a *MPI worker* executing the *FED* algorithm finds a match position, it is stored into a linked list together with the chunk and matching pattern identifiers. Thus the position in the reference sequence can be retrieved.

After all the text chunks have been processed, the application is finalised, and the *MPI Runtime* can download the results directly from the memory of SpiNNaker cores.

In the following, I analyse the efficiency and scalability of our optimised *Boyer-Moore* (*FED*) implementation on SpiNNaker. I compare it with the scalability on a traditional multi-core CPU using a server configuration with two Intel Silver Xeon 4114 processors, each with 10 cores and 20 threads. The *FED* algorithm is implemented in C and used to benchmark both Server and SpiNNaker architectures. The benchmark running on the general purpose Server architecture is written in C++ and compiled with g++ 7.4.0 and MPICH 3.3 parallel environment. The benchmark running on SpiNNaker architecture is written in C and compiled with gcc-arm-none-eabi 5.4.1 and SpinMPI 19w19. By using the SpinMPI library I ported the *FED* code written for a Server Architecture to the SpiNNaker hardware without applying any code transformation.

The text used for the sake of testing is the *Escherichia coli* genome, which is about 4 million symbols long, leading to an encoded text of about 1 MB size, which is then split into a set of about 4000 chunks, each 256 Bytes long.

There exist two types of strategies to evaluate the scalability of a problem in a parallel environment:

- *Strong-scaling* [6] keeps the size of the problem fixed and evaluates the application runtime when multiple processes are used. This strategy is suitable for CPU-bounded problems.
- *Weak-scaling* [37] is used to test the scalability of memory-bounded problems, as it keeps constant the ratio between the problem size and the number of working processes used.

The SpiNNaker platform provides a fast, core-local data memory (DTCM) of 64 kB. This memory constraint allows to store at most 100 *FED* patterns per node, totalling 40 kB in size. Given this memory constraint, I decided to use a *weak-scaling* benchmarking strategy to scale our benchmark up to the 768 nodes available on SpiNNaker. The problem size must be calibrated in order to claim a condition of equivalence and perform a fair comparison between different architectures; in our case, a condition of equivalence is met whenever the same *FED* execution time t_{FED} is observed using a single *FED* worker. When SpinMPI is requested to match 1000 *FED* chunks against 100 *FED* patterns on a single node, a run-time of 26,970 ms is measured; the same run-time, for the MPICH implementation with 1000 *FED* chunks, is obtained when the single *FED* worker used is in charge of 12,500 *FED* patterns. This preliminary assessment is needed to evaluate only the scalability features of the two architectures, without considering the difference in computing power of the single working node for the two architectures. The reason for this comparison is to put the performance of MPI on SpiNNaker in a familiar perspective, as the CPU-DualSocket server is a widespread general purpose machine that allows to use MPI.

A general strategy for evaluating the parallel scaling of an MPI application is computing the scaling efficiency, which measures how good the application is at using every node the parallel environment has. Given an environment with N workers and a problem that requires $t_{\text{FED},i}$ units of time to be solved with i workers, the *weak-scaling* efficiency E_N can be measured as in Equation 3.15. The speedup S_N can be easily inferred from the efficiency and computed with Equation 3.16.

$$E_N = \frac{t_{\text{FED},1}}{t_{\text{FED},N}} \tag{3.15}$$

$$S_N = E_N \cdot N \tag{3.16}$$

Figure 3.45 and Figure 3.46 report the speedup and efficiency of the *FED* with MPI algorithm on the Server and SpiNNaker architectures. The horizontal axis

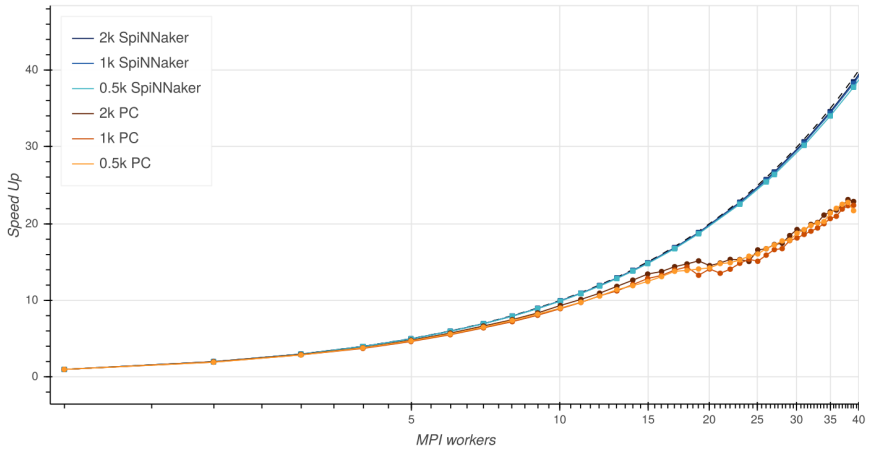


Figure 3.45: Comparison of Weak-scaling speedup for MPI-FED on a general purpose architecture and on SpiNNaker.

represents the number of MPI workers used; both systems were tested until saturation, with the Server reaching 40 parallel workers through Intel hyper-threading and the Spin5 board utilizing all 768 available physical cores. Tests were performed for genomes of 500, 1000 and 2000 chunks.

In Figure 3.45 we can see how the massively parallel architecture of SpiNNaker influences the speedup. The high number of physical cores on the machine lets the speed increase linearly, avoiding the discontinuities that a general-purpose processor has at critical points when hyperthreading is activated to provide the required number of workers (note, in the graph, the inflection point at 20 MPI workers for the PC version, i.e., the point at which the maximum number of physical threads on the Xeon is reached).

In Figure 3.46 SpiNNaker demonstrates excellent scalability, with efficiency values close to 95% for up to 200 workers. Additionally, we can see that the performance markedly improves for longer text sequences; the efficiency for 768 workers processing 2000 chunks is 87.83%. The reason for this happening is that as the size of the data to be processed increases, the ratio of processing time to communication time in the overall algorithm increases, since the data are only sent once at the beginning of processing and then gathered at the end. The bottleneck due to the communication overhead thus becomes less prevalent, and the efficiency improvement due to massive parallelism is more evident.

By contrast, the efficiency of the Server dips much faster, dropping below 90% as soon as the requested MPI workers outnumber the physical cores. It also remains fairly constant when changing the number of chunks. This appears reasonable as, for the high-speed CPU used in the test, the computation time is very small, but it

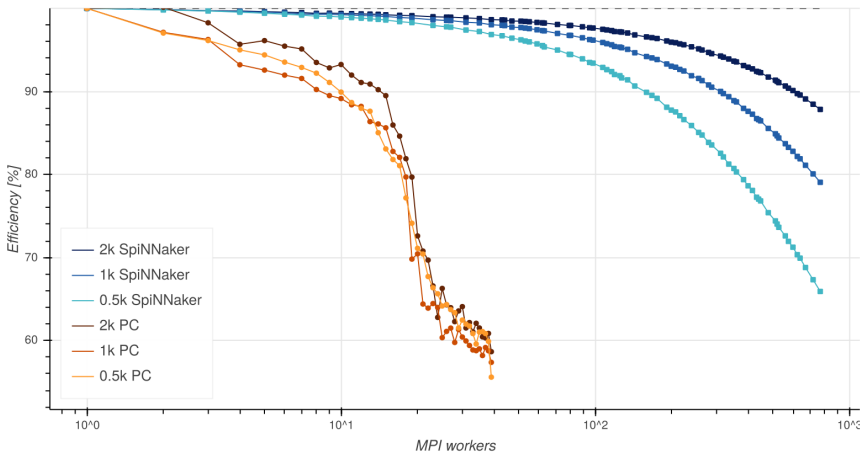


Figure 3.46: Comparison of Weak-scaling Efficiency for MPI-FED on a general purpose architecture and on SpiNNaker.

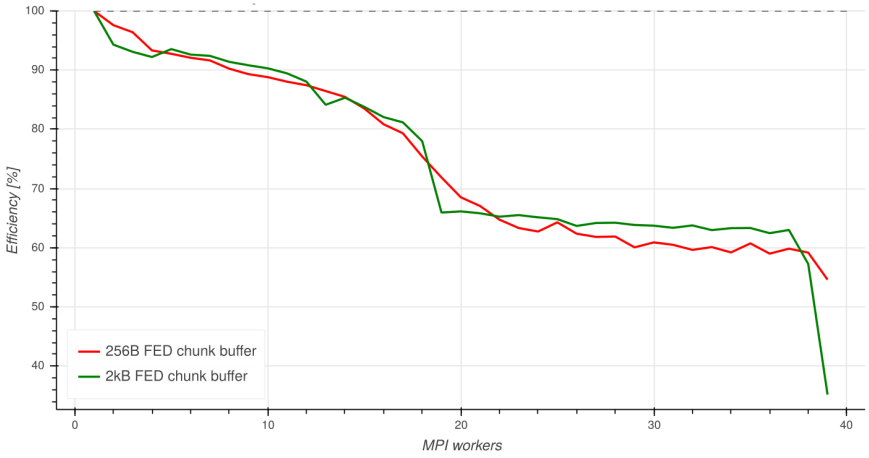


Figure 3.47: Efficiency of the general purpose architecture for different *FED* buffer sizes.

suggests that other phases of the computation such as inter-process communication and thread management have a significant impact on the efficiency of the algorithm.

As a side-experiment, I evaluated the impact of the size of the *FED* buffer distributing data among the *MPI workers* on the measured scaling efficiency. Figure 3.47 shows the scaling efficiency of two experiments—the former distributes the *FED* chunks to be analyzed as 1000 256-Byte packets. The latter broadcasts the same amount of data, formatted as 125 2-kB packets. Figure 3.47 highlights that the two scaling efficiency tracks are comparable, meaning that the size of packets used to distribute *FED* chunks among the *MPI workers* does not impact the benchmark results for the general purpose architecture.

Finally, we can make a comparison of the power efficiency on the two architectures by using estimated consumption based on the nominal values from the CPU and SpiNNaker [64] data-sheets. For the Intel Xeon, I consider the peak and idle powers at the values of P_{peak} = 11,030 mW and P_{idle} = 6320 mW, and I hypothesize that the number of active physical cores (out of the available 20), $f(x)$, can be expressed as a function of the active MPI workers x as $f(x) = ceil(\frac{x+1}{2})$.

The appearance of the term $x + 1$ rather than x is because there is one Controller process that has the task of distributing the data and patterns to the MPI workers. Based on this assumption, I assign a power consumption of P_{peak} to the active cores and of P_{idle} to every other core; thus the estimated power consumption with respect to the number of MPI workers x is $P(x) = P_{peak} \cdot f(x) + P_{idle} \cdot (20 - f(x))$.

On the other hand, for SpiNNaker I consider the values of Idle Power per Chip C_{idle} = 360 mW, Idle Power per Core P_{idle} = 20 mW, Peak Power per Core P_{peak} = 55.56 mW, and the Off-Chip-Link power, P_{link} = 6.3 mW. The power estimation for SpiNNaker depends on the MPI execution context, which can be described by a pair of values (p, k) where $p \in$ [1,16] is the number of active processors per chip and $k \in$ [1,48] is the number of active chips. The power estimation formula can be expressed as a function of the number of active processors and chips as $P(p, k) = k \cdot (C_{idle} + (P_{peak} - P_{idle}) \cdot (p + 1) + P_{link}) + (48 - k) \cdot C_{idle}$ Counting $p + 1$ processors to include the Monitor Processor on each core. Then, the estimated power given the number of MPI workers x is $P(x) = P(p, k)|min_k[p \cdot k = x + 1]$ As in the CPU case, I count $x + 1$ processes to include the Controller process.

Given the architectural difference between the SpiNNaker and CPU machines, it is necessary to outline a fair method to evaluate the efficiency of the algorithm's implementation. I define power efficiency as the energy consumed to align a single pattern to the reference, measured in units of mJ/pattern, as a function of the parallelisation effort of the given system, expressed as a percentage of the total resources. The maximum energy efficiency is obtained when all resources are in use, corresponding to a parallelisation effort of 100%. For SpiNNaker it is easy to assume that 100% utilisation occurs when all 768 cores are busy (i.e., at 767 MPI workers), corresponding to an average energy consumption of 37.3 mJ/pattern. For the CPU utilisation, I can either consider 100% utilisation to be the situation where all physical cores are active, or the one where all the virtual cores are active (20 physical + 20 virtual, providing 39 MPI workers). In the first case, the estimated average energy consumption is of 51 mJ/pattern, with an estimated power saving of 27% in favour of SpiNNaker. In the second case, the energy is 43 mJ/pattern, with SpiNNaker consuming 13% less.

3.3.6 Final Remarks

This work focused on the optimisation of the communication protocols for the SpiNNaker system. I first analysed the current methodologies to find possible weaknesses and noticed that the intrinsic concurrency of the system could have been exploited in order to improve the speed of the communication and that the current Point-to-Point protocol was inefficient for several tasks. For these reasons, I decided to develop a new middleware based on the more efficient Multicast protocol in order to implement a better communication system to be also used for configuring the system. The idea behind this new approach was also to reduce the complexity of the internal transmissions, by implementing unicast communications avoiding the supervision of the monitor processor that for several applications is time-expensive. I designed the Application Command Framework, and Application Command Protocol a new method to be adopted at the application level for spreading commands and manage the memory of the SpiNNaker neuromorphic platform. ACF allows the users to include in their distributed applications the subset of commands to carry out only the needed activities, hence saving memory for the code. On top of that, ACP allows the exchange of commands between application processors without involving the respective monitor processors using the multicast channel, thus optimising the communication flow. It provides a useful abstraction level of the memory which users can easily access through a virtual id to all the variables of the applications running on one application processors (AP) from any other AP of the system.

I modify two SpiNNaker applications in order to use the ACP inside the application used during the *Configuration* of SpiNNaker board, and into a neuron model used during the SNN *Simulation* phase. The ACP implementation in the first application enables a flexible and optimised set-up of the board during the configuration phase. Conversely, ACP in the second application allows the user to change some parameters at run-time during the simulation phase. In the first application, I demonstrated the advantages introduced by ACP interpreter in the run-time feeding of configuration applications. More specifically, I analysed the behaviour of the Monitor Processor of the node attached to the Ethernet interface, that is in charge of managing the communication with the external sources. Handling the configuration phase at the application level with ACP allows the configuration to be performed by all kinds of external sources. The use of this new method is straightforward and can speed-up host-to-boards data transmission during the configuration of SpiNNaker platforms. By exploiting the concurrency of the system and the ACP over MCM protocol, I have been able to get an improvement of 3 times on the data forwarding inside the board, providing the chance of building more efficient application through the new software. In the second application, I demonstrated the run-time flexibility introduced by the ACP interpreter embedded

in the neuron model application, implementing two different real simulation scenarios: i) a two-phases SNN-Classifier designed for discriminating the handwritten number and ii) a chain of neurons with run-time re-configuration parameters. Our results show that ACF allows to switch between training and testing phase in half of the time needed by the former workflow and to change model parameters (e.g. synapse delay) during the simulation.

This work opens the way to more flexible use of manycore neuromorphic platforms as brain simulators and as support for new computational brain-inspired paradigms. Moreover, in this work, I presented an implementation of the MPI paradigm on the SpiNNaker neuromorphic platform. The MPI standard exposes a programming model for the development of parallel applications in a distributed memory environment without knowledge of the interconnections between the computing units of the underlying architecture. The implementation of MPI for a specific architecture is therefore expected to implement the most suitable features in order to exploit the available resources and to synchronise the computing flow.

In the case of SpiNNaker, the implementation of MPI must deal with a resource limit both in terms of memory and computing power. However, it can take advantage of the technology offered by on-chip routers, obtaining efficient communication. SpinMPI is also in charge of managing communication between the *MPI Runtime* running on the host computer and the SpiNNaker cores; this is done by using the ACP protocol and memory entities. This software stack creates a simple working framework offering a universally known programming model capable of making the SpiNNaker architecture available for a wide range of applications. The SpiNNaker implementation of MPI is built on top of multiple abstraction-level, the following libraries: i) MCM, implements broadcast and unicast connection and synchronisation methods and a hash table ADT. ii) SpinACP, implements memory entities and network command functionalities. iii) SpinMPI, implements the MPI on SpiNNaker.

I benchmarking SpinMPI, showing that the scaling performances are kept linear when an increasing number of cores is used during the computation. As the second point, I demonstrated that by using the SpinMPI library, which provides MPI support for SpiNNaker, I could easily port algorithm implemented for standard computers on the manycore neuromorphic platform. I implemented an N-Body simulation to benchmark and evaluate the performance of the board in the execution of an MPI parallel application. In this simulation, 2 k particles were simulated on 240 processors with a speed-up of 194 x and an efficiency of 80% when compared to the serial version running on a single CPU. I also presented an implementation of an MPI-based DNA sequence matching algorithm. Results show that the scalability of the SpiNNaker board reaches an ideal profile (98% of efficiency) when using more than 100 processors, a 90% efficiency using 600 processors, reaching 88% efficiency

when all 767 application processors are used.

Chapter 4

Programming tools for heterogeneous platforms

In this chapter, I present *DeepLLVM*, a classification method for source code where I introduce the use of LLVM-IR and Deep Learning in code classification. Moreover, I report results of an extensive parameter exploration of the classification model that are used in some state-of-art approaches ([22, 9, 12, 3]).

DeepLLVM is divided into two modules:

- The source code preprocessing module that identifies the most significant syntactic elements and reducing them to a sequential list of integers.
- The language classifier component that can use two alternative Deep Neural Networks (DNN) models (Conv1D and LSTM) trained using a supervised learning method.

More specifically, *DeepLLVM* integrates Long Short-Term Memory (LSTM) cells, Convolutional layers (Conv1D), and Global Max-Pooling layers (GMP) to extract knowledge from syntactic language elements (tokens) of a kernel compiled in IR.

I trained the network using a dataset of OpenCL kernels whose execution time has been profiled on CPU and GPU [22]. I then evaluated the classification accuracy in mapping each kernel to the best compute unit. I evaluated not only the classification accuracy but also the overall speedup of kernels execution compared to a static mapping (all kernels in GPU or CPU).

I finally performed an extensive exploration of the hyper-parameter space composed of both network and training. The best set was used as a reference for comparison between the Convolutional Neural Network (CNN) model and the LSTM solution proposed in [9] and [22] accounting also for the impact of kernel optimisations and

token filtering strategies.

I tested the kernel-to-device allocation performance demonstrating that our LLVM-based classifier achieves an accuracy of 85% in selecting the best kernel allocation. Results confirm that IR based classifiers achieve similar or better performance than OpenCL based ones, with the advantage of the generality of the IR representation. Our results show that LLVM-IR keeps the informative content in the token sequence needed to perform an effective classification making possible the application of our classifier to any source code for which an LLVM compiler exists.

Moreover, I discover that CNN model outperforms RNN in terms of training time, classification accuracy and overall speedup.

4.1 Method

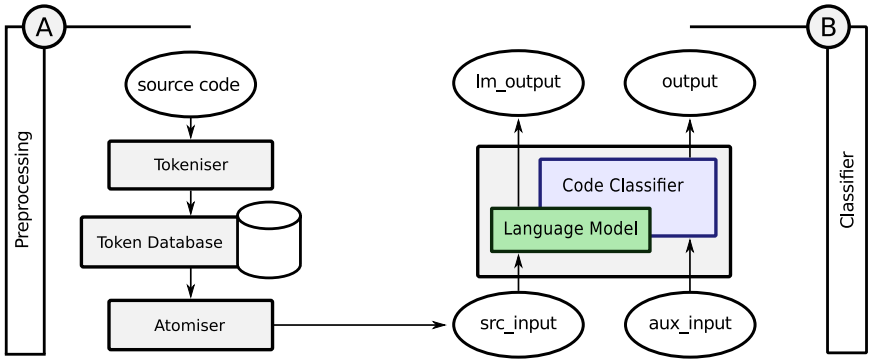


Figure 4.1: *DeepLLVM* flow representation of operations to be performed for the construction of the code classifier. A) Source code preprocessing steps for inserting the code into the classifier B) Black-box representation of the classifier

In this Section, I present *DeepLLVM*, a code analysis methodology based on deep learning.

I designed *DeepLLVM* to explore the feasibility to build a classifier able to analyse source code expressed in LLVM Intermediate Representation, which gives the advantage of decoupling a programming language from the target architecture. The applications that may take advantage of such analysis are numerous and range from the identification of sophisticated compilation strategies to the allocation of computing resources.

As shown in Figure 4.1, *DeepLLVM* is composed of two steps: i) source code preprocessing, which identifies the most significant syntactic elements (tokens) and reduces them to a sequential list of integers. ii) code classification, which exploits a

Neural Network based on layer models[1] trained using a supervised learning method.

Moreover, since no previous works introduced CNN for source code modelling, I performed an hyper-parameters exploration for devising a good network architecture and the impact of data preprocessing.

In subsection 4.1.1, I describe the *DeepLLVM* source code preprocessing phase composed of two steps: Tokenisation and Atomisation. In subsection 4.1.2, I describe the method used to implement the *DeepLLVM* code classifier. In subsection 4.1.3, I describe the Hyper-parameters exploration and in subsection 4.1.4 I introduced the quality metrics (e.g. the execution speedup) for evaluating the classification impact.

4.1.1 *DeepLLVM*: code preprocessing

LLVM-IR Code Fragment

```
%9 = and i64 %8, 4294967295
%10 = getelementptr inbounds <4 x float>, <4 x float>* %1, i64 %9
%12 = fsub <4 x float> <float 1.0e+00, float 1.0e+00, float 1.0e+00, float 1.0e+00>, %11
%13 = fmul <4 x float> %11, <float 3.0e+01, float 3.0e+01, float 3.0e+01, float 3.0e+01>
```

Tokenization

```
_local = and i64 _local , _integer_constant
_local = getelementptr inbounds _float_4 , _float_4 * _local , i64 _local
_local = fsub _float_4 _vector_constant , _local
_local = fmul _float_4 _local , _vector_constant
```

Atomization

```
10 11 13 14 10 12 15
10 11 16 17 18 12 18 19 10 12 14 10
10 11 20 18 21 12 10
10 11 22 18 10 12 21
```

Figure 4.2: Example of code transformations: The code in the top pane is an LLVM-IR code fragment. The code in the middle pane contains the result of the transformations applied in the tokenisation phases. The tokens sequence in the bottom pane is the network input, the result obtained after the atomization phase.

Since machine learning models work with numerical data, I need a procedure to convert source code into a form suitable to be processed by the input layer of the considered models.

[1] Convolution Layer, Recursive Cells, Dense Layer and Max Pooling

We start from a dataset of source code written using a high-level programming language. Then we compile all dataset elements using a compiler able to emit LLVM-IR. The *clang* compiler, for example, allows compiling the main C-like languages (C, C++, Objective C). It is also possible to compile OpenCL code for *nvptx* (NVidia), *amdgcn* (AMD) and *spir* (Standard Portable Intermediate Representation) architectures.

The LLVM-IR code obtained after the compilation needs to be cleaned and preprocessed before being fed into the neural network (Figure 4.1-Ⓐ).

The ***Tokenisation*** procedure identifies the most significant language syntactic elements (tokens) within the sequences. All the tokens are catalogued and placed in a dictionary. Then, the ***Atomisation*** procedure transforms code sequences replacing the characters that compose a token with the integer identifier of the token in the dictionary.

I implemented the *Tokenisation* procedure in two steps. The *pre-tokenisation* phase acts on each line of a kernel and performs the following operations:

- Remove empty lines and comments.
- Remove all lines outside the function body.
- Replace vector and array data-types with a simplified version.
- Replace vectors, arrays and float constants with a placeholder maintaining the type and removing the immediate value.
- Insert a space before and after the symbols
  Ⓒ Ⓒ Ⓒ Ⓒ Ⓒ Ⓒ Ⓒ Ⓒ Ⓒ Ⓒ Ⓒ Ⓒ

During this phase, the procedure simplifies complex data types and replaces constants with placeholders, obtaining a significant reduction of the code fragment length. For example, LLVM can express real constants in different ways: i) standard decimal notation (e.g. 6.563989), ii) exponential notation (e.g. 1.179029e+45), iii) hexadecimal notation (e.g. 0xFB160990091690BF). These representations are replaced with a placeholder ("_float_constant"). After *pre-tokenisation*, it is possible to identify as a token every sequence of characters separated by spaces.

The *post-tokenisation* transformations act directly on the tokens for applying the following higher level generalisations:

- Remove unnamed meta-data (tokens starting with Ⓒ) and attribute groups (tokens starting with Ⓒ).
- Replace variable and function names with a placeholder.

- Identification of special labels starting with “phi”, “pre”, “in”, “preheader” and “loopexit”.
- Identification and transformation of integer constants.
- Identification and transformation of global and local unnamed identifiers (e.g. %5 → _local, @16 → _global)

The *Atomisation* step replaces all the tokens with a unique integer identifier using the same approach proposed in [95]. At the end of this phase, each kernel source code is transformed into a sequence of integers, and it is ready to be used as input for the neural network. Figure 4.2 provides an example of the pre-processing pipeline I use.

The performance of text-based deep-learning systems (e.g. in the sentiment analysis) heavily depends on the dictionary chosen to transform the input into a numerical sequence [54]. Long sequences require many training samples and complex models capable of storing and correlating information for more extended periods. A common practice in the Natural Language Processing (NLP) field consists in removing less-informative tokens [47] for decreasing the mean length of the sequences to be analysed and reducing the burden of correlating distant tokens.

I used a weight function, the *term frequency - inverse document frequency* (*Tf-Idf*), to identify the less-informative tokens. The *Tf-Idf* can be obtained as in Equation 4.1

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) * \overbrace{ln\left(\frac{1}{\text{df}(t, D)}\right)}^{\text{idf}} \tag{4.1}$$

Given a dataset $D : (d_1, d_2, \ldots, d_n)$ (corpus of documents), a document $d : (t_1, t_2, \ldots, t_n)$ (sequence of tokens) and a token t, the *Tf-Idf* is the product between the term-frequency (tf) and inverse-document-frequency (idf). The idf is the natural logarithm of the inverse of document-frequency (df).

$$\begin{aligned} \text{tf}(t, d) &= \frac{|\{t' : t' = t, \forall t' \in d\}|}{|d|} \\ \text{df}(t, D) &= \frac{|\{d' : t \in d', \forall d' \in D\}|}{|D|} \end{aligned} \tag{4.2}$$

The term-frequency is the ratio between the occurrences of term t in a sequence of tokens d and the length, in terms of tokens count, of d. The document-frequency is the ratio between the number of sequences of tokens where the token appears and the total number of sequences. The idf reduces the term-frequency value for

very common tokens, and should be increases the term-frequency for specific tokens contained in few sequences.

We can use this weight to build a token-blacklist to delete the tokens with a low informative contribute from all documents. The token-blacklist can be used globally, or we can use the *Tf-Idf* score to create fine-grain filters removing from each sequence only the tokens with a poor local score.

4.1.2 *DeepLLVM*: classifier

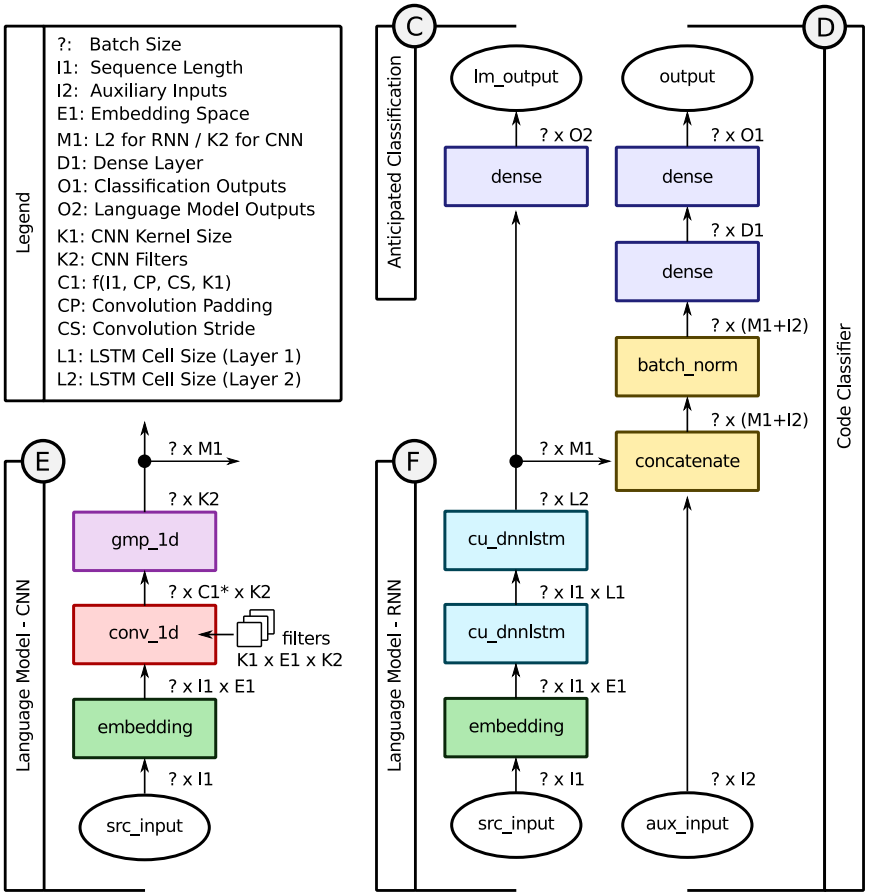


Figure 4.3: Informal representation of the classifier structure. The figure represents the layers of the classifier as rectangles identified by a label describing the operation performed. The arrows indicate the movement of data (tensors) whose dimensions are made explicit by symbolic values described in the legend. E) CNN-based language model F) RNN-based language model C) Early classification for specific language model training D) Overall classification network, takes into account the source code and context data flow.

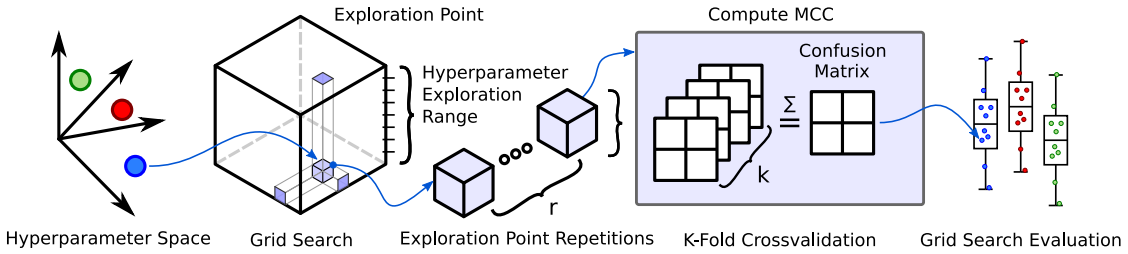


Figure 4.4: Schematic representation of Hyper-parameters exploration. Within the hyper-parameters space some points are chosen to be explored by means of a grid-search. For each hyper-parameter an exploration range is identified (Hyper-parameter Exploration Range). Each point is explored r times. Each exploration includes a k-fold cross-validation and training of k classifiers. The confusion matrices of the classifiers in cross-validation are added together, then the ACC and MCC are calculated. To evaluate the hyper-parameters space, each exploration point is seen as the distribution of MCC and ACC values of its repetitions.

The code classifier used in this work can be seen as a black-box with two input and two output, Figure 4.1-(B). The input is a tuple containing the preprocessed source code fragment (src_input), and auxiliary data that define the context of usage of the source code (aux_input). The auxiliary inputs are necessary because learning the relationships between them and the code allows contextualising the code.

Internally, the classifier is a neural network divided into two components: language model and features classifier. The language-model is in charge of reducing the token sequence into a point in $\mathbb{R}^{M1}$. The features classifier analyses the output of the language model.

The two outputs are the features-classifier output and the language-model output. Both outputs can be independently used to match the desired output, and each one is associated with a loss-weight. The loss weight is a scalar coefficient that defines the output loss contribute over the global-loss score.

The network structure proposed in this work is depicted in Figure 4.3. I propose two different language-model networks: the first one is based on a RNN (Figure 4.3-(F)) and the second one on a CNN (Figure 4.3-(E)).

The network input ("src_input") is a tensor composed of batch-size sequences, each one composed of sequence-length elements. We will refer to the input elements with the term token-indexes since each component represents the position of a token inside the token dictionary. Since the following layers need to work on comparable data, and the token indexes do not have this property because we cannot define a distance metric between two indices, the sequence of token-indexes must be projected into a metric space.

The Embedding layer is the first layer of the network that receives sequences of token-indexes and projects each element into an embedding space $\mathbb{R}^{E1}$. The output of the Embedding Layer is, therefore, a list of sequences each one composed of vectors belonging to the embedding space. The weights of the Embedding Layer determine how the token-indexes are projected in the embedding space. At the beginning of the training, the projection in the embedding space starts in a random condition.

The language model receives the output of the Embedding Layer (a vector in $\mathbb{R}^{I1,E1}$) and summarises the whole sequence in a single point in the features space ($\mathbb{R}^{M1}$). Features can now be passed to the second part of the network (Figure 4.3-(D)) that will perform the classification.

Our implementation of the CNN model consists of a one-dimensional convolution layer, followed by a global max-pooling layer. The global max pooling acts for each channel of convolution layer output and selects the maximum value. The result is a point in the feature space. This structure is inspired by the one adopted in [105] for sentence sentiment classification.

In our implementation or RNN model, I used two LSTM layers. The first layer elaborates the input sequence and produces another sequence in output. The second layer elaborates the output of the first layer maintaining only the last output element, considering it a point in the feature space. This structure is inspired by the one adopted in [9], which shows good performance.

4.1.3 Hyper-parameters exploration

All works based on machine-learning require an exploration strategy in the hyper-parameters space. If the model parameters (the network weights) are obtained trough steepest-descent and error back-propagation, the hyper-parameters must be explored heuristically to define a reference configuration. The reference configuration is necessary for comparison with other classifier models and to evaluate other choices in the problem context (e.g. compilation strategies and token filtering). The goal of the Hyper-parameters exploration is to define a different set of hyper-parameters and explore with a multi-stage grid-search the classifier performance.

I divided the hyper-parameters into three categories:

- *Network* hyper-parameters
- *Training* hyper-parameters
- *Dataset* hyper-parameters

The *network* hyper-parameters are specific to the network model that we consider. They are divided into CNN and RNN hyper-parameters. These two sets have in

common the parameters that define the sequence input length (I1) the output size of the embedding-layer (E1) and the output size of the last dense-layer (D1).

The CNN hyper-parameters are the kernel shape (K1), and the number of filters (K2). The RNN hyper-parameters are the cell-size of the first LSTM layer (L1) and the cell-size of the second LSTM (L2) layer.

The *training* hyper-parameters are instead specific to the supervised training method. They are training algorithm (SGD, Adam, ...) [35], training algorithm hyper-parameters (specific for the training algorithm), batch size (1, 16, 32, 64, ...) and loss weight (0.0, 0.1, 0.2, ..., 1.0) of the learning model output (*lm_output*).

The exploration of *dataset* hyper-parameters can give some insight into the ability of the language model to extract information from the source code sequence. The *dataset hyper-parameters* are: padding strategies (add a null token at the end or before the sequence), truncating strategies (keep the initial or final tokens), token filtering policy in order to eliminate tokens with a low information contribute (token blacklist, *Tf-Idf* threshold) and compilation optimisations (`-O0`, `-O1`, `-O2`, ...).

I define an exploration priority to decide which hyper-parameter set must be explored with the two-phase grid search. My strategy is to identify a good set of *network hyper-parameters* using a grid search followed by a second grid search on *training hyper-parameters*.

The evaluation process is depicted in Figure 4.4. The grid-search explores a subset of the hyper-parameters space. For each hyper-parameter, we decide a set of values of interest that define its exploration range. The exploration point is a precise configuration of hyper-parameters, and it is evaluated multiple time in order to take into account the intrinsic model variability during the training process and improve the statistic confidence of the process. The training of an exploration point repetition is performed in cross-validation. The whole dataset is split in k subsets, and in turn, we train k classifiers (fold-classifiers) changing the subset used as test-set. The confusion matrices of each fold-classifier are reduced using a sum operator. On the resulting matrix, we can compute the Accuracy (ACC) and the Matthews Correlation Coefficient (MCC[2]) [44].

$$C = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \text{MCC} = \frac{ad - cb}{\sqrt{(a+b)(a+c)(d+b)(d+c)}} \tag{4.3}$$

Given a confusion matrix $C \in \mathbb{R}^{2,2}$ the MCC is a metric that contrary to the Accuracy (ACC) considers the whole classifier behaviour and provides more consistent

[2]MCC varies between -1 and +1, being +1 the best result possible.

results when dataset labels are not balanced. Equation 4.3 reports the formula for computing MCC. Variables a and d represents true positives and true negative while b represents false negatives and c represents false positives.

All exploration points can be compared using the mean and the standard deviation of their performance distributions. An exploration point is evaluated given the performance (MCC and ACC) distribution of its repetitions.

4.1.4 Misclassification Impact

The only usage of Accuracy (ACC) or Matthews correlation coefficient (MCC) is not enough to evaluate the source code classifier. Evaluating the impact of a right or a wrong decision on other metrics (e.g. energy, power, runtime) is essential to evaluate the classifier performance. Given a metric, we measure the impact of wrong decisions made by the classifier using the percentage deviation from the optimal value obtainable using an oracle.

These metrics can be obtained during the labelling process of the classifier. Concerning the present work, the dataset collected in [22] was labelled with the compute unit showing the best runtime performance evaluated with the execution of OpenCL kernels. That is the reason why we also evaluate the speedup our system is able to grant, along with accuracy and MCC.

The tested devices for this dataset was a CPU, AMD-GPU, and Nvidia-GPU processors.

4.2 Results

I applied the methodology depicted in section 4.1 to define a CNN model to be used for performance evaluation. Moreover, I compared it with the RNN-based network already known in the literature. I also evaluated the impact of kernel misclassification on the speedup compared to a static mapping, that is kernel allocated all on GPU or CPU.

subsection 4.2.1 describes the main properties of the kernel dataset, as well as the preprocessing and filtering operations I applied for producing the symbol sequences fed into the machine learning models. subsection 4.2.3 details the hyper-parameters optimisation procedure I used for devising an appropriate CNN architecture. subsection 4.2.4 provides exhaustive comparisons between CNN and RNN and explores how token filtering impacts on classification accuracy. Then, it provides details regarding the time required for training the two architectures with different input and batch sizes. subsection 4.2.5 describes classifier performance taking into account runtime and speedup. Finally, subsection 4.2.6 summarises the findings coming

from the experiments analysis I performed.

4.2.1 Dataset description

Table 4.1: Labels distribution in the source dataset. For both GPU considered a slight 60%/40% unbalance in labels assignment can be observed.

Dataset	Device	CPU	GPU
AMD	Tahiti 7970	400 (58.8 %)	280 (41.2 %)
NVIDIA	GTX 970	293 (43.1 %)	387 (56.9 %)

For training and testing *DeepLLVM*, I used a composition of OpenCL kernels coming from six source code collections [22].

- AMD and NVidia OpenCL examples and benchmarks
- NPB, the NASA Advanced Supercomputing Parallel Benchmarks
- Parboil, computing applications for studying the performance of computing architecture and compilers
- PolyBench/GPU
- Rodinia, the University of Virginia Rodinia benchmark suite
- SHOC, Scalable HeterOgeneous Computing benchmark suite.

Each element of the data-set has a label denoting the best performing computation device between a CPU and a GPU (AMD Tahiti or Nvidia GTX). The authors of the data-set executed each kernel using different load (byte transfer) and different level of parallelism (workgroup size), keeping track of the time required for executing each kernel on the available devices. Each triple is composed of: a kernel, byte transfer size and workgroup size and it is labelled with the device exposing the best runtime performance. The full data-set is composed of 680 triples and 256 different kernels, and it is characterised by a slight unbalance in labels assignment, detailed in Table 4.1.

OpenCL kernels stored in the source dataset are not suitable to be classified as they are, because machine learning models detailed in subsection 4.1.2 require a sequence of numerical symbols as input. First of all, input sources were translated into LLVM intermediate representation running clang (v7.0.1) on each input kernel using the `-emit-llvm` parameter, set the desired OpenCL version using `-cl-std=CL2.0` and import OpenCL headers using the `-Xclang -finclude-default-header` parameters. The compilation of some kernels ended with the presence of errors. I then proceeded to manually fix the broken OpenCL kernels and use the entire dataset.

I created a second experiment trunk adding the `-O2` flag to the clang compiler command line for checking whether any middle-end code transformation impact on classification accuracy. Moreover, I produced a third kind of sequences tokenising OpenCL kernels as they are for the sake of comparing our pre-processing pipeline with the one of [22].

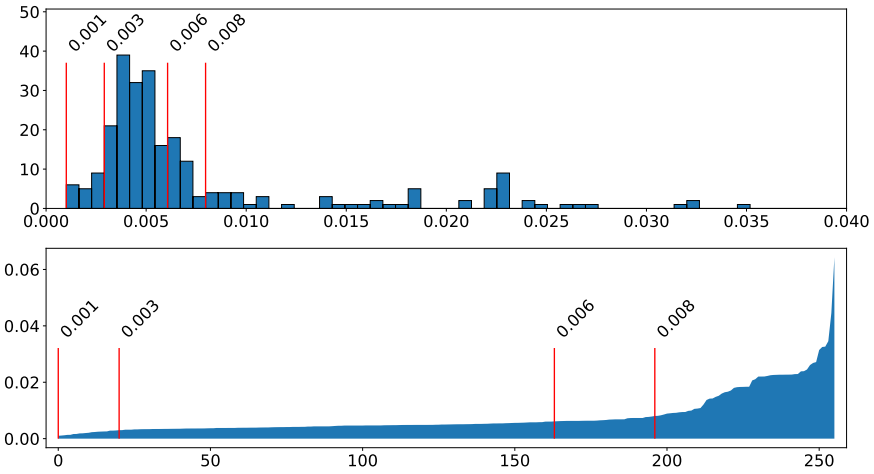


Figure 4.5: *Tf-Idf* analysis applied on LLVM `-O0` dataset. Distribution of the average *Tf-Idf* score measured per document (top). Average *Tf-Idf* score directly represented in a bar-plot. In the figure are depicted the score threshold used in *Tf-Idf* filtering evaluation.

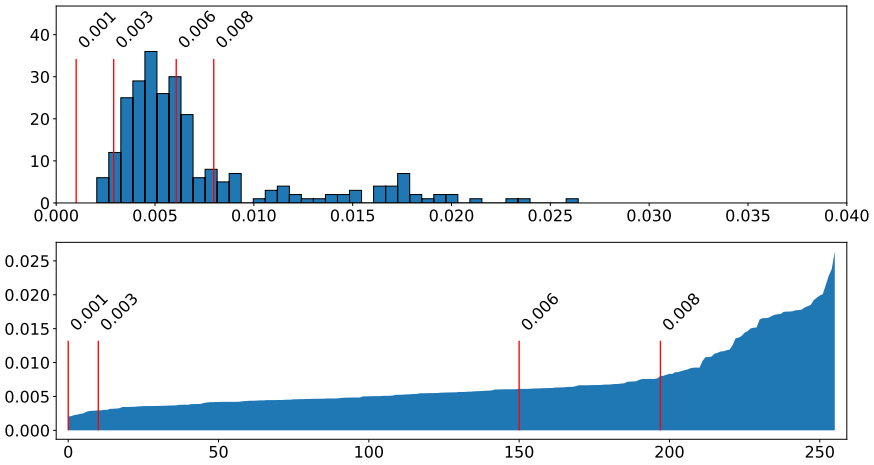


Figure 4.6: *Tf-Idf* analysis applied on LLVM `-O2` dataset. Distribution of the average *Tf-Idf* score measured per document (top). Average *Tf-Idf* score directly represented in a bar-plot. In the figure are depicted the score threshold used in *Tf-Idf* filtering evaluation.

The construction of the token dictionary can be done in three ways:

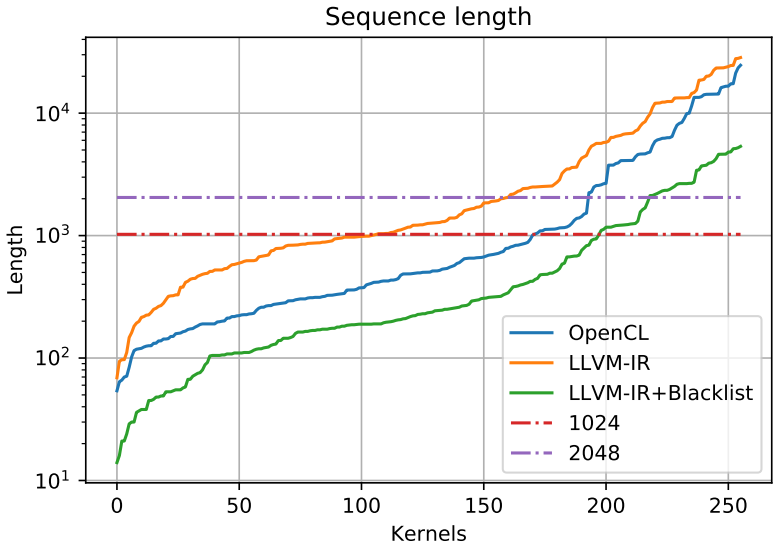


Figure 4.7: Length of code sequences in the three datasets: OpenCL, LLVM and LLVM with blacklist.

- Using a *pure-character dictionary*, it considers only the characters with the advantage of avoiding complex analysis for token construction, but it can be used mainly for short sequences.
- Using a *hybrid dictionary*, it allows a reduction of the length of the sequences by encoding the most common words with a single symbol and processing as single characters all the letters not recognised as dictionary words.
- Using a *pure-token dictionary*, it allows a substantial reduction of the length of the sequences through a transformation of the code where complex syntax artefacts are encoded with single symbols.

Even by simplifying the LLVM code through the tokenisation phase and using a pure-token dictionary, the sequences were too long to be correctly analysed by the network (Figure 4.7).

Usually, machine learning problems which focus on text classification employ token filtering strategies during pre-processing. Such techniques help cleaning input sequences from symbols with poor informative content. For exploring the impact of token filtering in the domain of source code analysis, we used two distinct methods:

- Blacklist filtering: removing a set of intermediate representation tokens from each kernel.
- *Tf-Idf* filtering: each token-kernel pair *(t, d)* is assigned a score as detailed in subsection 4.1.1. Whenever the score of *(t, d)* is lower than the given threshold, occurrences of token *t* are removed from document *d*.

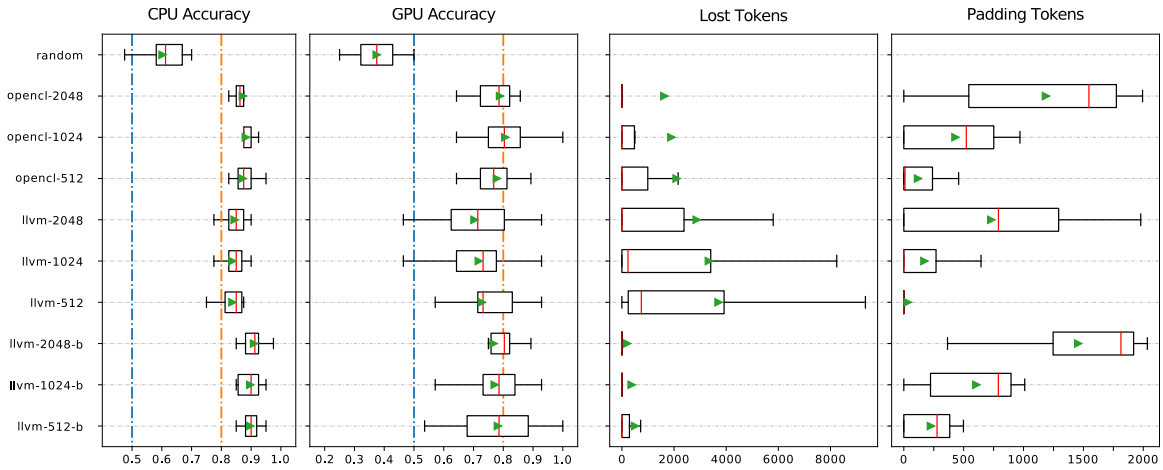


Figure 4.8: For each combination of datasets (OpenCL, LLVM), sequence lengths (2048, 1024, 512) and token-blacklist (used or not used): the first two box-plots show the distribution of the classification accuracy of the ten classifiers in cross-validation, the last two box-plots show the distribution of the lost-tokens (truncation) or added-tokens (padding) in the classifier input sequences.

Concerning blacklist filtering, I produced a list of common tokens that I assumed not to be strongly informative for kernel-device mapping. The Figure 4.7 shows the length of the kernels in OpenCL that are always shorter than LLVM-IR while kernels elaborated with the blacklist filtering reach a reasonable size (LLVM-B).

In *Tf-Idf* filtering, it is crucial to devise a score threshold for removing only redundant tokens. Since such thresholds are dataset dependent, and no previous literature works provide methods for computing them, I evaluated the distribution of the average *Tf-Idf* score per document and sampled four values from it. Figure 4.5 and Figure 4.6 show how the average *Tf-Idf* score per document distributes in the two LLVM datasets. Both distributions share the same shape and highlight a peak around 0.005 with a small tail up to 0.035 and 0.025. I chose to test four *Tf-Idf* scores around the most common average values, specifically 0.001, 0.003, 0.006 and 0.008.

4.2.2 Token Blacklist impact in RNN accuracy

I trained the network using three different datasets and three different sequence lengths. The first two box plots in Figure 4.8 show the accuracy distributions for each class (CPU, GPU) of the ten classifiers that were built to perform the cross-validation. I trained the model also using a dataset in which I replaced the code sequences with a random sequence (only the contribution of the auxiliary input remains). I notice the contribution of the code sequences compared to the random

Table 4.2: LSTM and blacklist filtering results.

	Seq.Len.	Median	Average
DeepTune [22]	1024	80.9%	82.2%
OpenCL	2048	83.1%	83.8%
	1024	84.6%	85.1%
	512	83.1%	83.4%
LLVM	2048	78.7%	78.7%
	1024	80.9%	78.8%
	512	80.1%	79.3%
LLVM-B	2048	86.0%	85.1%
	1024	86.8%	84.6%
	512	85.3%	85.0%

sequences and the difference in accuracy between the CPU and GPU classes due mainly to the unbalance of the dataset.

In the third and fourth box-plots of Figure 4.8 we can note the distributions of the number of padding tokens and tokens deleted for each dataset. The LLVM sequences are too long, and they are strongly disadvantaged by the truncation of more than 60% of the kernels. The introduction of the token blacklist has drastically reduced the length of the sequences, and the accuracy of the classifier has returned to the levels of the OpenCL dataset, with an improvement in the classification of the CPU class while maintaining, on average, the same levels of accuracy in the GPU class. The higher variance in GPU class accuracy disappears as the sequence length increases (llvm-b-2048). This behaviour is an indication that LLVM input is more difficult to be classified than OpenCL code. The difficulty depends on the highly rigid structure of an assembly-like language that requires a longer-term memory of the LSTM layers as the information is distributed over more extended sequences. The modest unbalance of the dataset contributes to creating difficulties for the classification of the disadvantaged class.

Table 4.2 shows the average results of each classifier in which we can see the improvement over DeepTune [22], and the growth of performances of the LLVM classifier when the token-blacklist is applied. The LLVM-B slightly exceeded the performance of the OpenCL classifier.

Table 4.3: Range of hyper-parameters values tested during the network and training grid-searches.

Grid search	Hyper-parameter	Values
Network	Input size	1024, 2048, 4096
	Padding/Truncating strategies	pre, post
	Embedding size	64, 128
	Conv kernel size	5, 7, 9
	Conv kernel number	32, 64, 128
	Dense layer size	64, 128, 256
Training	Batch size	16, 32, 48, 64
	Aux output weight loss	0.0, 0.1, ..., 1.0

4.2.3 CNN reference grid-search

To the best of our knowledge, no previous works introduced convolution neural networks for kernel-device mapping on heterogeneous platforms. The CNN architecture I propose for language modelling is inspired by the one adopted in [105] for sentence sentiment classification. Since it has a relatively high number of hyper-parameters, and the number of possible network configurations increases exponentially with the number of parameters considered, I selected a set of hyper-parameters of interest and split them into two groups, optimised separately using two successive grid-searches. I are interested in highlighting how useful this kind of networks may be at extracting features from source code. For selecting the best configuration among a set grid-search points, I adopted the following procedure:

1. Summarise each set of repetitions related to the same network configuration with its MCC mean and standard deviation.
2. Sort grid-search configurations by their average MCC.
3. Select the top 3 configurations.
4. Declare the configuration with the smallest standard deviation to be the best.

All experiments performed with Hyper-parameters exploration were made on the AMD dataset, using sequences in IR obtained trough a compilation without optimisation steps (`-O0`).

Figure 4.9 shows the performance of all network hyper-parameters (top) and training hyper-parameters (bottom) configurations considered, sorted by mean accuracy.

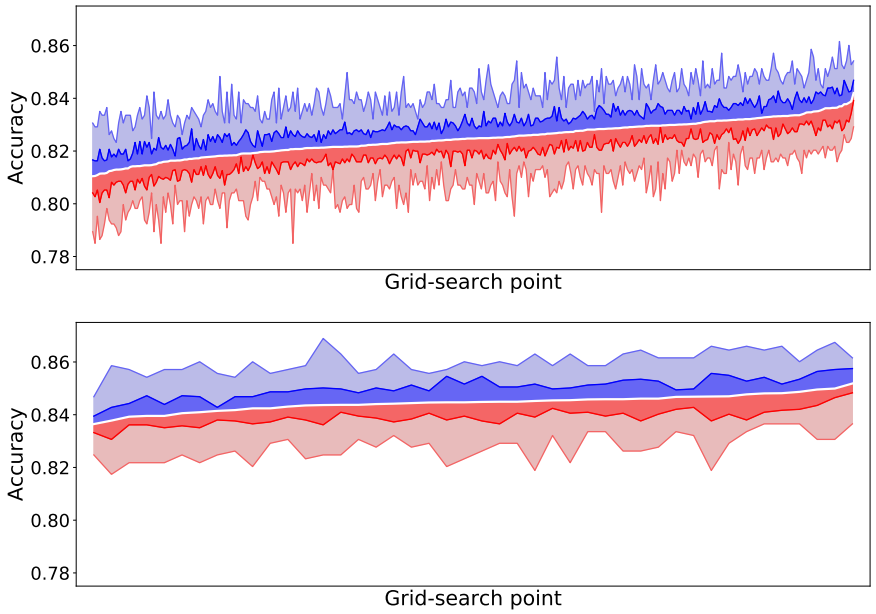


Figure 4.9: Classification accuracy improvements of network (top) and training (bottom) hyper-parameters exploration. The tracks represent classification performance quartiles of grid-search configurations explored, sorted by average accuracy. The mean trend is represented by the white track in the middle of each plot.

Table 4.4: Evolution of the CNN network and training hyper-parameters through the different grid-search phases. At first, we assumed batch-size equal to 32 and auxiliary output weight loss equal to 0.2. Then, we performed the network grid-search for establishing networks hyper-parameters. At last, we investigated batch-size and auxiliary output weight loss values with the training grid-search.

Hyper-parameter	Init	Network	Training
Input size	-	2048	-
Padding/Truncating strategy	-	pre	-
Embedding size	-	128	-
Conv kernel size	-	9	-
Conv kernel number	-	32	-
Dense layer size	-	256	-
Batch size	32	-	64
Aux output weight loss	0.2	-	1.0

The first grid-search aims at optimising hyper-parameters related to the CNN network structure. The search space is the hyper-cube defined by the Cartesian product of the network parameters in Table 4.3, while batch-size and auxiliary output

weight loss were fixed to 32 and 0.2, as specified by Table 4.4. The best configuration so far has a mean MCC of 0.672 and a standard deviation of 0.014. Mean classification accuracy equals 84.2% with a standard deviation of 0.7% and the best value of 86.2%.

The second grid-search improves the CNN performance selecting better batch-size and auxiliary output weight loss values, keeping the network hyper-parameters fixed. At the end of the two optimisation procedures, mean MCC approaches 0.693, with a standard deviation of 0.015, while classification accuracy reaches 85.2% with a standard deviation of 0.7% and the best value of 86.9%.

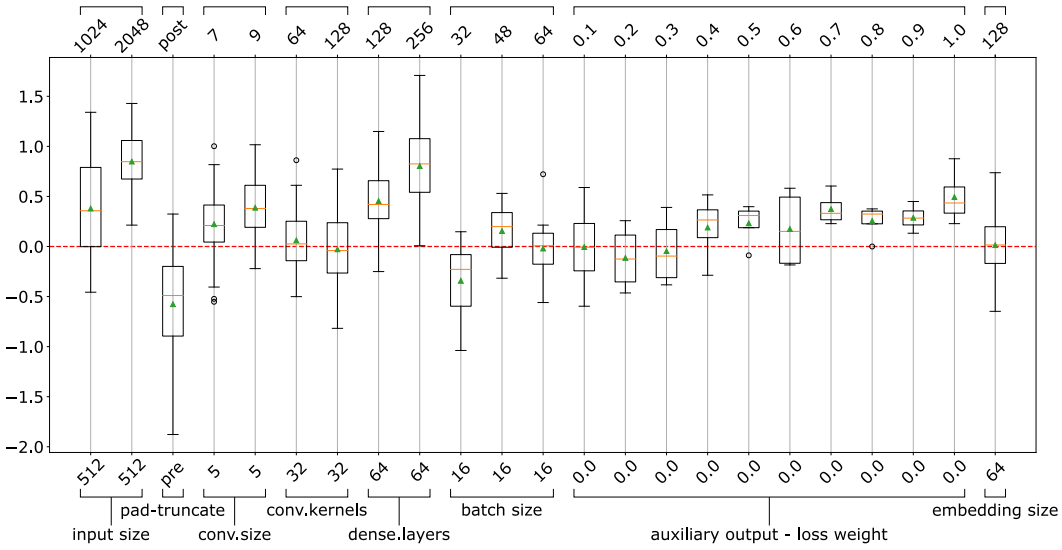


Figure 4.10: Single hyper-parameter sensitivity analysis in CNN models.

I analysed the outcome of the grid-search procedures for figuring out what hyper-parameters impact on classification accuracy most. Given an hyper-parameter p and two values A and B it assumes, we are interested in computing the distribution of the accuracy improvement granted by moving the value of p from A to B. For each CNN configuration C_A where p equals A, another one is selected, called C_B, characterised by having p equal to B and all the remaining hyper-parameters in common with C_A. Then, the classification accuracy difference is computed between C_A and C_B.

Figure 4.10 shows the outcome of such a procedure for all network and training hyper-parameters. It shows that sequence input-size and the number of neurons in the fully-connected layer of the classifier (dense-layers) are particularly promising. In essence, increasing them seems to be beneficial, and moving them from the minimum tested values to the maximum one always leads to better performance, no matter the values remaining hyper-parameters assume. In the best cases, accuracy

improvements are over 1%. Auxiliary output loss weight exposes a similar behaviour when exceeding 0.6. It is observed that increasing convolution kernel size from 5 to 9 generally leads to better results, while applying padding to the end of the sequences seem to be detrimental.

4.2.4 CNN-RNN comparison

Table 4.5: Architecture of the two machine learning models tested.

Hyper-parameter	RNN	CNN
Input size	1024	2048
Padding/Truncating strategy	pre	pre
Embedding size	128	64
Conv kernel size	-	9
Conv kernel number	-	32
LSTM layer 1 size	64	-
LSTM layer 2 size	64	-
Dense layer size	32	256
Batch size	32	64
Aux output weight loss	0.2	1.0

Table 4.6: Outcomes of experiments comparing different language modelling networks and token filtering strategies on the Nvidia data-set in terms of classification accuracy and MCC. Also, we reported speedup with respect to a static kernel mapper, mapping each kernel on GPU.

Dataset	Filtering	Length	RNN			CNN		
			ACC [%]	MCC {-1, +1}	S.Up [x]	ACC [%]	MCC {-1, +1}	S.Up [x]
LLVM -O0	-	3092	81.16	0.614	1.37	84.47	0.685	**1.69**
	Blacklist	900	82.22	0.636	1.40	85.40	0.704	1.58
	1 e-3	651	81.90	0.630	1.33	**85.60**	**0.708**	1.61
	3 e-3	553	81.16	0.615	1.08	84.79	0.691	1.45
	6 e-3	497	81.45	0.620	1.33	83.59	0.666	1.47
	8 e-3	465	79.24	0.576	1.03	81.78	0.629	0.97
LLVM -O2	-	3160	80.73	0.606	1.42	83.77	0.670	**1.60**
	Blacklist	949	82.53	0.643	1.41	84.20	0.681	1.50
	1 e-3	719	**82.60**	**0.644**	1.38	**84.78**	**0.691**	1.57
	3 e-3	602	82.39	0.640	1.03	84.63	0.688	1.54
	6 e-3	515	81.64	0.624	0.94	83.38	0.661	1.57
	8 e-3	495	80.60	0.604	0.77	83.31	0.660	1.51
OpenCL	-	2656	81.18	0.615	**1.49**	**83.00**	**0.656**	**1.51**

Table 4.7: Outcomes of experiments comparing different language modelling networks and token filtering strategies on the AMD data-set in terms of classification accuracy and MCC. Also, we reported speedup with respect to a static kernel mapper, mapping each kernel on CPU.

Dataset	Filtering	Length	RNN			CNN		
			ACC [%]	MCC {-1, +1}	S.Up [x]	ACC [%]	MCC {-1, +1}	S.Up [x]
LLVM `-O0`	-	3092	79.83	0.581	3.23	**85.32**	**0.695**	3.50
	Blacklist	900	81.97	0.625	3.23	83.97	0.667	3.50
	1 e-3	651	82.08	0.627	3.21	84.79	0.684	**3.88**
	3 e-3	553	81.47	0.615	3.17	84.76	0.683	3.80
	6 e-3	497	80.55	0.596	3.38	83.74	0.662	3.48
	8 e-3	465	79.55	0.575	2.16	83.76	0.662	2.15
LLVM `-O2`	-	3160	79.25	0.568	3.01	84.54	0.679	3.86
	Blacklist	949	81.82	0.622	2.97	83.81	0.663	3.59
	1 e-3	719	81.50	0.615	3.34	**84.78**	**0.683**	**3.91**
	3 e-3	602	**82.50**	**0.637**	3.20	84.23	0.672	3.33
	6 e-3	515	80.35	0.592	3.08	83.52	0.657	3.24
	8 e-3	495	79.56	0.575	2.07	83.96	0.666	3.63
OpenCL	-	2656	81.75	0.622	**3.69**	**84.78**	**0.684**	**3.26**

I tested how good the CNN detailed in subsection 4.2.3 is at classifying kernels, and I compared it with a network exploiting a RNN-based language modelling. Table 4.5 details the architecture of the two networks.

Table 4.6 and Table 4.7 report the outcome of experiments performed on both the AMD and the Nvidia datasets. For each kernel available, I applied the processing procedures detailed in subsection 4.1.1. Each kernel is tokenised and atomised using:

- the LLVM-based pipeline described in this work with two different *clang* optimisation flags, for investigating how source code transformation impacts on classification.
- the OpenCL-based approach proposed in [22].

Regarding LLVM token sequences, I tested two token filtering strategies:

- the blacklist approach proposed in [9].
- four different *Tf-Idf* thresholds.

I evaluated experiments in terms of classification accuracy, MCC and speedup. Red bold values highlight what preprocessing methodology gives the best result for each metric-model pair. Instead, the highlighted values stress the filtering strategies with which the CNN behaves better for each tokenising methodology.

The next two subsections discuss the classification performance of tested models and how CNN language modelling impact training time.

The CNN model always outperforms the RNN one, independently from the device used for kernel labelling, the preprocessing strategies, the filtering threshold.

Concerning classification metrics, the CNN performs better on LLVM `-O0` sequences in the AMD dataset, where it reaches a mean classification accuracy of 85.32% and a MCC of 0.695. The CNN grants a boost in classification performance between +3.00% and +5.50% on unfiltered sequences, providing a solid improvement over RNN. The same conclusion can be drawn from Nvidia results which are characterised by a slight reduction of the gap between classification accuracy and MCC offered by the two models. Here, the best mean classification metrics achieved by the CNN are 84.47% accuracy and 0.685 MCC on LLVM `-O0` sequences.

Token filtering improves models classification performance of LLVM sequences in most cases. Token blacklist, proposed in [9], and *Tf-Idf* filtering with thresholds 0.001 and 0.003 are the three strategies that often behaves better. RNN performances are more influenced by token filtering than the CNN ones. That is especially true in the AMD dataset, where using a threshold equal to 0.003 on the LLVM `-O2` sequences provides a boost of +3.25% in accuracy and +0.069 in MCC. Instead, the CNN is less sensitive to filtering strategies as the best improvement is +1.13%, achieved applying *Tf-Idf* filtering on LLVM `-O0` sequences. High *Tf-Idf* thresholds, such as 0.008, usually lead to bad performance, causing an accuracy drop of 2.69% in the worst case. In those situations, the filtering algorithm starts removing tokens with high informative content not homogeneously, making to learn the appropriate features for carrying on the classification tasks challenging.

Table 4.8: Amount of time required for training CNN and RNN models for different batch-size and input-size. Experiments were run on an Nvidia Titan XP GPU with 12GB RAM.

Batch size	Input size	RNN [s]	CNN [s]	Δ [%]	S.Up [x]
32	1024	1996	412	-79.4	4.8
	2048	3580	492	-86.3	7.3
64	1024	1216	299	-75.4	4.1
	2048	1812	414	-77.2	4.4

Training time is one of the most significant issues when dealing with deep machine learning models. Since input-size and batch-size are the two hyper-parameters that affect training time most, comparing the two proposed models as they are, would not have been fair. I computed the average time CNN and RNN require for being cross-validated using sequences of the same length and an equal number

of samples in mini-batches. Results reported in Table 4.8 show the CNN always performs consistently better than the RNN, ensuring between 4.1x and 7.3x speedup in training time.

4.2.5 Misclassification Impact

Table 4.9: Comparisons between the speedup obtained by *DeepTune* [22] and the one obtained by the best CNN on IR.

Dataset	*DeepTune* [22]	*DeepLLVM*	
		LLVM `-O0`	LLVM `-O2`
AMD	3.43	3.50	3.86
Nvidia	1.42	1.69	1.60

For devising how the proposed machine learning model behaves in a real-world scenario, evaluating classification accuracy and MCC is not sufficient. Typical machine learning metrics take care of checking the number of correct predictions over the total number of samples but do not consider the impact of mapping a kernel on the wrong device. Missing the best compute unit may not result in a significant penalty in terms of runtime or speedup.

For checking the impact of misclassified kernels, I measured the speedup granted by the two machine learning models tested. I computed Speedup using the same approach proposed in [22]. The time required for running all the OpenCL kernels on the device predicted by a classifier is divided by the time required for mapping all kernels on the device whose label is most common in the labelled dataset. Experiment results are summarised in Table 4.6 and Table 4.7. The CNN always outperforms the RNN model on IR sequences analysis. It reaches a mean speedup of 1.69x and 1.60x on the Nvidia dataset while reaching 3.50x and 3.86x on AMD labelled data. Moreover, the best average speedup obtained by the CNN outperforms results presented in [22] on both dataset as shown in Table 4.9. The RNN module obtains better results analysing OpenCL kernels from the AMD dataset, ensuring a speedup of 3.69x. Nonetheless, this speedup is smaller than the 3.91x ensured by the CNN on LLVM `-O2` sequences filtered with a *Tf-Idf* threshold of 1e-3.

4.2.6 Summary of findings

The results obtained highlight the following main findings:

- using CNN-based language modelling networks, in the context of kernel-device mapping, is a promising method for extracting features from source code.

They provide mean classification accuracy, MCC and speedup higher than the one provided by the RNN-based network, reaching 85.32% classification accuracy, 0.695 MCC and 3.86x speedup in the best cases for the considered dataset;

- CNN behaves comparably to the RNN models without requiring token filtering techniques which seem to be less effective on convolutive networks;
- CNN architecture I proposed requires much lower GPU training time with respect to the RNN one. Indeed, CNN ensures a 4x - 7x reduction in training time over RNN. It means that such models are more comfortable to explore for testing different architectures and hyper-parameters configurations extensively;
- The input sequences size, the number of units in the fully-connected layer of the classifier and the auxiliary output weight loss are the CNN hyper-parameters that promise to impact classification metrics most.
- The RNN network reports a more significant variation in performance when the sequence length is reduced. At the same time, CNN is less affected by *Tf-Idf* filtering because of the Global Max Pooling layer acts as a filter itself.
- Finally, I confirmed that using IR is a valuable representation for performing kernel analysis and using it does imply penalties neither on classification metrics nor in speedup.

4.3 Final Remarks

In this chapter, I presented a LLVM based code classification method and I explored the impact of neural network models on feature extraction and classification problems applied to source code in the intermediate representation.

Using an LLVM compiler, I obtained a general and optimised low-level representation (IR) of a source code written in a high-level programming language. At the LLVM-IR level, the code can be manipulated and filtered for condensing complex syntactic language elements in a restricted set of keywords (tokenisation procedure) that once translated in sequences of numbers (atomization procedure) are suitable for being used as input for the deep neural network classifier.

I evaluated the performances of our LLVM-based classifier using a dataset of OpenCL kernels properly manipulated with our tokenisation - atomization strategy. Furthermore, through a TF-IDF weight analysis, I remove less informative tokens thus reducing the input dimension and being able to obtain an accuracy of the classifier with a median value of 86% (5% better compared to 81% achieved by the state-of-the-art code classifier DeepTune, Table 4.2).

Given the absence, in literature, of a CNN-based language modelling networks for kernel-device mapping I explored the hyper-parameters of a CNN model in order to obtain a reference model. I explored 368 different hyper-parameters configurations, each cross-validated 20 times, reporting a statistical analysis of the results obtained.

I compared the best configuration of hyper-parameters for the CNN with the RNN-based network used in [22, 9] for different source code preprocessing and token filtering strategies, evaluating classification accuracy, MCC and speedup.

Results confirm that features extraction from IR is a valuable strategy for analysing sources without dealing with complex high-level constructs, and it can be done keeping all the information required for performing classification tasks in the context kernel-device mapping.

Chapter 5

Conclusions

During this research work, I worked on new generation manycore architectures for the development of innovative programming models in neuromorphic and heterogeneous architectures. My scientific contributions are framed on these architectures, in particular deep-learning for compilation chains, software-stack development and resource optimisation.

My achievements obtained in research on programming tools and middleware for manycore neuromorphic platforms are divided in two categories: optimisation of communication resources during SNN simulations and development of a software stack for the implementation of a parallel programming model based on message exchange. My achievements obtained in research on programming tools for heterogeneous platforms was the implementation of a source code classifier, analysing a LLVM-IR code with deep neural networks.

Optimisation of communication resources in SNN simulations

I proposed a methodology for profiling densely interconnected neuromorphic multi-chip manycore platforms for real-time SNN simulations. The methodology has been used to characterise reliability issues in the SpiNNaker platform, impossible to investigate using a biological network. I designed a custom SNN configurations to unveil both local and external network traffic issues. I proven that one of the causes of unreliability was due to packet conflicts in the internal router tree related to traffic congestion. This unreliability can be due to simultaneous usage of communication links of a router. Results show that, with a good neuron population placement, it is possible to improve simulation reliability by decreasing the total number of packets exchanged. I have modelled the mapping problem of complex directed graph (SNN) into the SpiNNaker processors-mesh. I have identified and

tested 4 methodologies to solve the problem. The *Naïve* method (a simple heuristics), the *Spectral* method (uses the graph eigendecomposition to obtain a planar representation of the SNN graph and performs the node association with the chip mesh through an ILP formulation), the *Scotch* method (uses the *Dual Recursive Bipartitioning* heuristic), the *Simulated Annealing* method (well-known *SA* procedure to minimise a cost function). I have defined the cost function of the placement problem using the synaptic elongation. I have chosen the cortical microcircuit as our benchmark network, and after performing several tests I highlight the performance of each method. The Spectral method was implemented in GHOST, a Python module compliant with the sPyNNaker tool-chain in order to demonstrate the effectiveness of the developed mapping approach with respect to random neuron placement. Finally comparisons were made between configurations produced by PACMAN and GHOST. From these simulations was evident that GHOST is capable to reduce the number of used cores, results in lower R2R traffic, 96X when GHOST is adopted.

Software stack for the implementation of MPI

The architecture provide an inefficient unicast communication protocol, unsuitable for the development of a communication library such as MPI. For these reasons, I developed a communication middleware (MCM) based on the Multicast protocol. I reduce the complexity of the internal transmissions, by implementing unicast communications avoiding the supervision of the monitor processor. On top of MCM I designed the Application Command Framework (ACF), and Application Command Protocol (ACP) a new method to be adopted at the application level for spreading commands and manage the private memory of the chip processors. It provides a abstraction level of the memory (Memory Entites). Users can easily access to all application Memory Entites.

To prove the advantages of our ACF I modify two SpiNNaker application enabling them to use our library. The first applicatioa is used during the configuration phase of SpiNNaker board, and the second application is a neuron model used during the SNN simulation phase. In the first application, I demonstrated the advantages introduced by ACF in the run-time feeding of configuration applications. The use of ACF can speed-up host-to-boards data transmission during the configuration of SpiNNaker platforms. Exploiting the concurrency of the system I have been able to get an improvement of 3 times on the data forwarding inside the board. In the second application, I demonstrated the run-time flexibility introduced by the ACF embedded in the neuron model application, implementing two different real simulation scenarios: i) a two-phases SNN-Classifier designed for discriminating the handwritten number and ii) a chain of neurons with run-time re-configuration parameters. Both the implemented applications were demonstrated to be flexible, scalable and expandable.

Lastly, I described an implementation of the MPI paradigm on the SpiNNaker neuromorphic platform exposing a programming model for the development of parallel applications without knowledge of the interconnections between the computing units of the underlying architecture. In the case of SpiNNaker, the implementation of MPI take advantage of the technology offered by on-chip routers, obtaining efficient communication by using the ACF and memory entities. This software stack creates a simple working framework offering a universally known programming model capable of making the SpiNNaker architecture available for a wide range of applications. I benchmarking our MPI implementation, showing its linear scaling performances executing two MPI programs. The first application was an N-Body simulation where 2 k particles were simulated on 240 processors with a speed-up of 194 x and an efficiency of 80% when compared to the serial version running on a single SpiNNaker core. I also presented an MPI implementation of a DNA sequence matching algorithm. Results show that the scalability of the SpiNNaker board reaches an ideal profile, 98% of efficiency, when using more than 100 processors, a 90% efficiency using 600 processors, reaching 88% efficiency when all 767 application processors are used.

Source code classifier with deep learning

The objective of this research is to provide the current compilation chains of a more complex code analysis mode capable of making complex decisions that would otherwise be difficult to codify in a set of rules. I have therefore trained deep learning models capable of automatically learning features from source code. In particular I wanted to show that it is possible to analyze code in intermediate representation. I developed a LLVM-IR code classifier using two neural network models (CNN and LSTM) and make some comparisons between them. The code sequences before to be passed to the neural network classifier need to be transformed for condensing language elements in a restricted set of keywords (tokens), filtered for removing less informative tokens, and then transformed in numbers. I evaluated the performances of our LLVM-based classifier using a dataset of OpenCL kernels labeled with the best compute units in terms of runtime (CPU or GPU). Then, I explored the hyper-parameters of a CNN model in order to obtain a reference model. I explored 368 different hyper-parameters configurations, reporting a statistical analysis of the results obtained. I compared the best configuration of hyper-parameters for the CNN with the RNN-based network for different source code preprocessing and token filtering strategies, evaluating classification accuracy, MCC and speedup. Results confirm that features extraction from IR is a valuable strategy for analysing sources without dealing with complex high-level constructs, and it can be done keeping all the information required for performing classification tasks in the context kernel-device mapping.

List of Publications

[**International Journals**]

Urgese, G., Barchi, F., Macii, E., & Acquaviva, A. (2016). **Optimizing network traffic for spiking neural network simulations on densely interconnected manycore neuromorphic platforms.** IEEE Transactions on Emerging Topics in Computing, 6(3), 317-329.

Barchi, F., Urgese, G., Siino, A., Di Cataldo, S., Macii, E., & Acquaviva, A. (2019). **Flexible on-line reconfiguration of multi-core neuromorphic platforms.** IEEE Transactions on Emerging Topics in Computing.

Urgese, G., Barchi, F., Parisi, E., Forno, E., Acquaviva A., & Macii E., (2019). **Benchmarking a manycore neuromorphic platform with an MPI-based DNA sequence matching algorithm.** MDPI Electronics.

[**Proceedings of International Conferences and Book Chapters**]

Urgese, G., Barchi, F., & Macii, E. (2015, September). **Top-down profiling of application specific manycore neuromorphic platforms.** In 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC) (pp. 127-134). IEEE.

Siino, A., Barchi, F., Davies, S., Urgese, G., & Acquaviva, A. (2016, September). **Data and commands communication protocol for neuromorphic platform configuration.** In 2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC) (pp. 23-30). IEEE.

Nittala, R. V., Barchi, F., Urgese, G., & Acquaviva, A. (2016, September). **Toolchain integration of runtime variability and aging awareness in multicore platforms.** In 2016 Forum on Specification and Design Languages (FDL) (pp. 1-8). IEEE.

Barchi, F., Urgese, G., Macii, E., & Acquaviva, A. (2017, September). **An efficient MPI implementation for multi-core neuromorphic platforms.** In 2017 New Generation of CAS (NGCAS) (pp. 273-276). IEEE.

Urgese, G., Peres, L., Barchi, F., Macii, E., & Acquaviva, A. (2018, September). **Multiple alignment of packet sequences for efficient communication in a many-core neuromorphic system: work-in-progress.** In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES). IEEE Press.

Barchi, F., Urgese, G., Macii, E., & Acquaviva, A. (2018, September). **Impact of graph partitioning on SNN placement for a multi-core neuromorphic architecture: work-in-progress.** In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES). IEEE Press.

Barchi, F., Urgese, G., Acquaviva, A., & Macii, E. (2018, October). **Directed Graph Placement for SNN simulation into a multi-core GALS architecture.** In 2018 IFIP/IEEE International Conference on Very Large Scale Integration (pp. 19-24). IEEE Press.

Barchi, F., Urgese, G., Acquaviva, A., & Macii, E. (2018, October). **Mapping Spiking Neural Networks on Multi-core Neuromorphic Platforms: Problem Formulation and Performance Analysis.** Chapter in VLSI-SoC: Design and Engineering of Electronics Systems Based on New Computing Paradigms, IFIP Advances in Information and Communication Technology (pp. 167-186). Springer.

Barchi, F., Urgese, G., Macii, E., & Acquaviva, A. (2019, June). **Code mapping in heterogeneous platforms using deep learning and LLVM-IR.** In Proceedings of the 56th Annual Design Automation Conference 2019 (p. 170). ACM.

Nomenclature

Acronyms / Abbreviations

ACF Application Command Framework

ACP Application Command Protocol

APs Application Processors

BNN Biological Neural Networks

CNN Convolutional Neural Networks

CRUD Create, Read, Update, Delete

DNN Deep Neural Network

DS Data Specification

DSE Data Specification Execution

DSG Data Specification Generation

FR Fixed Route

GHOST GrapH Optimiser Spinnaker Tool

GMP Global Max-Pooling

HBP European Human Brain Project

LLVM Low Level Virtual Machine

LSTM Long Short-Term Memory

MC Multicast packets

MCM Multicast Communication Middleware

ME Memory Entities

MP Monitor Processor

NMI Neuron Model Implementation

NN Nearest Neighbour

ODE Ordinary Differential Equations

PP Point-to-Point packets

RTE Runtime Error

SARK SpiNNaker Application Runtime Kernel

SCP SpiNNaker Command Protocol

SDP SpiNNaker Datagram Protocol

SNN Spiking Neural Networks

SpiNNaker Spiking Neural Network Architecture

SWE Software Error

TCM Tightly Coupled Memory

vME Virtual Memory Entity

Bibliography

[1] Larry F Abbott. "Lapicque's introduction of the integrate-and-fire model neuron (1907)". In: *Brain research bulletin* 50.5 (1999), pp. 303–304.

[2] M. Abeles. "Synfire chains". In: *Scholarpedia* 4.7 (2009). revision #150018, p. 1441. DOI: 10.4249/scholarpedia.1441.

[3] Rohit Aggarwal et al. "IR2Vec: A Flow Analysis based Scalable Infrastructure for Program Encodings". In: *arXiv preprint arXiv:1909.06228* (2019).

[4] Sacha J van Albada et al. "Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model". In: *Frontiers in neuroscience* 12 (2018), p. 291.

[5] Miltiadis Allamanis et al. "A survey of machine learning for big code and naturalness". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–37.

[6] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[7] Amir H Ashouri et al. "A survey on compiler autotuning using machine learning". In: *arXiv preprint arXiv:1801.04405* (2018).

[8] Amir Hossein Ashouri et al. "Cobayn: Compiler autotuning framework using bayesian networks". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.2 (2016), p. 21.

[9] Francesco Barchi et al. "Code Mapping in Heterogeneous Platforms Using Deep Learning and LLVM-IR". In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2019, pp. 1–6.

[10] Francesco Barchi et al. "Directed Graph Placement for SNN Simulation into a multi-core GALS Architecture". In: *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2018, pp. 19–24.

[11] Trevor Bekolay et al. "Nengo: a Python tool for building large-scale functional brain models". In: *Frontiers in neuroinformatics* 7 (2013).

[12] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. "Neural code comprehension: a learnable representation of code semantics". In: *Advances in Neural Information Processing Systems*. 2018, pp. 3585–3597.

[13] Basabdatta S Bhattacharya et al. "Engineering a thalamo-cortico-thalamic circuit on SpiNNaker: a preliminary study toward modeling sleep and wakefulness". In: *Frontiers in neural circuits* 8 (2014).

[14] Louis Blin, Ahsan Javed Awan, and Thomas Heinis. "Using Neuromorphic Hardware for the Scalable Execution of Massively Parallel, Communication-Intensive Algorithms". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. 2018, pp. 89–94. DOI: 10.1109/UCC-Companion.2018.00040.

[15] Kwabena Boahen. "Point-to-point connectivity between neuromorphic chips using address events". In: *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on* 47.5 (2000), pp. 416–434.

[16] Robert S Boyer and J Strother Moore. "A fast string searching algorithm". In: *Communications of the ACM* 20.10 (1977), pp. 762–772.

[17] Andrew D Brown et al. "SpiNNaker—programming model". In: *IEEE Transactions on Computers* 64.6 (2014), pp. 1769–1782.

[18] Roberto Capuzzo-Dolcetta, Mario Spera, and Davide Punzo. "A fully parallel, high precision, N-body code running on hybrid computing platforms". In: *Journal of Computational Physics* 236 (2013), pp. 580–593.

[19] Kristofor D Carlson et al. "GPGPU accelerated simulation and parameter tuning for neuromorphic applications". In: *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE. 2014, pp. 570–577.

[20] John Cavazos et al. "Rapidly selecting good compiler optimizations using performance counters". In: *Code Generation and Optimization, 2007. CGO'07. International Symposium on*. IEEE. 2007, pp. 185–197.

[21] Fan Chung. "Laplacians and the Cheeger inequality for directed graphs". In: *Annals of Combinatorics* 9.1 (2005), pp. 1–19.

[22] Chris Cummins et al. "End-to-end deep learning of optimization heuristics". In: *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE. 2017, pp. 219–232.

[23] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[24] Mike et al. Davies. "Loihi: A neuromorphic manycore processor with on-chip learning". In: *IEEE Micro* 38.1 (2018), pp. 82–99.

[25] Andrew P Davison et al. "PyNN: a common interface for neuronal network simulators". In: *Frontiers in neuroinformatics* 2 (2009), p. 11.

[26] Yufei Ding et al. "Autotuning algorithmic choice for input sensitivity". In: *ACM SIGPLAN Notices*. Vol. 50. 6. ACM. 2015, pp. 379–390.

[27] Charles M Fiduccia and Robert M Mattheyses. "A linear-time heuristic for improving network partitions". In: *Papers on Twenty-five years of electronic design automation*. ACM. 1988, pp. 241–247.

[28] Steve Furber. "Large-scale neuromorphic computing systems". In: *Journal of neural engineering* 13.5 (2016), p. 051001.

[29] Steve Furber. *SpiNNaker - a chip multiprocessor for neural network simulation. Datasheet. v2.02*. 2011. URL: `https://solem.cs.man.ac.uk/documentation/datasheet/SpiNN2DataShtV202.pdf`.

[30] Steve Furber, Steve Temple, and Andrew Brown. "On-chip and inter-chip networks for modeling large-scale neural systems". In: *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*. IEEE. 2006, 4–pp.

[31] Steve B Furber et al. "Overview of the spinnaker system architecture". In: *Computers, IEEE Transactions on* 62.12 (2013), pp. 2454–2467.

[32] Steve B Furber et al. "The spinnaker project". In: *Proceedings of the IEEE* 102.5 (2014), pp. 652–665.

[33] Francesco Galluppi et al. "A hierachical configuration system for a massively parallel neural hardware platform". In: *Proceedings of the 9th conference on Computing Frontiers*. ACM. 2012, pp. 183–192.

[34] Norman E et al. Gibbs. "A comparison of several bandwidth and profile reduction algorithms". In: *ACM Transactions on Mathematical Software (TOMS)* 2.4 (1976), pp. 322–330.

[35] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[36] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. "Portable mapping of data parallel programs to OpenCL for heterogeneous systems". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2013, pp. 1–10.

[37] John L Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[38] Kun He et al. "A novel task-duplication based clustering algorithm for heterogeneous computing environments". In: *IEEE Transactions on Parallel and Distributed Systems* 30.1 (2018), pp. 2–14.

[39] R Nigel Horspool. "Practical fast searching in strings". In: *Software: Practice and Experience* 10.6 (1980), pp. 501–506.

[40] Eugene M Izhikevich. "Simple model of spiking neurons". In: *IEEE Transactions on neural networks* 14.6 (2003), pp. 1569–1572.

[41] Yunlian Jiang et al. "Exploiting statistical correlations for proactive prediction of program behaviors". In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM. 2010, pp. 248–256.

[42] Xin Jin, S.B. Furber, and J.V. Woods. "Efficient modelling of spiking neural networks on a scalable chip multiprocessor". In: *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*. June 2008, pp. 2812–2819. DOI: 10.1109/IJCNN.2008.4634194.

[43] Xin Jin et al. "Modeling spiking neural networks on SpiNNaker". In: *Computing in science & engineering* 12.5 (2010), pp. 91–97.

[44] Giuseppe Jurman, Samantha Riccadonna, and Cesare Furlanello. "A comparison of MCC and CEN error measures in multi-class prediction". In: *PloS one* 7.8 (2012), e41882.

[45] George Karypis and Vipin Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SIAM Journal on scientific Computing* 20.1 (1998), pp. 359–392.

[46] Nikola K Kasabov. *Springer Handbook of Bio/neuroinformatics*. Springer Science & Business Media, 2013.

[47] Esther Kaufmann, Abraham Bernstein, and Lorenz Fischer. "NLP-Reduce: A naive but domainindependent natural language interface for querying ontologies". In: *4th European Semantic Web Conference ESWC*. 2007, pp. 1–2.

[48] Jin Wook Kim, Eunsang Kim, and Kunsoo Park. "Fast matching method for DNA sequences". In: *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer. 2007, pp. 271–281.

[49] Scott et al. Kirkpatrick. "Optimization by simulated annealing". In: *Science* 220.4598 (1983), pp. 671–680.

[50] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[51] Yann LeCun et al. “Generalization and network design strategies”. In: *Connectionism in perspective* 19 (1989), pp. 143–155.

[52] Chen Liu et al. “Memory-efficient Deep Learning on a SpiNNaker 2 prototype”. In: *Frontiers in neuroscience* 12 (2018).

[53] Gengting Liu et al. “Network traffic exploration on a many-core computing platform: Spinnaker real-time traffic visualiser”. In: *2015 11th Conference on Ph. D. Research in Microelectronics and Electronics (PRIME)*. IEEE. 2015, pp. 228–231.

[54] Andrew L Maas et al. “Learning word vectors for sentiment analysis”. In: *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*. Association for Computational Linguistics. 2011, pp. 142–150.

[55] Wolfgang Maass. “Networks of spiking neurons: the third generation of neural network models”. In: *Neural networks* 10.9 (1997), pp. 1659–1671.

[56] Thomas Martinetz, Klaus Schulten, et al. “A" neural-gas" network learns topologies”. In: (1991).

[57] Karlheinz Meier. “A mixed-signal universal neuromorphic computing system”. In: *2015 IEEE International Electron Devices Meeting (IEDM)*. IEEE. 2015, pp. 4–6.

[58] Paul Merolla et al. “A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm”. In: *2011 IEEE custom integrated circuits conference (CICC)*. IEEE. 2011, pp. 1–4.

[59] Don Monroe. “Neuromorphic computing gets ready for the (really) big time”. In: *Communications of the ACM* 57.6 (2014), pp. 13–15.

[60] Antoine Monsifrot, François Bodin, and Rene Quiniou. “A machine learning approach to automatic production of compiler heuristics”. In: *International conference on artificial intelligence: methodology, systems, and applications*. Springer. 2002, pp. 41–50.

[61] Saber Moradi et al. “A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs)”. In: *IEEE transactions on biomedical circuits and systems* 12.1 (2017), pp. 106–122.

[62] Javier Navaridas et al. “SpiNNaker: Enhanced multicast routing”. In: *Parallel Computing* 45 (2015), pp. 49–66.

[63] Javier Navaridas et al. “Spinnaker: fault tolerance in a power-and area-constrained large-scale neuromimetic architecture”. In: *Parallel Computing* 39.11 (2013), pp. 693–708.

[64] Eustace Painkras et al. "SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation". In: *IEEE Journal of Solid-State Circuits* 48.8 (2013), pp. 1943–1953.

[65] Eunjung Park, John Cavazos, and Marco A Alvarez. "Using graph-based program characterization for predictive modeling". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM. 2012, pp. 196–206.

[66] Eunjung Park et al. "Predictive modeling in a polyhedral optimization space". In: *International journal of parallel programming* 41.5 (2013), pp. 704–750.

[67] Biagio Peccerillo and Sandro Bartolini. "PHAST-A portable high-level modern C++ programming library for GPUs and multi-cores". In: *IEEE Transactions on Parallel and Distributed Systems* 30.1 (2018), pp. 174–189.

[68] François Pellegrini. "A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries". In: *European Conference on Parallel Processing*. Springer. 2007, pp. 195–204.

[69] François Pellegrini. "Static mapping by dual recursive bipartitioning of process architecture graphs". In: *Scalable High-Performance Computing Conference, 1994., Proceedings of the*. IEEE. 1994, pp. 486–493.

[70] Tobias C Potjans and Markus Diesmann. "The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model". In: *Cerebral cortex* 24.3 (2014), pp. 785–806.

[71] A. Rast et al. *AERIE-P: AER Intersystem Exchange Protocol*. Capo Caccia, Sardinia, Italy, 2015.

[72] Alexander Rast et al. "An event-driven model for the SpiNNaker virtual synaptic channel". In: *The 2011 International Joint Conference on Neural Networks*. IEEE. 2011, pp. 1967–1974.

[73] Alexander D Rast et al. "A location-independent direct link neuromorphic interface". In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2013, pp. 1–8.

[74] Alexander D Rast et al. "Scalable event-driven native parallel processing: the SpiNNaker neuromimetic system". In: *Proceedings of the 7th ACM international conference on Computing frontiers*. 2010, pp. 21–30.

[75] Alexander D Rast et al. "The leaky integrate-and-fire neuron: A platform for synaptic model exploration on the spinnaker chip". In: *Neural Networks (IJCNN), The 2010 International Joint Conference on*. IEEE. 2010, pp. 1–8.

[76] Oliver Rhodes et al. "sPyNNaker: A Software Package for Running PyNN Simulations on SpiNNaker". In: *Frontiers in neuroscience* 12 (2018).

[77] Nadav Rotem et al. “Glow: Graph lowering compiler techniques for neural networks”. In: *arXiv preprint arXiv:1805.00907* (2018).

[78] Martino et al. Ruggiero. “A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC platforms with communication awareness”. In: *International Journal of Parallel Programming* 36.1 (2008), pp. 3–36.

[79] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[80] John W Sammon. “A nonlinear mapping for data structure analysis”. In: *IEEE Transactions on computers* 5 (1969), pp. 401–409.

[81] Johannes Schemmel et al. “Live demonstration: A scaled-down version of the brainscales wafer-scale neuromorphic system”. In: *2012 IEEE International Symposium on Circuits and Systems*. IEEE. 2012, pp. 702–702.

[82] Michael Schmuker, Thomas Pfeil, and Martin Paul Nawrot. “A neuromorphic network for generic multivariate data classification”. In: *Proceedings of the National Academy of Sciences* 111.6 (2014), pp. 2081–2086.

[83] Catherine D Schuman et al. “A survey of neuromorphic computing and neural networks in hardware”. In: *arXiv preprint arXiv:1705.06963* (2017).

[84] Thomas Sharp, Cameron Patterson, and Steve Furber. “Distributed configuration of massively-parallel simulation on SpiNNaker neuromorphic hardware”. In: *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE. 2011, pp. 1099–1105.

[85] Thomas Sharp, Rasmus Petersen, and Steve Furber. “Real-time million-synapse simulation of rat barrel cortex”. In: *Frontiers in neuroscience* 8 (2014).

[86] Kapil Kumar Soni, Rohit Vyas, and Amit Sinhal. “Importance of string matching in real world problems”. In: *Int. J. Eng. Comput. Sci* 3.6 (2014), pp. 6371–6375.

[87] Mark Stephenson and Saman Amarasinghe. “Predicting unroll factors using supervised classification”. In: *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society. 2005, pp. 123–134.

[88] John E Stone, David Gohara, and Guochun Shi. “OpenCL: A parallel programming standard for heterogeneous computing systems”. In: *Computing in science & engineering* 12.3 (2010), pp. 66–73.

[89] Evangelos Stromatias et al. "Power analysis of large-scale, real-time neural networks on SpiNNaker". In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2013, pp. 1–8.

[90] Indar Sugiarto et al. "High performance computing on spinnaker neuromorphic platform: A case study for energy efficient image processing". In: *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE. 2016, pp. 1–8.

[91] XLA Team et al. *Xla–tensorflow compiled. post in the google developers blog.* 2017.

[92] Steve Temple. *AppNote 4 - SpiNNaker Datagram Protocol (SDP) Specification.* Available at `https://spinnaker.cs.manchester.ac.uk/`. 2011.

[93] Steve Temple. *AppNote 5 - Spinnaker Command Protocol (SCP) Specification.* Available at `https://spinnaker.cs.manchester.ac.uk/`. 2011.

[94] Gianvito Urgese, Francesco Barchi, and Enrico Macii. "Top-down profiling of application specific many-core neuromorphic platforms". In: *Embedded Multicore/Manycore SoCs (MCSoC), 2015 IEEE 9th International Symposium on.* IEEE. 2015.

[95] Gianvito Urgese et al. "Multiple alignment of packet sequences for efficient communication in a many-core neuromorphic system: work-in-progress". In: *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems.* IEEE Press. 2018, p. 3.

[96] Gianvito Urgese et al. "Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms". In: *IEEE Transactions on Emerging Topics in Computing* 6.3 (2018), pp. 317–329. DOI: `10.1109/TETC.2016.2579605`.

[97] Sacha J et al. Van Albada. "Full-scale simulation of a cortical microcircuit on SpiNNaker". In: *Front. Neuroinform. Conference Abstract: Neuroinformatics.* Vol. 10. 2016.

[98] Stijn Van Dongen. "Graph clustering via a discrete uncoupling process". In: *SIAM Journal on Matrix Analysis and Applications* 30.1 (2008), pp. 121–141.

[99] Zheng Wang and Michael O'Boyle. "Machine Learning in Compiler Optimisation". In: *arXiv preprint arXiv:1805.03441* (2018).

[100] Sandra Wienke et al. "OpenACC—first experiences with real-world applications". In: *European Conference on Parallel Processing.* Springer. 2012, pp. 859–870.

[101] Qianfei Xue et al. “A parallel algorithm for DNA sequences alignment based on MPI”. In: *2014 International Conference on Information Science, Electronics and Electrical Engineering*. Vol. 2. IEEE. 2014, pp. 786–789.

[102] Wenpeng Yin et al. “Comparative study of cnn and rnn for natural language processing”. In: *arXiv preprint arXiv:1702.01923* (2017).

[103] Aaron R Young et al. “A Review of Spiking Neuromorphic Hardware Communication Systems”. In: *IEEE Access* 7 (2019), pp. 135606–135620.

[104] Peng Zhang et al. “Optimizing Streaming Parallelism on Heterogeneous Many-Core Architectures”. In: *IEEE Transactions on Parallel and Distributed Systems* (2020).

[105] Ye Zhang and Byron Wallace. “A sensitivity analysis of (and practitioners’ guide to) convolutional neural networks for sentence classification”. In: *arXiv preprint arXiv:1510.03820* (2015).

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.