Hardware design of spiking neural networks for energy efficient brain-inspired computing

Nassim Abderrahmane

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Impact du codage impulsionnel sur l'efficacité énergétique des architectures neuromorphiques

Nassim ABDERRAHMANE

Laboratoire d'Electronique, Antennes et Télécommunications (LEAT)

Présentée en vue de l'obtention du grade de docteur en électronique **d'**Université Côte d'Azur
Dirigée par : Benoît Miramond
Soutenue le : 16 décembre 2020

Devant le jury, composé de :
Michel Paindavoine, Professeur, Université Bourgogne Franche-Comté
Timothée Masquelier, Chargé de recherche CNRS, Centre de Recherche Cerveau et Cognition
Benoît Miramond, Professeur, Université Côte d'Azur
Jean Martinet, Professeur, Université Côte d'Azur
Sébastien Bilavarn, Maître de conférences, Université Côte d'Azur
Olivier Bichler, Ingénieur chercheur, CEA LIST

Impact du codage impulsionnel sur l’efficacité énergétique des architectures neuromorphiques

Jury :

Président

Jean Martinet, Professeur des universités, Université Côte d’Azur

Rapporteurs

Michel Paindavoine, Professeur des universités, Université Bourgogne Franche-Comté

Timothée Masquelier, Chargé de recherche CNRS, Centre de Recherche Cerveau et Cognition, Toulouse

Examinateurs

Benoît Miramond, Professeur des universités, Université Côte d’Azur

Sébastien Bilavarn, Maître de conférences, Université Côte d’Azur

Olivier Bichler, Ingénieur chercheur, Commissariat à l'Energie Atomique et aux Energies Alternatives - CEA LIST, Palaiseau

Résumé

Dans le contexte actuel, l'Intelligence Artificielle (IA) est largement répandue et s'applique à de nombreux domaines tels que les transports, la médecine et les véhicules autonomes. Parmi les algorithmes d'IA, on retrouve principalement les réseaux de neurones, qui peuvent être répartis en deux familles : d'une part, les Réseaux de Neurones Impulsionnels (SNNs) qui sont issus du domaine des neurosciences ; d'autre part, les Réseaux de Neurones Analogiques (ANNs) qui sont issus du domaine de l'apprentissage machine. Les ANNs connaissent un succès inédit grâce à des résultats inégalés dans de nombreux secteurs tels que la classification d'images et la reconnaissance d'objets. Cependant, leur déploiement nécessite des capacités de calcul considérables et ne conviennent pas à des systèmes très contraints. Afin de pallier ces limites, de nombreux chercheurs s'intéressent à un calcul bio-inspiré, qui serait la parfaite alternative aux calculateurs conventionnels basés sur l'architecture de Von Neumann. Ce paradigme répond aux exigences de performance de calcul, mais pas aux exigences d'efficacité énergétique. Il faut donc concevoir des circuits matériels neuromorphiques adaptés aux calculs parallèles et distribués.

Dans ce contexte, nous avons établi un certain nombre de critères en termes de précision et de coût matériel pour différencier les SNNs et ANNs. Dans le cas de topologies simples, nous avons montré que les SNNs sont plus efficaces en termes de coût matériel que les ANNs, et ce, avec des précisions de prédiction quasiment similaires. Ainsi, dans ce travail, notre objectif est de concevoir une architecture neuromorphique basée sur les SNNs. Dans cette perspective, nous avons mis en place un flot de conception composé de trois niveaux, qui permet la réalisation d'une architecture neuromorphique dédiée et adaptée aux applications d'IA embarquée.

Dans un contexte d'efficacité énergétique, nous avons réalisé une étude approfondie sur divers paradigmes de codage neuronal utilisés avec les SNNs. Par ailleurs, nous avons proposé de nouvelles versions dérivées du codage fréquentiel, visant à se rapprocher de l'activité produite avec le codage temporel, qui se caractérise par un nombre réduit d'impulsions (spikes) se propageant dans le SNN. En faisant cela, nous sommes en mesure de réduire le nombre de spikes, ce qui se traduit par un SNN avec moins d'événements à

traiter, et ainsi, réduire la consommation énergétique sous-jacente. Pour cela, deux techniques nouvelles ont été proposées : "First Spike", qui se caractérise par l'utilisation d'un seul spike au maximum par donnée ; "Spike Select", qui permet de réguler et de minimiser l'activité globale du SNN.

Dans la partie d'exploration RTL, nous avons comparé de manière quantitative un certain nombre d'architectures de SNN avec différents niveaux de parallélisme et multiplexage de calculs. En effet, le codage "Spike Select" engendre une régulation de la distribution des spikes, avec la majorité générée dans la première couche et peu d'entre eux propagés dans les couches profondes. Nous avons constaté que cette distribution bénéficie d'une architecture hybride comportant une première couche parallèle et les autres multiplexées. Par conséquent, la combinaison du "Spike Select" et de l'architecture hybride serait une solution efficace, avec un compromis efficace entre coût matériel, consommation et latence.

Enfin, en se basant sur les choix architecturaux et neuronaux issus de l'exploration précédente, nous avons élaboré une architecture évènementielle dédiée aux SNNs mais suffisamment programmable pour supporter différents types et tailles de réseaux de neurones. L'architecture supporte les couches les plus utilisées : convolution, pooling (mise en commun) et entièrement connectées. En utilisant cette architecture, nous serons bientôt en mesure de comparer les ANNs et les SNNs sur des applications réalistes et enfin conclure sur l'utilisation des SNNs pour l'IA embarquée.

Mots clés— Réseaux de Neurones Artificiels, Intelligence Artificielle, Réseaux de Neurones Impulsionnels, Codage Neuronal, Calcul Neuromorphique, Architecture Matérielle, Consommation d'Energie, Systèmes Embarqués

Abstract

Nowadays, Artificial Intelligence (AI) is a widespread concept applied to many fields such as transportation, medicine and autonomous vehicles. The main AI algorithms are artificial neural networks, which can be divided into two families: Spiking Neural Networks (SNNs), which are bio-inspired models resulting from neuroscience, and Analog Neural Networks (ANNs), which result from machine learning. The ANNs are experiencing unprecedented success in research and industrial fields, due to their recent successes in many application contexts such as image classification and object recognition. However, they require considerable computational capacity for their deployment which is not adequate to very constrained systems such as 'embedded systems'. To overcome these limitations, many researchers are interested in brain-inspired computing, which would be the perfect alternative to conventional computers based on the Von Neumann architecture (CPU/GPU). This paradigm meets computing performance but not energy efficiency requirements. Hence, it is necessary to design neuromorphic hardware circuits adaptable to parallel and distributed computing.

In this context, we have set criteria in terms of accuracy and hardware implementation cost to differentiate the two neural families (SNNs and ANNs). In the case of simple network topologies, we conducted a study that has shown that the spiking models have significant gains in terms of hardware cost when compared to the analog networks, with almost similar prediction accuracy. Therefore, the objective of this thesis is to design a generic neuromorphic architecture that is based on spiking neural networks. To this end, we have set up a three-level design flow for exploring and implementing neuromorphic architectures.

In an energy efficiency context, a thorough exploration of different neural coding paradigms for neural data representation in SNNs has been carried out. Moreover, new derivative versions of rate-based coding have been proposed that aim to get closer to the activity produced by temporal coding, which is characterized by a reduced number of spikes propagating in the network. In this way, the number of spikes can be reduced so that the number of events to be processed in the SNNs gets smaller. The aim in doing this approach is to reduce the hardware architecture's energy consumption. The

proposed coding approaches are: First Spike, which is characterized using at most one single spike to present an input data, and Spike Select, which allows to regulate and minimize the overall spiking activity in the SNN.

In the RTL design exploration, we quantitatively compared three SNN architectural models having different levels of computing parallelism and multiplexing. Using Spike Select coding results in a distribution regulation of the spiking data, with most of them generated within the first layer and few of them propagate into the deep layers. Such distribution benefits from a so-called 'hybrid architecture' that includes a fully-parallel part for the first layer and multiplexed parts to the other layers. Therefore, combining the Spike Select and the Hybrid Architecture would be an effective solution for embedded AI applications, with an efficient hardware and latency trade-off.

Finally, based on the architectural and neural choices resulting from the previous exploration, we have designed a final event-based architecture dedicated to SNNs supporting different neural network types and sizes. The architecture supports the most used layers: convolutional, pooling and fully-connected. Using this architecture, we will be able to compare analog and spiking neural networks on realistic applications and to finally conclude about the use of SNNs for Embedded Artificial Intelligence.

Keywords— Artificial Neural Networks, Artificial Intelligence, Spiking Neural Networks, Neural Coding, Neuromorphic Computing, Hardware Architecture, Energy Consumption, Embedded Systems

Acknowledgements

I would like to first express my deepest gratitude to my thesis supervisor, Professor Benoît Miramond, for the support and trust in accepting the scientific directions of my research work. I would like to thank him for his expert advice all along this thesis, his kindness and his availability in challenging and difficult circumstances. I am very grateful for all these sharing moments.

I would like to thank all the researchers who accepted to be part of this thesis jury. I would like to express my gratitude to Michel Paindavoine and Timothée Masquelier for agreeing to report this thesis and for the time they devoted to reviewing my work. I would also like to thank Jean Martinet, Olivier Bichler and Sébastien Bilavarn for having accepted to be part of my thesis jury.

I would also like to sincerely thank the members of the ebrAIn group and the EDGE team. I particularly thank Lyes Khacef for the many discussions that we have had over the last three years and for his pleasure of sharing.

I do not forget, of course, all the staff and doctoral students of LEAT with whom I shared all these moments of uncertainty and pleasure. All those moments I shared over coffee or during soccer matches with them made the atmosphere of daily work so pleasant.

I would like to thank my family, especially my mother Ouardia and my father Abderrahmane, my brother Karim, my sisters Houria, Atika, Samira and Nassiba and my brothers-in-law, Nabyl and Nabil, and their children for sharing with me their happiness and supporting me in my efforts. I thank you for your presence regardless the distance. I thank all my friends, with a special thought for Alexis and his wife, 2 x Amine, Youcef and Said to whom I wish all the best.

A special thought goes to Yasmina with whom I have shared happy and memorable moments during the last years and for her numerous reviews of my thesis and scientific articles.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ALM	**A**daptive **L**ogic **M**odule
ANN	**A**nalog **N**eural **N**etwork
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
AXI	**A**dvanced e**X**tensible **I**nterface
CMOS	**C**omplementary **M**etal **O**xide **S**emiconductor
CNN	**C**onvolutional **N**eural **N**etwork
DMA	**D**irect **M**emory **A**ccess
DNN	**D**eep **N**eural **N**etwork
DPU	**D**eep Learning **P**rocessor **U**nit
DSP	**D**igital **S**ignal **P**rocessor
DVS	**D**ynamic **V**ision **S**ensor
EBS	**E**vent-**B**ased **S**ensor
FNN	**F**ormal **N**eural **N**etwork
FiFo	**F**irst **i**n **F**irst **o**ut
FPA	**F**ully-**P**arallel **A**rchitecture
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
GTSRB	**G**erman **T**raffic **Si**gn **R**ecognition **B**enchmark
HA	**H**ybrid **A**rchitecture
HICANN	**H**igh-**I**nput **C**ount **A**nalog **N**eural **N**etwork
IP	**I**ntellectual **P**roperty
(L)IF	(**L**eaky) **I**ntegrate-and-**F**ire
LUT	**L**ook-**U**p **T**able
MLP	**M**ulti-**L**ayer **P**erceptron
MNIST	**M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology database
N2D2	**N**eural **N**etwork **D**esign & **D**eployment
NPU	**N**eural **P**rocessing **U**nit
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem

RAM	**R**andom-**A**ccess **M**emory
ROM	**R**read **O**nly **M**emory
RTL	**R**egister **T**ransfer **L**evel
SDRAM	**S**ynchronous **D**ynamic **RAM**
SNN	**S**piking **N**eural **N**etwork
SOPS	**S**ynaptic **O**peration **p**er **S**econd
STDP	**S**pike **T**iming **D**ependent **P**lasticity
TMA	**T**ime-**M**ultiplexed **A**rchitecture
VHDL	**VHSIC H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**peed **I**ntegrtaed **C**irctuis

Author's publication list

Journal papers

— Nassim Abderrahmane, Edgar Lemaire, and Benoît Miramond (2020). "Design Space Exploration of Hardware Spiking Neurons for Embedded Artificial Intelligence". In: *Neural Networks* 121, pp. 366 –386. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.09.024.

Patents

— Nassim Abderrahmane and Benoît Miramond (2020b). *Hardware Architecture for Spiking Neural Networks and Method of Operating*. International Patent submitted to "Office européen des brevets".

International conference papers

— Nassim Abderrahmane and Benoit Miramond (2020a). "Neural coding: adapting spike generation for embedded hardware classification". In: *International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8.

— Nassim Abderrahmane and Benoit Miramond (2019). "Information coding and hardware architecture of spiking neural networks". In: *Euromicro Conference on Digital System Design (DSD)*, "1–8".

— Lyes Khacef, Nassim Abderrahmane, and Benoit Miramond (2018). "Confronting machine-learning with neuroscience for neuromorphic architectures design". In: *International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8.

Chapter 1

General introduction

1.1 Context

Artificial neural networks are derived and inspired from the biological brain, and have become the most well-known and frequently used form of Artificial Intelligence. Even though these neural models have garnered a lot of interest in recent years, they stem from the 1940s with the apparition of the first computer. Subsequent work and advancements have led to the development of a wide variety of neural models. Convolutional Neural Network (CNN) is the most used one today. It is inspired from the biological visual perception mechanism of living beings. In 1959, Hubel and Wiesel have found that animals possess biological cells in their visual cortex which are detecting the light present in their receptive field (Hubel and Wiesel, 1959). In 1980, Fukushima was inspired by this finding and proposed what is considered as the CNN's predecessor : the neocognitron (Fukushima, 1980).

A decade later, LeCun et al. proposed the CNN revolutionary framework that is presented as a Deep Neural Network (DNN) composed of several layers of different types, called "LeNet-5" (LeCun et al., 1990). This DNN is specifically used to ensure the handwritten digits classification task. More than a decade later, several models have been proposed, they came with improvements and facilitations in training DNNs. Krizhevsky, Sutskever, and Hinton proposed one of them, the AlexNet model, it has a similar, but deeper, structure to LeNet-5 with more layers (Krizhevsky, Sutskever, and Hinton, 2017). This model was very successful, it brought out many methods and materials coming from other works that improved classification performances. Among them, we can cite VGGNet (Simonyan and Zisserman, 2014), ZFNet (Zeiler and Fergus, 2014), GoogleNet (Szegedy et al., 2015) and ResNet (He et al.,

2015). Recent advances on DNNs can be found in the paper proposed by Gu et al. in (Gu et al., 2015).

Meanwhile, these neural algorithms have become more popular with applications in several domains such as image and video recognition, image classification, medical image analysis, and natural language processing. This success can be awarded to two factors. First, the high-performance computing capabilities of modern CPU/GPU based computers that accelerated the implementation, the training and the inference stages. Second, the huge amount of available open source labeled data for training DNN algorithms, which increased the number of applications and contributed to the improvement of these algorithms.

These neural networks could be separated into three different generations, distinguished by the neural computation and coding. The first generation is composed of neural models based on the traditional McCulloch and Pitts neuron, which outputs discrete binary values (Schuman et al., 2017).

The second generation is characterized by the use of continuous activation functions within their neurons rising to more complex architectures, such as Boltzmann Machines (Ackley, Hinton, and Sejnowski, 1985), Hopfield Networks (Hopfield, 1982), Perceptrons, Multi-Layer Perceptrons (MLP) (Rumelhart, McClelland, and PDP Research Group, 1986) and Convolutional Neural Networks (Krizhevsky, Sutskever, and Hinton, 2017).

Finally, the third generation is composed of neural algorithms called Spiking Neural Networks (SNNs). In SNN models, the neural information is encoded into spikes or action potentials, inspired from neuroscience. Indeed, the spiking models and inter-neuron connections used with SNNs mimic biological neurons and synaptic communication mechanisms based on action potentials. According to this spike-based coding paradigm, SNNs are characterized by an event-based processing where spiking neuron computations are operated only when receiving input spikes.

1.2 Problematic

The recent performance of DNNs in terms of image classification has given them a major role in machine-learning field and AI research. After a first phase of offline experiments, these methods have started to proliferate in our daily life through autonomous applications everywhere and close to the user.

Thus, an increasing number of applications such as smart devices, IoTs or autonomous vehicles are requiring an efficient embedded implementation.

However, the fact that these neural networks process analog and continuous data makes their computation more complex and requires to be executed on high-performance computing platforms. As a result, their initial implementation on classical Von-Neuman architectures (CPU/GPU) is too resource- and energy-intensive for such constraining systems (embedded systems). In addition to their computational complexity, these intrinsically parallel algorithms are not adapted to the general-purpose sequential processors. Therefore, it is essential to, first, reduce the computation complexity of these neural systems and, second, deploy them on dedicated neuromorphic systems. These architectures are designed to fit the parallel and distributed computing paradigm of ANNs, which allows their implementation on embedded systems.

1.3 Objectives

Recent literature considers the third generation of neural algorithms, i.e. Spiking Neural Networks (SNNs), as the alternative to Analog Neural Networks (ANNs) for embedded artificial intelligence applications. According to their spike-based coding paradigm, SNNs perform an event-based processing: computation is held by a spiking neuron when and only when it receives an input spike. Without any stimulation, the neuron remains idle. Hence, computation is strictly performed for relevant information propagation, in contrast to ANNs, where the states of every neuron are updated periodically. Moreover, the computation is usually much simpler in spiking neurons than in formal neurons. Hence, SNNs are much more promising for low-power embedded hardware implementations than ANNs, considering the advantages in terms of event-driven computation and resource consumption brought by the spiking neuron model. Consequently, the objective of this thesis is the hardware design of a generic neuromorphic computing architecture capable of running any embedded AI application based on SNNs. Indeed, the neural algorithm structure depends on the user-defined application data, where different types and sizes of the neural network models could be employed. Moreover, spiking data is naturally very sparse but using rate-coding a huge quantity is processed by the SNN, therefore, we explore novel coding schemes aiming to reduce this spiking activity.

1.4 Contributions

The thesis contribution consists in studying more precisely the question of the digital hardware design of spiking neural networks for energy-efficient implementation of embedded AI applications using static data. The SNNs are all the more advantageous as we plan to execute them in dedicated accelerators. Then they take full advantage of the event-driven nature of data flows, the simplicity of their elementary operators and their local and distributed computing and learning properties. Several specific hardware solutions have already been proposed in the literature, but they are only solutions isolated from the overall design space where network topologies are often constrained by the characteristics of the circuit architecture. We recommend the opposite approach, which consists in generating the architecture that best supports the network topology. This is done by: first, using a neural coding scheme generating, as little as possible, spiking events while keeping the desired accuracy; second, using an architectural model that optimally uses the available hardware resources to best fit the energy- and processing-efficiency requirements.

Through this study, we therefore propose an exploration framework that makes it possible to evaluate the impact of different spiking models on the effectiveness of their hardware implementation. With this framework, we start from a wide variety of hardware implementation choices to incrementally refine the scope to find the most suitable at the end.

Moreover, we focus on neural coding for spike generation with SNNs by studying its impact on neuromorphic system efficiency. Our intuition is that using time-based coding instead of rate-based coding leads to a system with reduced power consumption but with lower accuracy results. When using time-based coding a few numbers of spikes are used to encode data, whereas with rate-based coding a greater spiking activity is recorded in the network, thus increasing resource and energy intensiveness of the system. However, at the same time, higher accuracy results are obtained with rate coding than with temporal coding. Therefore, we explore additional spike coding methods attempting to mimic time-based coding paradigm in terms of spiking activity and rate-based coding in terms of accuracy performance.

In the context of hardware SNN implementation, we use a high-level neuromorphic architectural exploration simulator that provides rough energy

consumption, latency and chip surface estimates for several built-in architectural configurations. This simulator, acting as a first evaluation tool, provides results helping the filtering of the large panel of architectural choices. Therefore, fewer architecture candidates appear at the output of this stage that are then described using VHDL to be tested at Register Transfer Level, thus delivering precise timings, logic resources and energy measures. Notably, we propose a novel Hybrid Architecture, which combines the advantages of both multiplexed and parallel hardware implementations.

1.5 Thesis outline

The remaining part of this thesis manuscript is composed of six chapters. First, in chapter 2 we describe the background and the context of spiking neural networks and neuromorphic computing. Then, in chapter 3, we present the design framework of the thesis and give high-level results of some architectural models for hardware SNNs. Then, we discuss neural coding and report the results of some explored coding schemes ranging from rate-based to time-based paradigms, in chapter 4. Then, in chapter 5, we describe a Register-Transfer-Level (RTL) exploration of different neuromorphic architectures for SNNs that are based on fully-connected layers. This RTL exploration comes up with a set of architectural choices that are used to design a generic hardware architecture supporting spiking CNNs. This spiking CNN architecture is described in chapter 6. Finally, we present the thesis conclusions and propose some future directions in the final chapter 7.

Chapter 2

Theoretical background

2.1 Introduction

The objective of this thesis is designing a neuromorphic architecture based on bio-inspired approaches to adapt to the embedded system constraints. Moreover, the architecture has to respect some criteria in terms of configurability, programmability, scalability and genericity.

In this chapter, we describe the theoretical concepts and methods that will serve for the selection of some technical choices for achieving our objectives. First, we introduce the neural network models found in literature and the ones we are dealing with. Second, we present some spiking neuron models that are the main computing units of the spiking neural network. Afterwards, we discuss the SNN's training and neural coding which are two important concepts in the construction of spiking neuromorphic systems. Then, we define the basic paradigms and architectural models for the implementation of neuromorphic hardware chips. Finally, we present some related works dealing with neuromorphic computing.

2.2 Neural network models

In this work, we deal with the design of embedded electronic systems capable of performing tasks commonly considered as AI applications such as image classification or object recognition. For performing such tasks, the most commonly used solution is based on artificial neural networks.

There are many types of neural network topology, such as recurrent, feed-forward or self-organizing maps. In this work, we consider only feed-forward

networks because they are the simplest for implementation in hardware and provide the state-of-the-art accuracy performance for many AI applications.

These feed-forward networks are composed of neurons that are connected between each other through synapses that conduct neural activities. Based on these neurons, networks can be formed by organizing them in layers connected to each other (Bebis and Georgiopoulos, 1994). An ANN consists of an input layer and an output layer and at least one hidden layer. The input layer consists of input neurons that only transmit the information to the rest of the network by acting as buffers. The output layer contains the neurons that represent the different classes of the data that the network is classifying (Sze et al., 2017). The hidden layers are connected to the input and output layers and may be of different types and sizes (number of neurons). In this work, we have considered the most commonly used layer types, these are: convolutional, pooling and fully-connected layers. Figure 2.1 shows an example of such a CNN composed of 6 layer hidden layers, with, two convolutional, two pooling and two fully-connected layers. This kind of ANN is called "Convolutional Neural Network" or simply "CNN" because it has convolutional layers. In the following, we give a brief description of its different layer types.

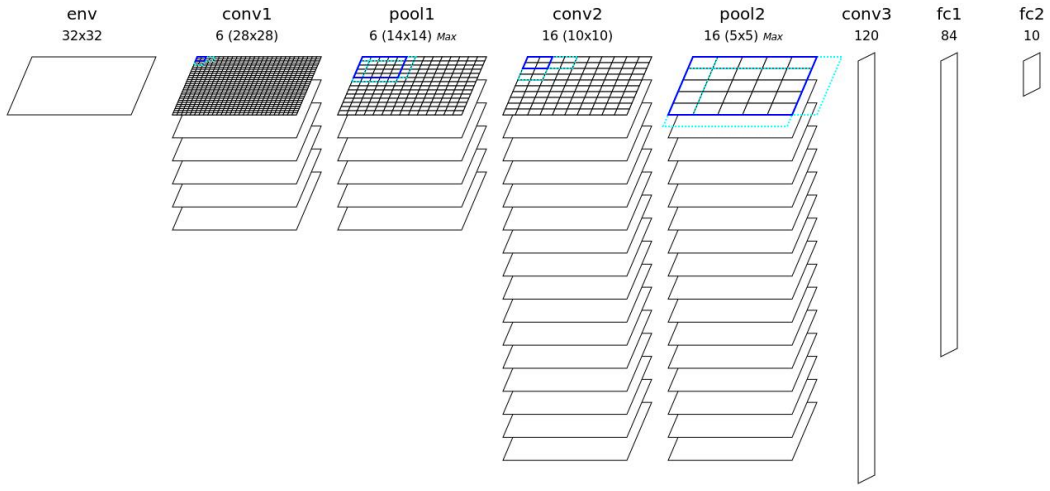


FIGURE 2.1: LeNet CNN architecture used with MNIST dataset – generated from the N2D2 framework (Bichler et al., 2017).

Convolutional layer

The convolutional layer is composed of kernels or filters that are applied to an input image or Feature Map (FM) to extract specific features. To do so, each kernel has its unique weights that are defined after the learning process

and applied at different positions of the input image or FM. The number of these positions correspond to the size of the convolution layer's output feature map that is computed using equations 2.1 and 2.2 .

$$FM_{size} = N_{out} * N_{out} \tag{2.1}$$

$$N_{out} = (N_{in} - F + 2 * P)/S + 1 \tag{2.2}$$

Where N_{out} is the width/height of the output feature map, F is the receptive field width (filter width), P is the padding, S is the stride and N_{in} is the input FM width/height. In figure 2.1 an illustration showing the network LeNet-5 when used for MNIST classification tasks. Here, there are three convolutional layers, one connected to the input and two others connected to the first and second pooling layers. The first layer is composed of 6 convolution kernels with the size of 5x5 that slide over the input image using a stride equal to 1 (the padding is equal to 0). This convolutional layer output is 6x28x28, i.e. 6 FMs of 28x28 size, this is obtained by applying the formulas in equations 2.1 and 2.2.

Pooling layer

Pooling layers, also called sub-sampling layers, are used to reduce the size of input FMs. Actually, they are composed of average or maximum pooling kernels that are applied to input FMs using the similar sliding technique used with convolutional layers. Therefore, to compute the output feature map size, we use the same formulas found in equations 2.1 and 2.2. For example, in the third layer from the left in figure 2.1, applying 6 pooling kernels (of 2x2 size and with a stride of 2) on 6x28x28 FMs results in feature maps of 6x14x14 size.

Fully-Connected layer

Fully-connected layers are generally located at the last stages of a CNN network. They come to process the output FM of the last CNN convolutional or pooling layer, they are considered as the network's classifier. These layers are composed of neurons connected to all previous layer nodes.

Neural models

Indeed, there are many models of artificial neural networks that can be classified into two families according to the type of neurons and the method of communication between them. First, there are the Analog Neural Networks (ANNs) which are widely used in the field of Machine Learning. They are called Analog because the data flowing and processed by the network is analog data (real valued data). Second, we have Spiking Neural Networks (SNNs) that rise from the field of neuroscience.

2.2.1 Analog Neural Networks (ANNs)

The analog models (ANNs) constitute the first neural family that we are dealing with in this work. As mentioned before, they are composed of analog neurons organized in layers and connected in a feed-forward manner. The data flowing in the network between its nodes represent neural activities of the different neurons. These neural activities are output of neurons that have processed input data coming from weighted input synapses. In this thesis, we have used two different ANN models: Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN).

Multi-Layer Perceptron

The best-known network model in the machine-learning field is the MLP, which is a feed-forward network that consists of only fully-connected layers. Each layer has a number of neurons called perceptrons that are totally connected to the neurons of the previous layer. The neuron j in layer l performs the computation shown in equations 2.3 and 2.4.

$$y_j^l(t) = f(s_j^l(t)) \tag{2.3}$$

$$s_j^l(t) = \sum_{i=0}^{N_{l-1}-1} w_{ij} \times y_i^{l-1}(t) \tag{2.4}$$

Where y_j^l is the output of the neuron j in the layer l, N_l is the number of neurons in the layer l, w_{ij} is the synaptic weight between neuron i in layer $l-1$ and neuron j in layer l, and $f()$ is the non-linear activation function. There are several activation functions that can be used with perceptrons such as Sigmoid (Narayan, 1997), hyperbolic tangent (TanH) (Lecun et al., 1998)

and Rectified Linear Unit (ReLU) (Hahnloser et al., 2000; Agarap, 2018). In this work, we use the ReLU function for two reasons: first, for its computing efficiency allowing the network to converge quickly and its prevention from over-fitting (Krizhevsky, Sutskever, and Hinton, 2017); second, when mapped to spiking domain, it results in a simple comparison with a threshold (Perez-Carrasco et al., 2013).

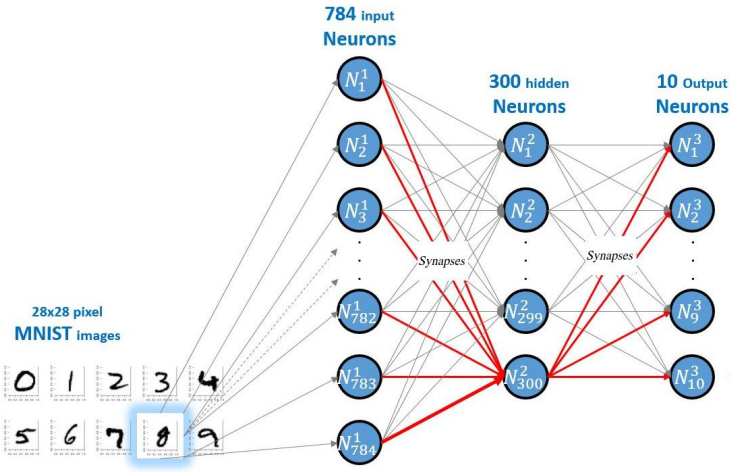


FIGURE 2.2: A one hidden layer MLP network applied to MNIST dataset.

The learning algorithm used to adapt the synaptic parameters of the MLP is back-propagation, this algorithm is described later in subsection 2.4.2.1. An example of an MLP with one hidden layer of 300 neurons applied to MNIST data-set is shown in figure 2.2.

Convolutional Neural Networks (CNNs)

CNN is the most used deep learning architecture. It is inspired from the biological visual perception mechanism of living beings. Indeed, Hubel and Wiesel have found, in 1959, that animals have cells in their visual cortex that detect the light present in their receptive field (Hubel and Wiesel, 1959). Inspired by this finding, Fukushima proposed in 1980 the neocognitron, which is considered as the CNN's predecessor (Fukushima, 1980). A decade later, LeCun et al. proposed the revolutionary framework of CNN that is presented as an ANN of several layers of different types, called "LeNet-5". This ANN is specifically used to ensure the handwritten digits classification task (LeCun et al., 1990).

More than a decade later, several models have been proposed, they came with improvements and facilitation in training CNNs. Krizhevsky, Sutskever,

and Hinton proposed one of them, AlexNet, it has similar structure to LeNet-5 but with more layers (Krizhevsky, Sutskever, and Hinton, 2017). This model was very successful, it brought out many methods coming from other works that improved classification performances. Among them, we can cite VGGNet (Simonyan and Zisserman, 2014), ZFNet (Zeiler and Fergus, 2014), GoogleNet (Szegedy et al., 2015) and ResNet (He et al., 2015). Recent advances on CNNs can be found in the survey paper proposed by Gu et al. in (Gu et al., 2015).

Meanwhile, CNNs have become more popular with applications in several domains such as image and video recognition, image classification, natural language processing, and medical image analysis. This success can be awarded to two factors. First, computing capabilities of modern CPU/GPU based computers that accelerated learning and inference stages. Second, the huge amount of available open source labeled data for training CNNs, which increased the number of applications and contributed in the improvement of CNN models.

2.2.2 Spiking Neural Networks (SNNs)

Spiking Neural Networks (SNNs) are the second family of neural models that we are studying in this thesis. In fact, SNNs are closer to the functioning of the brain than ANNs, thanks to the use of spiking neuron models. These spiking neurons mimic the behavior of biological neural cells by integrating and communicating with each other action potentials or "spikes" (Izhikevich, 2004; Pfeiffer and Pfeil, 2018; Tavanaei et al., 2019). Indeed, a spike is an electrical impulse that is traveling from one neuron to another and is considered as the elementary unit that is used to encode neural data (Gerstner and Kistler, 2002b; Brette, 2015). The neural integration consists in accumulating synaptic weights associated with pre-synaptic connections that have conducted input spikes. Various neuron models that can be used with SNNs will be presented in section 2.3.

The neuron's computation in SNNs is a spatio-temporal mechanism because: first, the neurons are spatially located in the SNN by belonging to neural regions identified generally by layers; second, the SNN processes data temporally where the neurons are activated at different dates depending on the firing of their predecessor neurons (Kasabov, 2018; Behrenbeck et al., 2018). This spatio-temporal property of SNNs fits perfectly event-based neuromorphic architectures. These neuromorphic systems have a structure adequate

to the spatial organisation of an SNN with Neural Processing Units (NPUs) representing layers (Esser et al., 2016; Schmitt et al., 2017; Rotermund and Pawelzik, 2018; Luo et al., 2018; Carbon et al., 2018). The different NPUs communicate between each other under the Address-Event-Representation (AER) protocol that fits the temporal aspects of SNNs. In the coming chapters, we will study neuromorphic architectures and their impact on the efficiency of SNNs.

2.3 Spiking neuron models

For the neuron model, among several models identified in neuroscience literature and in a machine learning context, we use spiking neurons. This spiking neuron model is used in the ANN-SNN conversion process to substitute the Perceptron of the originate ANN. In this section, we present some spiking neuron models found in literature and that can be used with neuromorphic systems.

Hodgkin-Huxley

The Hodgkin-Huxley (HH) model is the first biologically plausible mathematical neuron model that was proposed in 1952 by Hodgkin and Huxley. It is designed to describe the membrane's electrical behavior of a giant nerve fibre (Hodgkin and Huxley, 1952).

The HH model is expressed in equation 2.5, it shows how the synaptic current I flowing across the neuron's membrane is integrated on the membrane capacitance C_M.

$$C_M \frac{dV_M}{dt} = -g_L(V - E_L) - g_{Na} m^3 h (V - E_{Na}) - g_K n^4 (V - E_K) + I \quad (2.5)$$

With V_M the neuron's membrane potential, g the conductance, E the equilibrium potential and "m,h and n" gating variables for activation and inactivation. The different HH model's parameters have biophysical and biological meaning, which makes it largely used in the study of neurological diseases (Levi et al., 2018).

Izhikevich

Proposed by Izhikevich, this spiking neuron model is considered as biologically plausible because it is capable of reproducing spiking and bursting firing pattern behaviors of known types of cortical neurons (Izhikevich, 2003). It is used generally for real-time simulation of large-scale neural networks (Izhikevich, 2004). The formulations of this model are shown in equations 2.6 and 2.7.

$$\frac{dv}{dt} = 0.4v^2 + 5v + 140 - u + I \tag{2.6}$$

$$\frac{du}{dt} = a(bv - u) \tag{2.7}$$

With v the neuron's membrane potential, u the membrane's recovery variable. Activation and inactivation of K^+ and Na^+ ionic currents can be represented by the variable u which supplies v with a negative feedback. I is the synaptic delivery current and "a,b,c and d" are the model's dimensionless parameters.

$$\text{if } v \geq 30mV; \text{ then } \begin{cases} v = c \\ u = u + d \end{cases} \tag{2.8}$$

Equation 2.8 is used to update u and v when the amplitude of action potential reaches the threshold of 30 mV.

Integrate-and-Fire

The Integrate-and-Fire (IF) model is the simplest spiking neuron model that includes a hardware friendly computation with a simple addition and comparison. This IF rule is given in equation 2.9.

$$s_j^l(t) = p_j^l(t-1) + \sum_{i=0}^{N_{l-1}-1} (w_{ij} \times \gamma_i^{l-1}(t)) \tag{2.9}$$

With $p_j^l(t)$, the membrane potential of the j^{th} neuron of layer l, and $\gamma_j^l(t)$, the binary output of j^{th} neuron of layer l, being defined in equations 2.10 and 2.11. Note that θ is the activation threshold of the j^{th} neuron of layer l.

$$p_j^l(t) = \begin{cases} s_j^l(t) - \theta \text{ if } s_j^l(t) \geq \theta \\ s_j^l(t) \text{ otherwise} \end{cases} \tag{2.10}$$

$$\gamma_j^l(t) = \begin{cases} 1 \text{ if } s_j^l(t) \geq \theta \\ 0 \text{ otherwise} \end{cases} \tag{2.11}$$

As mentioned in equation 2.11, $\gamma_i^l(t)$ is either "1" or "0". Therefore, in a context of hardware implementation, the "$\times\gamma_i^{l-1}(t)$" expression in equation 2.9 can be represented as a simple accumulation operation performed within the hardware neuron. In case the neuron is enabled by a predecessor neuron of layer $l-1$, it does its computations. It accumulates the synaptic weight $(w_{ij} + p_j^l(t-1))$ and compares this amount to the threshold to update its internal potential and eventually fire an output spike. Note that, the Leaky Integrate-and-Fire model is slightly more complex (Liu and Wang, 2004) and implies a continuously decreasing membrane potential.

Spike Response Model

The Spike Response Model (SRM) is a bio-inspired spiking neuron that is describing more precisely the effect of input spikes on the membrane potential. The SRM model is slightly more complex than the LIF (Gerstner and Kistler, 2002b). Similar to the LIF model, SRM neuron generates spikes whenever its internal membrane potential reaches the threshold. However, in contrast to LIF, it includes a function dependent to reset and refractory periods. Moreover, unlike the LIF model that is modeled using differential equations for the voltage potential, the SRM is formulated using response kernels (filters). The SRM model mathematical formulation is expressed in equation 2.12.

$$v(t) = \eta(t - \hat{t}) + \int_{-\infty}^{+\infty} \kappa(t - \hat{t}, s) I(t - s) ds \tag{2.12}$$

With $v(t)$ being the neuron's internal potential that is changing over time t, $\hat{t}$ the emission time of the last neuron output spike, $\eta()$ describes the state of the action potential, $\kappa()$ is a linear response to an input spike. Both $\eta()$ and $\kappa()$ are kernel functions. $I(t)$ represents the stimulating or external current. The different functions of the formula are detailed in (Gerstner and Kistler, 2002a),

Equation 2.13 defines the spike emission condition where the internal potential $v(t)$ is compared to the threshold θ. In case $v(t)$ is equal or greater to the threshold an output spike is generated at $\hat{t} = t$.

$$\text{if } v(t) \geq \theta \text{ and } v'(t) > 0, \text{ then } \hat{t} = t \tag{2.13}$$

A recent work (Zhang et al., 2020) proposed a neuron model derived from the SRM model, called Exponential-and-Fire neuron. Like the SRM model, the EF model also uses kernel functions to process spiking data. These are two exponentially growing functions, the first is a PostSynaptic Potential (PSP) kernel that is used to integrate input spikes and the second one, an After Hyperpolarizing Potential (AHP) kernel, is used to generate output spikes.

Spiking neuron for embedded AI

Among the spiking neuron models described so far, we notice that the computationally simplest model is the Integrate-and-Fire model. In a machine learning context, spiking neurons are most often based on this simple IF model (Abbott, 1999). Moreover, the IF model is already known to be sufficient for spike-based classification applications (Cao, Chen, and Khosla, 2015; Cassidy et al., 2013; Cruz-Albrecht, Yung, and Srinivasa, 2012; Merolla et al., 2011), where SNNs that are based on the IF neuron obtain higher results than SNNs are based on the other spiking neuron models. Therefore, the only model that results in equivalent accuracy is the ANN's perceptron neuron which is a non-spiking model. In this context, the IF model's computations when compared to the perceptron, show that this model is much more competitive where multiplicative operations and a nonlinear function are found in perceptrons which are much more resource-intensive than the addition and comparison found in IF model. In this thesis, we use the IF model due to its computation efficiency and accuracy performance.

2.4 SNN's training

2.4.1 Unsupervised learning with STDP

Unsupervised learning of SNNs is based on the Hebbian rule that consists in adapting the network's synaptic connections to the data received by the

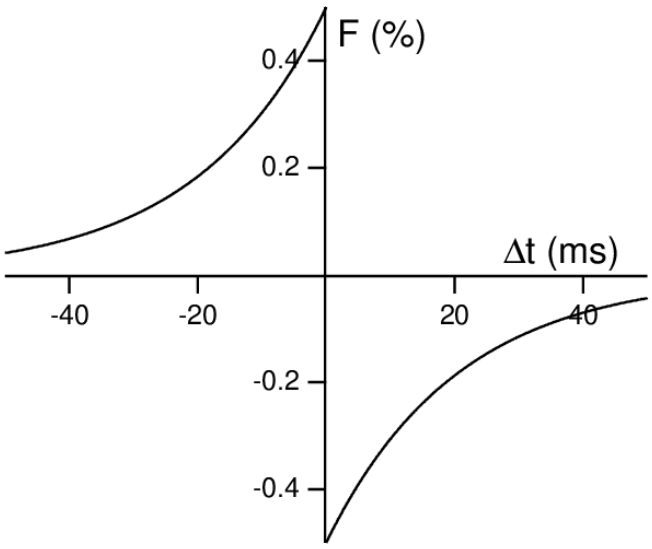


FIGURE 2.3: Spike-time-dependent synaptic modification rule presented in (Song, Miller, and Abbott, 2000).

neurons. Here, we present the Spike Timing Dependent Plasticity (STDP) algorithm which is an implementation of Hebb's rule.

STDP is an unsupervised brain-inspired learning algorithm that is used to train SNNs on unlabeled data. The STDP rule is based on the Hebbian synaptic plasticity rule that is applied to SNN's synapses locally. This is done by considering only the timings of spikes released by the pre- and postsynaptic neurons of a synaptic connection to either reinforce or depress it's weight.

The concept is to detect the causality between the neurons for each input. First of all, an expiry time Δt is defined to strengthen or weaken a synaptic connection of a neuron. Second, the synapses are classified into two categories of synapses depending on their arriving time and contribution to the emission of output spikes. Long-Term Potentiation (LTP), reinforcement of synaptic weights, is applied to the first category of synapses which are the ones that have conducted spikes before the neuron fires ($\Delta t > 0$). The second category of synapses are the ones that have conducted spikes after the neuron's firing. Long-Time Depression (LTD) is applied to these synapses, which is the decrease of their synaptic weights ($\Delta t < 0$).

Several variants of STDP rule with different biological plausibility and computational complexity are found in literature (Song, Miller, and Abbott, 2000; Cruz-Albrecht, Yung, and Srinivasa, 2012; Diehl and Cook, 2015; Mozafari et al., 2018; Kheradpisheh et al., 2018; Thiele, Bichler, and Dupret, 2018). They have been discussed in the recent work (Vigneron and Martinet, 2020). The formulation of the original STDP rule proposed in (Song, Miller, and Abbott,

2000) is shown in equation 2.14. Figure 2.3, from (Song, Miller, and Abbott, 2000), shows a curve indicating the amount of synaptic modification originating from a single pre- and post-synaptic pair of spikes with a separation of Δt time.

$$F(\Delta t) = \begin{cases} A^+ e^{(\Delta t/\tau^+)} & if \Delta t < 0 \\ -A^- e^{(-\Delta t/\tau^-)} & if \Delta t > 0 \end{cases} \tag{2.14}$$

With τ^+ and τ^- being the ranges of presynaptic to postsynaptic interspike intervals over which the synaptic connections are modified. When Δt is close to zero, the maximum amounts of synaptic modifications are determined by A^+ and A^-.

2.4.2 Supervised learning

The second approach of training SNNs is using supervised learning methodologies. The supervised methods rely on labeled data-sets to learn ANNs, where a huge amount of labeled data examples are presented to the network to incrementally adapt its synaptic parameters. Despite the fact that this approach is less biologically plausible, it achieves much higher performance and remains more deployed in embedded AI than unsupervised learning. Since the objective of the thesis consists in implementing SNNs for inference in hardware, we have adopted this learning approach to get the ANN's state-of-the-art accuracy performance equivalent.

There are two possible ways to train SNNs in a supervised manner: mapping ANNs to SNNs and learning the SNNs in spiking domain. Let's briefly describe both the approaches.

2.4.2.1 ANN-SNN conversion

The ANN-SNN conversion approach has been already studied in several works (Cao and Grossberg, 2012; Perez-Carrasco et al., 2013; Rueckauer et al., 2016; Cao, Chen, and Khosla, 2015; Diehl et al., 2015; Rueckauer et al., 2017; Zhang et al., 2020). This approach is based on mapping trained ANNs using a supervised (back-propagation) learning algorithm to SNNs. Indeed, the ANN is trained using a mature learning approach that is largely deployed in machine learning and deep learning fields (Sze et al., 2017). Before describing

the mapping approach, let's give a brief description of the back-propagation algorithm.

Backpropagation

The most common learning algorithm used with ANNs is the backpropagation (Schuman et al., 2017; Sze et al., 2017), a supervised off-line learning algorithm (LeCun et al., 1990). As shown in figure 2.4, the principle is to calculate the gradient of the error between the desired output and actual output and to back-propagate it to the neurons of the previous layers in order to adjust their synaptic weights. The process is repeated through all the learning data-base until either the number of learning iterations or the validation rate is reached. The synaptic weights are updated as shown in equation 2.15.

$$w_{ji}^{l}(t+1) = w_{ji}^{l}(t) + \eta \times \delta_{j}^{l}(t) \times y_{i}^{l-1}(t) \tag{2.15}$$

Where t is the learning iteration, η is the learning rate and δ_j^l is the error gradient of the neuron j in the layer l, such that:

— At the output layer, $\delta_j^l(t) = f^{'}(s_j^l(t)) \times e_j^l(t)$ where e_j^l is the error, i.e.: the difference between the correct and the actual network output of the neuron j in the layer l.

— At the hidden layer, $\delta_j^l(t) = f^{'}(s_j^l(t)) \times \sum_{k=0}^{N_{l+1}} \delta_k^{l+1}(t) \times w_{kj}^l(t)$ where $f^{'}$ is the derivative of f.

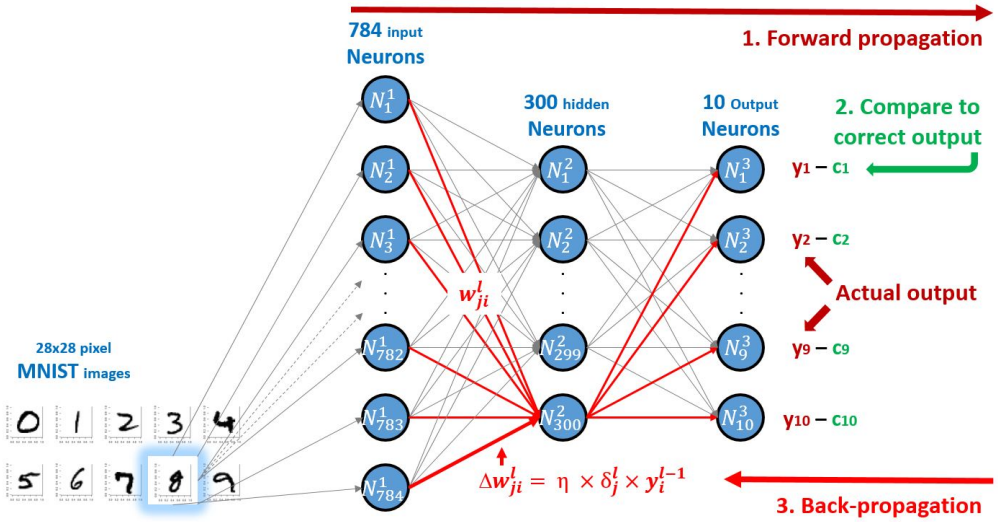


FIGURE 2.4: Back-propagation algorithm applied to a one-hidden layer neural network.

Conversion

Figure 2.5 illustrates a schematic diagram showing the different steps of the ANN-SNN conversion. First, we train the neural model in the analog domain using a backpropagation learning algorithm. Then, we export the resulting synaptic scalar weights to use them with the SNN, refer to figure 2.5. The SNN is composed of IF neurons organized in layers that follow the same organization as the original ANN's topology. The input data are converted in the spiking domain using a neural coding methodology, refer to 2 for more details about spike generation. Second, we test both analog and spiking ANNs on the patterns of the testing data-set. Finally, if the accuracy results of both the SNN and ANN are satisfying, then these networks are ready for inference. Note that, building SNNs in the N2D2 framework follows the same approach, refer to section 3.3.1 for more details.

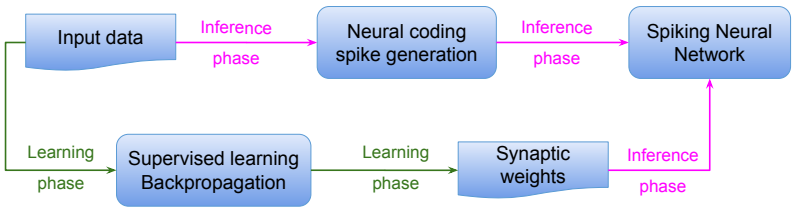


FIGURE 2.5: ANN-SNN conversion mechanism.

2.4.2.2 Learning in spiking domain

Spiking neurons, like IF and LIF models, are of a non-differentiable and discontinuous nature, which renders the application of back-propagation based on gradient descent on SNNs difficult. In practice, SNNs perform less well than ANNs in terms of accuracy on traditional learning tasks. This fact has motivated several works in recent years to propose algorithms and learning rules to implement spiking CNNs as efficiently as traditional CNNs on complex visual recognition tasks (Wu et al., 2019; Cao, Chen, and Khosla, 2015). In addition to ANN-SNN conversion methods, several works investigate direct spike-based supervised learning algorithms. In this context, spike-based back-propagation rules have been proposed to perform gradient-based training over a complete SNN on spiking data. In spike-based back-propagation approaches, two solutions are possible to overcome the non-differentiable aspect of spiking neurons: 1- by using a continuous approximation of the real gradient called a surrogate gradient (Wu et al., 2018; Bellec et al., 2018;

Neftci, Mostafa, and Zenke, 2019); 2- by using an approximation of the neuron model making it continuous and thus differentiable (Huh and Sejnowski, 2018). Among these works, we can cite Spike-Prop (Bohté, Kok, and Poutré, 2000), temporal-based learning (Mostafa, 2018), tempotron (Gütig and Sompolinsky, 2006), Remote Supervised Method (ReSuMe) (Ponulak and Kasiński, 2009), SpikeGrad (C., Bichler, and Dupret, 2019), spiking back-propagation using rank-order coding (Kheradpisheh and Masquelier, 2020), etc.

The adopted learning in this thesis

A recent work (Lee et al., 2020) has shown that SNNs resulting from spike-based learning methods have an overall inference latency 10 times lower than those stemming from conversion approaches. However, in terms of learning time (development process), spike-based learning requires more total iterations for learning compared to conversion approaches. Moreover, ANN-SNN conversion technique takes advantage of the advance of research in the field of deep-learning which has optimized and improved learning methods.

2.5 Neural coding

In this section, we present different neural coding paradigms for representing neural data in SNNs. In this part, we explore mainly the spike generation techniques that can be used to convert analog data to spike-based stimulus. Therefore, when we refer to the notion of "neural coding", we mainly insinuate the generation of spikes. Indeed, the first step of SNN's implementation is selecting a spikes generation scheme for converting static input data to spike-based stimuli that the SNN will process. Note that this spike generation is limited to only static input data. Actually, much research has been conducted to unravel neural coding in biological neurons, but so far this "problem of neuronal coding" (Gerstner and Kistler, 2002b) remains unsolved (Brette, 2015; Tavanaei et al., 2019). However, two paradigms stand out, one leading to temporal coding (Thorpe, Delorme, and Rullen, 2001) and the other to rate coding (Gerstner and Kistler, 2002b). Therefore, to perform the spikes generation task with SNNs we have to select a neural coding technique rising either from one of these coding paradigms.

2.5.1 Rate-based coding

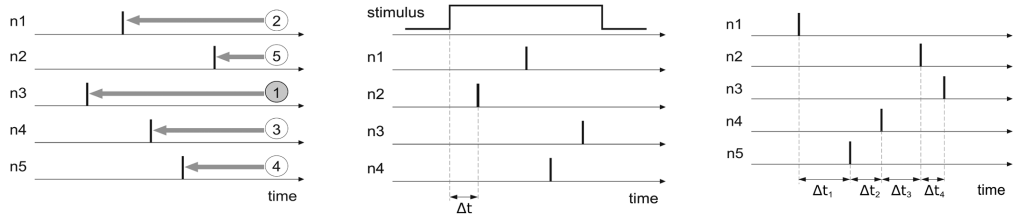
With rate-based coding the neural data is represented as trains of spikes with different firing rates. In this coding paradigm, an input neuron can be seen as an analog to frequency converter: the neuron responds to input data with output spikes represented as a train of spikes with a firing rate proportional to the intensity of the input stimulus. This firing rate defines the number of output spikes and their firing timings. Therefore, the information flowing in the SNN is encoded using rates dictating the amount. Considering the spike generation at the SNN's input, the data is somehow converted to a firing rate (or frequency) that is used to generate spike trains. Doing so, the input stimulus is represented by a number of spike trains with different firing, this number is equal to the input size. The most common technique that is used to compute the firing rates of analog data to generate spike trains is the Poisson distribution (Cao, Chen, and Khosla, 2015; Diehl et al., 2015).

2.5.2 Time-based coding

With Time-based coding, neural data flowing in the SNN are represented in a code scheme based on the precise timing of spikes. In this coding paradigm, neurons are considered as analog-to-delay converters rather than analog-to-frequency converters, as in rate-coded networks (Thorpe and Gautrais, 1998). Thus, when a neuron receives strong input data, its reaction is faster by triggering a spike within a small delay. In case of weak data, the firing delay is longer and the spike is triggered later. The generation of spikes using this coding technique from input stimulus is done by converting each analog input value into precise firing timing. Indeed, these precise times are inversely proportional to the intensity of the input data. Thus, the spikes are generated earlier for high intense values and later for less intense ones. Several techniques exist to determine these precise delays. Some of them are shown in figure 2.6 and described below: rank order coding, time-to-first-spike and relative latency coding.

Rank order coding

Rank order coding is a temporal coding scheme proposed by Thorpe and Gautrais in 1998 (Thorpe and Gautrais, 1998). This coding technique is a simpler version of temporal coding, where the arriving times are only used to order the spikes. To generate spikes from an analog input data, the data



(A) (B) (C)

FIGURE 2.6: Temporal coding: (A) rank order coding; (B) time-to-first-spike coding and (C) relative latency coding. Legend: $n1$ – $n5$ represent the neurons labels; the vertical bars represent the neural firing times; the circled numbers indicate the arriving order of the spikes; Δt is the latency between the stimulus onset and the first spike; $\Delta t1$ – $\Delta t4$ are the inter-spike latencies (from Ponulak and Kasinski, 2011).

would be first transformed to time amounts (delays) and then, based on these timings the input stimulus spikes are ordered. In a context of hardware implementation, rank order coding would be an effective solution for event-based architectures where the spikes are ordered naturally. Moreover, the spiking events will contain only the address of the emitting neuron because there is no need to register an associated timestamp. Figure 2.6a shows a group of output spikes coded using rank order coding, the spike arriving from neuron 3 has the highest order compared to others because it is the first to arrive.

Time-To-First-Spike

The Time-To-First-Spike (TTFS) coding scheme is another temporal coding technique. To encode input stimulus in the spiking domain using TTFS scheme, each input data is transformed to a spike with an emission timestamp equal to the latency between the start of the stimulus and the first spike of the neural response. Figure 2.6b shows how output spikes of four neurons are encoded using the TTFS scheme. Each spike of the neurons is emitted at different timestamps Δt which are the latencies defining time to the beginning of the stimulus. This method, representing neural data by considering only the time to first spike, has been shown to carry enough information to encode, in the tactile system, touch signals at the finger tips (Johansson and Birznieks, 2004). This coding scheme reduces drastically the neural data flowing in the SNNs and thus enables accelerating processing (Ponulak and Kasinski, 2011).

Latency coding

Latency coding can be considered as an extension of TTFS coding where the time difference is relative, in addition to the first spike, between all consecutive spikes. These relative time differences, especially between presynaptic and postsynaptic spikes, have been shown to play an important role in synaptic learning processes (Markram et al., 1997). Latency code is also very efficient In terms of information capacity, it is efficient where a few spike timings are sufficient to carry a substantial information amount (Borst and Theunissen, 1999; Ponulak and Kasinski, 2011). Figure 2.6c shows a group of output spikes coded using latency coding, the spike arriving from neuron 1 has the lowest latency compared to others and thus arrives before all the rest.

2.5.3 Contributions

At this stage of the thesis, we have discussed the theoretical background of spiking neural models, therefore, let's select the neuron model, the synaptic connection model, the neuronal coding and the learning method to be used in the remaining parts of this thesis. First, for the neuron model, we have chosen the IF neuron because it performs state-of-the-art recognition rates in classification tasks while at the same time being the simplest computational spiking model.

Second, a large number of "synapse models" found in the literature exhibit a variable level of biological plausibility by reproducing different numbers of neuronal behaviors and properties such as: synaptic delays, size, shape and strength of synapses (Brette et al., 2007). For the purpose of reducing SNNs complexity, we have selected the simplest model that reproduces only the strength of the synaptic connections, which are given by their synaptic weights.

Next, we adopt the ANN-SNN conversion approach to train the SNNs because, in addition to its maturity, it is characterized by a faster design time. It should be noted that the training algorithm does not affect the final hardware architecture and that both the conversion and the spiking learning approaches are possible. The architectures presented in chapters 5 and 6 are generic and programmable by first configuring the SNNs topology and second, uploading the synaptic weights issued from one of these training algorithms.

Finally, in designing the SNNs, the choice of the neural coding scheme used to generate spikes is crucial. This coding scheme must perform the state-of-the-art accuracy and generate as less spiking activity as possible in the SNN. In this work, we study and explore techniques ranging from rate-based to temporal-based coding paradigms.

2.6 Neuromorphic hardware

2.6.1 Hardware targets

Many application domains which are subject to the design constraints of embedded systems, like sensor networks, IoT, smart devices, aeronautics or autonomous vehicles require dedicated electronic systems for meeting the requirements of area occupation, latency, heat dissipation and energy consumption.

Under these conditions, general purpose programmable solutions on CPU (Central Processing Unit) or GPU (Graphical Processing Unit) are no longer suitable, and it becomes essential to implement dedicated hardware solutions for ANNs instead of the software ones. These solutions are hardware neuromorphic architectures that take advantage of the fundamentally parallel and distributed nature of these ANN algorithms.

Several computing electronic substrates are available to deploy ANN models and can be classified into two categories: analog and digital substrates. In this work, we use digital substrates to implement the neuromorphic architectures because they benefit from the maturity of the associated manufacturing technologies and their reprogramming / configuration facilities (Philippe et al., 2015). Moreover, it is shown in (Joubert et al., 2012) that below 22nm, digital implementation becomes more attractive in terms of area and scalability for LIF-based SNNs.

Indeed, the digital solutions could be implemented either on Field Programmable Gate Array (FPGA) or on Application Specific Integrated Circuit (ASIC). These hardware targets have been studied for several decades but no complete solution has dealt with convolutional SNN.

FPGA

In previous studies, FPGAs have been frequently employed for the design of neuromorphic computing circuits (Pani et al., 2017) (Rotermund and Pawelzik, 2018). This technology can be used either for prototyping and delivering a sub-part of a greater system, or directly as the final chip design implementation. The main advantages of this technology are its high programming / reconfiguration, and moderate cost. In the context of this thesis, which mainly concerns an architectural exploration of SNN's hardware design, we are interested in a re-configurable platform : indeed, the chip must be reconfigured for each architecture. Thus, an FPGA device is an adequate technology for our purpose.

Some devices, namely SOCs (System On Chips), include one or several CPUs alongside the programmable logic array, which offers possibilities for both software and hardware programming. As we aim to develop a generic neuromorphic Intellectual Property (IP) capable of executing any feed-forward SNN configuration, this type of device would suit the programmability requirement.

ASIC

ASIC chips have also been widely employed for neuromorphic digital hardware implementations (Painkras et al., 2013; Benjamin et al., 2014; Akopyan et al., 2015).

In contrast with conventional processor architectures, which are designed to handle a wide variety of tasks, ASICs are fully customized to run a particular type of application. Some of those chips, such as TrueNorth (Merolla et al., 2014) (Akopyan et al., 2015), are very highly specific: they are designed for one particular neuron model with very low reprogramming abilities, whereas others such as SpiNNaker (Furber et al., 2014) (Painkras et al., 2013), are designed with a much higher capacity for flexibility. Usually, these chip architectures are designed to support the high level of parallelism and distribution found in neural algorithms. Thus, most of the time they are based on a massively parallel computation paradigm, with great care given to the communication between computing units. However, these ASICs focus not only on pure computation acceleration, but also on the constraints of their application domain.

For integration in embedded systems for example, particular attention has to be paid to the chip surface and energy consumption limitations. These application-related constraints are also of major concern for ASIC design, and can be found in the differences between TrueNorth and SpiNNaker: the first is focused on energy savings, whereas the second is focused on flexibility. Even if the task is very similar, the implementation design is dramatically different, and so are performances: TrueNorth (Merolla et al., 2014) (Akopyan et al., 2015) shows an energy consumption of 12pJ per synaptic connection, in contrast to 20nJ for SpiNNaker (Furber et al., 2014) (Painkras et al., 2013). In this thesis, design space exploration requires a high reprogramming / reconfiguration abilities, and we have thus targeted FPGA design instead of ASIC. By this way, we favour the automatic generation of architectures on a re-configurable substrate rather than the definition of a programmable architecture on a fixed one.

2.6.2 Event-based computing

Neuromorphic Computing systems are said to be event-based or event-driven if their states are updated asynchronously by incoming events. A spiking neural network is by nature an event-based system because the spiking neurons are triggered by input spikes or spiking events. When the neuron is not stimulated, i.e. no input spikes, it's internal state (membrane potential) is not changed and thus it may stay in an idle state. Therefore, SNNs are completely event-based systems that can be efficiently implemented on event-based computing architectures. Moreover, the spiking data flowing in SNNs are characterized by a high degree of temporal and spatial sparsity. Unlike the analog data, spikes are temporally sparse where, for example, neurons belonging to the same layer will not fire at the same computing moment. The spatial sparsity aspects concern the amount and location of spiking data. For example with SNNs, the number of spikes is different to the number of neurons and due to the filtering characteristic of the neurons when going deeper in the SNNs this number of spiking events decreases incrementally.

To sum up, SNNs are completely event-driven algorithms that are favorable to event-driven computation. In addition to this event-driven aspect, the spiking data are spatio-temporally sparse, which can influence the parallelism of their computation which will be discussed in the following.

2.6.3 Time-multiplexed and parallel computing

Since SNNs are intrinsically parallel algorithms, computation parallelization should result in great acceleration of processing. Nevertheless, a high level of parallelization requires a large number of NPUs (ideally, one per logical neuron), resulting in the drastic increase of chip surface and makes it a non-scalable solution. As mentioned before, the spiking data is known to be spatio-temporally sparse. First, the temporal aspect is due to the asynchronous arriving of the spikes. Second, the spikes are spatially distributed over the SNN's layers and their number is going-down while going deeper in the network. Therefore, SNNs can be implemented in a time-multiplexed fashion benefiting from this spatio-temporal sparsity characteristic. Time-multiplexing computing is a temporal serialization of parallel computing, where the parallel tasks will be processed one after the other. Doing so, the computing chip surface is reduced and becomes scalable. However, the latency of the SNN will be increased. So, we need to find a good latency-surface trade-off to efficiently implement SNNs for embedded AI applications. Therefore, in this thesis we explore different levels of architectural parallelism and multiplexing in order to better take advantage of spiking data sparsity. In chapter 4, we compare three hardware architectures with different degrees of parallelism.

2.7 Related works : neuromorphic hardware chips

In this section, we introduce some of the most recent neuromorphic hardware architectures found in the literature. Those systems consist of ASIC or FPGA chips, designed to simulate large numbers of spiking neurons. We give a brief description of their features, alongside energy consumption information. Those information are summed up in table 2.1.

SpiNNaker

SpiNNaker (Furber et al., 2014) is a fully digital system aiming to simulate very large spiking networks in real-time, and in an event-driven processing fashion.

A SpiNNaker board is composed of 864 ARM9 cores, divided into 48 chips containing 18 cores each. The memory is highly distributed, as there is no global memory unit, but one small local memory unit for each core and a

TABLE 2.1: Neuromorphic hardware architectures. Legend: "D" for digital, "A" for Analog, "On" for online learning and "off" for offline learning.

Chip - year	Year	Electronics	Learning	Programming	Neuron
SpiNNaker	2010	D	On & Off	PyNN	LIF, IZH, HH
NeuroGrid	2014	A & D	On & Off	NGPython	Dimensionless
TrueNorth	2014	D	Offline	Corelets	LIF
Loihi	2018	D	On & Off	Loihi API	CUBA LIF
Minitaur	2014	D	Offline	RTL	LIF
Fast pipeline	2015	D	Offline	RTL	LIF
HFirst	2015	D	Offline	RTL	Complex-IF
BrainScales	2017	A & D	On & Off	PyNN	Exp-IF
DYNAPS	2017	A & D	Offline	CHP	Exp-IF
ConvNode	2018	D	Offline	RTL	LIF
This work	2020	D	Offline	N2D2/TF	IF

shared memory for each chip. The main feature of SpiNNaker is its efficient communication system: all the nodes are interconnected through high-throughput connections designed for small packet routing, which contain Address Event Representation (AER) spikes, i.e., the address of the transmitter neuron, the date of the emission, and the destination neuron. This communication scheme has been conceived to tolerate the intrinsic massive parallelism of the ANNs. The SpiNNaker board is programmable thanks to the PyNN interface, PyNN being a Python library for SNN simulation (Davison et al., 2007; Davison et al., 2009), which provides various neuron models (LIF, Izhikevich, etc.) and synaptic plasticity rules such as STDP (Spike-Time-Dependent Plasticity)(Thiele, Bichler, and Dupret, 2018; Kheradpisheh et al., 2018). In terms of energy usage, a SpiNNaker board has a peak power consumption of 1W.

SpiNNaker is used to implement massively parallel hardware SNNs in the litterature, such as NeuCube in (Behrenbeck et al., 2018), where a SNN is implemented on SpiNNaker to capture and classify spatio-temporal information from EEG (Electro-EncephaloGram). Notably, this architecture offers the possibility to pause classification process to learn new samples or classes, in an Incremental Learning (Carpenter et al., 1992) (Polikar et al., 2001) fashion, which is an interesting property.

Configurable event-driven convolutional node

The authors in (Camunas-Mesa et al., 2018) proposed a configurable event-driven convolutional node with rate saturation mechanism in order to implement arbitrary CNNs on FPGAs. The designed node consists of a convolutional processing unit formed by a bi-dimensional array of IF neurons and a router allowing to build large 2D arrays dedicated for ConvNets inference. In this structure, each node is directly connected to four other neighboring nodes through ports that carry bidirectional flow of events. Internally, all input and output ports are connected to a router, which dispatches events to its local processing unit or to the appropriate output port. The network described in (Perez-Carrasco et al., 2013) for high-speed poker symbol recognition was implemented on ®Xilinx ®Spartan 6 FPGA. With more than 5 K neurons and 500 K synapses, the generated circuit occupied 21,465 slices, 38,451 registers and 202 of block RAMs. The slower versions of the architecture showed recognition rates around 96% when all the input events were processed by the network, while less than 20% of the events were processed at real time, obtaining a recognition rate higher than 63% with a power consumption of 7.7 mW when the stimulus was being processed at real time, and even lower consumptions for slower processing: 5.25 mW when it was 10 times slower, and 0.85 mW for a slow-down factor of 100.

Conv core

The work in (Yousefzadeh, Serrano-Gotarredona, and Linares-Barranco, 2015) proposed a pipe-lined architecture for processing spiking 2D convolutional layers in a fully event-driven system. Indeed, this system takes asynchronous input data from a Dynamic Vision Sensor (DVS) (Lichtsteiner, Posch, and Delbruck, 2008) (Delbrück et al., 2010), a bio-inspired vision sensor which outputs a continuous flow of spikes corresponding to brightness gradient variations in a dynamic image. This architecture benefits from the parallelism offered by FPGAs by implementing a 3-stages-pipeline, thus reaching the great performance of updating 128 pixels of the layer in 12ns; while running on ®Xilinx ®Spartan 6 FPGA. On the same board, the implementation of a spiking convolutional layer with a 128x128 pixel input and a 23x23 convolution kernel occupies 48% of logic resources and 68% of block RAM. This architecture uses the LIF neuron model, a bit more complex than our simple IF neuron. This system is adapted to asynchronous spiking input, whereas our system is adapted to conventional CCD (Charge Coupled Device) vision

sensors, however, by the use of a spike-based learning algorithm, we could adapt our architecture to DVS to benefit from the asynchronous input in parallel implementations.

HFirst

HFirst (Orchard et al., 2015) is a Spiking CNN architecture. It is based on a frame-free paradigm, as it takes inputs from a DVS. HFirst's particularity is to focus on relative timing of spikes across neurons, benefiting from the continuous flow of input data. Hence, HFirst is dedicated to temporal pattern recognition, whereas our architecture is dedicated to static image recognition, and uses the accessible CCD sensor. Moreover, HFirst uses another IF neuron version which is more complex than ours, emulating the physical behavior of an IF Neuron. This model uses several multiplications, which results in a more resource intensive implementation (17 DSP in HFirst versus 0 for ours). HFirst runs on Xilinx®'s Spartan® 6 FPGA, with a 100MHz clock, and consumes between 150mW and 200mW. It performs 97.5% accuracy on HFirst Cards data-set (4 classes), and 84.9% on HFirst Characters data-set (36 classes).

Minitaur

Minitaur (Neil and Liu, 2014) is an event-driven neural network accelerator dedicated to high performance and low power consumption. This is an SNN accelerator on ®Xilinx ®Spartan 6 FPGA board. The example LIF-based network implemented on the board performs 92% accuracy on MNIST dataset and 71% on 20 newsgroups dataset. The Minitaur architecture consists of 32 LIF-based cores dedicated to parallel processing of spikes. The input spikes arrive from a queue where they are stored as packets through the USB interface. Those packets are encoded on 6 Bytes : 4 Bytes for timestamp, 1 Byte for layer index and 2 Bytes for the neuron address (Address-Event-Representation). This is a semi-parallel architecture, where some layers are processed in parallel, and others are processed sequentially. Minitaur achieves 19 million neuron update per second on 1.5 W of power and it supports up to 65K neurons per board within fully-connected layers based SNN.

Loihi

Loihi (Davies et al., 2018) is again a fully-digital chip containing 128 cores, each of which are able to simulate up to 1024 different neurons. The memory is also largely distributed, with each core having a local 2MB SRAM memory unit. The chip also contains 2x86 cores and 16MB of SRAM synaptic memory. Accordingly, it is able to support up to 130 000 neurons and 130 million synapses. In contrast with previous systems, the Loihi board is able to perform learning. The chip can be programmed to implement various learning rules, notably STDP. The chip is able to simulate up to 30 billion SOPS, with an average of 10pJ per spike.

TrueNorth

TrueNorth (Akopyan et al., 2015) is another fully-digital system, capable of simulating up to 1 million spiking neurons. A TrueNorth board is composed of 4096 Neurosynaptic cores dedicated to LIF neuron emulation. Each core contains a 12.75 KB of local SRAM memory, and is time-multiplexed up to 256 times so that one core can simulate 256 different neurons. Similarly to SpiNNaker, the communication scheme is asynchronous, event-based and able to tolerate a very high level of parallelism. TrueNorth can perform 46 billion synaptic operations per second (SOPS) per Watt, with a power consumption of 100mW when running a 1 million neurons network. The system is programmable thanks to the Corelet programming language (Amir et al., 2013), allowing to tune neuron parameters, synapse connectivity and inter-core connectivity.

DYNAPs

DYNAPs (Moradi et al., 2017) is a reconfigurable hybrid analog/digital architecture. Its hierarchical routing network allows the configuration of different neural network topologies. This interesting method also tries to solve the compromise between point-to-point communications and request broadcasting in large neural topologies.

The use of mixed-mode analog/digital circuits allowed to distribute the memory elements across and within the computing modules. As a counterpart, this requires the addition of conversion circuits. The analog parts are operated in the subthreshold domain to minimize dynamic power consumption and to implement biophysically realistic behaviors.

The approach is validated by a VLSI design implementing a three-layer CNN network. If the circuit consumption is low (about ten pJ per data movement in the network), the implementation of the 2560 neurons of the targeted spiking CNN requires the use of a PCB composed of 9 circuits. The overall consumption and scalability of the approach therefore remains to be confirmed.

BrainScaleS

BrainScaleS (Schemmel et al., 2010) is a mixed digital-analog system. The processing units (neuron cores) are analog circuits, whereas the communication units are digital. BrainScaleS implements the adaptive exponential IF neuron model, which can be configured to reproduce many biological firing patterns. BrainScaleS is composed of HiCANN (High-Input Count Analog Neural Network) chips, which are able to simulate 224 spiking neurons and 15 000 synapses. Several HiCANN units can be placed on a wafer, so that a single wafer can simulate up to 180 000 neurons and 40 million synapses. The system also integrates general purpose embedded processors, which are able to measure relative spike timings, thus plasticity rules such as STDP can be implemented. Other plasticity rules can also be programmed, and a PyNN interface allows users to program the network in a similar fashion to SpiNNaker. The BrainScaleS platform consumes between 0.1nJ and 10nJ per spike depending on the simulated network model, and reaches a maximum of 2kW of peak power consumption per module.

NeuroGrid

NeuroGrid (Benjamin et al., 2014) is also a mixed digital-analog system, which targets real-time simulation of large SNNs. It employs subthreshold circuits, to model neural elements. The synaptic functions are directly emulated thanks to the physics of the transistors operating in the subthreshold regime. The board is composed of 16 NeuroCore chips, interconnected by an asynchronous multicast tree routing digital communication system. Each core is composed of 256*256 analog neurons, so that NeuroGrid is able to simulate up to 1 million neurons and billions of synaptic connections. Concerning energy, NeuroGrid consumes an average of 941pJ per spike and has a peak power consumption of 3.1W.

2.8 Conclusion

In this chapter, we have described the basic theoretical notions that are necessary for selecting the neural and architectural models to use for achieving the objectives of the thesis. Therefore, we have organized the chapter in three parts. First, we presented the different neural network models and the corresponding neuron and training models. Afterwards, we discussed neural coding by focusing on the spike generation and its impact on SNNs performance. Finally, we have presented some architectural models that can be used with neuromorphic systems and discussed their influence on the performance of a hardware implementation of SNNs. In the context of low-power embedded AI, we also discussed how to adapt the computing parallelism in a neuromorphic architecture to get advantage of sparsity of spiking data. In the last part of the chapter, we provided a description of several neuromorphic chips found in literature with some related metrics and key information. We have seen from this review chapter that SNNs models are emerging at the beginning of their use, yet only few works have dealt with them from a low-power embedded systems point of view. Those research studies dealing with the subject of neuromorphic computing, approach it from different sides, either on the neural model (training or coding) or on the hardware implementation side. However, few of them deal with the subject from end-to-end, starting from the model to the hardware implementation. Our thesis goes in this direction, where we set a design flow to go from the neural model to the hardware architecture of an AI accelerator. During the design flow, the different constraints of the embedded systems, such as hardware resources, energy budget and real-time restrictions, will be considered.

Chapter 3

Design Space Exploration Methodology

3.1 Introduction

In this chapter, we present a design flow framework that we have adopted for the design space exploration of hardware spiking neural networks. Indeed, there are plenty of architectural and neural possibilities to design and implement SNN-based AI applications on powerful computing platforms. However, when targeting hardware restricted platforms, such as embedded systems, the number of possible solutions is drastically reduced due to the low hardware budget in terms of power consumption, chip surface and real-time processing constraints. Therefore, for dealing with the integration of AI applications on embedded systems, we propose a design space exploration framework that assists the designer in the implementation process of such systems. The objective of this framework to organise the exploration of SNNs for a hardware implementation by starting from the neural model to the structure of the hardware architecture.

3.2 Design flow framework description

This design flow, synthesized in figure 3.1, follows a funnel philosophy: knowing the application context, we start from a wide variety of hardware implementation possibilities and incrementally refine the scope to find the most suitable at the end. In our case, the application context will be embedded image classification.

First, a behavioral software simulation using the N2D2 framework (Bichler et al., 2017) (available online at *N2D2*) is carried out to perform learning, test

and validation for several SNN topologies with different neural coding techniques. The most suitable model in terms of prediction accuracy and spiking activity (the amount of spikes processed by the SNN to perform classification inference) is selected for the following steps, and the learned parameters are then extracted.

Concurrently, a preliminary analytic study is carried out to get the first estimations of flat hardware resources and memory intensiveness corresponding to the chosen SNN model: these first results will serve as a frame for the next steps of our design flow, giving indications for the most suitable hardware architectural choices.

Second, we perform a high-level architectural exploration aiming to confirm or invalidate the assumptions resulting from the preliminary analytic study. The NAXT simulator has been developed for that purpose. The simulator is configured with the model and parameters extracted from N2D2. The software will generate SystemC architectures corresponding to different high-level architectural choices, such as memory distribution, memory technology and processing parallelism. Then, it performs high-level simulation of the operations on the specific user-defined application task. For each simulation, we obtain coarse estimates for power consumption, surface and latency: those results allow us to discriminate suitable architectural models.

Finally, the resulting architectures from the previous round are described in hardware using the parameters extracted from N2D2. This architectural design is done using the mean of the VHDL hardware description language (Navabi, 1998), and a physical synthesis is performed. Thus, this last step leads to a fine-grained evaluation of a suitable architecture on FPGA (Field-Programmable Gate Arrays) or on ASIC (Application Specific Integrated Circuit).

3.3 Preliminary analytical exploration

The aim of this preliminary analysis is to refine the large amount of the design possibilities combining various architectural and neural models. In the following, the preliminary analysis is described in two parts: firstly the neural network model, and secondly the hardware architectural model.

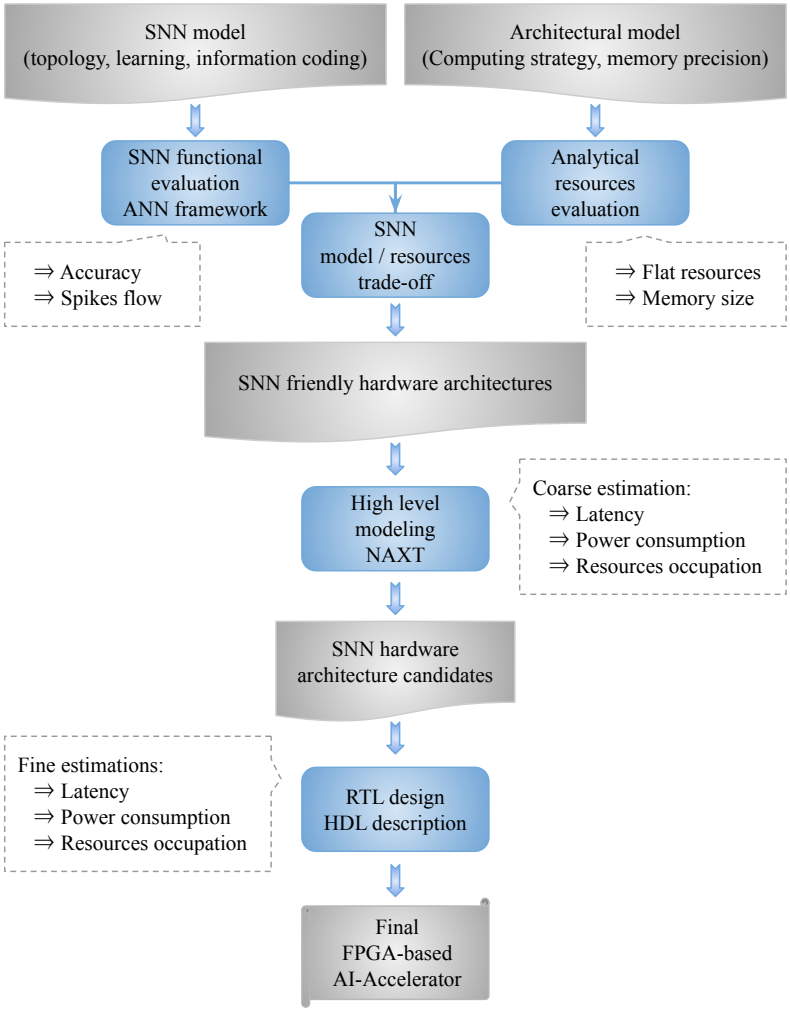


FIGURE 3.1: Design space exploration framework diagram.

3.3.1 Spiking neural models exploration

Indeed, most of the algorithms used to perform image classification are based on ANNs. In this work, we focus on SNNs to deal with embedded AI because they are very competitive in a hardware implementation context. As mentioned in chapter 1, the neuron model and inter-neurons communication in SNNs are more adequate to embedded systems compared to what is used with analog neural models.

Therefore, in the neural model analysis step, we analyze SNNs in two phases:

— first, we explore different learning parameters to get state-of-the-art prediction accuracy;

— second, we explore different neural information coding techniques in terms of spiking activity and their impact on SNN performances.

N2D2 : learning, validating and testing SNNs

In this work, the neural models are learned, validated and tested using the open source Neural Network Deployment and Design (N2D2) framework (Bichler et al., 2017). This software is an event-based simulator for DNNs that we have selected from a wide variety of deep-learning frameworks which are described in literature (Parvat et al., 2017; Mozafari et al., 2019). Actually, we opted to N2D2 essentially for two reasons: first, it is an open source solution that gives the ability to develop new methods without designing a whole simulator. Second, N2D2 offers ANN-SNN conversion possibility to transcode ANNs into spiking domain, which is essential for our purpose. In order to perform simulations of an SNN with N2D2, we follow these steps:

1. Define ANN topology;
2. Learn, validate and test the defined ANN;
3. Define the ANN-SNN conversion method to generate the SNN;
4. Test the SNN defined in 3.

Note that our configuration parameters are listed in table 3.1. We have chosen Xavier Filler as a weight initialization method as it is a popular method which offers state-of-the-art performance (Glorot and Bengio, 2010). Moreover, we have chosen to implement the rectifier activation function (Krizhevsky, Sutskever, and Hinton, 2012; Nair and Hinton, 2010) in hidden layers because it offers state-of-the-art classification performances according to literature (Krizhevsky, Sutskever, and Hinton, 2012). As in (Diehl et al., 2015), we use for classification purposes a linear activation function in the output layer coupled with a classification unit. The classification module follows a Terminate Delta procedure to determine the winning class, this method is discussed in section 4.3.

Neural coding analysis

In an energy consumption context, the neural coding used with SNNs is very crucial, because it influences directly the SNN's spiking activity and thus its power consumption. Actually, the energy consumption for performing a classification task can be estimated using the number of spikes generated to process an input pattern. To do so, the amount of energy a spike consumes on known neuromorphic architectures can be used to estimate the total energy consumption by doing a simple multiplication between this amount and the

TABLE 3.1: ANN's learning hyper-parameters used in this work. Legend : LR stands for Learning Rate.

Hyper-parameter	Value
Weight Initialization	Xavier Filler
Learning Rate	0.01
Momentum	0.9
Decay	0.0005
LR Policy	Step Decay
LR Step Size	1
LR Decay	0.993
Activation function	Linear - last layer Rectifier - other layers

total number of spikes (Cao, Chen, and Khosla, 2015). Indeed, this method is not precise enough but it gives us first estimates that can help us to refine our implementation choices even before moving to RTL design.

Therefore, in the second phase of the neural model analysis, we evaluate the impact of neural coding methods on the spiking activity. These coding methods are presented in chapter 4. In fact, N2D2 simulation tool offers intrinsically Single Burst, Poissonian, Periodic and Jittered Periodic as coding methods with an ability to customize and develop new coding schemes. Therefore, we integrated 2 new schemes : Spike Select and First Spike to N2D2 tool in order to evaluate all the coding techniques under the same conditions and fairly select the most suitable one for the user-defined application.

In this context, we apply these neural coding techniques to different image classification tasks (MNIST, GTSRB) using various network topologies. The networks are first learned and validated in analog domain as described in the previous step. The discrimination of these methods is based on two metrics: the spiking activity and the prediction accuracy. In fact, the neural coding impact on energy-efficiency of SNN architectures is mainly reflected by the prediction accuracy and the amount of spikes propagating in the networks. In other words, we can judge that a coding method is effective if, when we use it, we are able to reduce the spiking activity while keeping the state-of-the-art recognition rate performance.

3.3.2 SNN architectural models exploration

The second part of the preliminary analysis within our design flow is an architectural exploration of different implementation options of SNNs. In

the following, some preliminary results are presented that guide further investigations. These results focus on on-chip memory capacity and resource requirements, which need to be addressed upstream. Indeed, these two restrictions strongly influence later choices in terms of architectural model and hardware target.

Memory capacity : a limiting factor

For any given hardware target, the on-chip memory capacity is always limited. Indeed, even the most recent FPGA devices such as Xilinx® Virtex® Ultrascale™ + and Intel® Stratix® 10 offering high on-chip memory capacity, remain insufficient to handle most neural network models. For reference, the VGG16 ANN model requires a total of 230 MB to store the weight of (Simonyan and Zisserman, 2014).

Consequently, we investigated the evolution of the required memory capacity with respect to the number of weighted synaptic connections. The analytical model for memory capacity is based on the total memory footprint of network parameters and neurons activity, in our case: synaptic weights storage and neurons internal potential. Hence, the analytical approach is related to the parameters coding precision and the topology of the SNN. Figure 3.2 depicts the evolution of required memory capacity with respect to the number of weighted synapses in the SNN. Indeed, the requirement in memory does not depend on the architectural model but only depends on the quantity of synaptic parameters. We observe on this figure that the required memory increases drastically with respect to the number of weighted synapses of the network. Evidently, encoding weights in an 8 bits precision results in less memory requirement than using 16 bits precision. The data in this figure confirm our intuition about memory, which is the fact that it is a major limiting factor when implementing neural networks in general.

Another aspect to take into consideration when dealing with SNNs is the internal potential of the logical neurons. Indeed, due to the temporal computation characterizing SNNs, the internal potential of a spiking neuron must be saved during all the classification period. This is to be able to process input spikes arriving asynchronously. Therefore, the neuron's activity memory cannot be reused by other neurons like in analog models.

In this context, we present in tables 3.2 and 3.3 some analytical results showing memory requirements of different neural models in terms of weights and

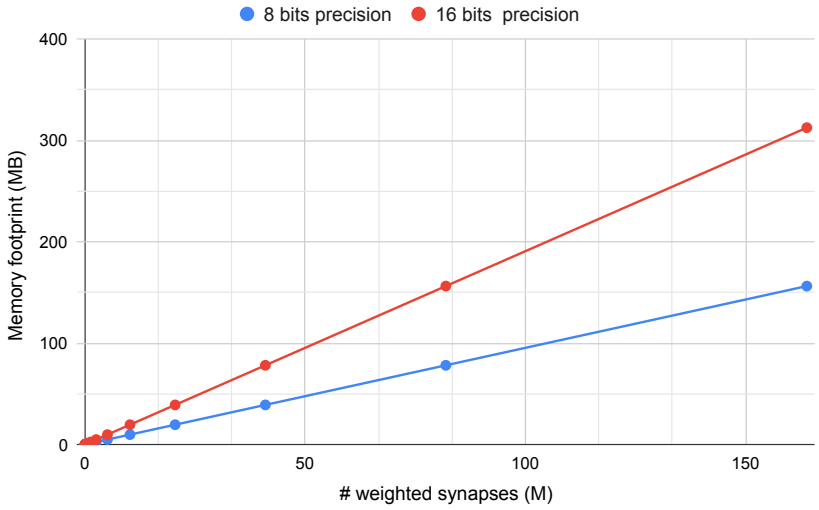


FIGURE 3.2: Memory occupation of two encoding precisions (8 and 16 bits) versus the number of weighted synaptic connections.

TABLE 3.2: Neurons' internal potential and synaptic weights memory usage versus the number of hidden conv. layers – 8 bits encoding precision.

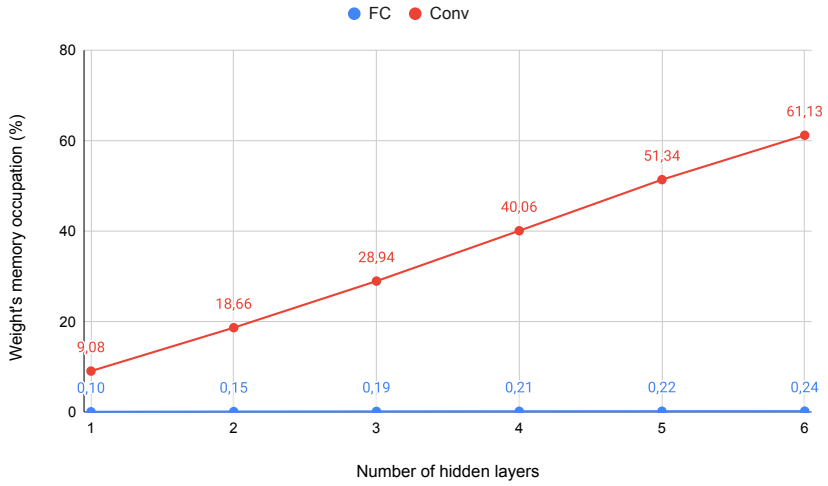
Topology		32x32-Nx(6C5)-10					
Number of hidden layers		1	2	3	4	5	6
Memory usage (bytes)	**Weights**	47190	35610	25950	18114	12294	8490
	Neurons	4714	8170	10570	12106	12970	13354
	Total	51904	43780	36520	30220	25264	21844

TABLE 3.3: Neurons' internal potential and synaptic weights memory usage versus number of hidden FC layers – 8 bits encoding precision.

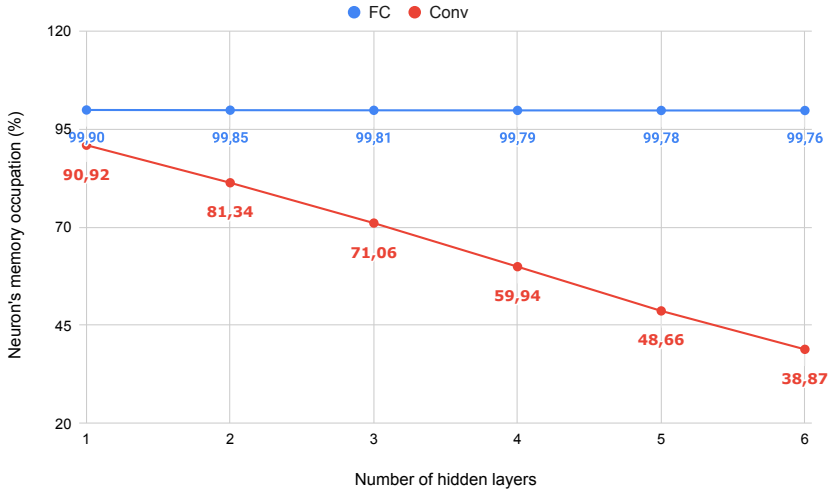
Topology		32x32-Nx(300)-10					
Number of hidden layers		1	2	3	4	5	6
Memory usage (bytes)	**Weights**	310200	400200	490200	580200	670200	760200
	Neurons	310	610	910	1210	1510	1810
	Total	310510	400810	491110	581410	671710	762010

neurons memory requirements. Using these data we have plotted curves showing memory requirements evolution versus the size of the network (number of hidden layers) and layer types. These curves are illustrated in figure 3.3, where 3.3a is dedicated to synaptic weights storage and 3.3b is dedicated for neurons internal potential storage.

For the fully-connected based SNNs shown in blue, we observe that most of



(A) Synaptic weights



(B) Neurons' internal potential

FIGURE 3.3: Neuron and weight memory requirements versus the SNN size.

the memory is occupied by the weights (more than 99%) for all SNN sizes, this because in this layer type there are a lot of connections and thus weights. Whereas, with the spiking CNNs, the occupied memory by the neurons internal potential increases with respect to the number of layers. This is due to: first, there are a considerable number of neurons when adding convolutional layers; and second, only few weights are added due to the weights sharing. Therefore, when dealing with deep CNNs having large feature maps, this internal potential memory would be very large, which may limit the implementation of such neural systems. However, for small network topologies this factor is yet neglectable.

FPGA occupation: towards multiplexing

Logic resources occupation is the second limiting factor we encounter when implementing SNNs on FPGA devices. The FPGA occupation statistics can be obtained by FPGA synthesis simulation tools. However, this synthesis requires a long processing time, especially when synthesizing a large-scale network. Moreover, there is a large number of possible architectural models that have to be designed in order to have robust statistical results. In order to bypass this long implementation and processing time, we have built an analytical model capable of estimating the number of logic cells occupied on an FPGA according to the network topology and size.

To build our analytical model, we have decomposed a generic SNN hardware architecture in elementary modules (neurons, spike generation cell, counters, etc.). For each elementary element, we have measured the corresponding hardware implementation cost in terms of logic cells, using Quartus Prime 18.1.0 Lite edition. Quite straightforwardly, every SNN topology is then expressed as a combination of those elementary modules, and hence can be related to an estimation of its flat hardware implementation cost. Note that some parts of the system can be multiplexed in the final design, but this model only outputs the flat hardware resources as an indicator. The analytical results for a fully-parallel implementation of an SNN with 784 inputs, 10 outputs and a variable number of hidden layers with 100 neurons, are shown in figure 3.4. As depicted in the figure, FPGA occupation grows drastically with respect to network size. Note that this model does not reproduce the organization optimization that the synthesizer may apply when synthesizing the architecture (for example, a single logic unit can be used to perform two different functions in some cases). However, this model is sufficient to give a proper estimation of FPGA occupation against network size in the early stages of the design space exploration.

The analytic model shows, consistent with our expectations, that such fully-parallel implementations face FPGA capacity limits: according to the model, a fully-parallel implementation of 900 IF-neurons would cover 5465 logic cells on FPGA.

Compared to real convolutional networks, which present several tens of thousands of neurons (65,000 for AlexNet (Krizhevsky, Sutskever, and Hinton, 2017)), it is quite obvious that the fully-parallel implementation paradigm

is not viable when using FPGA devices. Therefore, we assume that time-multiplexed architectures are way more viable when dealing with the hardware implementation of deep SNN.

On the other hand, time-multiplexing consists in implementing fewer neurons in hardware than there is in the model : each hardware neuron will thus operate successively for several neurons of the model. This method results in slowing down computation, but allows one to implement large-scale networks with fewer resources, notably for FPGA implementation or cost reduction purposes. Those assumptions will be evaluated in chapter 5.

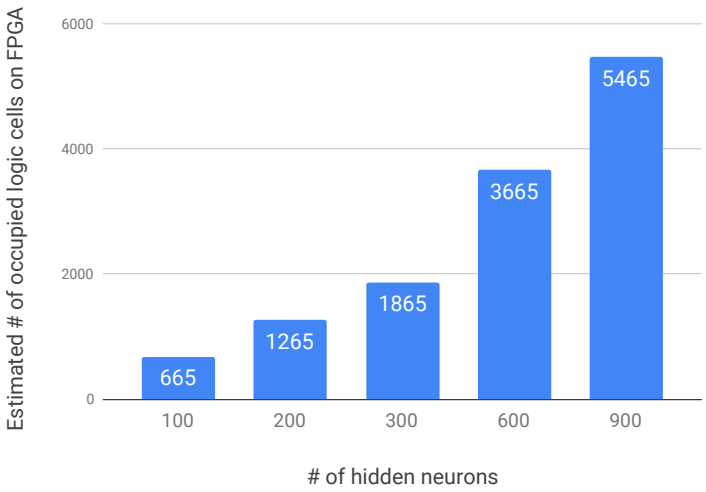


FIGURE 3.4: Theoretical FPGA occupation versus hidden layer size – MNIST.

3.4 High-level SNN's architectural modeling

In this section, we introduce a tool developed in collaboration with Edgar Lemaire (a PhD student in our eBrain team), namely "NAXT" for Neuromorphic Architecture eXploration Tool, which aims to simulate SNN hardware implementations with various architectural choices, such as processing parallelism, memory distribution and memory technology. The goal is to match application specific constraints (power, consumption, logic resources) with high-level architectural choices. The simulator is configured with the SNN parameters extracted from N2D2 (topology and learned synaptic weights) and with user-defined architectural choices (i.e., level of multiplexing, level of memory distribution and memory technology). It subsequently generates a SystemC code corresponding to those parameters, and performs inference on a test data-set. The simulator estimates the chip surface, average latency

and energy-consumption per inference. Hence, the role of NAXT simulator, in the funnel-like architectural exploration workflow, is to easily and quickly provide coarse estimates for different architectural paradigms. Although RTL modeling gives much finer results, it requires a long design and development time. Therefore, the NAXT simulator is used to clear the path, as its results guide further and finer architectural exploration in the following steps of the workflow. Hence, the NAXT simulator is quite innovative as it brings hardware estimation at a very early stage of a design flow, based on a functional description of the network.

Note that the chip surface estimations are relative to an ASIC target, and are analogous to Gate Array occupation for an FPGA target. Indeed, these two metrics are relative to the same "hardware resource" evaluation: a hardware resource can be seen as a piece of circuit from the ASIC point-of-view, or as a group of logic cells from an FPGA point-of-view. Accordingly, chip surface estimations can be taken for FPGA occupation qualitative estimations.

SystemC modeling

To develop our simulator, we used SystemC (Panda, 2001), a behavioral-level hardware description library for C++. This language is often chosen for architectural exploration purposes at a high-level description, as it enables simple functional system description, overcoming the usual finer-level description constraints (transaction modeling, etc.). SystemC enables users to develop functional modules that run concurrent processes and communicate with each other via signals. Thus, we developed three different modules as "elementary bricks" of our hardware architecture models: a Neural Processing Unit, a Memory Unit, and a spike generation module. Before we start a more precise description of each module, it is important to note that our simulator was developed according to a synchronous paradigm: every process is executed at a clock rising edge. The clock signal is generated by the spike generation module. Despite our enthusiasm for asynchronous processing, we have chosen a synchronous simulation paradigm for the purposes of coding ease. We plan to enable asynchronous processing simulation in future development of NAXT simulator.

Neural Processing Unit

The NPU is basically a digital implementation of a spiking neuron. Thus, it is fully dedicated to the Integrate and Fire task. At every clock rising edge,

it integrates synaptic weights corresponding to spikes received during the last cycle. The integration is done in a simple accumulator. After integration, the accumulator value is compared to the membrane's threshold value: if the threshold is exceeded, a spike is emitted at the neuron's output, and the accumulator is reinitialized. If not, it waits until the next clock rising edge to start a new integration, and so on.

Memory Unit

Synaptic weights are stored in memory units. Thus, NPUs must access memory units whenever a spike is integrated. As our architectures work on a synchronous paradigm, integration processes are run simultaneously by all NPUs. Consequently, a memory module can receive several access requests at the same time, but real memory uUnits can only answer one request at a time. Thus, the memory unit model focuses on this aspect: this module must store incoming requests in the right order, and answer those requests one by one in that same order.

Spike Generation module

The spike generation module is dedicated to input image transcoding. Indeed, we have to translate input data from the analog domain to the spiking domain. Various spike coding techniques exist, we will explore them in chapter 4. The spike generation module is responsible for input data transcoding and spike train injection into input neurons of the SNN. Ultimately, this module should disappear as we aim to simulate and evaluate a fully spiking implementation, with true spiking data coming from an asynchronous camera for example.

3.4.1 Parallelism and distribution

As previously described, there are two main exploration steps available in the NAXT simulator. The first is processing parallelism. Indeed, as SNNs are intrinsically parallel algorithms, computation parallelization should result in great acceleration of processing. On the other hand, a high level of parallelization requires a large number of processing elements (ideally, one NPU per logical neuron), resulting in the drastic increase of chip surface. This first level of exploration allowed us to evaluate the trade-off between chip surface savings and processing acceleration.

In the NAXT simulator, this exploration level is modeled by two different architectural paradigms: Fully-Parallel Architectures, and Time-Multiplexed Architectures.

Fully-Parallel Architecture

Fully-Parallel Architecture (FPA) in NAXT stands for the extreme case where every logical neuron in the algorithm is implemented by the mean of one NPU on the chip. This architectural choice should result in fast processing but a large area.

Time-Multiplexed Architecture

Time-Multiplexed Architecture (TMA) in NAXT simulator stands for the case where each layer is composed of only one NPU. Indeed, we could have chosen one single NPU for the whole network as an extreme case, but this would be the equivalent to conventional 'Central Processing Unit' (CPU) based architectures, which is not interesting as we want to explore innovative neuromorphic architectures. We have chosen one NPU per layer to pipeline computations at network level, where there is at least one processing element per layer. In future work, we plan to let the user choose the number of NPUs per layer, for more flexibility and finer exploration purposes. Multiplexed architectures should result in slower processing, but will be interesting in terms of chip area savings.

3.4.2 Memory organization

The second rung of architectural exploration in NAXT simulator is the memory distribution. Thus, three levels of memory distributions have been developed: a 'Centralized Memory' architecture (one memory unit for the whole network), a 'Layer-Shared Memory' architecture (one memory unit per layer), and 'Fully-Distributed Memory' architecture (one memory unit per NPU). Note that in the case of TMA, in the current version of NAXT simulator, layer-shared and fully-distributed memory organizations are the same (1 NPU per layer = 1 memory unit per layer in both cases).

These three different memory architectures allow users to evaluate, once again, the trade-off between processing latency, energy consumption and chip surface (i.e., FPGA occupation). For example, a 'Centralized Memory'

architecture will be more compact than a multitude of 'Distributed Memories', but will slow down processing as it can only answer one single NPU request at a time. 'Layer-Shared Memory' architecture is an intermediate between both architectures. Therefore, in this work we adopt one memory block per layer.

Figure 3.5 depicts all memory distribution levels for fully parallel architectures, and figure 3.6 shows all memory distributions for multiplexed architectures.

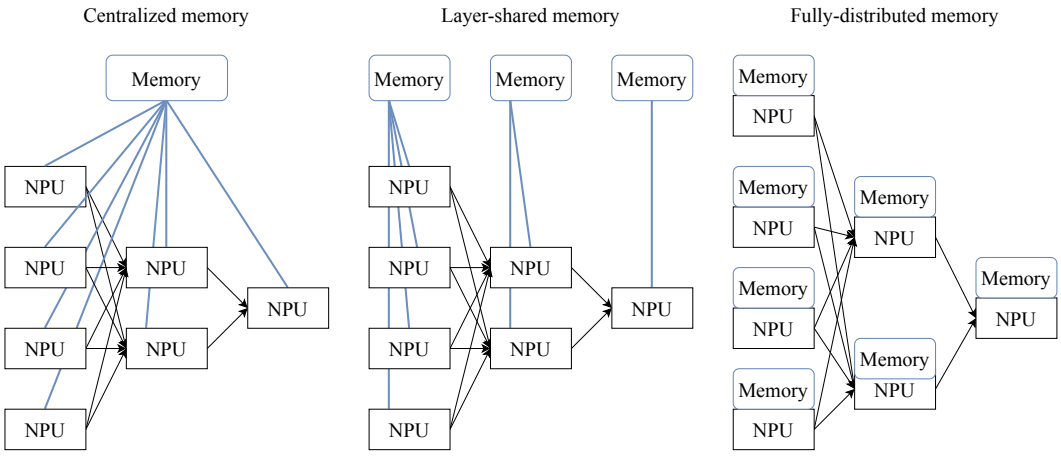


FIGURE 3.5: Memory organizations for FPA architectures.

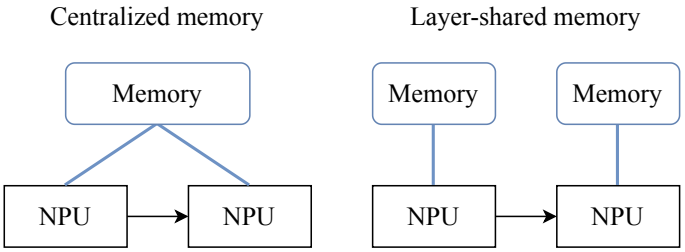


FIGURE 3.6: Memory organizations for TMA architectures.

3.4.3 Latency, Power and Surface estimations

The aim of NAXT Simulator is to give an estimation of power, latency and logic resources for a user defined SNN topology considering different architectural paradigms. To do so, estimations are performed 'a posteriori', using traces generated during inference simulation. More precisely, during inference, all events are recorded: spike emission, read memory access, write memory access, etc. These records, alongside with the number of clock cycles spent for processing, constitute the trace used for estimations.

Latency

Latency is calculated as the product of the number of clock cycles spent for processing and the clock period. The number of clock cycles are being recorded in the trace file, thus we only have to estimate the clock period. To do so, we have chosen to constrain the clock period to the maximum memory access latency, as it is often the limiting factor of computer architectures (worst-case critical path). This latency is estimated using NVSim (Dong et al., 2012), an open-source software aiming to simulate memory behavior for different memory technologies and technology nodes, which return various estimations, including memory access latency.

Hardware Resources

The hardware resources estimation is calculated in a similar fashion than in 3.3.2: we separate our architecture in elementary modules, for each of which we measure the hardware implementation cost in terms of resources. Each architecture is expressed as a combination of elementary modules, and hence can be related to a global hardware resources cost estimation. In this case, the architectural model is taken into consideration, where for example, if we select a time-multiplexed architecture, the resource cost will be equal to the number of layers times the cost of a single NPU. Note that this estimation method does not take into account placement and routing optimizations performed by FPGA design software tools (®Quartus, ®Vivado, etc.).

Power

The power estimation is calculated as a sum of energy consumption of the two main sub-parts of the system: memory and processing. Concerning memory, static and dynamic energy consumption of Memory Units are extracted from NVSim offline simulations. Static energy consumption of Memory Unis are then multiplied by the total inference latency, and summed together. Dynamic energy consumption are multiplied by the number of memory accesses (read and write), and summed together. Both of those results are summed together to give the total power estimation for memory units. Concerning Neural Processing Units, we have taken from literature (Mayr et al., 2015) the average energy consumption per spike of a state-of-the-art hardware digital spiking neuron. This average energy consumption per spike is multiplied by the number of spikes emitted during inference to obtain a

global power estimation of Neural Processing Units. Although this power estimation method is not directly related to our developed Neural Processing Unit architecture, it is a relevant approximation, as it concerns state-of-the-art hardware digital spiking neurons. Finally, both power estimations (for Memory Units and for Neural Processing Units) are summed together to give a global power estimation for the whole system. This power consumption evaluation method is quite approximate and gives coarse estimations, hence it should be improved in future works.

3.4.4 High-level modeling results

Here, we show some data obtained with the NAXT simulator. Note that these estimations are made 'a posteriori' thanks to the network activity traces (the number of spikes processed by each NPU, the number of memory accesses per memory unit, etc.). Simulations have been run for a relatively small network "784-10-10". Our simulator achieves 62% accuracy on MNIST dataset, which roughly corresponds to the equivalent N2D2 recognition accuracy for the same network. NAXT performs latency, surface and power estimations based on traces generated during processing: during each inference, we record the spiking activity of each NPU, alongside all memory accesses for each memory unit. Memory-related estimations have been computed using SRAM technology.

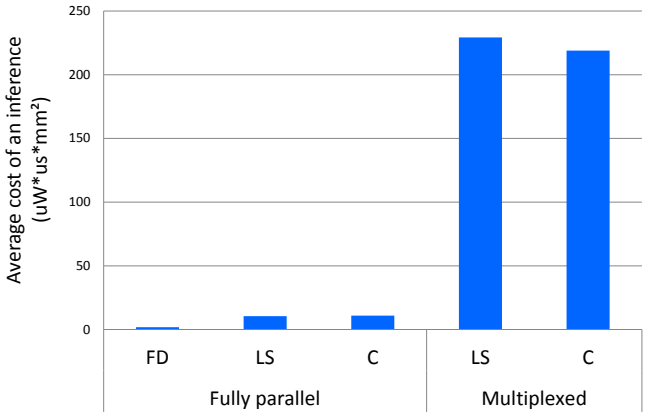


FIGURE 3.7: Qualitative cost function for a 804-hardware-neurons SNN for the different architectures available in NAXT.

For a better understanding of the results presented in table 3.4, we depicted them in figure 3.7 by virtue of a qualitative cost function. This cost function

is calculated as the product of three parameters (latency, energy and chip surface), as we seek to minimize those parameters at the same time. Note that this representation is purely qualitative, but gives a good indication of which architectures are the most suitable for embedded implementation.

TABLE 3.4: Simulation results for a 784-10-10 SNN hardware for the available architectures in NAXT, with SRAM on-chip memories. Legend: LD: Layer Distributed; LS: Layer Shared; C: Centralized.

Architecture	Fully parallel			Multiplexed	
Memory organization [1]	**FD**	LS	C	**LS**	C
Chip area (mm^2)	13	13	13	1.3	1.3
Energy consumption per inference (uJ)	3.34	3.37	3.35	27.9	27.2
Latency per inference (us)	0.042	0.24	0.25	6.32	6.19

The obtained results are consistent with our expectations: the trade-off between chip surface on one side, and energy consumption and latency on the other side, is clearly visible in these estimations. These results show that fully-parallel architectures globally decrease latency and energy cost at the expense of chip surface, while time-multiplexed architectures have the opposite effect. This acknowledgement is quite straightforward, as TMA is based on an opposite design paradigm compared to parallel architectures: they are more compact, but processing serialization results in higher latency, increasing energy consumption (notably because of leakage power). Moreover, we confirmed that the more memory is distributed among processing units, the faster processing will be. Indeed, when memory is centralized, parallel access to stored data is impossible and must be serialized as explained in subsection 3.3.2. This involves a severe increase in latency when memory is centralized. On the other hand, memory architecture does not significantly influence energy consumption and chip surface.

Therefore, we found that both multiplexed and parallel architectures have their own advantages and drawbacks, that is, the trade-off between processing latency, energy consumption and chip surface (i.e., FPGA occupation). In light of these findings, we will develop three architectures: Fully-Parallel, Time-Multiplexed, and the novel Hybrid Architecture, which uses

both paradigms to optimally fit the spiking activity in the network.

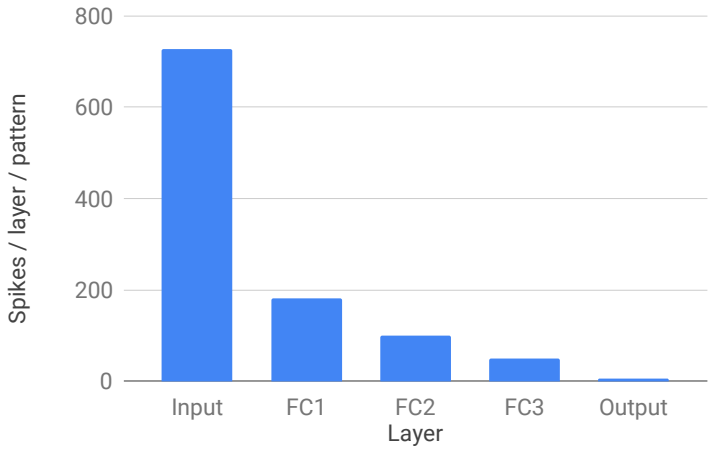


FIGURE 3.8: Average number of spikes generated for one pattern per layer – "784-3x(300)-10" SNN on MNIST.

Indeed, as shown in figure 3.8, in a feed-forward SNN, the number of input spikes per layer decreases drastically as we go deeper in the network: the first layers are much more solicited than deeper layers during inference. This effect is even more prominent when using our novel Spike Select information coding method (see chapter 4). Consequently, we assume that the first layers must be implemented in a fully-parallel fashion to prevent spike bottlenecks, whereas deeper layers can be implemented in a multiplexed fashion. From this assumption, a hybrid architecture has been developed in VHDL and simulated at the Register Transfer Level (RTL), which provides finer estimations than the NAXT simulator.

3.5 Hardware architecture description of SNNs

From the high-level modeling step of the design space exploration, we have identified two possible ways for hardware implementation of SNNs: parallel and time-multiplexed. On another hand, the analysis of spikes distribution over SNN's layers motivated the use of another architectural model consisting in gathering both parallelism and multiplexing. In this section, we give a brief description of these architectural models and how we implement them on hardware.

Fully-Parallel Architecture (FPA)

As described earlier, in FPA all the logical neurons of an SNN are implemented physically in hardware. In other words, for each neuron of the SNN a hardware neuron module is instantiated. Figure 3.9 shows a simplified structure showing the connectivity and organisation of different components that compose the architecture. Indeed, in this architecture, a parallel processing is held in each layer of the network where an input spike is processed by all the layer neurons simultaneously. The input layer is simply a buffer that forwards input spikes to the first hidden layer. The output winner class of the FPA is determined by a classification module that is connected to the last layer of the SNN. Indeed, whenever an output neuron satisfies the classification rule, the processing is ended and its address is considered as the winner class.

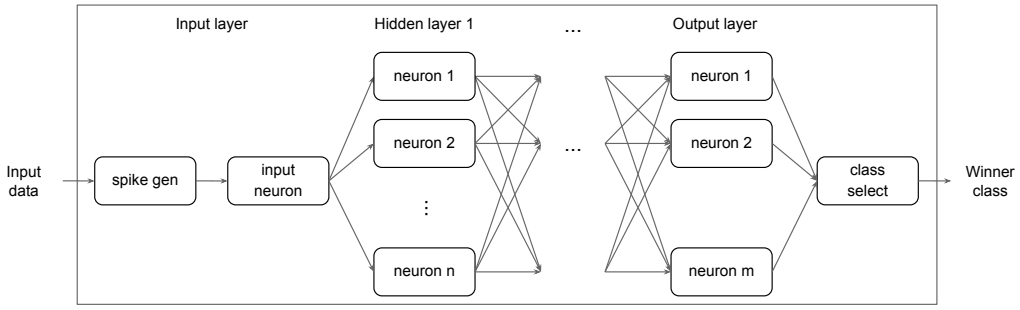


FIGURE 3.9: FPA simplified representation.

This parallel architectural choice should result in fast processing but with high logic resource intensiveness. In the following subsection, we present the second model which takes the opposite architectural choice : Time-Multiplexed Architecture (TMA).

Time-Multiplexed Architecture (TMA)

The TMA comes to save hardware resources and overcome the limitation of a parallel architecture. In this architectural type, the NPUs are used to process several logical neurons. Doing so, the number of hardware neurons will be smaller than the logical ones. One of the possible ways to multiplex computations is to represent each layer by one single NPU, an example of that structure is shown in figure 3.10. This one NPU per layer is required to perform network-level pipelined computations. This is necessary to benefit from the sparsity of spiking data that was discussed in chapter 2. Indeed, within

this TMA structure, each NPU computes successively for the layer's logical neurons in a time-multiplexed manner. Indeed, temporal multiplexing of computations results in higher latency when compared to parallel computing. Therefore, this architecture should drastically diminish the hardware occupation, but increase the system latency as a counterpart.

To sum-up, we have discussed two architectures (TMA and FPA) situated in the two extremes of latency and hardware intensiveness. In the next subsection, we will describe a middle ground between those two extremes, taking advantages from both to fit the reality of spiking activity in SNNs : the Hybrid Architecture (HA).

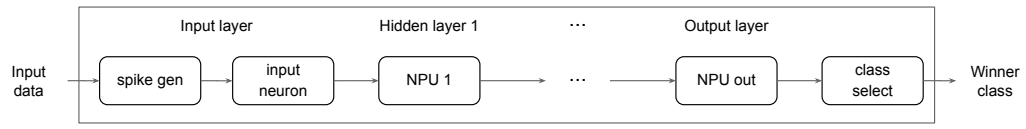


FIGURE 3.10: TMA simplified representation.

Hybrid Architecture (HA)

HA represents the middle ground between those fully-parallel and time-multiplexed extremes. The aim of this architecture is to take advantage of both parallelism and multiplexing to fit the behavior of spiking data in SNNs. The concept comes from the fact that spiking data are extremely sparse, with most of the data located at first layers of the SNN, refer to section 2.6.2. Therefore, having parallel computation in the first layer will speed-up the processing since most of the spikes are located there. Moreover, since fewer spikes are propagated to the deeper layers, NPUs are sufficient to efficiently process those spiking events. A simplified representation version of this hybrid architecture is shown in figure 3.11. Note that, the parallelism in the first hidden layer is represented in a similar way to FPA, for each logical neuron a physical one, and in the remaining layers is represented by NPUs to multiplex computations.

3.6 Conclusion

In this chapter, we have described a three-level funnel-like framework for the design space exploration of hardware SNNs. This is a methodology for addressing the efficient hardware implementation of SNNs for embedded AI applications. In the first level, we analytically explore various neural models

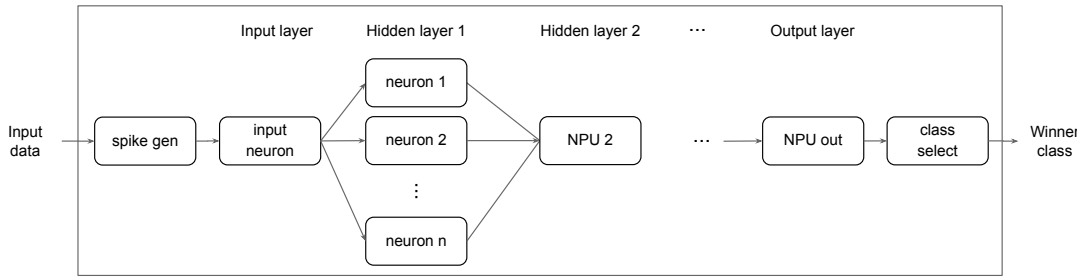


FIGURE 3.11: HA simplified representation.

and architectural structures for inferring SNNs. The aim of this early stage is to eliminate the solutions that do not fit embedded systems' restrictions. Doing so, we reduce the choice panel and have only a few possibilities presented as friendly hardware architectures. In the second level, we model the resulting architectures in high-level. In this context, we have seen that some memory organizations, such as centralized memory and fully distributed memory, are not suitable for our objectives due to the difficulty of controlling their accesses (centralized) or their high implementation cost (fully-distributed), refer to section 3.4.4. The most adequate one is the layer-shared organisation which facilitates the memory accesses and reduces their number. This gives coarse estimations that help us to further refine the design models to get some architecture candidates. At the final level, we do an RTL description of these architecture candidates to get more accurate estimates. In this way, these latest results serve as the baseline for selecting the optimal solution as the final AI-accelerator. The resulting architectural organisations have different computing parallelisms going from fully-parallel implementations to time-multiplexed ones.

In the following chapters, we are going to discuss the other framework levels, which concern neural coding and hardware architectures.

Chapter 4

Neural coding

4.1 Introduction

Indeed, one of the biggest challenges of neuroscience researchers is to understand how the human brain processes and represents the very dynamic and complex external world stimuli. This is a research question that has still not been answered and is the subject of much scientific debate and this in spite of the number of researchers and means made available. The debate concerns mainly the question: does the brain use rate-based coding or spike-based coding? This controversial question forms the subject of the article in (Brette, 2015). In this chapter, we deal with this neural coding, which is the subject of the first level of the framework presented in the previous chapter (chap. 3), by exploring different techniques for SNN's spike generation. First, we present the basic rate-based and time-based neural coding paradigms and show how they are inspired from the information representation in the brain. Second, we discuss the thesis contributions that consist in exploring rate-based and time-based coding schemes and in proposing modified versions targeting low-power embedded hardware classification. Finally, we present the obtained results from experimenting the coding schemes in two different classification use-cases and then conclude about the spike generation for SNNs in the embedded AI context.

Indeed, the brain is composed essentially of neurons and synaptic connections linking them. A neuron is connected to several neighboring input neurons through connection synapses called "dendrites". These input synapses transmit the information coming from the input neurons under the form of

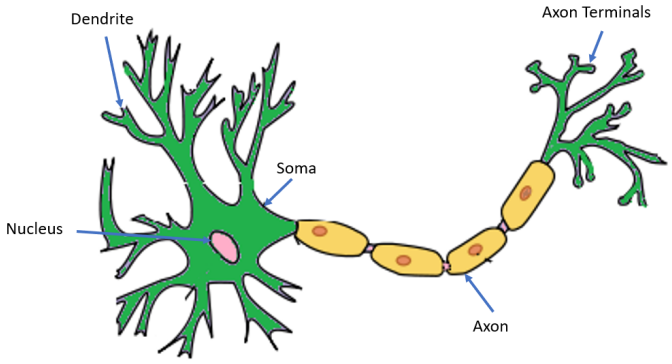


FIGURE 4.1: Simplified representation of a biological neuron - from (Yu et al., 2014).

electrical action potential called spikes. These spikes are continuously integrated to internal potential in the SOMA until the membrane potential exceeds the potential threshold. At this moment, the neuron emits and transmits a spike to the output neurons through the axon and resets its internal membrane potential, refer to figure 4.1. These spikes are used to exchange data between neurons, and to represent input data using a rate-based or time-based paradigm. In this chapter, we present coding techniques that can be used to represent data and generate spikes for the spiking neurons of an SNN into classification tasks. The different techniques are ranging from temporal to rate coding.

4.1.1 Rate-based coding

From (Brette, 2015), we find three definitions of rate coding: over time: the firing rate is captured by the number of spikes emitted by a neuron in a predefined duration time; over neurons: rate represents the average number of spikes produced by a defined number of neurons; and over trials: the firing rate corresponds an average of spikes over a number of trials. Therefore, a firing rate defines how the spikes are propagating in a network of neurons. In this work, we focus on the generation of spikes at the entrance of the spiking network by converting analog data into trains of spikes. In other words, the data are represented in signals carrying flow of spikes, where each spike train corresponds to an input data value. First, the time window is divided to periods equivalent to the input value. Then in each period, a spike is emitted at time "t" determined by applying a distribution function (Poissonian). In figure 4.2, three pixels of a gray-scale image are transformed to spike trains.

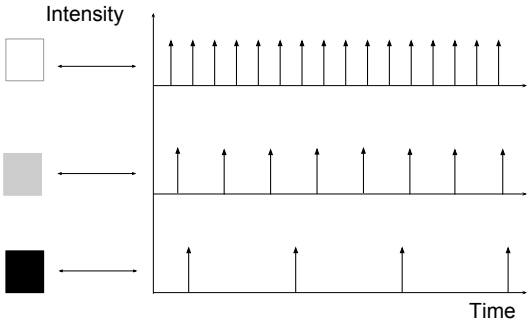


FIGURE 4.2: Rate-based coding paradigm

Several distribution functions can be used to determine the emission time of spikes, we will present and compare them in the rest of this chapter.

4.1.2 Time-based coding

The time-based coding defined as spike-based coding is where the spikes are representing discrete events occurring at relatively well-defined times (Brette, 2015). Moreover, a single spike can represent a physically measurable quantity. Therefore, with time-based coding the emission time of a spike is what matters in decision making. Indeed, in this coding paradigm the information is encoded into precise spike emission time. In an image processing context, a pixel of an image is represented by a single spike emitted at a date t that is inversely proportional to the pixel's intensity, refer to figure 4.3.

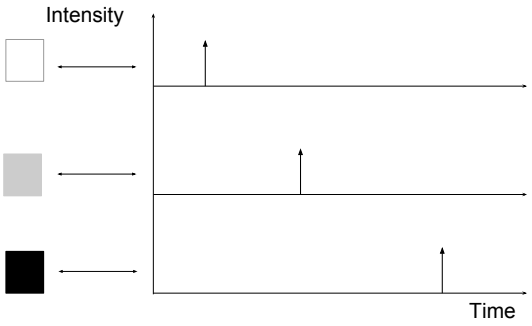


FIGURE 4.3: Temporal coding paradigm

4.1.3 Neural coding versus energy-efficiency

In SNN architectures, information is encoded in spikes. The spike, also called "action potential" or "nerve pulse" in biology, is generated by a spiking neuron, in a process called "firing". In a feed-forward SNN with Fully-Connected (FC) layers, spikes are transmitted to all the next layer neurons.

Several neural information coding approaches have been proposed by neuroscience researchers, including Rate Coding, Time Coding, Phase Coding, Rank Coding, Population Coding, etc. (Brette, 2015). In this study, we focus on Rate Coding and Time Coding for two reasons:

1. Rate coding: for its maturity. When using this method, SNNs reach State-of-the-Art performance on classification applications (Cao, Chen, and Khosla, 2015; Khacef, Abderrahmane, and Miramond, 2018);
2. Time coding: when used, fewer spikes are propagated in the SNN, which reduces computation and resource intensiveness during inference (Yu et al., 2014; Mostafa, 2018).

Based on these methods, we propose some modified versions of the standard Rate Coding to make trade-offs with the temporal coding paradigm: maintain high accuracy and reduce the number of spikes flowing in the network. Indeed, the energy consumption of an SNN hardware implementation is directly proportional to the number of spikes it generates. As mentioned in (Cao, Chen, and Khosla, 2015), an estimation of the energy consumed by processing an image is calculated using the equation 4.1.

$$E_{total} = N_{spikes/image} \times \alpha \qquad (\mathrm{J/image}) \tag{4.1}$$

Where E_{total} is the average energy consumed for processing an image, $N_{spikes/image}$ is the total number of spike emissions per input pattern, and α is the energy consumption of a spike emission. Note that the spiking-activity-related energy consumption varies from one accelerator to another and obviously depends on the specific architecture.

4.2 ANN-SNN conversion

The Spiking Neural Networks are bio-inspired ANNs issued from neuroscience. Compared to Analog Neural Networks, different neuron models

(LIF, IF, Izhikevich, etc.) and neural coding are used as depicted in neuroscience literature (Brette, 2015). In fact, we use the non-leaky IF model which is composed of an adder and a threshold comparator, as shown in figure 4.4. The leaky part is not considered in the implementation of the hardware neuron for two reasons:

— The Leaky part implies a more complex hardware structure with a larger chip surface;
— The non-leaky IF model gets equivalent accuracy records to analog networks using rate coding.

In addition, the data exchanged between neurons are coded with one-bit signals using rate or time based coding. These aspects make the SNNs more suitable for embedded AI applications.

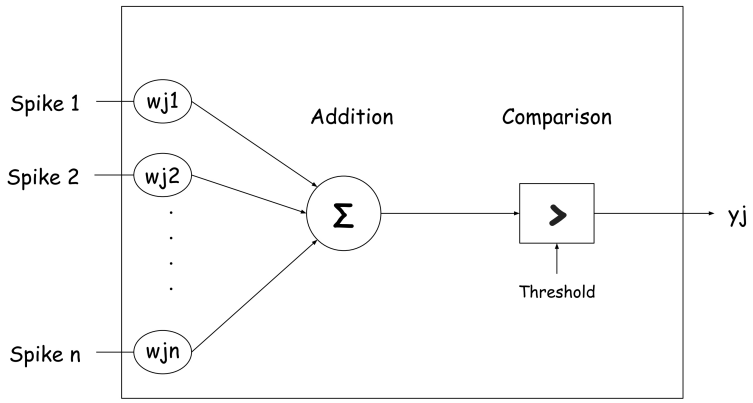


FIGURE 4.4: Non-leaky Integrate-and-Fire neuron.

In this work, we build SNNs using the ANN-SNN conversion methodology that is already described in (Diehl et al., 2015; Perez-Carrasco et al., 2013). Indeed, there is other techniques to build and learn spiking neural neural networks like temporal back-propagation (Kheradpisheh and Masquelier, 2020; Yu et al., 2014), spike-based back-propagation (Rotermund and Pawelzik, 2019; Lee et al., 2020; C., Bichler, and Dupret, 2019), STDP learning (Thiele, Bichler, and Dupret, 2018; Kheradpisheh et al., 2018) or even reward-STDP (Mozafari et al., 2018). However, in this work we adopted ANN-SNN conversion because we get more easily the state-of-the-art classification results. First, we define the analog neural networks topology and then use the rectifier function for the neurons, set biases to zero and learn the network with back-propagation. Then, the resulting weights from this first step are used to define the SNN that is composed of IF neurons. At this point, the SNNs is

ready for inference on spiking data that are generated using one of the neural coding techniques that will be described in the remaining of the chapter. Finally, we test the converted SNN on spiking data generated by the spikes generation cell. The classification result of the SNN is decided using the classification module that is described in the following.

4.3 SNNs classification policy

Usually, in ANNs, a Softmax layer is used at the output layer for classification purposes. In the spiking domain, we often replace this classification method by a linear activation function at the output layerDiehl et al., 2015 followed by a spike-based classification procedure to determine the winning class.

As previously mentioned, the spiking data are different from the ANN analog data, where neurons produce spike trains instead of activities. Therefore, one possible softmax equivalent function with SNNs would be a function which can identify the most spiking neuron at the output layer. That neuron can then be selected as the winning class of the SNNs. Hereby, we present two spiking classification methods that can be used with SNNs: Delta Terminate and Max Terminate. Software versions of these procedures have been implemented in the N2D2 tool. The figures 4.5 and 4.6 represent flowcharts corresponding to operations of Terminate Delta and Max Terminate methods, respectively.

On one hand, with the Terminate Delta procedure, class prediction occurs when the most spiking neuron has fired "TD" times more than the second most spiking neuron. Note that, a maximum number of spikes amount that a neuron fires is defined to stop the processing if the terminate delta rule is not reached. As depicted in the flow chart on figure 4.5, the output neuron activations are recorded and updated at each incoming spiking event, by checking the status of "Empty". Then for two neurons, those with the first and second highest activity, the number of activations and address (Max1/2 and @Max1/2) are registered. For each new spike, its address is compared to @Max1 for checking completion of the classification. Indeed, if the input event corresponds to the most spiking neuron (@Max1), then the corresponding number of activations is incremented (Max1++). Then, the difference between its activation (Max1) and the one of the second most spiking neuron (Max2) is compared to the delta value (TD). If this difference is higher than

the delta value, the processing is stopped and @Max1 is selected as the winner class. There is another case where the input event does not match with @Max1. In this case, there are two possibilities: first, if the input event address is equal to @Max2, we check if this neuron becomes the first instead of second most spiking neuron; second, the input event address is different from @Max1 and @Max2, here we check if this new neuron has spiked more the neuron @Max2.

On another hand, in Max Terminate, the classification process is completed whenever an output neuron (the most spiking neuron) reaches the maximum number of spikes ("Delta" in figure 4.6). Its detailed operation is depicted by the flowchart on figure 4.6, it works similarly to the Terminate Delta procedure but with a lot of simplifications. Therefore, compared to terminated delta technique, the operation of this method requires fewer parameters (Max, @Max and act) and is doing less computations. In section 4.5.2, we confront the two methods in terms of accuracy and number of generated spikes to select the most appropriate to use with SNNs.

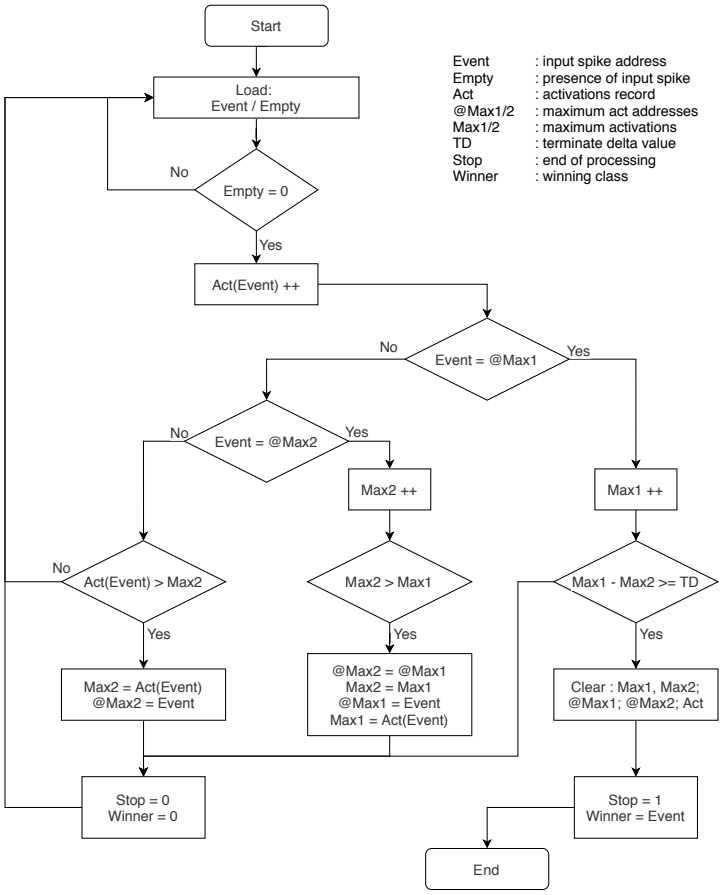


FIGURE 4.5: Terminate Delta flowchart.

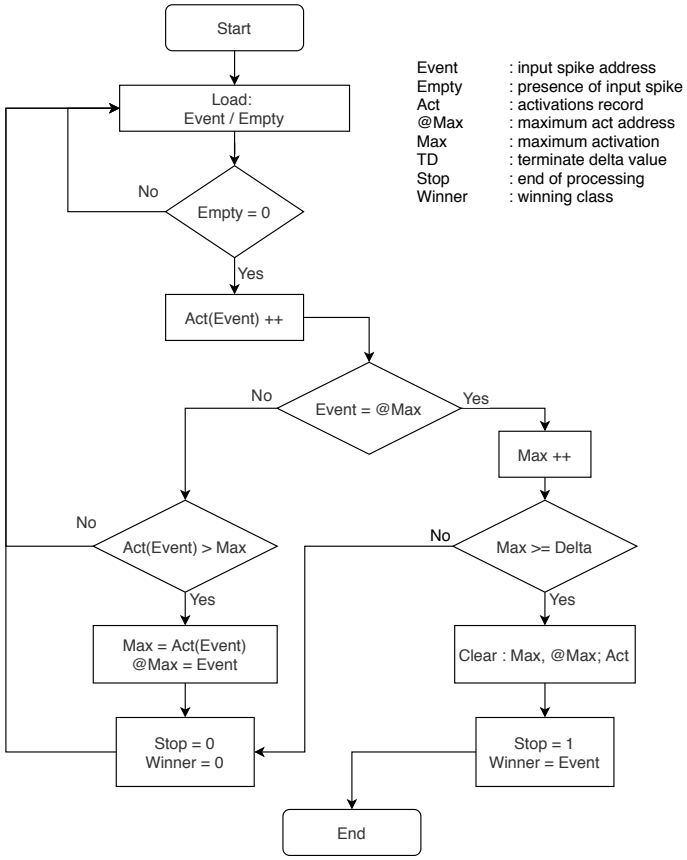


FIGURE 4.6: Max Terminate flowchart.

4.4 Spikes generation methodologies

4.4.1 Rate-based coding

Rate-based coding is the most widespread method for converting analog data into spike trains. Based on the analog data value, a corresponding period, that is equivalent to the spikes emission frequency, is computed using equation **??**. In figure 4.2, three gray-scale pixels of an image are transformed into spike trains: each pixel is represented by a spike train whose frequency is proportional to its intensity.

In N2D2 (Bichler et al., 2017), there are several rate coding techniques for converting input stimulus data to spiking domain: Poissonian, Jittered Periodic and Periodic. These conversion methods are developed to feed an event-based system, thus the output data are in a form of events defined by their timestamps. They are computed as follows: First, some parameters are defined: p_{min} and p_{max} the minimum and maximum mean periods corresponding to separation time between consecutive spikes, s_{dev} the relative standard deviation and P_{min}^{sys} the minimum spikes separation time supported

by the system. Second, the input data *value*, which is first re-scaled between 0 and 1, is converted to a time period p using the formula in equation 4.2.

$$p = 1/(f_{max} + (1 - \mid value \mid) * (f_{min} - f_{max})) \tag{4.2}$$

Where $f_{max} = 1/p_{min}$ and $f_{min} = 1/p_{max}$ the maximum and minimum mean periods. Then, from this period we compute a time step Δt that is used to get the input value spiking time, it is computed with one of the three methods:

— **Poissonian**: $\Delta t = f_{Edist}(p)$, apply an exponential distribution function to the period.
— **Periodic**: $\Delta t = f_{Ndist}(p, s_{dev})$, apply a normal distribution function to the period, in our experiments we fix s_{dev} to 0.
— **Jittered Periodic**: $\Delta t = f_{Ndist}(p, s_{dev}) * f_{Udist}()$, multiplication of the normal distribution function of p with a random uniform variable.

These methods are injecting white noise to the time period p to get the time step Δt. Then, Δt will be replaced by p^{sys}_{min} ($\Delta t = p^{sys}_{min}$) if it is too small ($\Delta t < p^{sys}_{min}$), i.e. less than the spikes minimum separation time. Finally, the spiking time is given by: $t = t_{previous} + \Delta t$, where $t_{previous}$ is the timestamp of the last generated spike.

4.4.2 Temporal coding : Single Burst

With the Single Burst coding method the input data stimulus is mapped to temporal domain. An input data value is represented using a one spike, which is emitted at a specific time t, computed by "$t = \mid 1 - v \mid * w_t$", with t the emission time, w_t the time window dedicated for the generation of the spikes, and v the input value. A software version of this method is present in N2D2 (Bichler et al., 2017) as a stimulus transcoding type.

4.4.3 Temporal coding : First Spike

Derived from standard rate coding, the First Spike method is an intermediate version between time-based and rate-based coding paradigms, by having aspects in common with both methods. On one hand, as for time-based coding, it uses only one spike to represent information. On another hand, similar to rate-based coding, the data flowing inside the SNN between the IF-neurons is encoded in a rate-based paradigm. A pseudo-algorithm, presented in figure 4.7, shows a transformation of an input value (v) to spiking domain using

the First Spike method. First, some parameters that will be used in the conversion process are defined. Afterwards, a period p is calculated based on the value v using equation 4.2.

Where, f_{min} and f_{max} are minimum and maximum frequency parameters. Then, the period p is used to compute the time step Δt, which corresponds to the date of the next spike emission, thanks to the function $\Delta t()$ presented in equation 4.3.

$$\Delta t(p) = f_{Udist}(f_{Ndist}(p, s_{dev})) \tag{4.3}$$

With s_{dev} being the standard deviation, $f_{Ndist}()$ the Random normal distribution function and $f_{Udist}()$ the Random uniform distribution function.

Next to that, the Δt value is compared to $Tmin$, which is the minimum spike delay (no spike can be emitted earlier than $Tmin$) to fire a spike. If $\Delta t > Tmin$, a spike is emitted at time Δt, otherwise, the spike will be emitted at time $Tmin$. This process is done only once for each pixel, because this method consists in generating only one spike per input pixel.

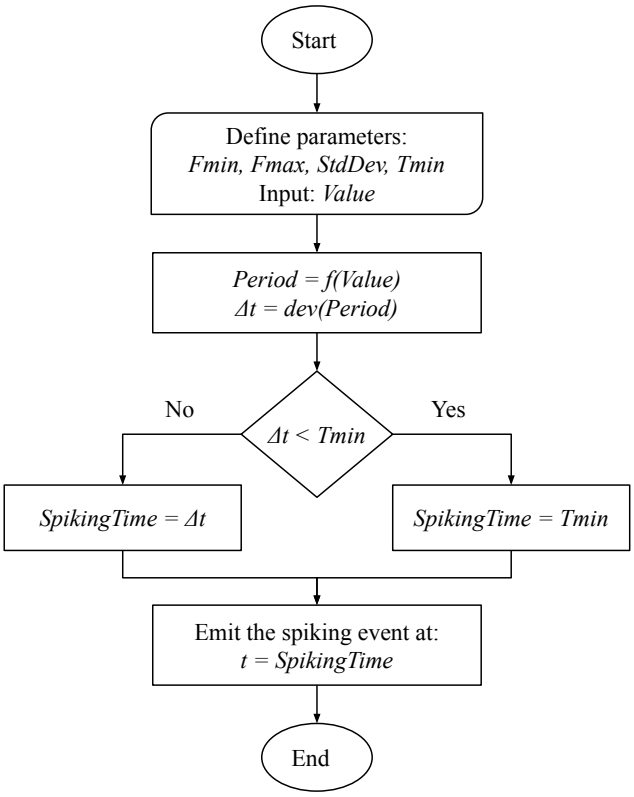


FIGURE 4.7: First Spike method flow-chart.

4.4.4 Hybrid coding : Spike select

Rate coding techniques, such as Jittered Periodic, rely strongly on the high number of output spikes or the terminate delta rule to improve the accuracy. When using the terminate delta classification module, increasing the delta value results in more spikes at the output layer. This fact allows us to better differentiate the data classes that are represented by the SNN output neurons. Moreover, this delta value increase produces a larger number of spikes over all the SNN's layers. To diminish this number of spikes, especially in the deeper layers, an opposite approach is used with the Spike Select method.

Instead of increasing the delta value of the Terminate Delta classification approach, which influences the overall activity over the network, spike select uses the first hidden layer's threshold to control the propagation of the spikes and to improve the accuracy. This is done by increasing the threshold of this first layer while keeping the base threshold in the other layers. Furthermore, the delta value of the classification module is simply replaced by the base threshold. As a result, the spikes generated by the neurons of the first hidden layer are reduced where most of these spikes come from the frequently firing neurons when using Jittered Periodic coding. This spike selection allows the use of a small delta value, that is equal to the base threshold, with Terminate Delta without affecting the accuracy.

In figure 4.8 shows an example showing a group of neurons for which the spike select method is applied, which is done by increasing their threshold. The original threshold (base) used with Jittered Periodic for the neurons (in left of the figure) is 1 and the one used with spike select ones (in right of the figure) is 3. The neurons with spike select, having higher threshold, have a reduced output spikes amount compared to the neurons with Jittered Periodic. This is due to the fact that the neurons with spike select reach their threshold less frequently because their threshold is 3 times higher than the original one. On the same figure 4.8, a zoom-in is made on neuron number 0 (N0) to show the change of its internal potential state at each output spike. Using Jittered Periodic coding, when an output spike is emitted, the internal potential is greater than or equal to 1.0, which is the threshold amount, then it is reset. Since for both the coding methods, the same input spikes are fed to the neuron, then the internal potential is incremented similarly with both coding methods. Since the difference between them is only the value of their threshold, the difference is in their firing of output spikes and reset of the potential. In the case of Spike Select, the firing of a spike and the potential

reset occurs when the internal potential reaches 3.0 instead of 1.0 in case of Jittered Periodic. Due to this difference, when going from Jittered Periodic to Spike Select, the amount of output spikes is reduced from 8 spikes to only 2 for the neuron N0. Therefore, applying this method to a deep SNN would drastically reduce the amount of spikes generated over the network. Note that the amount of the spike select threshold is experimentally fixed through several trials similarly as with Terminate Delta.

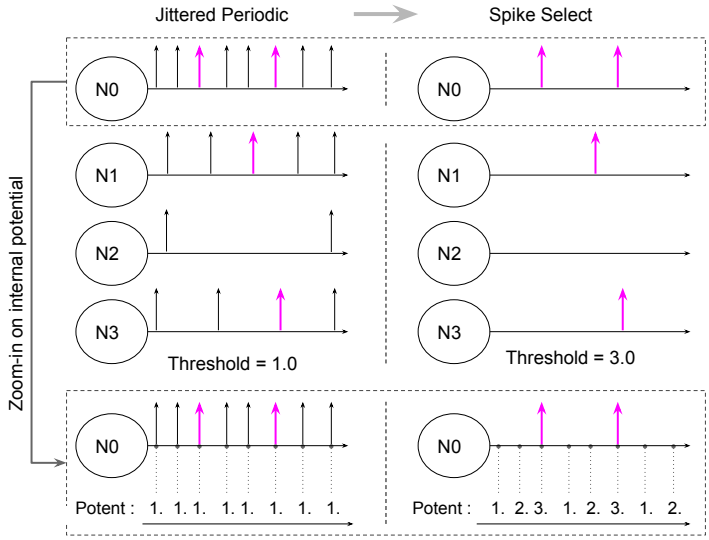


FIGURE 4.8: Spike Select and Jittered Periodic codings effect on output spikes of an integrate-and-fire neurons group.

Naturally, spiking data are characterized by a high level of sparsity with the quantity of spikes decreasing increasingly as one goes deeper and deeper in the network layers. Indeed, statistical results of SNNs with rate-based coding, such as Jittered Periodic (refer to results section 4.5), have confirmed this sparsity. The results show that most of the spiking activity is located in the first layer and only few spikes are present in the deeper layers. Spike select would result in a more sparse data since it is coming to enforce this aspect by letting fewer spikes propagate in the deeper layers. Therefore, if this method results in a to rate-based coding equivalent accuracy, it will be one of the most suitable methods for implementing SNNs in hardware. These assumptions will be verified in the experiments and results section 4.5.

From the hardware perspective, this is a very promising method that allows for efficient hardware usage, especially with the hybrid architecture presented in section 3.5. Indeed, in this architecture, the first hidden layer is implemented in parallel, and the deeper layers are time-multiplexed : this

architectural configuration fits well with the spiking distribution implied by the spike select coding method.

4.5 Experiments and results

In this section, we present the experiments and results of the different neural information coding techniques presented so far: analog to spiking domain transcoding methodology, spike generation coding and classification policy with SNNs.

In this section, the experiments and results that concern the neural coding, that is covered in this chapter, are presented. First, we present data concerning the ANN-SNN conversion, then the classification methods are reviewed, and finally we discuss the results of the spike generation techniques.

4.5.1 Experiment setup

To check the validity of the ANN-SNN conversion method, we build several ANN topologies using the N2D2 framework following the steps presented earlier. For both the spiking and analog models, a rectifier function is used for the hidden layers and a linear function is used at the output layer. The weights initialization is done using the Xavier filler function and the learning is done using the back-propagation algorithm.

To define our ANN models, we use a ".ini" file where a typical convolutional layer is defined as follows:

```
[conv1]
Input=sp
Type=Conv
KernelWidth=4
KernelHeight=4
NbOutputs=32
Stride=1
WeightsFiller=XavierFiller
ActivationFunction=Rectifier
ConfigSection=common.config
```

Where "[conv1]" indicates the name of the current layer, "Input" defines the previous layer, "Type" defines the layer's type, "KernelWidth" and "Kernel-height" are used to configure the size of the filter and "NbOutputs" indicates the number of filters in the layer.

In the same file, a configuration section "ConfigSection" that contains all the learning hyper-parameters is defined as follows:

```
[common.config]
NoBias=1
WeightsSolver.LearningRate=0.01
WeightsSolver.Momentum=0.9
WeightsSolver.Decay=0.0005
Solvers.LearningRatePolicy=StepDecay
Solvers.LearningRateStepSize=[sp]_EpochSize
Solvers.LearningRateDecay=0.993
Solvers.Clamping=-1:1
```

Where "NoBias=1" means that we are not using bias neurons, the other lines define the learning rate, momentum, decay, the learning hyper-parameters (learning rate policy, etc.). "Solvers.Clamping=-1:1" is used to normalize the weights between -1 and 1. Similarly to most of the ANN-SNN conversion works, in this work we are not using bias neurons, which when combined with ReLU activation functions allows to improve the SNN's accuracy (Cao, Chen, and Khosla, 2015).

Training and testing data-sets

For training, validating and testing the neural networks of this study, we used two different data-sets: the handwritten digits MNIST dataset and the traffic signs GTSRB dataset. On one hand, we have MNIST data-set, which is a handwritten digit's database of 70000 images (60000 for learning and validation, 10000 images for test). Figure 4.9b shows some pattern examples of this dataset.

On the other hand, we have GTSRB dataset that has 43 classes spread on 51840 colored (RGB) image samples (figure 4.9b), 50% of them is used for training, 25% for validation and the remaining 25% is used for testing. The

images have different sizes ranging from 15x15 to 250x250. Therefore, re-scaling the input within the CNN is mandatory (Stallkamp et al., 2012).

(A) MNIST dataset examples.

(B) GTSRB dataset examples (Stallkamp et al., 2012).

FIGURE 4.9: Pattern examples of the datasets used in this work.

ANN topologies

In this work, we used two types of ANN topologies: the first one is composed of networks having only fully-connected layers and the second one is composed of CNNs that are constituted from convolutional, pooling and fully-connected layers.

For both data sets, we defined 4 CNN topologies and 2 FC-based ones. First, the FC-based networks are composed of an input layer of 28X28 (784) neurons, one to three hidden layers of 300 neurons and 10 neurons in the output layer for MNIST dataset and 43 neurons for GTSRB dataset. Second, the CNNs used with MNIST are : *Conv29*: a CNN with a mono-channel (1C) input with a size of 29x29 followed by 2 convolution layers, 2 pooling layers and 2 Fc layers; *ConvFc*: a CNN composed of only convolution and fully-connected layers; *ConvPool*: a CNN composed of only convolutional and pooling layers; and the famous LeNet network. Finally, the 4 CNN topologies used with GTSRB data-set are two mono-channel networks, that process the gray-level version of the data-set: *LeNet-1C* and *Conv-1C*, and two three-channel networks having three-channel inputs that support the RGB version of the data-set: *LeNet-3C* and *Conv-3C*.

TABLE 4.1: ANN topologies used with MNIST and GTSRB data-sets. Legends: "c" stands for convolution, "p" stands for max pooling, "s" stands for stride and no letter means FC layer.

Dataset	Network	Topology
MNIST	FcNet1	784-300-10
	FcNet2	784-3*(300)-10
	Conv29	29x29-32c4s1-32p2s2-48c5s1-48p3s3-200-10
	ConvFc	28x28-16c4s2-24c4s2-150-10
	ConvPool	28x28-12c5s1-12p2s1-64c5s1-64p2s1-10
	LeNet	32x32-6c5s1-6p2s2-16c5s1-16p2s2-120c5s1-84-10
GTSRB	FcNet3	784-300-43
	FcNet4	784-3*(300)-43
	Conv29-1C	29x29x1-32c4s1-32p2s2-48c5s1-48p3s3-200-43
	Conv29-3C	29x29x3-32c4s1-32p2s2-48c5s1-48p3s3-200-43
	LeNet-1C	32x32x1-6c5s1-6p2s2-16c5s1-16p2s2-120c5s1-84-43
	LeNet-3C	32x32x3-6c5s1-6p2s2-16c5s1-16p2s2-120c5s1-84-43

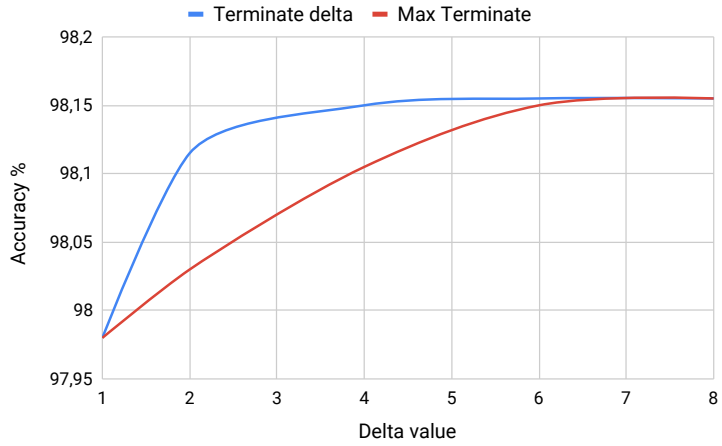
4.5.2 SNN classification policies

In this section, we evaluate the impact of the classification policy on SNN's performances. To do so, we compare performances, in terms of accuracy and spiking activity, of Max Terminate and Terminate Delta winner-class selection methods. For both methods, we used a three-hidden layers network with the 784-3x(300)-10 topology. The recorded results for both accuracy and average number of spikes generated per image are listed in tables 4.2.

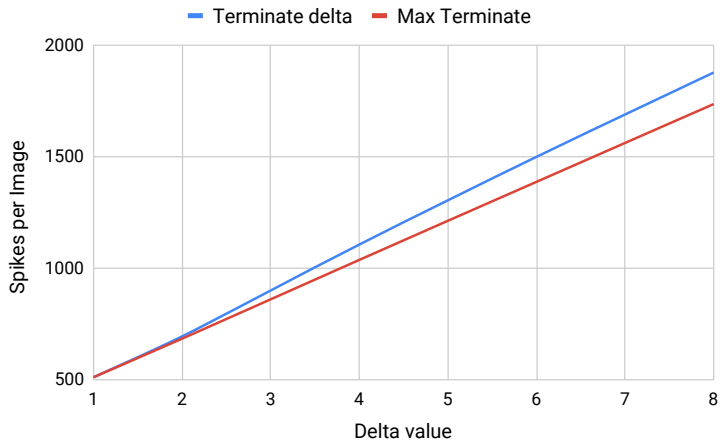
TABLE 4.2: Terminate Delta and Max Terminate versus accuracy and spiking activity on MNIST dataset. Neural coding: Jittered Periodic; Topology: 784-3x(300)-10.

Delta	Terminate Delta		Max Terminate	
	Accuracy	Spike/image	Accuracy	Spike/image
1	97.98	511.47	97.98	511.47
2	98.11	694.49	98.03	684.32
4	**98.15**	**1106.24**	98.10	1037.44
6	98.15	1499.56	**98.15**	**1387.47**
8	98.15	1877.50	98.15	1736.16

For better representation and based on the results of this table 4.2, we have plotted the curves in figure 4.10. The first sub-figure 4.10a represents the evolution of the accuracy versus different delta values for both methods, where the red curve represents Terminate Delta and the blue one is for Max Terminate. The second sub-figure 4.10b represents the evolution of the number of spikes generated over the SNN in average per pattern versus the delta value.



(A) Accuracy



(B) Average number of spikes per image

FIGURE 4.10: Terminate Delta and Max Terminate versus accuracy and spiking activity - 784-3x(300)-10 - MNIST data-set.

We notice that for both methods, the accuracy and the average number of spikes increase continuously with respect to the TD/Max delta values. Moreover, using the Terminate Delta method the SNN performs slightly higher accuracy than using Max Terminate while having a thin increase in the number of spikes. For delta values greater than 4, the accuracy begins to stabilize and gets its maximum for Terminate Delta. Whereas with Max Terminate method, this maximum accuracy is reached only when the delta value is greater or equal to 6. Considering the spiking activity, we notice that the maximum accuracy is reached with Terminate Delta when the number of spikes is equal to 1106.24. Whereas with the Max Terminate method, the maximum accuracy is reached when the number of spikes is equal to 1387.47. From these results, we conclude that Terminate Delta is more interesting since for

equivalent accuracy, less spiking events are generated than with Max Terminate approach. For this reason, we use the Terminate Delta method to implement the SNNs in the remaining parts of the work.

4.5.3 ANN-SNN conversion versus accuracy

The mapping of ANNs to SNNs methodology is evaluated by comparing the accuracy results in both spiking and analog domains. Note that Jittered Periodic coding method is used to generate input spikes for the spiking models. From the table 4.3, which summarizes accuracy results of some SNNs on both MNIST and GTSRB datasets, we observe that the records are slightly similar for both spiking and analog models. These observed data validate the adopted ANN-SNN conversion approach and allow us to continue our investigation by exploring the different neural coding techniques.

TABLE 4.3: Accuracy results of analog and spiking models.

ANN topology	Accuracy (%)	
	Analog	Spiking
FcNet1	97.85	97.74
FcNet2	98.35	98.24
Conv29	99.16	99.16
ConvPool	99.18	99.21
LeNet	99.18	99.15
ConvFc	99.09	99.09

(A) MNIST data-set

ANN topology	Accuracy (%)	
	Analog	Spiking
FcNet3	90.25	89.81
FcNet4	91.18	90.68
Conv-1C	96.80	96.68
Conv-3C	97.86	97.75
LeNet-1C	95.84	95.76
LeNet-3C	96.25	96.19

(B) GTSRB data-set

4.5.4 State-of-the-art accuracy results

Before dealing with neural coding, let's compare the obtained accuracy results using the ANN-SNN conversion approach to some records that are found in literature. These comparison results, both on MNIST and GTSRB data-sets, are listed in tables 4.4 and 4.5. First, with MNIST data-set, we obtained accuracy results equivalent to the ones that are found in literature. Second, with the GTSRB data-set, we get lower results compared to human (Stallkamp et al., 2012) and architecture_32 network found in (Yang et al., 2020) but higher than the other networks. From these results, we can validate the mapping approach to be used for constructing the Spiking Neural Networks.

In the context of embedded hardware classification, this step allows us to use this Jittered-Periodic-based conversion approach to explore other neural

TABLE 4.4: Classification accuracy results of different SNNs on MNIST data-set

SNN topology	Accuracy (%)
(Diehl et al., 2015): 784-2x(1200)-10	98.60
(Du et al., 2015): 784-300-10	95.40
(Mostafa, 2018): 784-800-10	97.55
(Kheradpisheh and Masquelier, 2020)	97.40
This work: FcNet1	97.74
This work: FcNet2	98.24
This work: LeNet	99.15
This work: Conv29	**99.21**

TABLE 4.5: Classification accuracy results of different network topologies on GTSRB data-set

Network topology	Accuracy (%)	Spiking
Human (2012)	**97.98**	-
LeNet (2020)	92.68	No
Random forests (2011)	96.14	No
Architecture_32 (2020)	97.79	No
9 layers CNN on TrueNorth (2016)	96.50	Yes
This work: LeNet-1C	95.84	Yes
This work: LeNet-3C	96.25	Yes

coding techniques and to evaluate them in terms of accuracy and spiking activity.

4.5.5 Spikes generation

In this part, we present the process of validating the spike generation neural coding techniques which is organized in two parts: First, we test different basic rate-based coding techniques; Second, we explore the different neural coding schemes ranging from rate to temporal codings.

Rate-based coding

A very important part of SNNs is the neural information encoding or the spikes generation scheme. In this section, we compare three different rate-based coding techniques that are presented in section 4.4: Poissonian, Jittered Periodic and Periodic. In this comparison experiment, we focus on the accuracy and the number of spikes propagated over the network using different

SNN topologies. Accuracy results are illustrated in table 4.6 and spiking activity results showing the amount of spikes propagated over the some SNNs are listed in table 4.7.

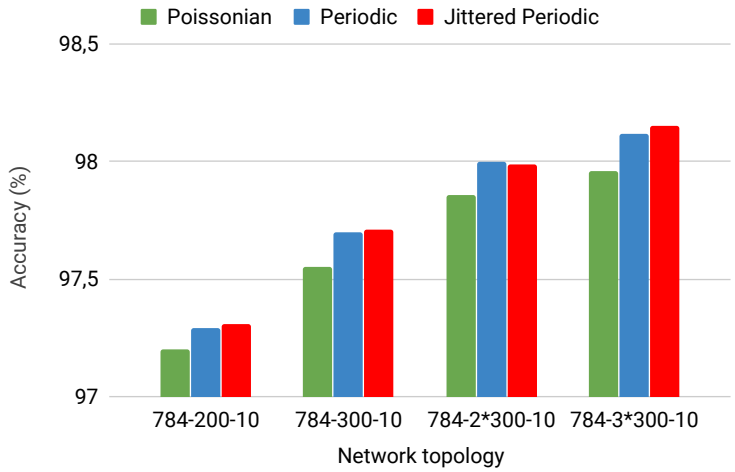
TABLE 4.6: Rate-based coding versus accuracy - MNIST.

SNN topology	Accuracy (%)		
	Poissonian	**Periodic**	**Jittered Periodic**
784-100-10	96.29	**96.31**	96.30
784-200-10	97.20	**97.31**	97.29
784-300-10	97.55	**97.71**	97.70
784-300-300-10	97.86	97.99	**98.00**
784-300-300-300-10	97.96	**98.11**	**98.11**

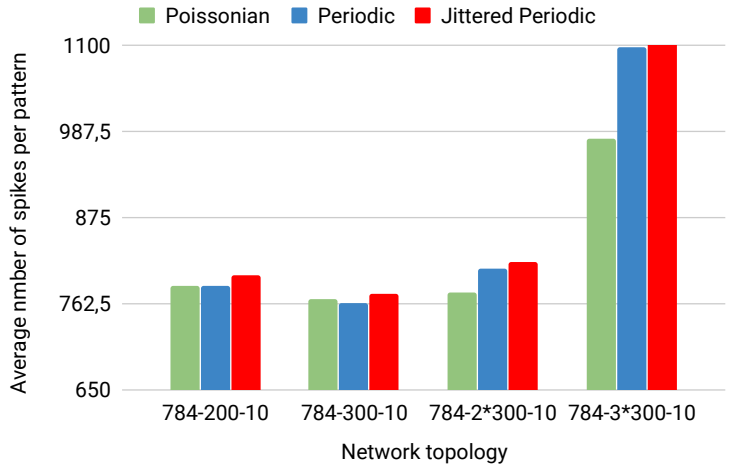
TABLE 4.7: Rate-based coding versus spiking activity - MNIST.

SNN topology	Spikes / pattern		
	Poissonian	**Periodic**	**Jittered Periodic**
784-100-10	845.22	865.43	**754.92**
784-200-10	**785.89**	786.70	799.63
784-300-10	768.03	**763.14**	774.76
784-300-300-10	**776.66**	808.06	816.26
784-300-300-300-10	**978.70**	1096.64	1106.24

Based on the data presented in these tables, we have plotted histograms showing the influence of the coding schemes on accuracy and spiking data flow. First, figure 4.11a shows the impact on accuracy for different SNN topologies and second, the figure 4.11b shows the influence on the average number of spikes propagated over the SNN for an MNIST pattern. On one hand, for most of the SNN topologies, Jittered Periodic and Periodic techniques perform the highest accuracy results while generating approximately similar number of spikes per pattern to Poissonian coding. On the other hand, the Poissonian method, while being the most competitive in terms of spiking activity, always performs the lowest recognition scores. Note that, the Periodic coding technique is performing almost similar results to Jittered Periodic, but, due to its periodicity, it is less robust to noise. Indeed, with Jittered Periodic, a white Gaussian noise is injected to the signal generated for the input data, thus making it more robust to noise than the Periodic method. Thus, for the remaining parts of this neural coding exploration and for the design of SNN's hardware architectures, we will adopt Jittered Periodic.



(A) Accuracy



(B) Number of spikes

FIGURE 4.11: Rate-based coding – MNIST.

Exploring novel neural coding schemes

In this step, we compare the explored coding techniques with Jittered Periodic by analyzing accuracy and spiking data flow. In this context, we have recorded, in tables 4.8 and 4.9, accuracy results of different CNNs using Jittered Periodic, Spike Select, First Spike and analog coding techniques on both MNIST and GTSRB data-sets. From these data, we notice that the First Spike method performs the lowest accuracy results on MNIST data-set, with more than 9% accuracy loss using Conv29 and LeNet CNNs. Moreover, a huge loss is recorded when applied to the GTSRB dataset, with scores varying from 21.57% to 68.61%. This method shows a competitive spiking activity compared to others due to the fact that it uses only one spike to encode an input value. However, the important accuracy losses makes it not viable for classification applications. A possible solution to improve the accuracy with

First Spike is the use of a dedicated learning algorithm that is adapted to "at most one spike per neuron" like in (Kheradpisheh and Masquelier, 2020).

TABLE 4.8: Spike generation versus accuracy – MNIST dataset.

Coding method	CNN model			
	Conv29	**ConvPool**	**LeNet**	**ConvFc**
Jittered Period	99.16	**99.21**	99.15	**99.09**
Spike Select	**99.21**	97.07	99.13	98.68
First Spike	87.21	96.56	90.87	96.16
Analog domain	99.16	99.18	**99.18**	**99.09**

TABLE 4.9: Spike generation versus accuracy – GTSRB dataset.

Coding method	CNN model			
	Conv-1C	**Conv-3C**	**LeNet-1C**	**LeNet-3C**
Jittered Period	96.68	97.75	95.76	96.19
Spike Select	96.56	97.83	95.72	96.18
First Spike	21.57	45.49	25.46	68.61
Analog domain	**96.8**	**97.86**	**95.84**	**96.25**

This accuracy difference is more clearly shown on figure 4.12. For GTSRB data-set, the bars representing First Spike method are too small compared to the bars representing the other coding techniques. Concerning the other coding schemes, we observe that Jittered Periodic and Spike Select methods are more accurate and show equivalent records to the analog CNNs. For example, applying the "Conv-1C" CNN on GTSRB using Jittered Periodic and Spike Select results, respectively, in 96.68% and 96.56% accuracy which are roughly similar to the analog's accuracy that performs 96.80%. All the networks perform almost similar scores on both MNIST and GTSRB datasets using Spike Select and Jittered Periodic coding schemes. Therefore, in the following part, we will compare these two methods in terms of spiking activity and prediction accuracy.

Rate-based coding versus hybrid coding

In this section, we compare rate-based coding to a hybrid coding, represented by Spike Select and Jittered Periodic. This experiment is done because results obtained so far show that these methods are the ones that lead to the highest accuracy on both datasets. In this experiment, we have applied the networks LeNet-1C, LeNet-3C and FcNet2 to MNIST and GTSRB datasets and then recorded the obtained results in terms of accuracy and spiking activity.

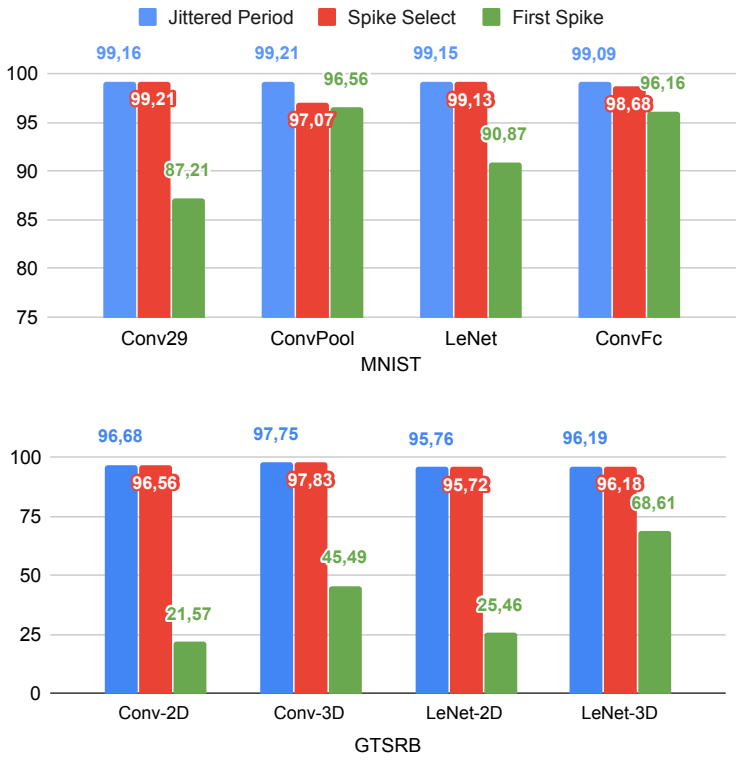


FIGURE 4.12: Spike generation versus accuracy - MNIST & GTSRB.

First in table 4.10, we show the spiking distribution over the layers of a fully-connected based SNN FcNet2 and its accuracy on MNIST dataset using the two coding schemes. With Spike Select coding scheme, FcNet2 reaches an accuracy of 97.87% on MNIST data, which is very close to the Jittered Periodic one (98.24%). However, the amount of spikes generated over the deeper layers of this SNN, from FC1 to Output, is much smaller using Spike Select than Jittered Periodic.

TABLE 4.10: Neural coding methods versus spiking data distribution on MNIST. Network : "FcNet2 : 784-3x(300)-10".

Layer	**Spikes per pattern**	
	Jittered Periodic	**Spike Select**
Input	724	1547
FC1	173	74,5
FC2	103,5	35
FC3	39	4
Output	4	1
Total	1043.5	1661.5
Accuracy %	**98.24**	97.87

TABLE 4.11: Spike Select and Jittered Periodic versus total spiking data – MNIST.

Network model	**Conv29**	**ConvPool**	**LeNet**	**ConvFc**
Jittered Periodic	15153	8367	5359	1097
Spike Select	12281	1397	1698	380

TABLE 4.12: Spike Select and Jittered Periodic versus total spiking data – GTSRB.

Network model	**Conv-3D**	**Conv-1D**	**LeNet-3D**	**LeNet-1D**
Jittered Periodic	102089	98036	35027	27066
Spike Select	39526	34975	18075	11563

Tables 4.11 and 4.12 show the average total number of spikes generated for processing a single pattern of four different spiking CNNs when applied to GTSRB dataset. Using the data on these tables, we have plotted histograms, shown in figure 4.13, that represent this average total number of spikes with both Spike Select and Jittered Periodic coding schemes. The data on these illustrations confirm that Spike Select, represented with red bars, is more efficient than Jittered Periodic by generating less spikes for all the 4 topologies. For example with GTSRB dataset (GTSRB on figure 4.13), Spike Select is generating around 50% less spikes than Jittered Periodic for all the spiking ConvNets while keeping approximately similar recognition rates.

TABLE 4.13: Spike Select and Jittered Periodic versus spiking data distribution – GTSRB.

Layer	LeNet-1C		LeNet-3C	
	Jittered Periodic	Spike Select	Jittered Period	Spike Select
Input	3295	3110	7692	6698
Conv1	12581	3028	15048	3553
Pool1	4160	1046	4956	1207
Conv2	4585	1463	4799	1542
Pool2	2209	723	2346	762
Conv3	115	37	91	29
Fc1	77	23	67	20
Output	44	8	28	5
Accuracy %	95.76	95.72	96.19	96.18
Total	27066	**9438**	35027	**13816**

In order to confirm the efficiency of spike select, we have performed statistical measurements to show the distribution of the spiking data over the different layers of the network. To do so, we have applied two CNNs on the GTSRB dataset. The first network is a mono-channel CNN (LeNet-1C)

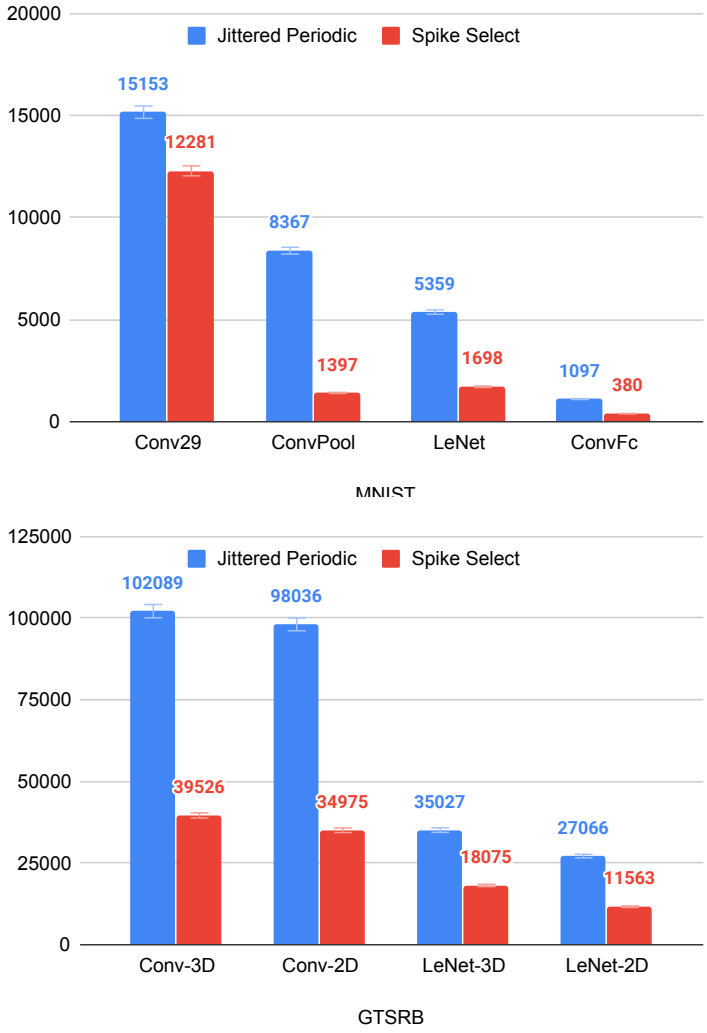


FIGURE 4.13: Spike Select and Jittered Periodic versus total spiking data.

that is processing the "gray-level" version of GTSRB and the other is a "three-channel" CNN addressing the colored (RGB) version of GTSRB. This statistical results, presented in table 4.13, are then used to plot two column charts, shown in figure 4.14, showing the amount of spikes in each layer of the two LeNet CNNs. These representations confirm that Spike Select is more promising than Jittered Periodic, where fewer spikes are generated on all the layers of both LeNet networks. Moreover, we observe a regulation of spiking distribution over the CNN's layers when applying Spike Select, by reducing the huge amount of spikes present in the first layers present with Jittered Periodic. In addition, we notice a non uniform spikes distribution with Jittered Periodic whereas when using Spike Select the number of spikes decreases continuously from one layer to another.

FIGURE 4.14: Spike Select and Jittered Periodic versus spiking data distribution – GTSRB.

4.6 Discussions

The objective of neural coding exploration is to find the most efficient coding schemes that minimize the spiking activity while performing the state-of-the-art accuracy on known benchmarks. Therefore, we have first analyzed rate-based coding that is the most used coding technique with SNNs to perform classification and which is rising to the state-of-the-art accuracy. Thus,

we have set this coding as the reference in terms of accuracy. Since the objective is reducing spiking activity then naturally the best approach would be the use of time-based coding which is known to represent information with much fewer spikes defined by a precise timing. However, this coding approach rises lower accuracy than the state of the art one because of this spike's amount reduction. Therefore, we have set time-coding as the reference in terms of spiking activity but not accuracy. Afterwards, we have used these as references and explored other different coding techniques: Jittered Periodic, Spike Select, First Spike and Single Burst. The results have shown that Single Burst and First Spike methods are not adequate to spiking CNNs because they present important accuracy losses compared to analog CNNs. Therefore, we have selected Spike Select and Jittered Periodic as potential solutions for spike generation technique with SNNs. In this way, we have compared their accuracy records and spiking activities on two data-sets (MNIST and GTSRB) using different topologies. In terms of recognition rate, both methods perform results equivalent to analog CNNs scores. Whereas, in terms of spiking activity, with Spike Select method we are able to regulate the spiking distribution over the network to get a more sparse spiking data activity, where the number of spikes decreases when going deeper into the network. Moreover, the overall amount of spikes is drastically reduced using this method. Therefore, these findings make Spike Select a promising solution for embedded AI-application when dealing with spiking CNNs. In addition, combining this coding scheme with the hybrid architecture (cf. chapter 3) would offer a better latency-resources trade-off.

From this perspective, figure 4.15 shows how the use of the spike select method would be adequate to the hybrid architecture, which involves both parallel and multiplexed computings. Indeed, as mentioned in chapter 3, the first layer is implemented in a massively parallel fashion and with multiplexed hardware for the remaining layers. This structure fits well the spikes distribution when using Spike Select method. Because, the parallel part of the architecture will process for the first layers where most of the spikes are generated. In the remaining layers, there are only a few spikes which will be processed by the multiplexed part of the architecture.

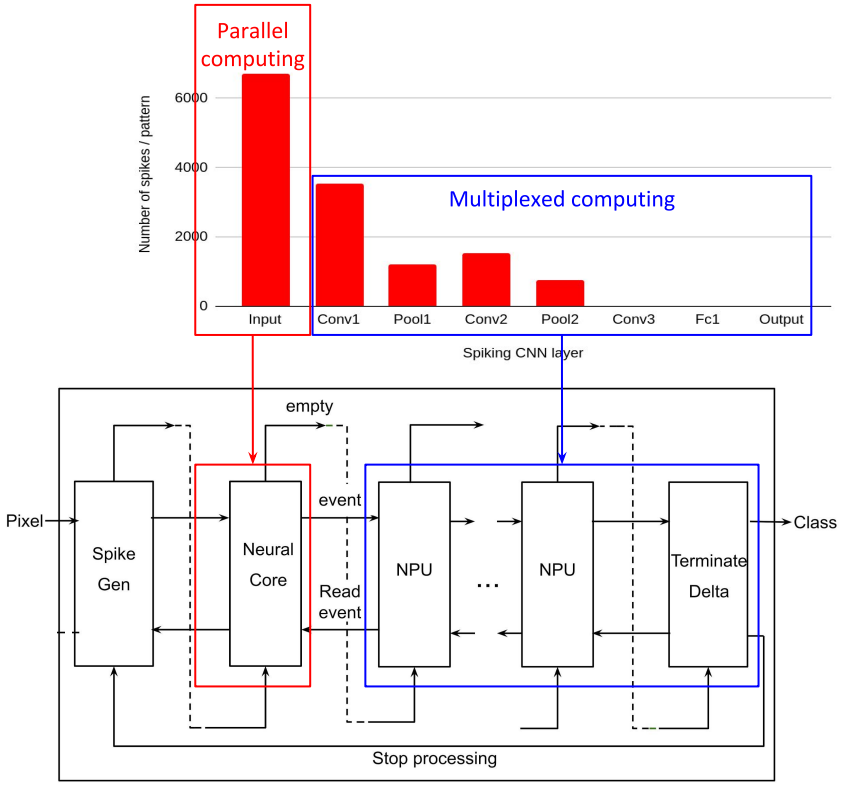


FIGURE 4.15: Spike Select and Hybrid Architecture for embedded hardware classification.

4.7 Conclusion

In this chapter, we have explored different neural coding schemes for spike generation with SNNs. The objective is to select an efficient neural coding technique that is adequate with low-power embedded systems. The coding scheme must perform, in classification tasks, the state-of-the-art accuracy while minimizing the amount of spiking events. To this end, we have set rate-based coding and temporal-based coding as references. First, rate-based coding is the most commonly used coding scheme with SNNs that ensures state-of-the-art accuracy performances but at the same time it induces the use of a huge amount of spiking data. Second, temporal-based coding is set as a reference for spiking activity because it is characterized by the use of a smaller amount of events to encode data. Second, we set time-based as the second for spiking activity for energy efficiency even though, it does not achieve the state-of-the-art accuracy level. These two references are thus used to explore other models that aim at meeting both activity and precision requirements.

In this context, we have first explored the rate-based coding methods: Poissonian, Periodic and Jittered Periodic. These coding techniques resulted in similar performances to those obtained by analog neural networks in terms of recognition rate, where they reach up to more than 98% on MNIST and 97% on GTSRB for some spiking CNN topologies. Afterwards, we evaluated other forms of coding schemes ranging from rate-based to temporal-based coding paradigms. This study shows that the most suitable neural coding paradigm was the novel Spike Select coding, as it ensures high prediction accuracy and sparse spiking activity in the network. Spiking data sparsity implies a lower number of spikes per pattern, resulting in a shorter processing and a lower energy consumption, which is suitable for embedded system applications. Indeed, the use of this method drastically reduces the network's activity. Moreover, we noticed a regulation of the spiking data distribution, where most spikes are generated in the input layer and few of them propagate in the remaining ones. This spiking distribution would benefit from the hybrid architecture that has a fully-parallel input layer and multiplexed deeper layers; this architecture will be described in more detail in the next chapter. Therefore, from this chapter we conclude that spike generation methods influence the global activity of the SNN and may reduce the energy consumption of the neuromorphic chip. We have seen also that SNNs are characterized by a very sparse data over its layers and in deep networks most of the spikes are present in the first layers but only few are present in the deeper ones. Therefore, two architectural aspects would benefit from this sparsity and which are: first, the event-based computing which will allow the processing of only spiking events to reduce the processing time; second, adaptation of parallelism and multiplexing of computations within the architecture to the spiking data distribution like it is done with the hybrid architecture.

In the next chapter, we discuss the register transfer level design of SNNs on FPGA using various event-based architectural models adopting different levels of parallel and multiplexing computing.

Chapter 5

RTL exploration of neuromorphic architectures

5.1 Introduction

As mentioned earlier, there are a lot of applications requiring embedded implementation of neural networks such as smart devices or autonomous vehicles. Their classical implementation on CPU/GPU based systems is still too expensive for an embedded context, because their computational model is inadequate to the hardware target. To overcome this limitation, it is necessary to design dedicated neuromorphic accelerators that fit the parallel and distributed computation paradigm of neural networks. Moreover, an adequate neural model would be SNNs that, in addition to their greater ease of integration onto neuromorphic hardware, naturally take advantage of event-driven computation. This paradigm assumes that only events are processed, reducing the global chip activity to the units triggered by the flow of events. Therefore in this chapter, we explore different architectural models in a Register Transfer Level, i.e., a hardware implementation on FPGA, for designing neuromorphic accelerators for SNNs inference. Thus, the objective of this chapter is the exploration of different architectural models for implementing SNNs. To this end, we consider only fully-connected deep SNNs and assume that similar architectural models can be adapted to spiking CNNs. Hence, in the next chapter we will use the results obtained in this chapter to efficiently implement the spiking CNNs. Indeed, an RTL design of such architectures rises to accurate estimation results serving as a quantification that can be used to discriminate different architectural models. In this exploration we first implement simple neural models rising from machine-learning and neuroscience and second study and implement deep SNNs with different

architectural models. The different hardware models are: Fully-Parallel Architecture, Time-Multiplexed Architecture and Hybrid Architecture. The architectural models are the result of the previous steps of the design space exploration framework.

5.2 Preliminary SNN to ANN confrontation

In this section, we present a preliminary comparative study that is being conducted to compare SNNs to ANNs. This study comprises a hardware implementation of two simple neural networks stemming from two families: an ANN from machine learning and an SNN from neuroscience. The ANN and SNN models are feedforward neural networks composed of one hidden layer of 300 neurons, an input layer of 784 neurons and an output layer of 10 neurons. In chapter 4, these models have been already applied to MNIST dataset, where the ANN accuracy was 97.85% and the SNN accuracy was 97.74% (FcNet1 on the table 4.3). For this preliminary comparative study, we have used a very simple topology in order to facilitate the implementation. The goal of this experiment is to provide a preliminary estimation of the potential hardware cost gain that SNNs could have, which will drive further implementations of more complex topologies. Both ANN and SNN have been implemented using a similar structure but with different neuron models. The integrate-and-fire neuron model is used with the SNN and the perceptron is used with the ANN, these neuron models are described in chapter 2. The ANN and SNN architectures have the structure that is illustrated in figure 5.1, it is composed of three parts representing the different layers of the network topology. The first part represents the input layer and is temporally multiplexed using a single neuron that is simply buffering input data (spikes for the SNNs and pixels for the ANN) to the neurons of the hidden layer. The second part is dedicated to perform the processing of the hidden layer neurons, where each neuron is represented by a hardware *Neuroni* module. Similarly, the third part is composed of hardware *Neuronj* units that represent the network's output layer. Two counters are used to indicate the address of the input spike/activity for the hardware neurons, as shown in figure 5.1. The address given by one of these counters is used by the hardware neurons to retrieve the synaptic weight corresponding to the input spike/activity. Note that spikes are communicated between SNN neurons, while activities are communicated between ANN neurons. Output spikes/activities of the hidden layer neurons are processed, one after the

other, by the output neurons. This is done by using a multiplexer that selects the spikes/activities, one after the other, according to the address given by the output counter.

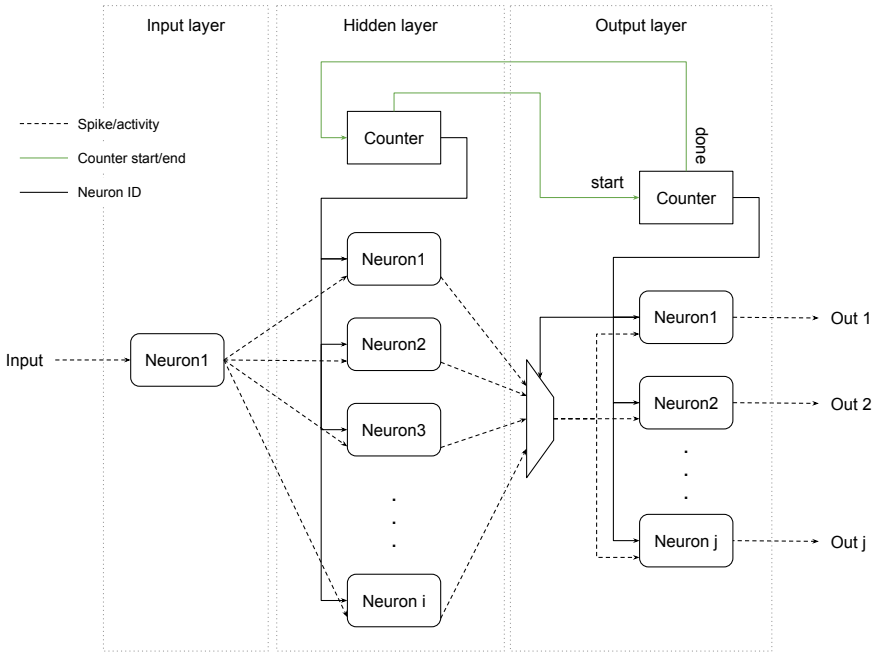


FIGURE 5.1: Input-time-multiplexed SNN's architecture.

Comparison results

These architectures are then implemented both on FPGA and ASIC using VHDL hardware description language. For the FPGA implementation, we used Altera FPGA Cyclone V: 5CGXFC9E7F35C8 device that gives a good flexibility in terms of resources (ALMs, memory bits and DSP blocks). For the ASIC implementation, we targeted a CMOS 65nm technology to implement our neural networks and we compiled our designs using Synopsys Design Compiler.

The obtained comparison results on FPGA and ASIC in terms of resource occupation and power consumption combined with the accuracy of both the ANN and the SNN are summarized in table 5.1.

In terms of accuracy, as it has been shown in chapter 4, the SNN achieves practically the same performance as the ANN. In terms of hardware cost, first the FPGA implementation shows important gains for the SNN in terms of logic utilization (43.46%) due to the extra ALMs used for the activation function in the MLP, total pins (87.62%) due to the information coding and DSP blocks (100.00%) due to the multipliers that are used for the MLP only,

TABLE 5.1: SNN vs. ANN comparison results on the topology : 784-300-10. The ASIC results correspond to a CMOS 65nm technology obtained using Synopsys Design Compiler. The FPGA device is an Intel Cyclone V (5CGXFC9E7F35C8).

Neural model		MLP	SNN	SNN Gain (%)
Accuracy		97.85	97.74	-0.11
Events/Synapse/Pattern		1	0.26	**73.20**
FPGA	Logic (ALMs)	61809	34950	**43.46**
	# registers	6275	6801	-8.38
	Max # neurons	342	1007	194.45
	DSP blocks	310	0	100
	Dynamic power (mW)	12.03	4.95	**58.86**
	Total power (mW)	538.78	530.21	1.60
ASIC	Total cell area (mm)	1.88	0.86	**53.98**
	Cell internal power (mW)	0.75	0.40	45.91
	Net switching power (mW)	0.14	0.026	81.62
	Dynamic power (mW)	0.88	0.42	51.63
	Cell leakage power (mW)	51.22	28.04	45.26
	Total power (mW)	52.11	28.46	**45.37**

with a small loss in the total registers (-08.40%), due to the stored spike signal in the SNN. In terms of power consumption, the SNN is more efficient in dynamic power (58.86%), even if this gain is irrelevant in the total consumption of the FPGA device (01.60%). Second, the ASIC implementation shows a clear advantage for the SNN, as it is more efficient in terms of number of ports (87.62%) and total cell area (53.98%). These results are coherent since they reflect the SNN gains in FPGA resources. In terms of power consumption, the SNN is more efficient in total dynamic power (51.63%), that is approximately the same result found in FPGA. However, the SNN is also more efficient in cell leakage power (45.26%) as it is proportional to the used area, and therefore more efficient in terms of total power (45,37%).

TABLE 5.2: Area gain compared to the architecture in Du et al., 2015 study. The network topology is 784-300-10.

Network	ANN		SNN	
	This work	Du et al., 2015	This work	Du et al., 2015
Learning	Back-propagation		ANN-SNN conversion	STDP
Area (mm^2)	01.89	79.63	00.87	38.89
Area gain	97.62%		97.76%	

Now, the obtained results are compared to those of networks in the comparative study conducted in (Du et al., 2015). Approximately the area gain of 50% is obtained using similar precision. However, there is a very large difference

in terms of SNN energy gain. In (Du et al., 2015), the ANN has an energy gain of 96.38%. In this study, the SNN's total energy gain is 45.37%, which fits the theoretical study on the computational complexity of the two models, where the spiking neuron performs a simpler computation based on spiking events. Therefore, it is not the spike-based coding that penalizes the SNN in the study conducted by Du et al., but the fact that they have implemented an online STDP learning. When implementing the ANN and the SNN on the same target technology (CMOS 65nm) with Synopsys Design Compiler, we have an area gain of about 97% compared to (Du et al., 2015), as shown in table 5.2.

5.3 Complete hardware architecture overview

The designed architectures in this thesis are to infer SNNs are implemented in a modular structure based on a first-in first-out memory interface and an event-based communication protocol. These architectures are modular, scalable and adaptable to the user-defined application because there are as many hardware modules as the defined SNN has layers. The architecture is composed of neural processing units, a spike generation module and class select module. We have designed three types of NPUs: convolutional, pooling and fully-connected. In this chapter, only fully-connected layers are used, the pooling and convolutional layers will be used in the next chapter.

Within these architectures, each layer is represented by a mean of one of these processing units depending on its type (convolutional, pooling or fully-connected), refer to figure 5.2. Indeed, each NPU is processing spiking events in a sequential fashion and whenever it has an output event it is stored in its FiFo memory for the next layer's NPU. Doing so, many computations could be held concurrently with this event-based architecture. This pipeline computational aspect, enabled by the use of the event-based communication protocol, when combined with the Terminate Delta classification policy accelerates the computations and reduces the processing time. Due to the sparsity of the spiking data in SNNs, the Terminate Delta module enacts the classification by only receiving a few spiking events. Therefore, ensuring the path of spikes from input to output of the SNN would benefit from the temporal nature of spiking data and activate the terminate-delta earlier. Indeed, if enough spikes reach the output layer before receiving the whole input stimulus, the classification module can designate the winner class and stop the

processing in the architecture. In this case, which is so often with SNNs, the processing time is drastically reduced because fewer events are processed before acting the classification. In this context and in order to benefit from this temporal and sparsity aspect of SNNs, we have implemented this hardware architecture using one processing module per layer to ensure the path of the important spikes (first spikes) from input to output. In the remaining parts of the chapter, the different modules of the architecture are described and then some hardware cost quantification results are presented and discussed.

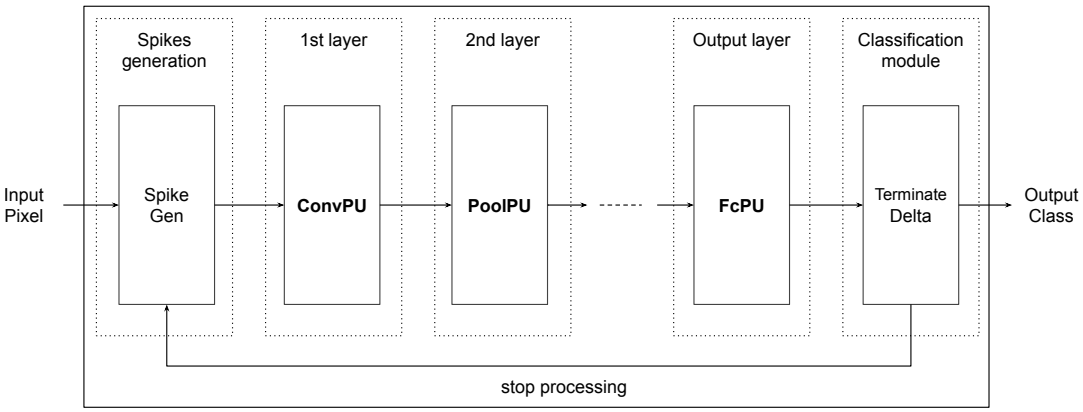


FIGURE 5.2: Schematic diagram of proposed hardware architectures.

5.4 Event-based communication protocol

As mentioned previously, the implemented architectures are implemented in an event-based structure. This is possible by the use of FiFo memories that handle the flow of the spiking data through the different modules constituting the neural system. The spiking data flowing through this communication protocol are events that indicate the location of neurons that emit spikes in the network. Due to the feed-forward characteristic of the SNNs, an event represents only the location of its emitting neuron inside the layer without indicating the source and destination layers. The events are not assigned a timestamp because of three reasons: first, since we are using FiFo modules, the arriving order of spikes from one layer is respected; second, since each layer is represented by a processing unit that uses an output FiFo module then its output spikes are ordered, and since the layers are connected to each other in a feedforward manner, then the global spikes of the hole network are ordered; finally, what matters with Integrate-and-Fire neurons is the arriving

order of spikes but not their exact time (similarly as in (Thorpe and Gautrais, 1998)).

Input / Output ports description

Actually, the different components of the system representing the spike generation module, the classification module and the different SCNN layer processing unit modules have the same input and output interfaces that are based on FiFo memory I/O ports. A typical processing unit module of the SCNN system is illustrated in figure 5.3, it has three input ports and three outputs.

— Previous layer ports:

1. i_Event : this is the port that is used to communicate the addresses of the previous layer's neurons that emitted spiking events. The size of this 'i_Event' input port depends on the output Feature Map (FM) (or number of neurons for FC layers) size of the previous layer.
2. o_Rd_Event : this output signal indicates to the previous layer an event has been read from its FiFo and therefore it has to remove it from its memory.
3. i_Empty : this input indicates the presence of input spiking events coming from the previous layer. This means that if this signal is true (equals to '1'), the previous layer's FiFo is empty and thus there is no input event for processing.

— Next layer ports:

1. o_Event : this output port is used to communicate the addresses of the current layer's neurons that emitted spiking events. The size of this port depends on the output Feature Map (FM) (or number of neurons for FC layers) size of the present layer.
2. i_Rd_Event : this input port is used by the next layer to inform the current layer that it has read an event from the FiFo and therefore it has to be removed.
3. o_Empty : this indicates the presence of output spikes in the current layer's FiFo memory.

Note that, in addition to these I/O ports, a clock and reset inputs are used.

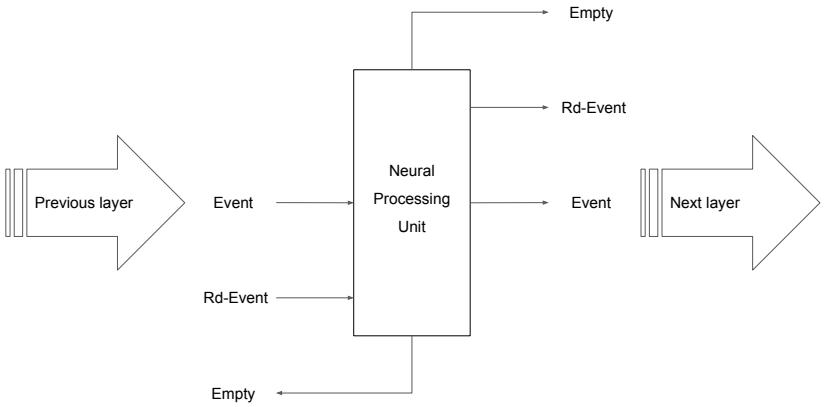


FIGURE 5.3: FiFo-based communication system used to link the architecture layers.

Communication protocol description

After defining the I/O ports of the architecture modules, let's describe the event-based communication protocol used to propagate spiking events between the different units from input to output of the SNN.

First, the architecture component checks the presence of input spiking events for processing by looking at 'Empty' input. Indeed, this empty input is high when there is no event to process in the previous layer's FiFo. Therefore, in this case (Empty='0'), the component does nothing and stays in an idle state. In the other case, empty is low (Empty='1'), the component processes the event by reading the input 'Event'. After reading the event, the output 'Rd-Event' is set to high to indicate to the previous layer that the current event is read. Second, when the component has an output event that is saved in its FiFo it sets the output 'Empty' to low and the output 'Event' points to the first event in the FiFo. When the next layer reads an event from the FiFo, it sets the input 'Rd_Event'. In this case (Rd_Event = '1'), the first event in the FiFo is removed and the component's 'Empty' and 'Event' outputs are updated. Note that, the 'Full' signal of the FiFo is internal to the component and is influencing the readings and writings of input/output events, refer to section 6.2.1.

We have used this communication protocol for its ease of integration in hardware, to easily make the architecture modular, scalable and thus generic (able to change the number of layers and neurons easily) and finally to take advantage of the event-driven property of spiking neural networks.

5.5 Deep SNNs hardware implementation

In this section, we describe the implemented SNN architectural models on FPGA for exploration purpose and which are: Fully-Parallel Architecture (FPA), Time-Multiplexed Architecture (TMA), and the Hybrid Architecture (HA). We have selected those architectural paradigms according to NAXT results, which shows how those three architectures are adequate to evaluate the trade-off between power consumption, resource intensiveness and latency. To do so, we first describe the elementary modules that are used in the different designs, then we present the complete architectures. As mentioned in chapter 3, we use N2D2 to extract the different parameters of SNNs then move to the hardware implementation of the architectures. This phase is realized with the Intel® Quartus® Prime 18.1.0 Lite edition for FPGA prototyping, and ModelSim® for the validation with simulation of the design behavior.

5.5.1 Elementary hardware modules

Here we present the elementary components used to construct the different SNN's architectures. The modules are: a counter used for synchronization, a FiFo memory to store spiking events, a ROM to save the synaptic weight parameters, a spike generation module to generate spikes and a classification module.

Counter

The counter modules are used for synchronizing the communication between neurons of different layers. On one hand, they are ordering the start and end of computations by the neurons and indicate the synaptic connection addresses corresponding to input spikes (in TMA architectures). On the other hand, they are ensuring a coherent flow of spikes in the network to synchronize the different layers, for more details refer to their usage in FPA architecture (figure 5.11).

First-in First-out memory

The First-in First-out (FiFo) memory modules are used in the different hardware architectures to serve as buffers between the different processing modules representing layers or neurons. These FiFos are used to ensure coherent data flow in the SNN in an event-based processing way, where the neurons

output spikes are interpreted as spiking events and, according to their arrival dates, they are sorted in an ascending order. Doing so, the spikes are processed by the different neurons in the correct order. Figure 5.4 illustrates a schematic diagram representation of the designed module showing its I/O ports. Indeed, the input and output data correspond to neurons addresses (origin of the received spikes). Due to the huge number of weights, the spikes are stored in this event-based format for facilitating the access of the related weights in the next layer.

Read Only Memory

For the neuromorphic system proper operation, memory blocks are needed. In FPGA devices, there different memory types: ROM, RAM, latches or registers. The ROM are used to store the SNN logical neurons' synaptic weights, where one ROM block is used for each layer, as concluded in chapter 3. Therefore, different ROMs of different sizes are used which depend on the amount of emulated synaptic weights. The ROM block I/O ports are illustrated in figure 5.4.

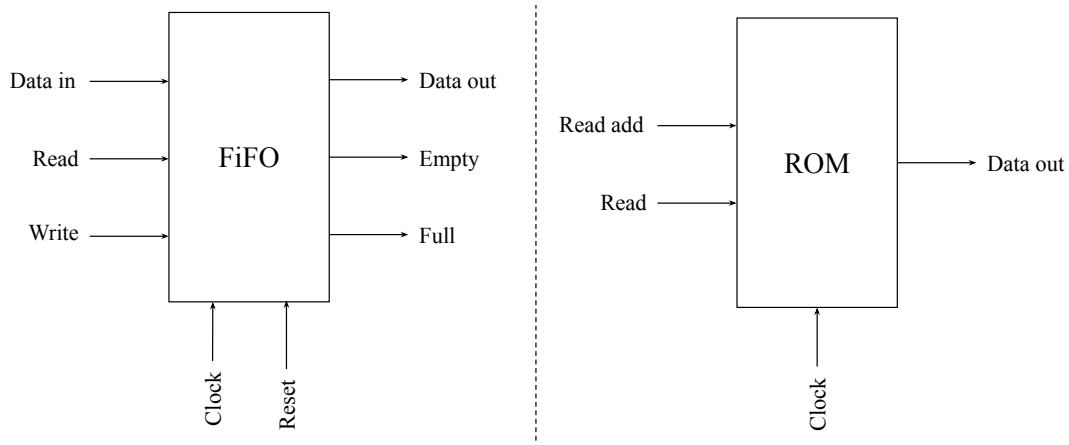


FIGURE 5.4: I/O ports of FiFo and ROM memory blocks.

Spike generation

The input data received by the SNN architecture, in case of non-event-based sensors, is in analog domain. Therefore, a spike generator must be used to transform these analog data to spiking domain before its processing by the SNN architecture. Indeed, some of the different neural coding techniques presented in chapter 4 are implemented through the 'Spike Gen' hardware module. Till now, the coding method implemented through this module was

Jittered Periodic. To get the other coding methods, some adjustments are made:

— Periodic: use the periods as the emission times of the spikes;
— First spike: generate only the first spike per pixel;
— Spike select: keep the same spikes generation methodology, modify the first layer's threshold and reduce the Terminate Delta value.

The spikes generation module is designed to reduce computation cost as much as possible: the input values **v** being 8-bit coded (to pixel's intensity ranging from 0 to 255) and the values of f_{max} and f_{min} being constant, the frequency encoding function can be implemented as a simple Look-Up-Table of 256 cases. This procedure helps save logic and energy to reduce the cost of the transcoding module. Within this module a counter is used to represent the time in hardware and a FiFo module is used to memorize the output spikes as events. Spike emission times 't_{spike}' are computed from the input pixels and whenever a spike emission time 't_{spike}' is equal to the time displayed by the counter, an event holding the address of the corresponding pixel is saved in the output FiFo. The SNN accelerator reads the input spikes through this FiFo module.

SNN class selection modules

During inference, the winning class is selected from one of the output neurons using either Terminate Delta or Max Terminat. In the Terminate Delta procedure, the classification is done when a neuron has spiked delta times more than all the other neurons. On other hand, in Max Terminate, the classification process is enacted whenever an output neuron has fired max-value spikes.

These classification modules are implemented in hardware using the structures shown in figure 5.5. The input of the module is a vector of activations containing the state of all the output neurons of the SNN. These activations are used to update the number of spiking times of the neurons and which is then used to verify the classification. First, within the Terminate Delta two maximum operation blocks are used to detect the maximum value of the array holding this number of firings, which are then used to determine the winner class and to stops the processing. The first maximum block, namely Max1, detects the most spiking neuron having the highest number of spikes maximum, and the second, namely Max2, detects the second most

spiking neuron. These Max blocks output also the number of spikes emitted by both the neurons. Afterwards, a difference between these numbers of spikes is computed and compared to the delta-value. Finally, if the difference is greater than this delta-value, the class given by Max1 Module is enacted as the winner. Otherwise, the same procedure is repeated until a class is selected as a winner, more detailed description is given in chapter 4.

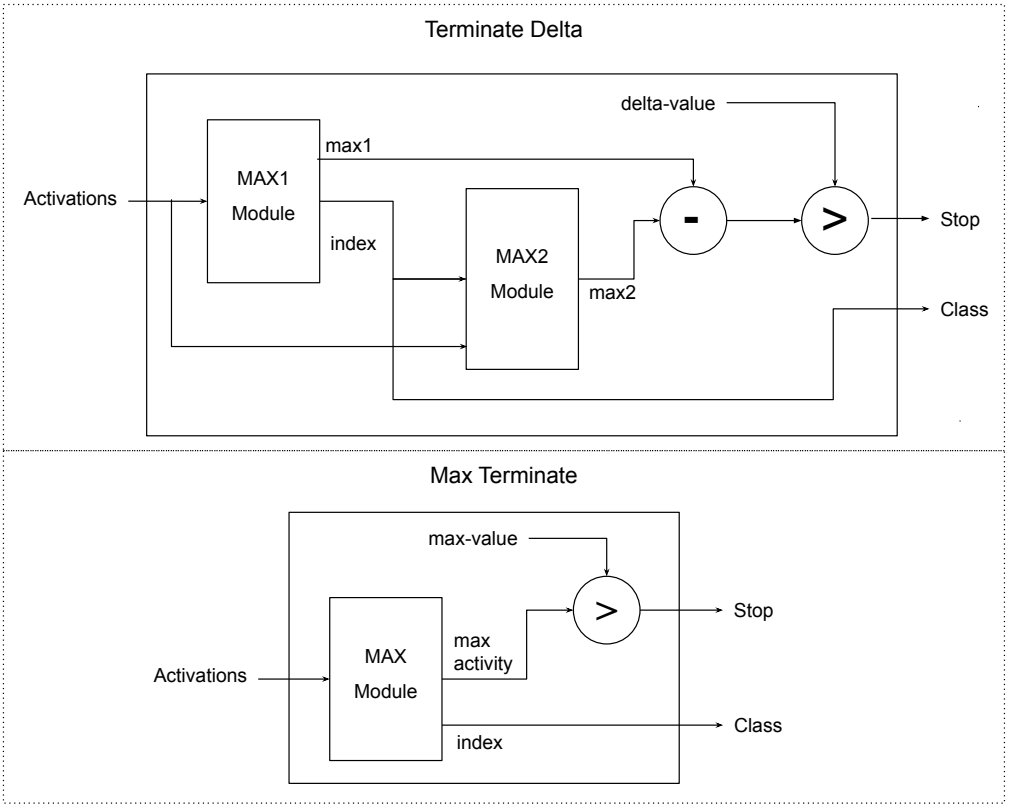


FIGURE 5.5: Schematic diagram of the classification modules.

On the other hand, the Max Terminate module integrates only one maximum operation block that returns the address of the output neuron with the highest number of spikes. Then its activity is compared to a user-defined max-value. If the maximum spiking activity is greater than this max-value, the corresponding output neuron is enacted as the winner class, and the processing is stopped.

Integrate-and-Fire neuron

The IF-neuron hardware structure is illustrated in the simplified schematic diagram presented in figure 5.6. In contrast to the perceptron, it does not have a multiplier and thus results in cheaper hardware with only elementary

components. The module has two inputs: the input spike and its corresponding synaptic weight; and one output port for output events.

When the neuron receives an input spike, its corresponding weight is accumulated with the last internal potential update stored in a register. Afterwards, this accumulated potential is compared to a potential threshold to fire whenever it is exceeded. In a firing case, the internal potential is decreased by the threshold, otherwise, the internal potential is not modified.

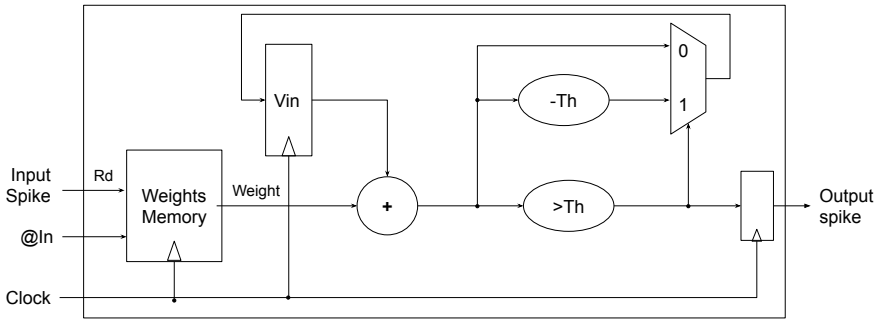


FIGURE 5.6: IF neuron module's internal structure.

Neural Processing Unit module

The Neural Processing Unit (NPU) module is used to emulate time-multiplexed layers. A single IF Neuron module will operate successively for all neurons in the layer. Moreover, the NPU includes a counter module, a FiFo Memory module and an NPU controller. These modules are used to build an NPU that processes coherently spiking event, refer to figure 5.7.

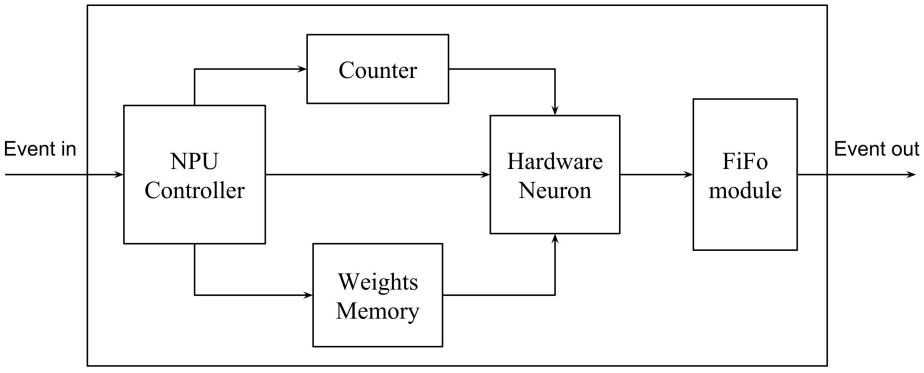


FIGURE 5.7: Neural Processing Unit simplified block diagram.

Figure 5.8 is a flow chart of the steps for operating the Neural Processing Unit module. The process begins by a first step of loading an input event,

empty input signal and the stop processing signal. Whenever the Terminate Delta module activates the stop processing signal, the process is ended. Otherwise, the NPU checks the presence of input events by verifying the state of the empty input 'i_Empty' signal. Then, depending on the layer type (i.e. a fully-connected layer or a convolutional layer), the addresses of the logical neurons are forwarded to the hardware neuron to retrieve internal activities and weights to perform the integrate-and-fire rule. Note that the 'map()' function present on this flow chart will be described in the next chapter (6.2.1). The address of the computed logical neuron is saved in the FiFo buffer if its output spike is high (Spike=1). This process, controlled by a counter, is repeated for all the logical neurons of the layer. Once all these neurons are processed, new inputs are loaded to compute for the next input spiking event.

Neural Core module

The Neural Core module, shown in figure 5.9, is a computation unit which emulates two layers (an input layer and a first hidden layer). The input layer comprises an Input Neuron module which forwards 'input events', i.e. a flow of spikes, to downstream neuron circuits which are implemented on a parallel structure allowing to process in parallel the spiking events of input layer to filter the number of spikes and generate a reduced number of spikes. Each logical neuron is represented by a dedicated hardware circuit 'Neuron 1 to Neuron N'.

A reduced flow of spikes output from the first hidden layer is input to a Neural-Core Control. The Neural-Core Control is composed of a '1:N' counter, a multiplexer (MUX) and an output First-in First-out (FiFo) buffer. One can note that 'N' is the number of neuron circuits of the FP layer. When the N neuron circuits of the FP layer have processed in parallel an input spiking event (i.e. a spike from the input flow of spikes), their output spikes are connected to the write-enable of the output FiFo buffer sequentially through the multiplexer MUX. The MUX block is configured to select the output spikes one after the other by using their addresses (@Neuron) given by the 1:N counter. These addresses are also connected to the input data of the FiFo module. In the case the output spike is high (spike=1), the output of the counter (i.e. the neuron's address) is written into the output buffer (FiFo). Once the counter ends the forward of all the fully-parallel layer output spikes, it resets the count and repeats the same procedure for the next

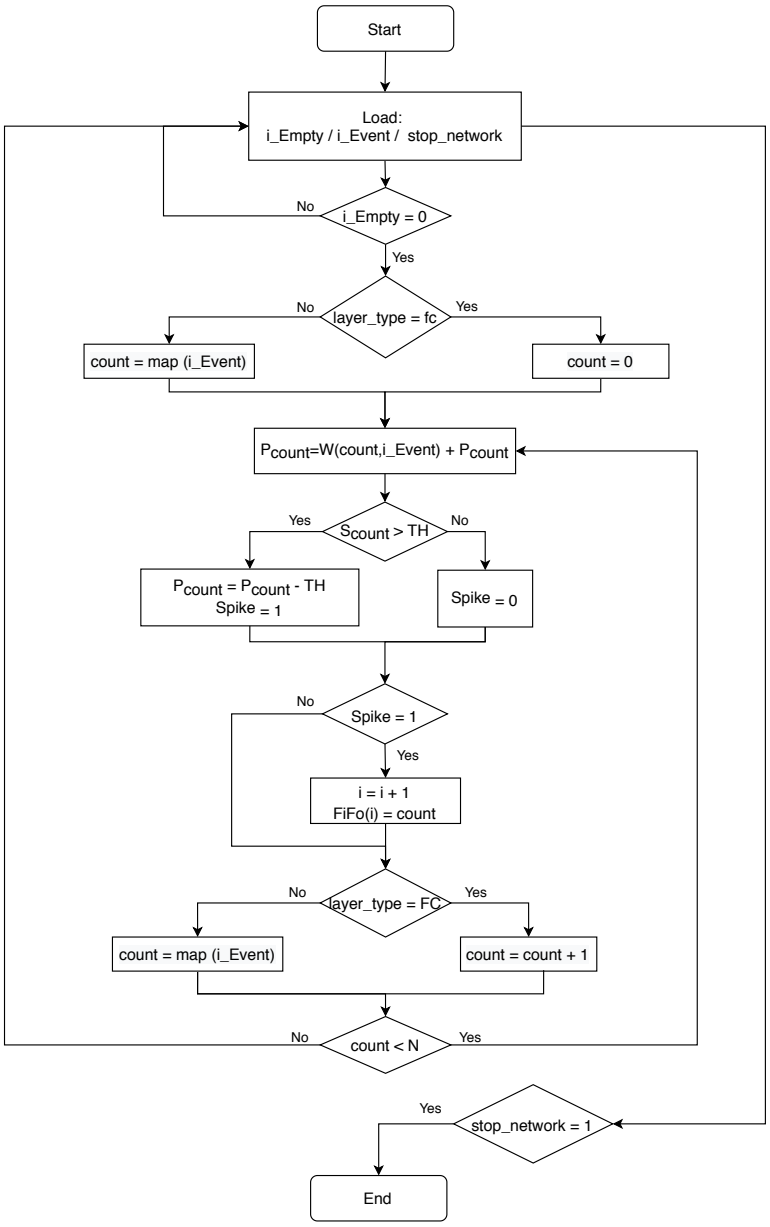


FIGURE 5.8: Flow chart of the NPU operating steps.

spikes. Next the output of the Neural Core - 'Output Event'- becomes the input of the Time-Multiplexed part of the hybrid architecture.

Figure 5.10 is a flow chart of the steps for operating a Neural Core module. The process begins by a first step of reading the input spike address (@In) and a 'stop_network' signal provided by the Terminate Delta module. On one hand, the process allows verifying if the 'stop_network' signal is equal to '1' to end the process. On the other hand, the input spike address '@In' is forwarded to the neuron circuits of the first hidden layer to perform the

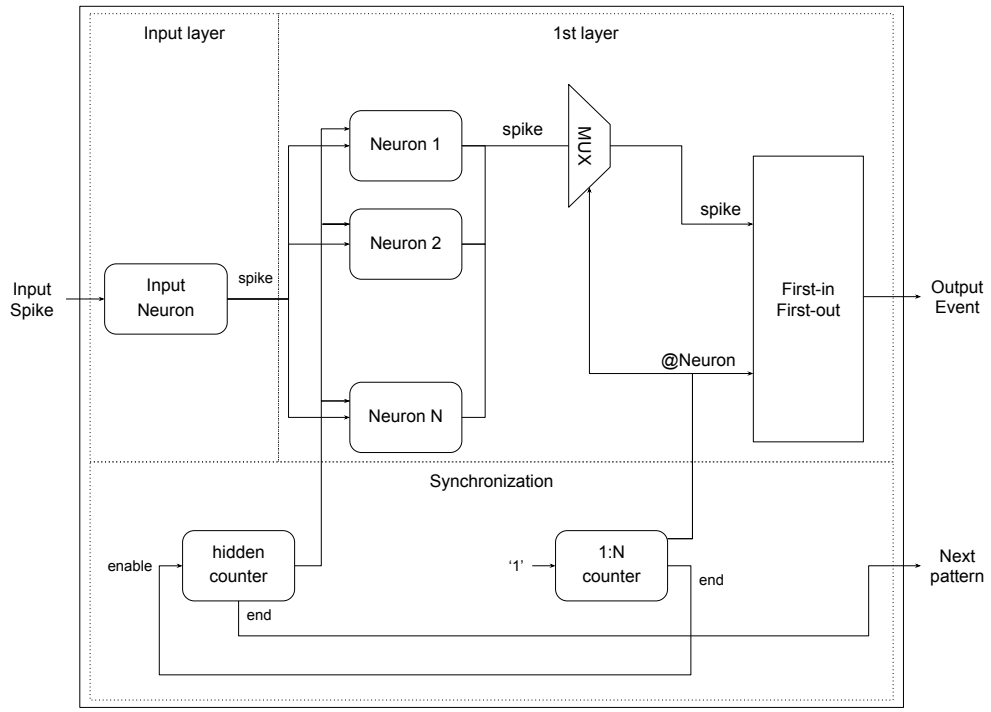


FIGURE 5.9: Neural Core module schematic diagram.

integrate-and-fire rule. Each of the neurons is computed by using the input address @In being to retrieve the corresponding weight that is accumulated to its internal potential 'Si'. Each accumulated potential is compared to a threshold 'TH' in a parallel way. When this potential is higher than the threshold, a spike is emitted, i.e. Spike=1, and the potential is updated by reducing from it the threshold 'TH'. These spikes are then used to write output spike addresses as spiking events in the FiFo. A multiplexer controlled by a counter is used to sequentially forward the spikes one-by-one. If a spike is emitted, the address of the neuron that has emitted it (spiking event), is saved in the FiFo buffer. Once the counter has forwarded all the spikes, 'count = N-1' which is verified, the count is reset (count=0), and then the process is repeated by reading new inputs (@In and stop network).

5.5.2 Fully-Parallel Architecture

This sub-section describes the FPA architecture. This architecture has been designed alongside TMA architecture for evaluating the trade-off between resource, latency and intensiveness at a lower level than using NAXT Simulation tool.

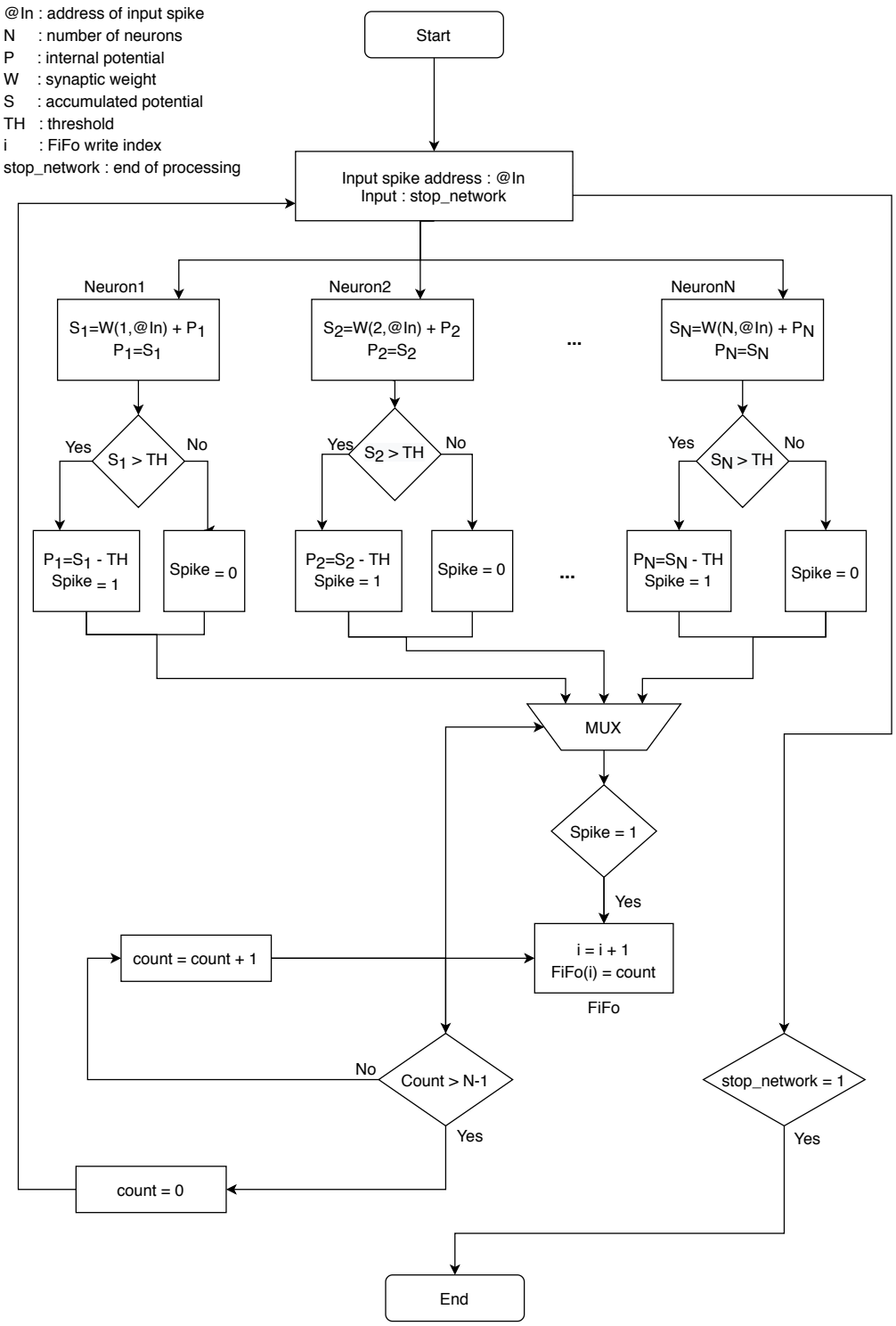


FIGURE 5.10: Flow chart of the Neural Core operating steps.

In the FPA architecture, all the SNN logical neurons are implemented in hardware. That is, the IF neuron hardware module is instantiated as many times as the SNN has logical neurons. Figure 5.11 shows the internal structure and components of the FPA architecture. It is composed of counter modules, with

one per layer to synchronize the overall SNN's computations. In this architecture, each layer waits until the previous layer neurons finished all their processing to start: all the spikes are processed layer after layer. The Input Neuron forwards the data to the first hidden layer, spike by spike, where a counter is managing their addresses. At each cycle, the IF neurons integrate the input spike and buffer their output spikes. When all the input spiking events are processed, the hidden counter triggers an end signal to the next layer. Note that all the hidden layers do the similar process on their incoming spikes, layer by layer. The last hidden layer's counter enacts the completion of the process to the output layer. Afterwards, the output layer's counter enables the output neurons to process their incoming spikes. Finally, the output layer spikes are processed by the classification module to check the ending of the computations and selecting the winner class.

This FPA architecture should result in fast processing but high logic resource occupation. In the following part, we present the second architecture which takes an opposite architectural choice : Time-Multiplexed Architecture.

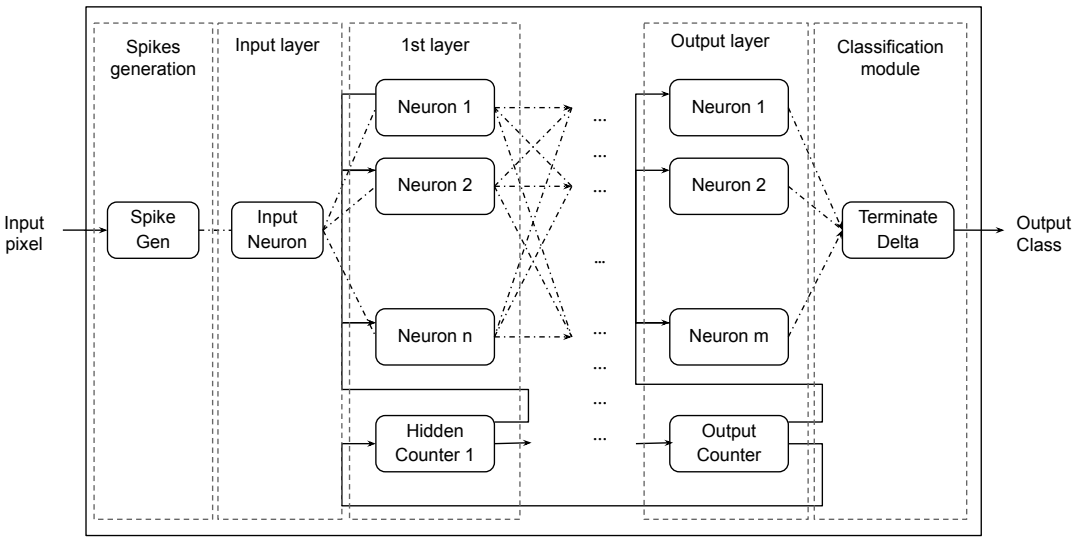


FIGURE 5.11: FPA simplified schematic diagram.

5.5.3 Time-Multiplexed Architecture

The TMA architecture is designed to save hardware resources, in contrast with FPA architecture. In this implementation, the main computation unit is the NPU module described in section 5.5.1. Contrary to FPA, the number of hardware neurons is smaller than the number of logical neurons: each layer

is represented by one single NPU, instead of one NPU per neuron. The complete hardware architecture consists of NPU modules, interconnected with each other as shown in figure 5.12. As in FPA, the input layer is represented by a dedicated Input Neuron module, which forwards input spikes to the first hidden layer. Each one of the other layers are represented by one single NPU, which successively computes the layer's logical neurons in a time-multiplexed manner. These NPUs have their own ROM memory containing their parameters. This architecture should drastically diminish the hardware occupation, but increase the system latency as a counterpart. In other words, TMA and FPA represent the two extremes of the latency versus hardware intensiveness trade-off. In the next subsection, we will describe a middle ground between those two extremes, taking advantages from both to fit the reality of spiking activity in an SNN : the novel Hybrid Architecture (HA).

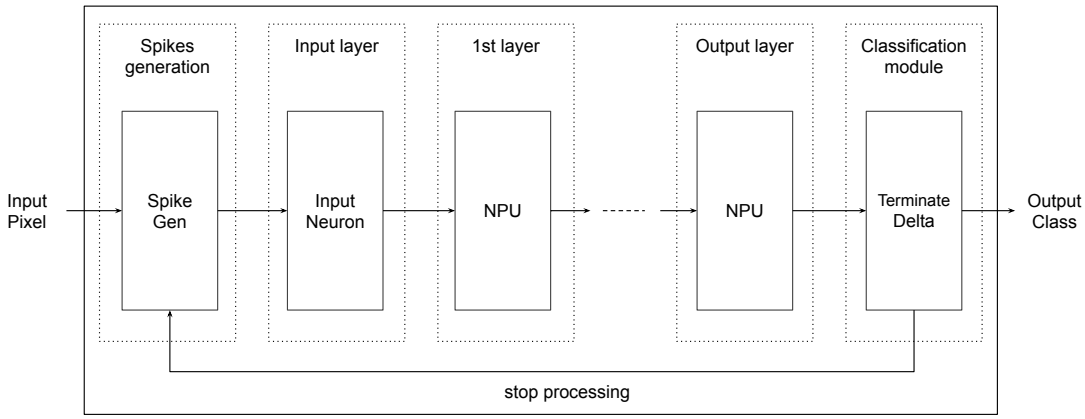


FIGURE 5.12: TMA simplified schematic diagram.

5.5.4 Hybrid Architecture

In chapter 4, it was shown that most of the spiking activity in the network is located in the first layer. Therefore, the first hidden layer is the most solicited during processing. To take advantage of this aspect, the Hybrid Architecture (HA) is designed, mixing both TMA and FPA. Moreover, this novel hybrid architecture is appropriated for the use of the Spike Select method that is described in chapter 4, in which spiking activity is concentrated in the first layers. This implementation derives from the findings and observations we made thanks to the funnel-like Design Space Exploration framework. As shown in figure 5.13, this architecture is a mixture of FPA and TMA, where: first, the initial two layers are implemented using a Neural Core module as

in FPA; second, the remaining layers are time-multiplexed using one NPU per layer, as in TMA.

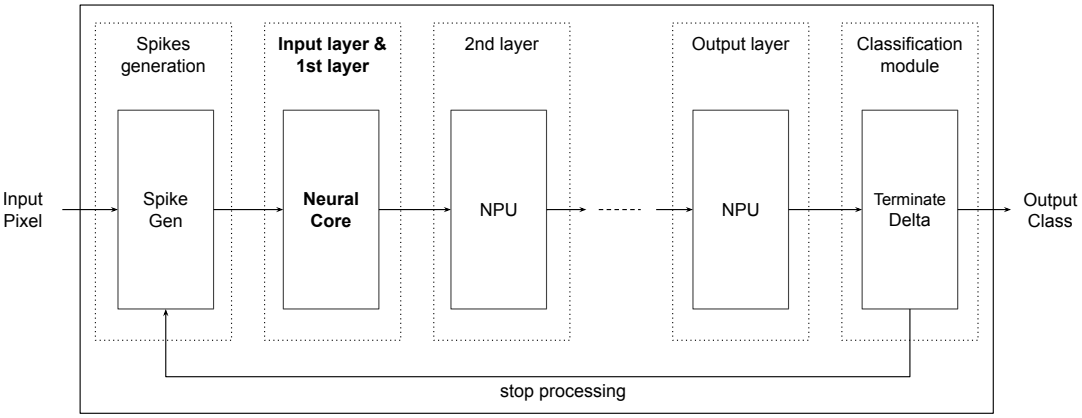


FIGURE 5.13: HA simplified schematic diagram.

Advantageously, as the number of spikes is drastically reduced by the Neural Core module, there is no such need of parallel computing and the plurality of NPU modules are implemented as a time-multiplexed structure to allow a sequential processing of the spikes. There is as much as computing cycles as the number of logical neurons implemented in one NPU. The output of the last NPU becomes the input of the classification module represented by the Terminate Delta, which allows determining if the classification process is ended or not, by determining if a sufficient number of spikes has been received to classify the input image. If not, the process is iterated, or the process stops 'Stop Processing'.

Figure 5.14 is a flow chart of the general steps for operating the hybrid architecture. The process begins by a first step of loading or receiving input data. Next, the process allows the spike generator to generate a flow of spikes from the analog input data. On a next step, the process allows the Neural Core module to process the flow of spikes in a fully-parallel processing to reduce the number of output spikes. On the next steps, the process allows each output event to be sequentially processed by the plurality of Neural Processing Units. Afterwards, the process allows the output of the last NPU to be processed by the Terminate Delta or Classification module to determine a winner class. During this last step, if the Terminate Delta determines a winner class, the process allows the activation of the 'stop_network' signal to stop the process. Note that, like in the other architectures, all the steps of this hybrid architecture process work in a pipelined way to optimally use the components of the architecture over time. For example, while loading the next input data,

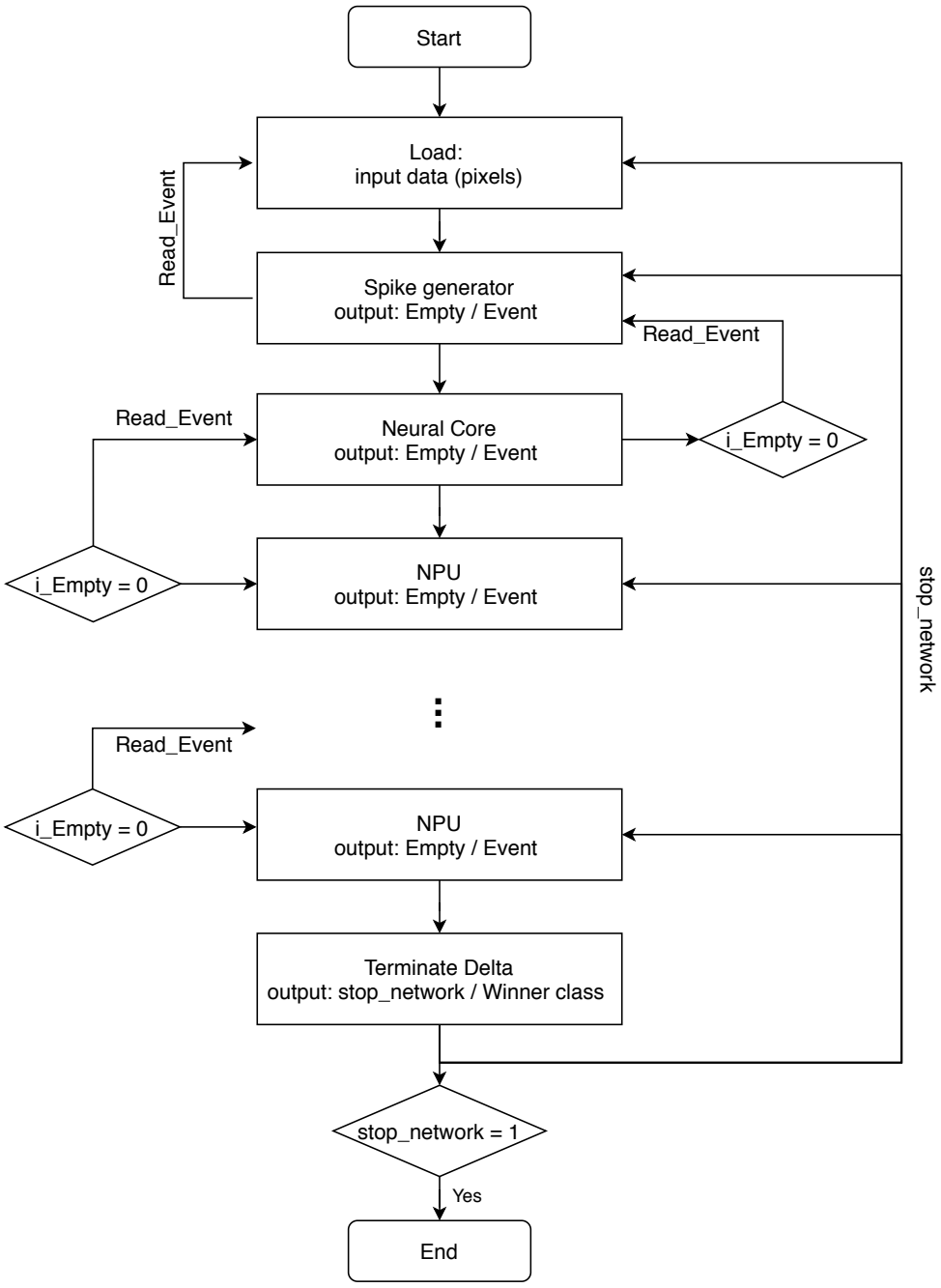


FIGURE 5.14: Operating steps of the Hybrid Architecture flow chart.

the spike generator translates progressively the pixels of a previously generated flow of spikes. At the same time, the neural core is processing these input spikes, the plurality of NPUs process other recent data, and the Terminate Delta verifies the classification of current data received from the last NPU.

Note that the process on this figure can be used in one hand to describe the

TMA operation by simply replacing the neural core coming after the spike generator by an NPU. On the other hand, it can describe the flow of the FPA architecture by using instead of the NPUs neural cores to process data in parallel (here, the neural cores would not have the input neuron).

5.6 Experiment and results

In the design of embedded neuromorphic architectures, it is important to consider the resources occupation in order to estimate the required silicon area. In this section, we quantify the hardware cost estimations of the different architectural models presented within this chapter. The architectures are described using generic VHDL code, which is compatible with any fully-connected multi-layer SNN topology. The VHDL codes describing the hardware models are configurable by the parameters of the SNNs that are rising from N2D2 or any other neural networks building framework. Since the objective of this chapter is exploring architectural models to implement SNNs for embedded AI applications, we have used FPGA as the hardware target and implemented SNNs with different topologies but based only fully-connected layers. As mentioned in chapter 2, FPGA devices are reconfigurable platforms that fit well to our exploration purposes. Therefore, we have synthesised the SNNs using the different architectural models using the Quartus®Prime Lite 18.10 edition tool and targeting the "5CGXFC7C7F23C8" Cyclone®V FPGA board.

In the following, we present the different hardware cost results that we have obtained within this RTL architectural exploration of deep SNNs hardware implementation. These results are issued from the synthesis of several SNN topologies having different sizes using the three hardware architectures: FPA, TMA and HA.

Fully-Parallel Architecture : towards multiplexing

In this exploration, we begin by synthesising different SNNs using the first architectural model and with the fully-parallel architecture. Indeed, this model is the extreme case that is targeting computing efficiency in terms of processing time and speed. As mentioned earlier, each logical neuron of the SNN is synthesised in hardware through the use of an IF-Neuron module. These FPA synthesis results are summarized in table 5.3, giving the logic (ALMs) and registers occupation related to the SNN topology. Then, those

results are used to plot two graphs showing the evolution of resource intensiveness against the number of neurons (figures 5.15 and 5.16).

TABLE 5.3: FPGA Logic ALMs and registers utilization by the FPA architecture.

SNN: Topology	Logic	Registers
784-100-10	13317	3836
784-200-10	26225	7048
784-300-10	31461	10974
784-300-300-10	47257	24008
784-3x(300)-10	60628	40600

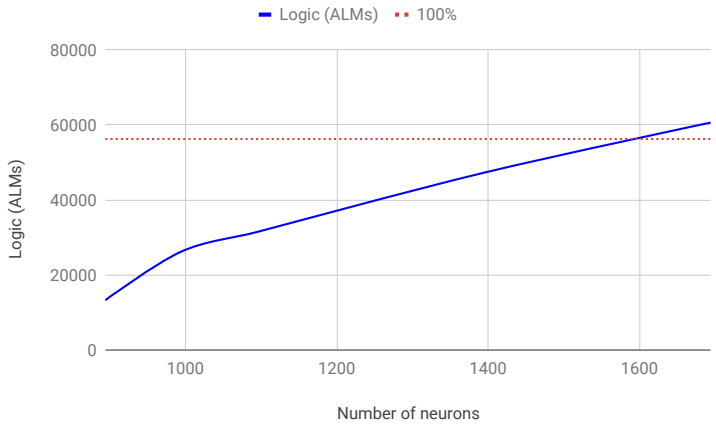


FIGURE 5.15: FPA architecture: FPGA logic (ALM) utilization versus the SNN number of neurons; Different SNN topologies are used, refer to table 5.3.

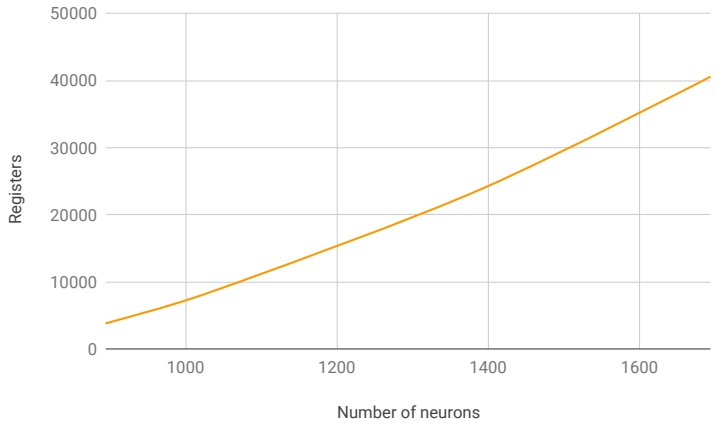


FIGURE 5.16: FPA architecture: FPGA registers occupation versus the SNN number of neurons; Different SNN topologies are used, refer to table 5.3.

TABLE 5.4: FPGA resource occupation of different SNNs by the TMA architecture.

SNN topology	Logic	Registers	BRAM (KB)
784-100-10	690	1255	64
784-200-10	1192	2168	128
784-300-10	1714	3082	230
784-300-300-10	3235	5937	241
784-3x(300)-10	4736	8799	249

TABLE 5.5: FPGA cyclone V resources occupation of different SNNs with by the HA architecture.

SNN topology	Logic	Registers
784-100-10	2440	1383
784-200-10	7478	2434
784-300-10	21406	3455
784-300-300-10	22638	6318
784-3x(300)-10	22859	9336

From these results, we observe that the FPA logic occupation is directly proportional to the SNN's topology (depth/size), i.e. increases linearly with the number of neurons. Nevertheless, the generated circuits are supported by the FPGA fabric when the networks are smaller than the *784-300-300-10* topology, but not for larger ones. Therefore, our first intuition regarding the limited scalability of *FPA* when used for deep SNNs is confirmed. But, we are yet to confirm if the *TMA* or *HA* architectures occupy less resources, and are thus more viable.

Time-Multiplexed and Hybrid Architectures

In this context, we have synthesized the same SNN topologies using these two architectures (TMA and HA), the results are shown in tables 5.4 and 5.5. With these architectures, instead of using only registers and logic for memory, we also use on-chip BRAM memory. Besides the use of these different memory types and organizations, the total memory footprint should be the same since the same SNN topologies are implemented, i.e., equal amounts of parameters and activities to store in memory. Therefore, we focus on the occupation of FPGA logic cells, where the major difference between the three architectures should be found. For improved clarity, we have plotted the histogram shown in figure 5.17 representing the logic occupation of the three architectural models.

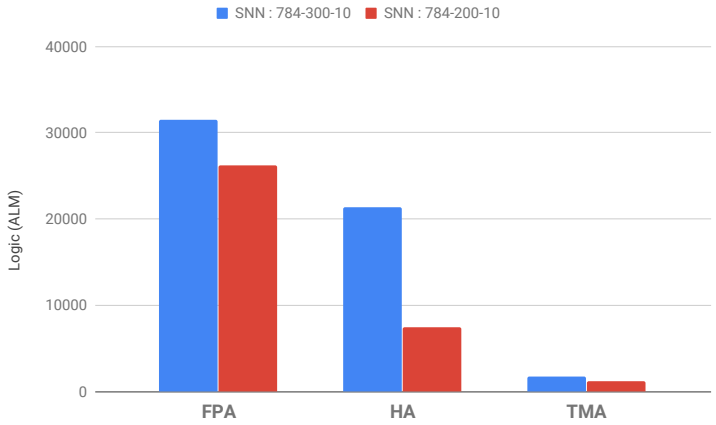


FIGURE 5.17: Logic occupation of two SNNs by the three architectures: FPA, TMA and HA.

As expected, compared to FPA, these architectures, TMA and HA, occupy much less logic resources due to multiplexing. Indeed, one NPU per layer is used for all the SNN's layers with TMA and from the second layer with HA. An NPU is using one hardware neuron coupled with some control logic to represent all the neurons of a layer. Using the 784-3x(300)-10 for example, to represent the three hidden layers 3 NPUs are sufficient with TMA but with FPA 3×300 neuron modules are required. Therefore, multiplexing hardware to implement large-scale SNNs would be the more adequate solution. In order to estimate the latency of these architectures, we have used the average number of spikes generated for processing a pattern using different SNN topologies on MNIST dataset (refer to chapter 4) and computed the number of cycles, the results are summarized in table 5.6.

TABLE 5.6: Latency (number of cycles / image) of the three architectures using three neural coding schemes – SNN : 784-3x(300)-10.

Coding method	Latency (cycles)		
	FPA	HA	TMA
Jittered Periodic	1039,5	84064	300540
Spike Select	1660,5	34437	496990
First Spike	332	23540	74370
Single Burst	3077	441432	459970

We observe that the processing latency is higher using TMA than HA and even more than FPA. If we transpose these latency results on the hardware logic occupation, we notice that the latency is inversely proportional to the

logic occupation, where we find that the TMA, while having the highest latency, occupies the lowest logic resource amount compared to the other structures. Indeed, time-multiplexing allows to reduce the quantity of hardware resources, but relies on the serialization of a parallel task, thus resulting in a higher processing latency. This is why three different architectures have been designed to evaluate the trade-off between hardware resources and processing latency. In this context, the HA is an intermediary solution with a significant reduction in the amount of hardware resources, while maintaining reasonable latency. Referring to the latency table 5.6 and to the logic occupation tables 5.3, 5.4 and 5.5, we notice that on average, TMA has a gain of **56.19%** in terms of latency when compared to TMA and **57.05%** in terms of logic occupation when compared to FPA.

Finally, to analyze the performance of our architectures, a measurement of Synaptic Operation per Second (SOPS) was performed on all the architectures using the 784-3x(300)-10 SNN topology on the same FPGA board target. The FPA achieves the best performance with 51.02 billion SOPS, whereas TMA achieves only 283.80 million SOPS. The HA is below FPA, where it achieves 23.12 billion SOPS using similar topology and FPGA device. Their respective measured maximum frequencies are 83.51 MHz for FPA, 76.3 MHz for TMA and 70.95 MHz for HA.

5.7 Discussions

As a reminder, the high-level results obtained with the NAXT simulator (in chapter 3) are summarized in table 3.4 and figure 3.7. As already explained, surface estimations provided by NAXT correspond to an ASIC target, and can be seen as qualitatively equivalent to logic occupation for an FPGA target. In these results, the trade-off between latency and FPGA occupation (i.e., chip surface in the figure and table) was clearly visible: *FPA* had low latency but high FPGA occupation, and the opposite was true for *TMA*. Fine-grained results provided by RTL synthesis and latency estimations for the "784-3x(300)-10" SNN are summarized in figure 5.18.

This figure shows Pareto curves representing FPGA logic occupation versus latency (number of cycles) for the three hardware architectures according to the neural coding methods described in chapter 4. These results are consistent with the high-level estimations, where the same trade-off can be seen between latency and logic occupation: *FPA* has a low latency but high logic

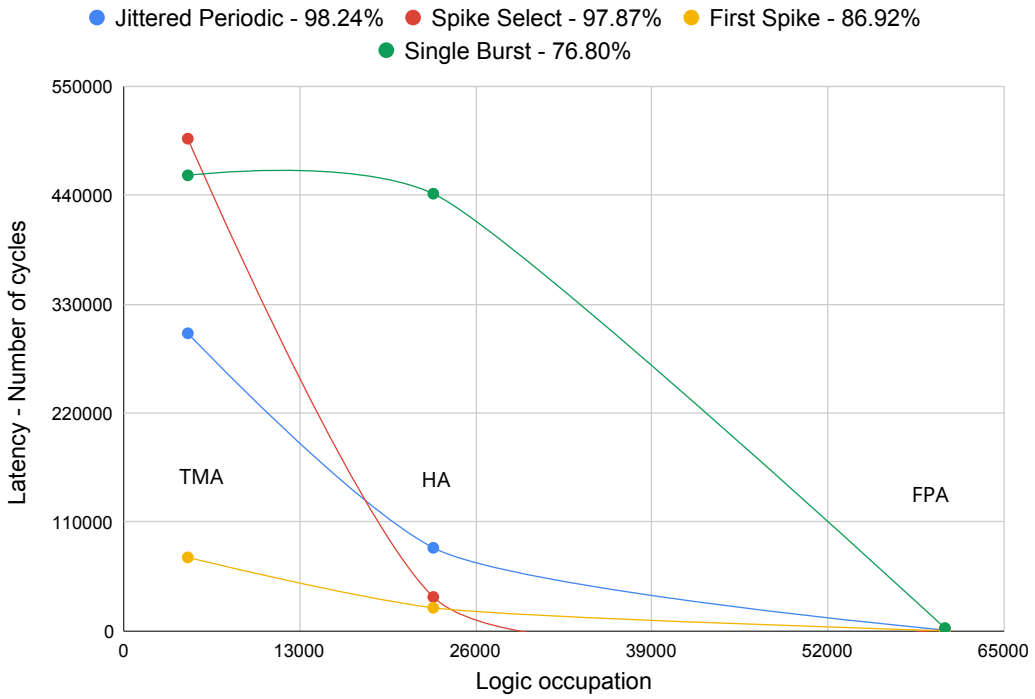


FIGURE 5.18: Trade-off between logic and latency of the hardware architectures according to three neural coding schemes.

occupation, whereas *TMA* has high latency but low logic occupation. If we consider the neural coding scheme without looking at the recognition rate, the First Spike method combined with TMA architecture has the best 'latency / chip surface' trade-off. However, this method has a loss of around 10% in terms of accuracy compared to Spike Select and Jittered Periodic methods, see section 4.5. Therefore, taking into account the accuracy criterion, the Spike Select method combined with HA architecture has the best 'latency / logic' occupation trade-off. The method, while performing 97.87% accuracy on MNIST dataset, allows only few spikes propagating in the deeper layers of the SNN, which fits well the HA architecture making this combination well-tailored for deep SNNs implementation.

The coherence of these results is shown in figure 5.19, which depicts the evolution of FPGA occupation (in terms of logic cells) against the number of neurons, for both theoretical estimations and Quartus® experimental results. The considered network has a fully-parallel architecture in both cases. Both curves are very similar for a low number of neurons, which shows coherency between estimations and experimental results. The divergence observed for higher numbers of neurons is due to synthesis optimizations performed by the Intel Quartus® tool, which are not taken into account in our estimations.

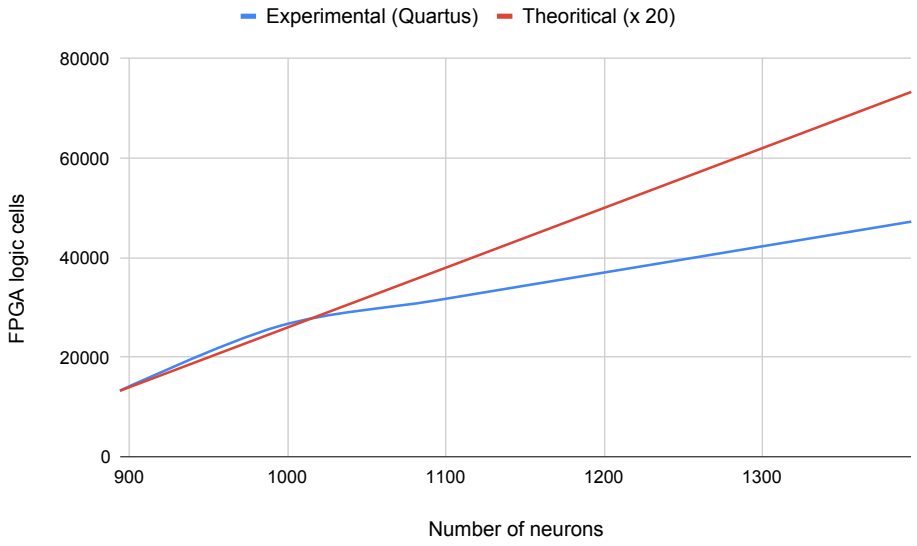


FIGURE 5.19: Theoretical vs. experimental FPGA occupation with respect to the number of neurons – FPA.

However, the two curves remain qualitatively coherent, as they follow similar linear growths. Hence, the results are coherent between high-level and low-level parts of the design flow. Even if the estimates could be improved in the future.

EBS sensor: towards frame-free SNNs for video recognition

In this work, we have focused on static image recognition. Thus, we based our approach on ANN-SNN conversion method in which input data are translated into spikes (see chapter 4). This conversion step is one of the main drawbacks of SNNs usage, as it may counterbalance the latency, energy and surface savings that are achieved thanks to the spiking processing. For video recognition, however, this issue can be tackled by using event-based cameras.

Indeed, in contrast to static images, videos can be directly recorded in an event-based fashion, with so-called asynchronous cameras Delbrück et al., 2010. In contrast to classical cameras, which output a succession of discrete frames, an Event-Based Sensor (EBS) emits a continuous flow of events: each pixel outputs a spike whenever an edge crosses its receptive field. In other words, an event-based sensor outputs a flow of spikes representing the movement happening in its receptive field. We expect that SNNs could benefit from the use of such EBS sensors, as the processing would eliminate the time and energy consuming spike generation phase.

5.8 Conclusion

In this chapter, we dealt with the last level of the framework presented in chapter 2. This represents the low-level RTL description that allows us to determine the best implementation by providing fine-grained evaluations of the architecture. Indeed, we have first implemented naively an SNN and MLP with the same simple topology, then, realized three different SNN implementations: first, *Fully Parallel Architecture* and *Time-Multiplexed Architecture* that are developed to emphasize the two extremes of the latency versus hardware resources trade-off; second, a novel and innovative *Hybrid Architecture* that is a middle ground, deriving from the findings and observations of our Design Space Exploration work.

First, the confrontation of neuroscience to machine-learning algorithms showed that the neuroscience approach reaches similar performance to machine learning one in terms of accuracy, while in terms of hardware implementation cost (area, power), the SNN is about twice as efficient as the ANN.

Second, for deep SNNs, and according to our design flow, the most suitable architecture is the Hybrid Architecture, as it takes advantage of the increasing spiking activity sparsity as we go deeper into the network. This novel architecture has been developed in our lab, and to the best of our knowledge, is completely original. Combined with Spike Select Coding, presented in chapter 4, is a well-tailored approach for future Deep SNN implementation into embedded systems.

Toward spiking CNNs

At this point, our work focused on fully-connected based networks, the so-called classifiers (Zhang, 2000). However, this neural networks type is restrained to simple classification tasks: they are not able to perform classification on complex data, and are not resilient to image rotation or translation. Thus, modern ANNs for complex data recognition and classification involve convolution and pooling layers (LeCun and Bengio, 1998; Krizhevsky, Sutskever, and Hinton, 2017). The Convolutional and Pooling layers enable feature extraction resulting in a Feature Map that can be afterwards fed to a simple classifier. In order to simulate state-of-the-art ANN hardware implementations, in the next chapter, we present a generic event-based architecture dedicated to spiking CNNs and also classifiers based on the fully-connected layers developed in this chapter.

Chapter 6

Spiking CNN hardware architecture

6.1 Introduction

The current trend with artificial neural networks is towards the use of very deep networks with a continuous increase in the number of layers. Networks like ResNet (He et al., 2015), AlexNet (Krizhevsky, Sutskever, and Hinton, 2012) or GoogLeNet (Szegedy et al., 2015), that are comprised of a large layers and neurons number, require a lot of hardware resources for their proper execution. Parallel architectures are hardly conceivable with this type of ANN topologies because their need in hardware resources is very important. Consequently, to meet this need, the use of multiplexed and event-driven architectures is more than essential. In this chapter, we describe an architecture implemented in a completely time-multiplexed way with the possibility to integrate in future parallel parts in order to implement the hybrid architecture presented in the previous architecture. The current architecture can infer any deep spiking CNN network topology. The architecture has a similar structure of the architectures presented in the previous chapter, refer to section 5.3, but also includes processing units dedicated to pooling and convolutional layers. Figure 5.2 shows the complete overview of this hardware architecture for spiking CNNs. As shown in this figure, the architecture is composed of three different neural processing unit types: Convolutional Processing Unit 'ConvPU', Pooling Processing Unit 'PoolPU' and Fully-Connected Processing Unit 'FcPU', a spike generation module 'Spike Gen' and classification module 'Terminate Delta'. The different computing modules of this architecture communicate through FiFo-like interfaces and use the event-based communication protocol that is presented in the section

5.4 of the previous chapter.

6.2 Convolutional processing unit – ConvPU

In this section, we present a thorough description of the designed convolutional processing unit (ConvPU). A schematic representation of ConvPU internal structure is illustrated in figure 6.1.

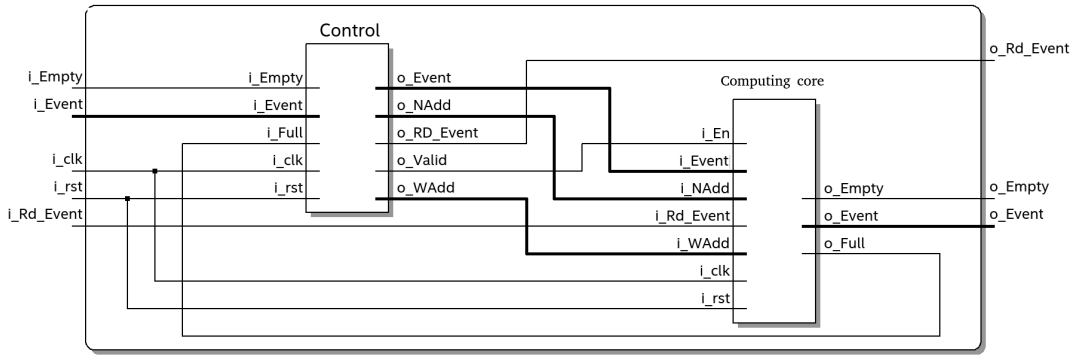


FIGURE 6.1: Convolution Processing Unit schematic diagram.

This processing unit is composed of two main parts that are the convolution control module and the convolution computing core. The control module is connected to the inputs of ConvPU and the computing core is connected to outputs. As shown in this figure, the control and computing units are connected between each other through four signals communicating the neurons and weights addresses, an enable of computations and a full signal. On one hand, the control module is designed to manage the state of the ConvPU by controlling its computing core. On the other hand, the computing core is dedicated for processing the logical neurons of the convolutional layer. These control and computing modules are described more precisely in the following sub-sections.

6.2.1 Control module

The convolution control module is, as mentioned before, a sub-module of the ConvPU and is used for managing and controlling the processing unit. Figure 6.2 shows the internal structure of this module, where it has three sub-modules that are a state machine, an address compute module and an address generator.

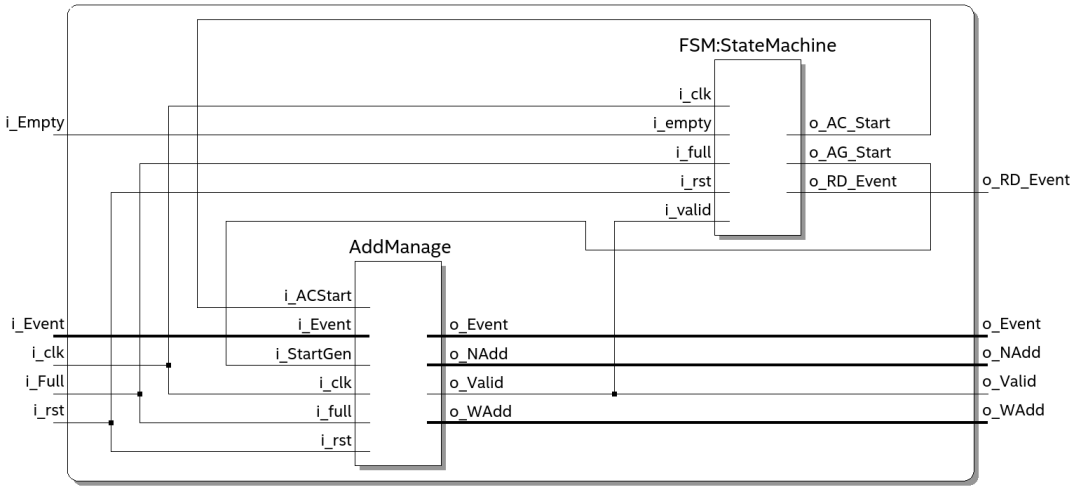


FIGURE 6.2: Convolution control module schematic diagram.

Finite State machine

The first module is a Finite State machine (FSM) that monitors the other sub-modules and also enables the computing core. A graph representation of this FSM is shown in figure 6.3, and states transitions in table 6.1.

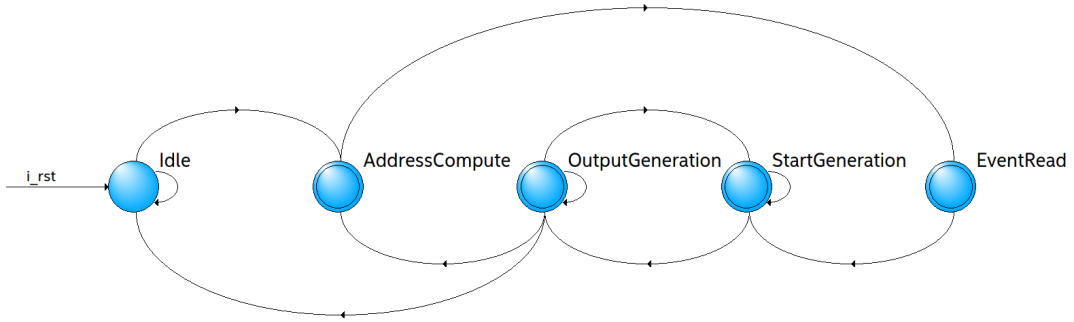


FIGURE 6.3: Convolution control : finite state machine.

TABLE 6.1: Convolution FSM states transitions.

Source state	Destination state	Condition
AddressCompute	EventRead	None
EventRead	StartGeneration	None
Idle	Idle	(!Empty).(!Full)
	AddressCompute	(Empty)+(Full)
OutputGeneration	Idle	(Empty).(!Valid).(!Full)
	OutputGeneration	(Valid).(!Full)
	StartGeneration	Full
	AddressCompute	(!Empty).(!Valid).(!Full)
StartGeneration	OutputGeneration	!Full
	StartGeneration	Full

The FSM is composed of 5 states : "Idle", "AddressCompute", "StartGeneration", "EventRead" and "OutputGeneration". At reset, the state machine is at "Idle" state where all the outputs are equal to '0'. Then, if there is an input event and the FiFo is not full ((Empty = '0') and (Full= '0')) the FSM goes to "AddressCompute" state otherwise it stays in "Idle" state.

In "AddressCompute" state, the module "Address Compute" is enabled by setting "Add-Compt-Start" to '1' and the FSM goes directly to "StartGeneration".

Once it is in "StartGeneration", the module "Address Compute" is disabled and the output generator module is enabled by setting "Add-Compt-Start" to '0' and "Out-Gen-Start" to '1'. In case the FiFo is full, the FSM stays in the current state otherwise it goes to "OutputGeneretation".

When the state machine is in "OutputGeneretation" state, the computing core is receiving the neurons and weights addresses for doing its computations. If the FiFo module is full, the state machine goes to the previous state and waits until full signals goes low to go back at "OutputGeneration" state. Once the Output Generator module finishes transmitting the addresses for the current event it changes the state of valid to low '1'. Therefore, the FSM changes the state to "AddCompute" if there is an input event for processing (Empty='0'). If it is not the case, i.e. Empty='1', the FSM goes to "Idle" and waits for input events. This process is repeated until the Terminate Delta sends a stop processing signal.

Address management

In the previous section, we have seen how the different modules of the control module are managed and enabled by the state machine. In this section, we describe the functioning of the address compute and address generator modules.

Address compute

In order to reduce the number of hardware connections between the CNN neurons, we have designed an algorithm to compute the addresses of the weigths. It indicates the convolution neuron addresses corresponding to the input event, refer to algorithm 1. Note that, "AddCompute" module is a hardware implementation of this algorithm. The role of this module is to find

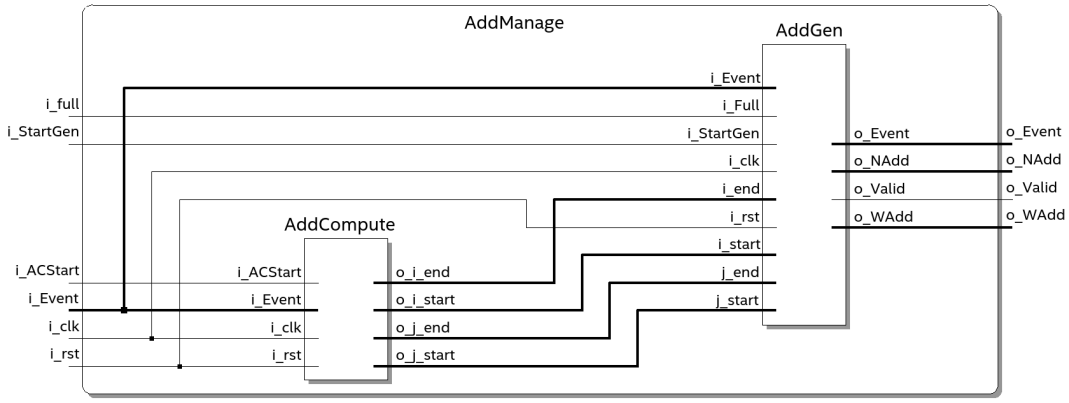


FIGURE 6.4: Address management module's internal structure.

the height and width coordinates of the convolution neurons intersecting the input event from the previous layer feature map.

Algorithm 1: Address compute algorithm

$(E_C, E_H, E_W) = Map(Event)$;
Neuron's width coordinates :
Result: $(i_{start}$, $i_{end})$
if E_W - F + $1 < 0$ **then**
    $i_{start} = 0$;
    $i_{end} = \lfloor E_W / S \rfloor$;
else if $E_W + F - 1 >= N_{in}$-1 **then**
    $i_{start} = \lfloor (E_W - F)/S \rfloor$+1;
    $i_{end} = N_{out}$-1;
else
    $i_{start} = \lfloor (E_W - F)/S \rfloor$+1;
    $i_{end} = \lfloor E_W / S \rfloor$;
end
Neuron's height coordinates :
Result: $(j_{start}$, $j_{end})$
if E_H - F + $1 < 0$ **then**
    $i_{start} = 0$;
    $i_{end} = \lfloor E_H / S \rfloor$;
else if $(E_H + F - 1 >= N_{in}$-$1)$ **then**
    $i_{start} = \lfloor (E_H - F)/S \rfloor$+1;
    $i_{end} = N_{out}$-1;
else
    $i_{start} = \lfloor (E_H - F)/S \rfloor$+1;
    $i_{end} = \lfloor E_H / S \rfloor$;
end

Where:

E_C : input event channel's coordinate

E_H : input event height's coordinate
E_W : input event width coordinate
i_{start} : convolution neuron width coordinate start-index
i_{end} : convolution neuron width coordinate end-index
j_{start} : convolution neuron height coordinate start-index
j_{end} : convolution neuron height coordinate end-index
F : kernel's height/width
S : convolution stride
N_{in} : input FM height/width
N_{out} : output FM height/width
$\lfloor./.\rfloor$: integer division

This algorithm maps the input event into (E_C, E_H, E_W), i.e. channel, height, width, coordinates, and then gives its exact position in the input feature map. We have separated the calculation of the output neuron addresses according to the position of the input event. Two of them concern the case where the event is located in the bordering extremities of the input feature map and the remaining one is where it is inside the FM (normal case).

So let's first consider the normal case. Indeed, a neuron of the convolution layer with a width coordinate n_W is intersecting all input events situated in $[(n_W$*S), $(n_W$*S+F-1)]. Where "n_W*S" and "n_W*S+F-1" are the start and end coordinates of the convolution kernel when projected to the input feature map. Therefore, all the convolution neurons intersecting the input event with width coordinate E_W must have E_W in $[(n_W$*S) , $(n_W$*S+F-1)], thus, n_W is located in $[(\lfloor E_W - 1/S \rfloor + 1), (\lfloor E_W/S \rfloor)]$. This range is given in the algorithm by i_{start} / i_{end} for width coordinates and j_{start} / j_{end} for height coordinates.

The second case is where E_W is situated between 0 and F-1: here the first output neuron width coordinate is 0 ($i_{start} = 0$) because the other positions $(\lfloor (E_W - F)/S \rfloor + 1)$ are negative and cover positions out of the input FM.

The last case is where the event is situated between $N_{in} - 1$ and $N_{in} - F$: here the last position is $N_{out} - 1$ because the other position are greater than $N_{out} - 1$ and also cover positions out of the input FM.

This methodology is then applied similarly to the "E_H" coordinate of the input event to get the height coordinates range (j_{start} / j_{end}) of the convolution neurons intersecting the input spike.

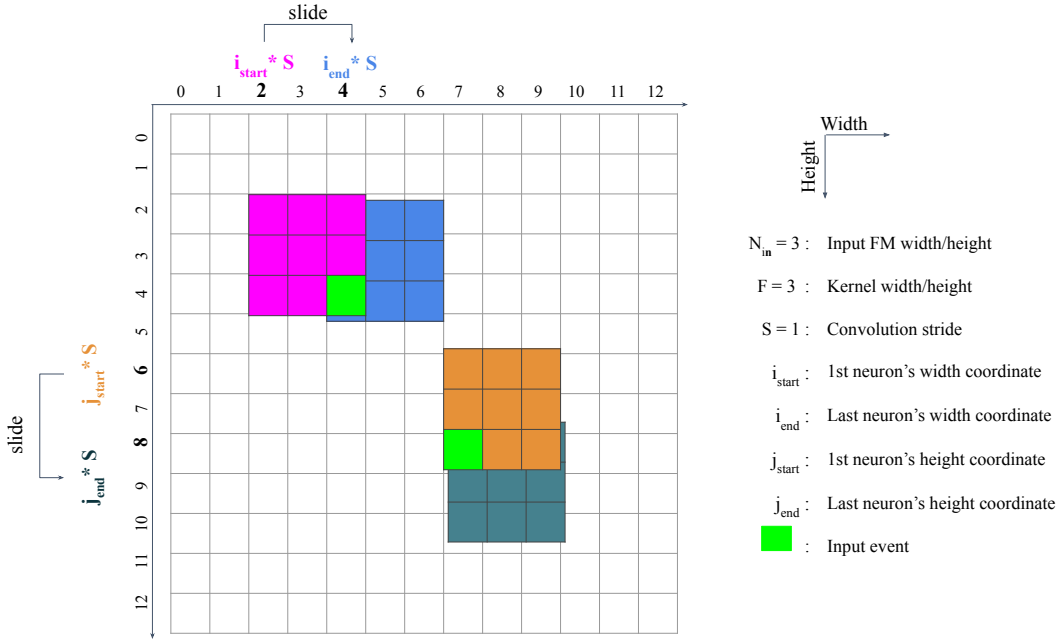


FIGURE 6.5: Address compute: coordinates of the first and last convolutional neurons having input event in their receptive field.

In figure 6.5, we illustrate the computation with an example of applying a convolution kernel to a 13x13 (N_{in}=13) input feature map with a kernel of size equal to 3x3 and a unity stride (S=1). In this example, we illustrate the coordinates range of the neurons having the input event in their vision field. With the first event which is located at the position (4,2), we consider only the width axis. The first position of the kernel covers the input event with its last cell. Then when we slide the filter to the right, we reach the last position where the input event is located in its first cell. The same methodology is applied to get coordinates in the height axis and with the second event located at the position (8,8).

Addresses generation

Algorithm 2 describes how the neurons and weights (@Neuron, @Weight) are computed using the start and end indices of convolutional neurons intersecting the input spiking event. These indices are generated by the "Add-Compute" module as shown in figure 6.4.

Where:
N_{Kernel} : is the number of convolution kernels (filters);

Algorithm 2: Convolutional layer's address generation algorithm

for ($i = i_{start}$; $i \leq i_{end}$; $i++$ **)** {
  for ($j = j_{start}$; $j \leq j_{end}$; $j++$ **)** {
    for ($k = 0$; $k \leq N_{Kernel} - 1$; $k++$ **)** {
      $@Neuron = i + j * N_{out} + k * N_{out}^2$;
      $WAdd_W = E_W - i * S$;
      $WAdd_H = E_H - j * S$;
      $WAdd_C = E_C$;
      $@Weight =$
       $WAdd_W + WAdd_H * F + WAdd_C * F^2 + k * N_{Channel} * F^2$;
    }
  }
}

$N_{Channel}$: is the number of channels (number of input FMs).

In this algorithm, there are three nested loops iterating over width, height and kernel coordinates. The first loop is going from i_{start} to i_{end} and it represents the width coordinates of the convolution neurons. The second one goes from j_{start} to j_{end} and represents the width coordinates and the last one goes from 0 to $N_{Kernel} - 1$ and represents the kernel coordinates. These indices are used in the following lines to compute first the neuron address and second the weight address. For the neuron address, the computation is straightforward, which is just a mapping computation going from (Kernel, Height, Width) coordinates to an address ranging from 0 to $N_{out} * N_{out} * N_{Kernel}$. The weight addresses computation is done in two parts, first the different coordinates and second the weight address itself (@Weight). Figure 6.6 shows a convolution filter covering an input event (orange cell) and showing these different parameters. In this representation, the convolution kernel is mapped and superposed to the input feature map. The distances between the beginning of the filter and the input event ($WAdd_H$ in width axis and $WAdd_W$ in height axis) represent the coordinates of the synaptic input that are needed to compute the synaptic weight.

The channel's coordinate $WAdd_C$ is simply the input event's channel coordinate E_C. The address of the input synapse conducting the input event to the first convolution kernel is computed by simply mapping the synapse coordinates ($WAdd_W$, $WAdd_H$, $WAdd_C$) to an address ranging from 0 to $N_{Channel} * F * F$. For the next kernel, an offset equals to the number of synapses in one FM, so ($N_{Channel} * F * F$) is added to the previous address. By doing

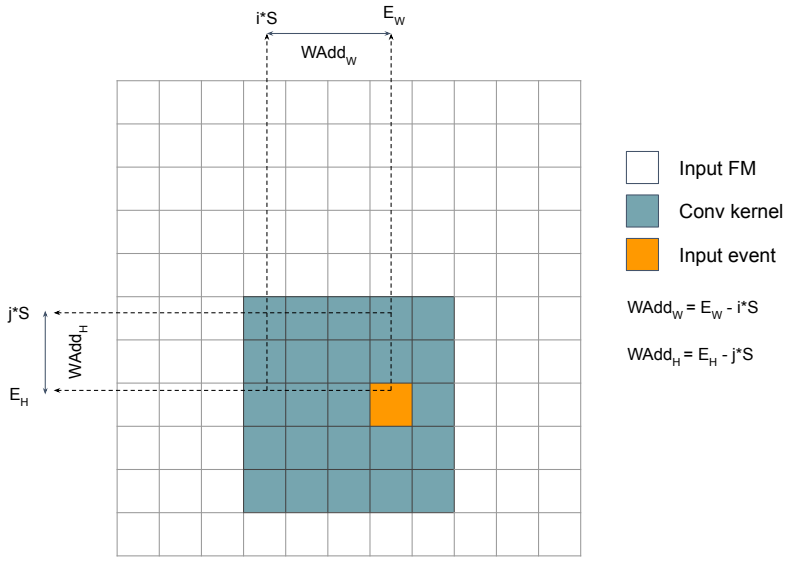


FIGURE 6.6: A convolution filter covering an input event.

this addition incrementally, we get the weight address equals, as depicted in algorithm 2, to: $@Weight = WAdd_W + WAdd_H * F + WAdd_C * F * F + k * N_{Channel} * F * F$.

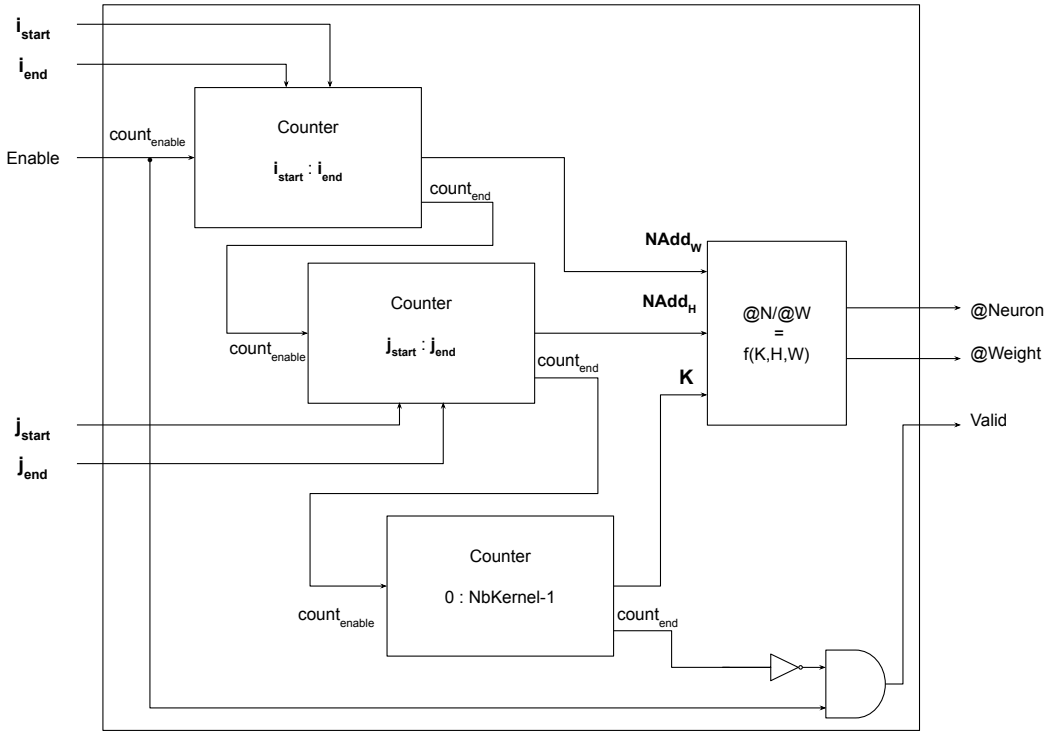


FIGURE 6.7: Convolution address generator module.

This algorithm is then implemented in hardware to get the address generation "AddGen" module that is shown in figure 6.7. The nested for loops in

the presented algorithm are unrolled and represented in the hardware component as three superposed counters that output the width ($NAdd_H$), the height ($NAdd_W$) and the kernel (K) coordinates. These coordinates are then used to compute the neuron and weight addresses (@Neuron and @Weight) as described in algorithm 2. The addresses are then communicated to the computing core of ConvPU that uses them to retrieve from memory the internal potential of neurons and the synaptic weights. In the next section, this computing core will be presented in detail.

6.2.2 Computing core

The second part of the convolution processing unit is the computing core that is responsible for integrating input spiking events and generating output spikes. The computing core is composed of two memory blocks, an integrate-and-fire neuron and an output FiFO buffer that is used to write the output spikes. One of the memory blocks is used to store the synaptic weights and the other is used to register the internal potential of the neurons. The internal structure of the computing core is illustrated in figure 6.8.

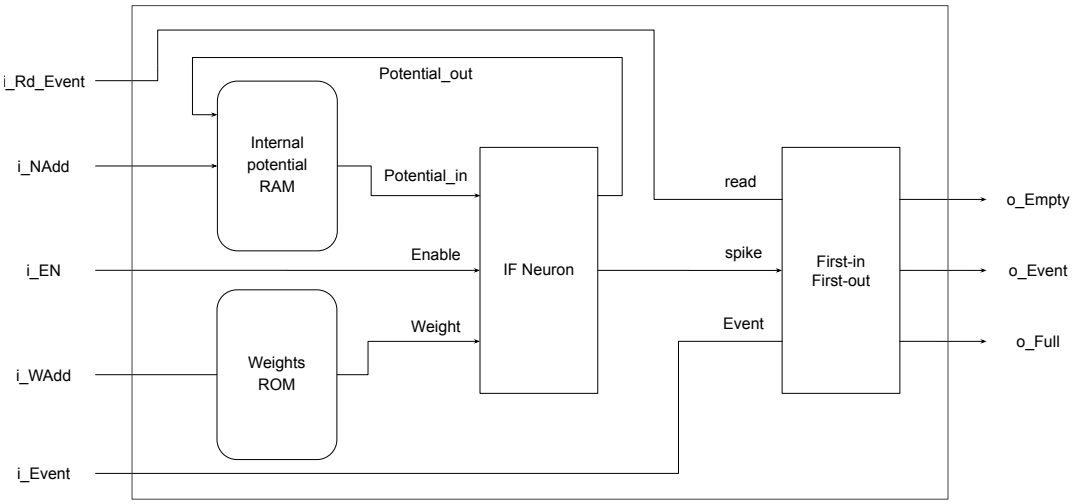


FIGURE 6.8: Convolution computing core schematic diagram.

The computing core is connected to the control module of the ConvPU and the NPU of the next layer. On one hand, it uses the input port "i_Rd_Event" and the output ports "o_Empty" and "o_Event" to communicate with the next layer's NPU. On the other hand, it is connected to the control module through the inputs "i_NAdd", "i_WAdd" and "i_En", and the "o_Full" output port.

The next layer's NPU or component sets "i_Rd_Event" to high when it has read an event from the output FiFo. Indeed, before reading the event on the "o_Event" output port, this NPU verifies first the presence of output events by looking at "o_Empty", referring to the FSM control section 6.2.1. When "i_Rd_Event" is high the content of the FiFo buffer is updated by removing the read event.

When the computing core is enabled by the control module, the input i_En is set to high and the data present on the other inputs are valid. First, using the neuron address (i_NAdd) the core accesses the RAM memory block and retrieves the internal potential of the neuron. In the same clock cycle, the weight's address (i_WAdd) is used to get the synaptic weight from the Conv_ROM block. Second, the IF-Neuron module integrates these weight and internal potential data and applies the integrate-and-fire rule to them. The neuron module accumulates these synaptic amounts and checks if the result is higher than the threshold. In this case, a spike is generated and the internal potential is reset otherwise the content of o_Spike is low and the potential is kept at its state after the integration. Afterwards, the new potential is written back to the RAM and the neuron's output spike (o_Spike) is used as a write-enable of the FiFo block. The input data of the FiFO is i_Event which represents the coordinates of the logical neuron that has been computed in the IF-Neuron module. Therefore, if the neuron's output o_Spike is high the coordinates present in i_Event input are written to the FiFo buffer.

6.3 Pooling Processing Unit – PoolPU

The second layer type we deal in this work is pooling layers that are widely used with ConvNets to reduce the size of convolutional layers. In this section, we present the Pooling Processing Unit (PoolPU) that is a hardware implementation of pooling layers found in spiking ConvNets. The ConvPU module is designed with a structure similar to ConvPU with two submodules, one for control and the for computing. In this work, due to its large use compared to average pooling we implemented max-pooling. However, due to the modular structure of the PoolPU it would be easy to change the pooling function to average.

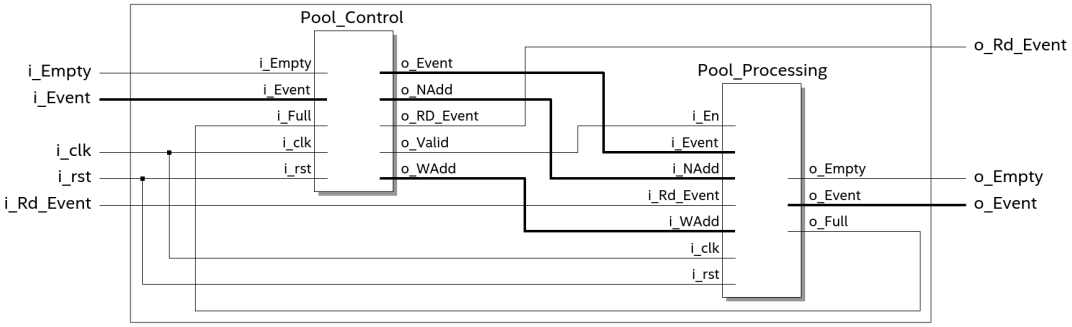


FIGURE 6.9: Pooling Processing Unit schematic diagram.

Control module

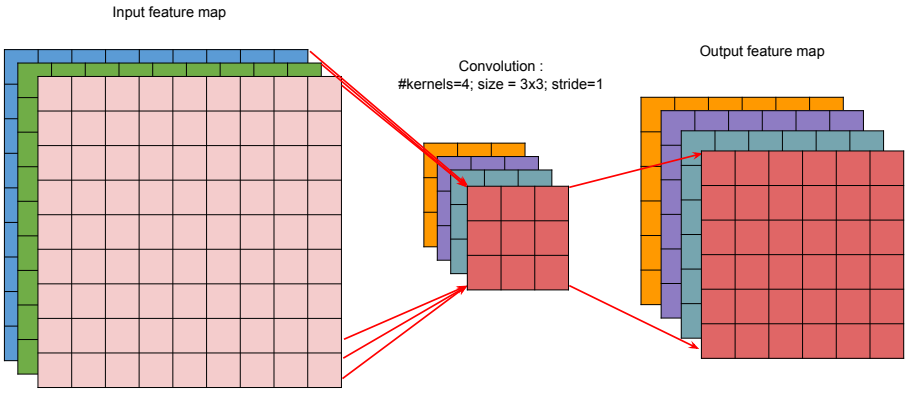
The control module is composed of an FSM and an address manager exactly like in convolution control module. We used the same state machine as in convolution with the pooling control unit. However, with the address manager the same "AddCompute" component is instantiated and connected to an address generator that is quite different. Indeed, with ConvNets the number of pooling kernels is equal to the number of channels and the connectivity of these kernels to the input feature maps is different from convolution layers, refer to figure 6.10. In pooling layers, each kernel is connected to only one channel of the input feature maps whereas a convolution kernel is connected to all the input channels.

Algorithm 3: Pooling layer's address generation algorithm

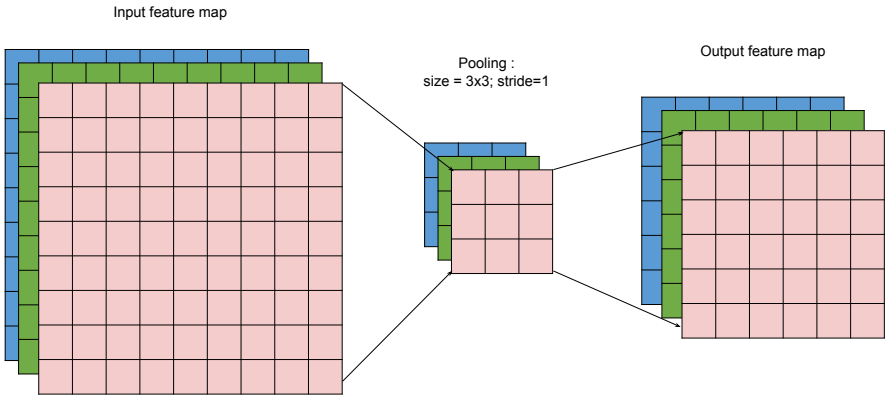
for ($i = i_{start}$; $i \leq i_{end}$; $i++$ **)** {
  for ($j = j_{start}$; $j \leq j_{end}$; $j++$ **)** {
    $@Neuron = i + j * N_{out} + E_C * N_{out}^2$;
    $Synapse_W = E_W - i * S$;
    $Synapse_H = E_H - j * S$;
    $@Synapse = Synapse_W + Synapse_H * F + @Neuron * F^2$;
  }
}

Therefore, we have designed a quite different algorithm for address generation to be used pooling layers, this is depicted in algorithm 3. If we compare this algorithm to the convolution one (algo. 2), we observe two differences concerning the number of loops and the way addresses of neurons and weights are computed.

Since each pooling kernel is connected to a unique input channel, an input event will trigger neural computations of neurons belonging to a single feature map as shown in figure 6.10b. Therefore, the third loop in algorithm 2 is



(A) Convolution



(B) Pooling

FIGURE 6.10: Convolution versus pooling features mapping.

not necessary in this case because only neurons of one kernel are triggered.

For the computation of the pooling neuron address, we replace "k" by "E_C" because the neurons that will process the input event are belonging to a kernel that is in the same depth-level as the input event's channel (E_C).

Indeed, in pooling layers we do not have synaptic weights since the pooling neurons are doing only maximum or average functions that don't require parameters. Thus, here we deal with synapse addresses ($@Synapse$) instead of weight addresses. These synapse addresses are used by the computing core to determine for each neuron which synaptic input is the most spiking. As mentioned earlier, a pooling kernel is connected to only one channel and thus has only width and height coordinates. Therefore, the channel's coordinate of the synapse ($WAdd_C$ in algo. 2) does not exist. Thus, to compute the synapse address only width and height coordinates are used ($Synapse_W$ and $Synapse_H$). Consequently, the synapse address $@Synapse$ is a result of adding its address in a single 2-dimensional kernel ($Synapse_W + Synapse_H * F$) and

an offset equals to $(@Neuron - 1) * F^2$. Indeed, each pooling neuron has F^2 input synapses, thus the offset $(@Neuron - 1) * F^2$ is the number of synapses preceding the synapses of the actual neuron at the address $@Neuron$.

Computing core

The Max pooling in formal is applied to activities of a neural group which is different from spiking domain because the spiking neurons output spikes instead of activities. With the formal CNNs, a max pooling cell delivers the maximum activity of an input group of neurons representing its receptive field. With the spiking CNNs, instead of finding this maximum activity, the pooling cell searches for the most active input neuron. The activity of a formal neuron is equivalent to a spiking rate of an SNN's neuron. Thus, the most active input neuron represents the neuron having the highest spiking rate compared to its neighbors. To do so, the pooling neuron records a spiking rate for each input neuron by assigning an activity rate for its input synapses. Figure 6.11 shows the internal structure of the computing core that is used to represent the pooling neuron in the hardware architecture.

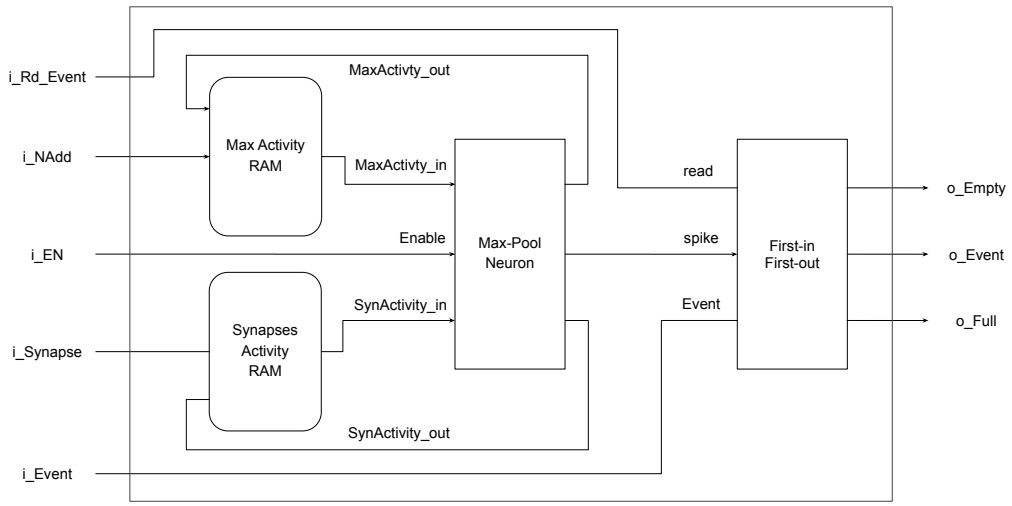


FIGURE 6.11: Pooling computing core internal structure.

The pooling computing core is composed of two read/write memory blocks, the Max-Pool neuron and an output FiFo buffer. When an input spike arrives, the computing core receives the corresponding pooling neuron (i_NAdd) and input synapse ($i\_Synapse$) addresses and the Max-Pool neuron is enabled ($i\_En =' 1'$). The pooling neuron address is used to access the Max-Activity RAM to get the activity of the synapse with the highest spiking rate. The synapse address is used to retrieve the activity of the input synapse in the

Synapses Activity RAM. Afterwards, the Max-Pool neuron increments the activity of the input synapse ($SynActivity\_in + 1$) and compares it to the neuron maximum activity ($MaxActivity$) to emit an output event if the activity is higher or remains silent if it is not the case. The Max-Pool cell writes the updated input synapse activity to Synapses Activity RAM ($SynActivity\_out = SynActivity\_in + 1$). If the Max-Pool neuron has emitted a spike, the new maximum activity of the pooling neuron is changed by the updated input synapse's activity ($MaxActivity\_out = SynActivity\_in + 1$). Otherwise, the pooling neuron keeps its previous maximum activity ($MaxActivity\_out = MaxActivity\_in$). Finally, similar to the convolution computing core, the output spike of the pooling cell (*spike*) is used as a write-enable to the FiFO buffer. In the case there is an output spike ($spike =' 1'$), the pooling cell coordinates present in the *i_event* port are written in the FiFo.

6.4 Fully-connected Processing Unit

In this section, we present the "FcPU" unit that is dedicated for processing a fully-connected layer of a spiking CNN that is inferred in hardware using our architecture. Indeed, in fully-connected layers, which are the third CNN's layer type that we are dealing with, each neuron is connected to all the previous layer neurons. The FcPU processing unit role is to represent these fully-connected neurons. To do so, it follows the same structure philosophy as convolution and pooling processing units, where it also has a control module and a computing core with similar roles. Here, we will briefly describe the internal structure of this FcPU with emphasis on the changes made to suit the FC layers. In the following subsections, we will present the two components of this processing unit.

Control module

Similar to the other layers, we used a control module to manage a fully-connected layer in hardware represented by its processing unit FcPU. This control module is composed of a finite state machine and address management module. Due to the connectivity of this type of layers, we do not have an address compute module in the composition of the address manager. Therefore, the address manager is simply the address generator.

This change at the address management level implicates a modification in the state machine, where the address computation state is removed. On figure

6.12 a graph representation of the fully-connected FSM is illustrated showing its different states and transitions. The FSM transitions from a state to another following the conditions are listed in table 6.2.

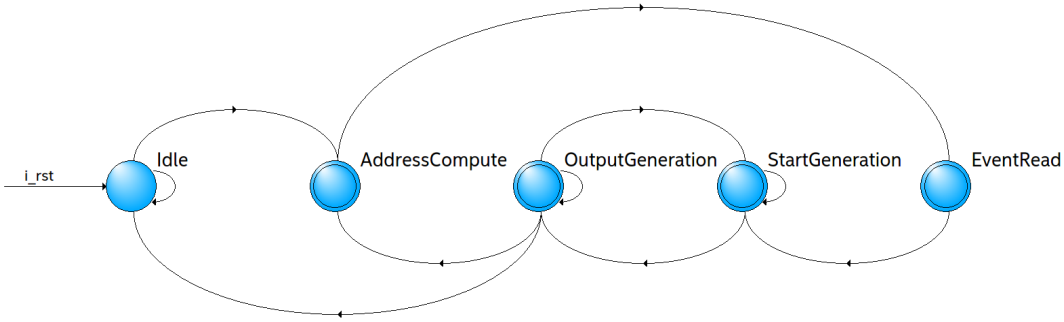


FIGURE 6.12: A graph representation of the fully-connected finite state machine.

TABLE 6.2: Fully-connected FSM states transitions.

Source state	Destination state	Condition
AddressCompute	EventRead	None
EventRead	StartGeneration	None
Idle	AddressCompute	(!Empty).(!Full)
	Idle	(Empty)+(Full)
OutputGeneration	Idle	(Empty).(!Valid).(!Full)
	OutputGeneration	(Valid).(!Full)
	AddressCompute	(!Empty).(!Valid).(!Full)
	StartGeneration	Full
StartGeneration	OutputGeneration	!Full
	StartGeneration	Full

The address generator module used within this control module is composed of a counter and a block indicating the logical neuron's address that is currently computed and the address of the input synapse. The counter is going from 0 to N-1, with N the number of the Fc-layer's neurons, which correspond to the logical neuron addresses. The block that is calculating the synaptic weight addresses takes as arguments the output of this counter and the address of the input neuron encoded in *i_event* signal to compute the weight address. Algorithm 4 details the behavior of the module by showing, on one side, the counter represented by the loop, on the other side, the calculation of the synapse address.

Algorithm 4: Fully-connected layer's address generation algorithm.

$NAdd_{IN} = Map(Event)$;
for ($i = 0$; $i \leq N - 1$; $i++$ **)** {
    $NAdd = i$;
    $WAdd = i + N * NAdd_{IN}$;
}

Computing core

The computing core used with a fully-connected layer has exactly the same structure as the one used with convolutional layers. Indeed, it is composed of a ROM to store the synaptic weights of the layer, a RAM to write and read the activities of the logical neurons of the layer, an IF neuron and a FiFo to record the output events.

6.5 Experiments and results

In this section, we will present the results of a functional validation of the architecture and other results concerning the occupation of FPGA resources. First, we use the ModelSim software to perform the functional validation by running simulations on the MNIST database. Then, we will present resource utilization results obtained through RTL synthesis using the Quartus tool.

6.5.1 Functional validation

In this part, we validate the functioning of the architecture by simulating a hardware classification with the CNN spiking architecture using ModelSim. With this tool, we visualize different signals of the modules that compose the architecture and check their proper behavior. Indeed, we first validate the elementary modules on synthetic data. Then, we proceed to the validation of the complete architecture which is, in other words, the assembly of these modules. The purpose of the functional validation is to resolve possible malfunctions before moving on to hardware synthesis and on-chip operation.

We will now show signal captures of different regions of the in-phase architecture of a simulation of a classification of a MNIST image.

First, we visualize the top-level of the architecture which shows the input of the image pixels, the activity of the neurons of the output layer, the winning

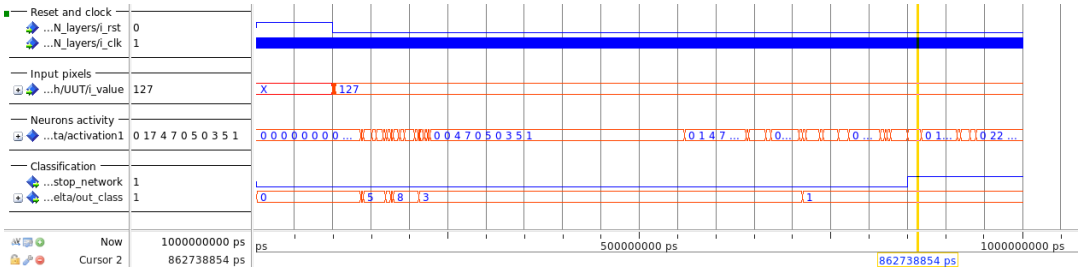


FIGURE 6.13: Example of simulation under Modelsim tool of the architecture showing the classification of the number "1".

class and the signal that stops the processing in the network. In this example, illustrated in figure 6.13, we show a classification of an MNIST image that contains the number "1". At the beginning, we observe the pixels that arrive on "i_value" just after the release of the reset (goes to "0"). Then, these values are processed by the architecture where the activation of the output neurons changes progressively. By the end, the number of activations of neuron 1 exceeds all the neurons by "10" which is the "terminate delta", this will activate the stop network signal while the class winning signal is displaying "1".

Spiking generation

In figure 6.14, the inputs/outputs and internal signals of the spikes generation module are illustrated. At the beginning of this diagram, the arrival of the pixels is observed, followed by a time interval where the signals are unchanged. During this period, the pixels are loaded and the various memories are initialized. Then, the empty signal is reset to "0" after which spikes are generated.

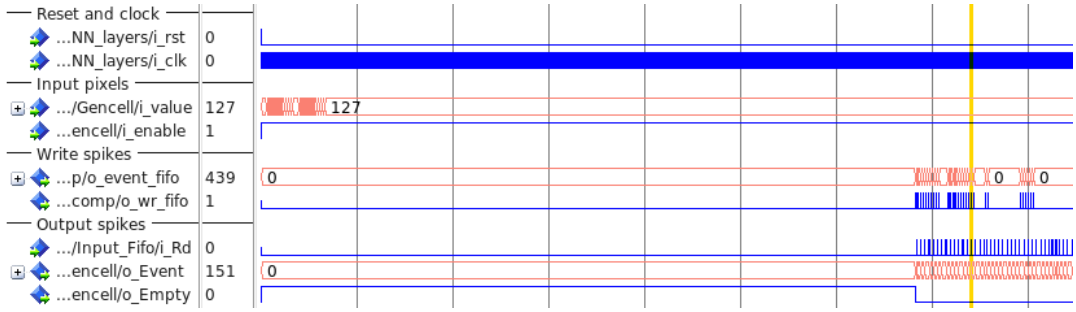


FIGURE 6.14: Spike generation simulation.

A zoom-in on this phase in figure 6.15 is showing a generation of two spikes. These two spikes are generated by pixels at addresses 37 and 38 where we

notice that the signal "o_wr_fifo" goes to "1" enabling the write of these addresses in the FiFo. The ConvPU of the first hidden layer reacts to this generation with the reading of a first spike by setting the "i_RD" to "1", this results in the update of "o_event" from 37 to 38 for the next reading.

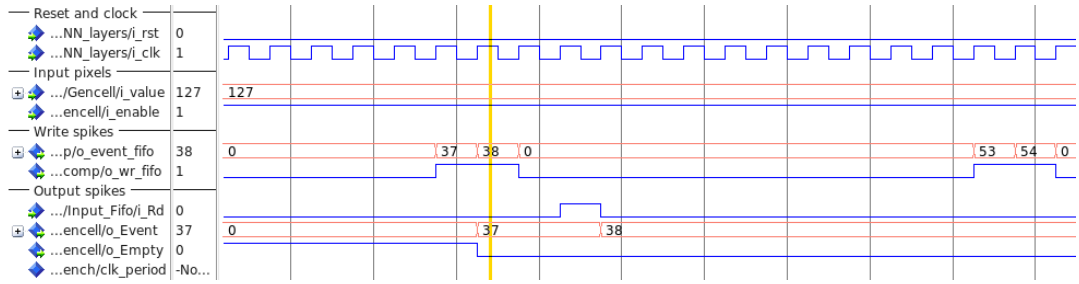


FIGURE 6.15: Zoom-in on spike generation simulation.

Convolutional layer

As indicated in section 5.3, the "spike gen" module is connected to the first NPU that represents the first hidden layer of the spiking CNN. In this case, the first layer is a convolution which is represented by a ConvPU. Simulation results are shown in figure 6.16 that capture the behavior of this ConvPU I/Os when receiving input spikes resulting from the "spike gen" cell. We have previously seen that this cell has generated two spikes with the addresses 37 and 38. These two addresses appear on the "i_event" input just after the "i_Empty" input goes from 1 to 0. The spiking event "37" is then processed by the ConvPU, which, in its turn, emits output spikes that are reflected by the transition from 1 to 0 in "o_Empty" output and by the state change of " o_Event" output. This process is then repeated for all the input spikes and by all the NPUs of the spiking CNN.

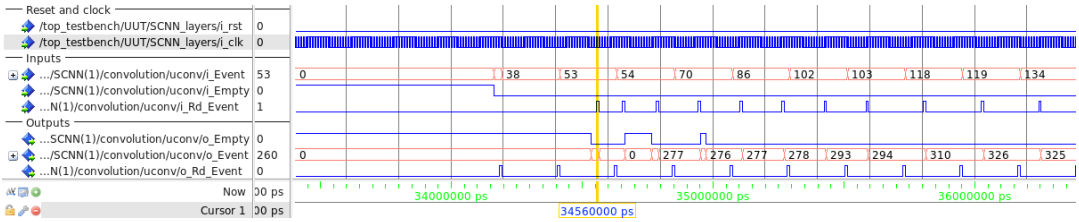


FIGURE 6.16: Convolution layer simulation.

Control module

The ConvPU unit, as previously mentioned, contains a control module and a computing core. The control module consists of a state machine "FSM"

and an address manager. Figures 1 and 2 show diagrams resulting from the simulation of the two modules.

The diagrams in Figure 1 show the behavior of the FSM by indicating the PresentState and NextState depending on the content of the inputs and outputs. Indeed, the desired behavior of the FSM, described in section 6.2.1, can be observed on these diagrams. At the beginning, the two states (PresentState and NextState) are at "Idle" and when "i_Empty" goes to zero the FSM process described later is triggered.In this process, NextState links the states: "AddressCompute" - "EventRead" - "StartGeneration" - "OutputGeneration" in order. This series of states is observed one cycle later in PresentState. This state (PresentState) is then used to indicate the contents of the outputs. It is clear that the outputs correspond to the displayed state, for example: "o_AC_Start" changes to 1 when the displayed state is "AddressCompute" which is used to activate the address generation module.

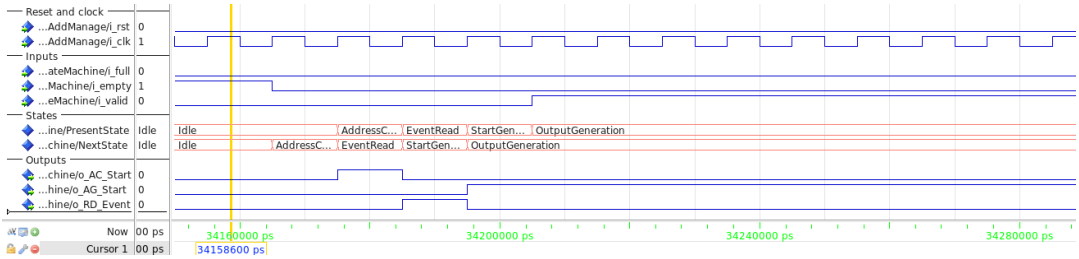


FIGURE 6.17: FSM simulation.

Figure 6.18 shows a simulation of the operation of the address generation module. This module is enabled by the FSM which changes the state of "i_AC_Start" to compute the address start indices that will be generated. This generation begins when the input "i_AG_Start" is changed from 0 to 1. At this point, the addresses are sequentially written in "o_NAdd" and "o_WAdd" and then transmitted to the processing core. Notice that during this generation, the output "o_Valid" displays 1 until all the logical neuron addresses of the layer have been transmitted.

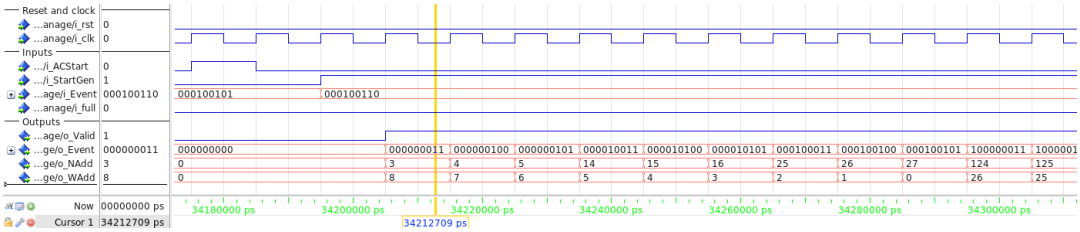


FIGURE 6.18: Address generation module simulation.

Processing core

The second component of the ConvPU is the processing core, which itself consists of a neuron and three memory blocks of different types. Therefore, the most interesting module to visualize and check its correct operation is the hardware neuron. In this context, we show a simulation of a part of the operation of this neuron in figure 1. Indeed, from the moment the input "i _En" goes to "1" the neuron is activated. In this case, the neuron adds the value of the synaptic weight with the internal potential of the neuron present on "i_Weight" and "i_Potent" consecutively. In the figure, where the vertical indicator in yellow is positioned, the neuron emits two spikes. Indeed, the sum of the "weight" and "potent" observed at that moment exceeds the threshold, which causes the emission of these spikes.

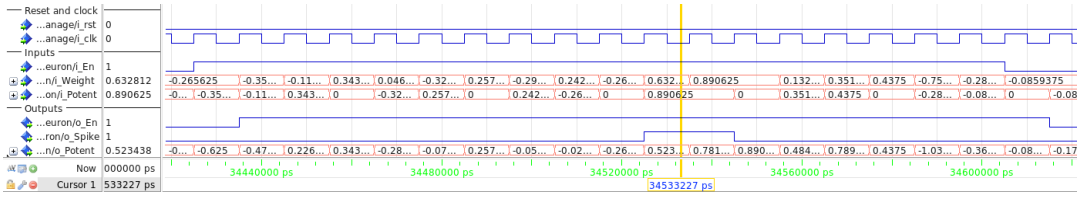


FIGURE 6.19: Simulation of hardware neuron.

On-chip validation

An on-chip functional validation is performed with the spiking CNN hardware architecture. This on-chip validation is carried in collaboration with Edgar Lemaire, who is a second year student in the eBrain team. The purpose of this collaboration is to use this spiking CNN hardware architecture in the context of his thesis project by first applying it to spacial applications and second upgrading it to perform online learning (STDP) (Lemaire et al., 2020). Indeed, in this context of satellite applications, a platform running on a Xilinix FPGA board and which can host an AI accelerator has been developed. Figure 6.20 represents the structure of this platform by showing its components, the spiking CNN accelerator is in the right-top of the figure. Several AXI-bus interfacing modules are used to connect the PS and PL parts and to manage memory accesses. The Xilinx® Deep Learning Processing Unit (DPU), which is a programmable accelerator for formal CNNs targeting FPGA devices, is present in the platform for the context of Edgar Lemaire thesis, which is to compare it with the spiking accelerator.

The winner class output of the spiking CNN is communicated to the Zynq processor through the AXI bus when the stop network is set to high by the

Terminate Delta module. With this implementation setup, we successfully integrated our Spiking CNN architecture, using Vivado tool, and tested it on the MNIST database using the LeNet-5 network. The results were equivalent to those obtained on the N2D2 tool.

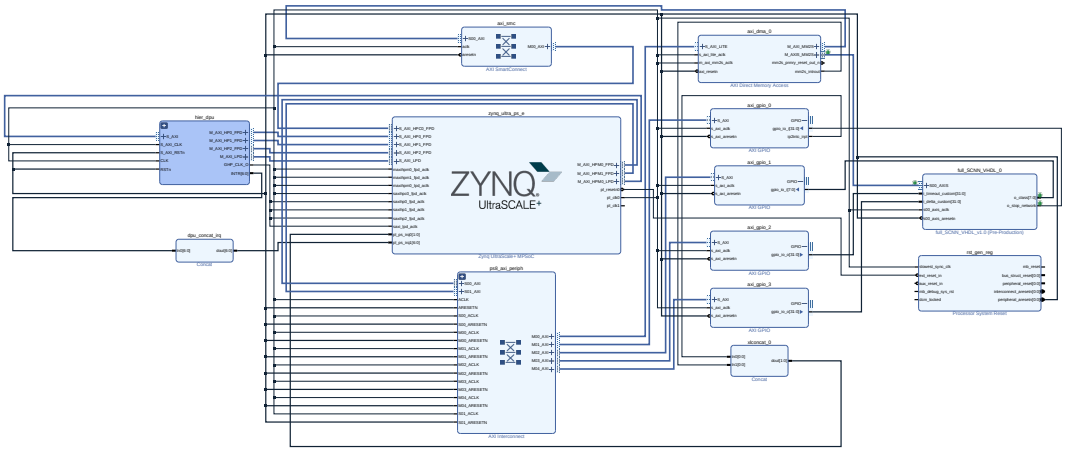


FIGURE 6.20: On-chip validation platform structure (generated from Vivado tool).

6.5.2 Hardware cost

Once the functional validation of the time-multiplexed architecture for spiking CNNs in both ModelSim and on-chip simulation, we were interested in its hardware resources occupation. To do so, we have made several synthesis with different topologies by varying the number of FC and Conv layers. The results of those synthesis, obtained with the Quartus tool and by instantiating the Intel Cyclone V board (5CGXFC7C7F23C8), are given in the tables 6.3 and 6.4. On one hand, the table 6.3 represents the results of the occupation of the spiking CNNs that have two convolutional layers followed by a different number of fully connected layers. On another hand, the table 6.4 presents the occupation results of other spiking CNNs that have a different number of convolutional layers followed by two fully-connected layers. The results are presented in terms of logical occupation and memory block and register utilization. We used this Cyclone V board because it is large enough to contain a large amount of synaptic weights and thus be able to do several RTL synthesis. From these result tables we have plotted curves associating the resource occupation to the size of the network, these curves are illustrated in the figures 6.21 and 6.22.

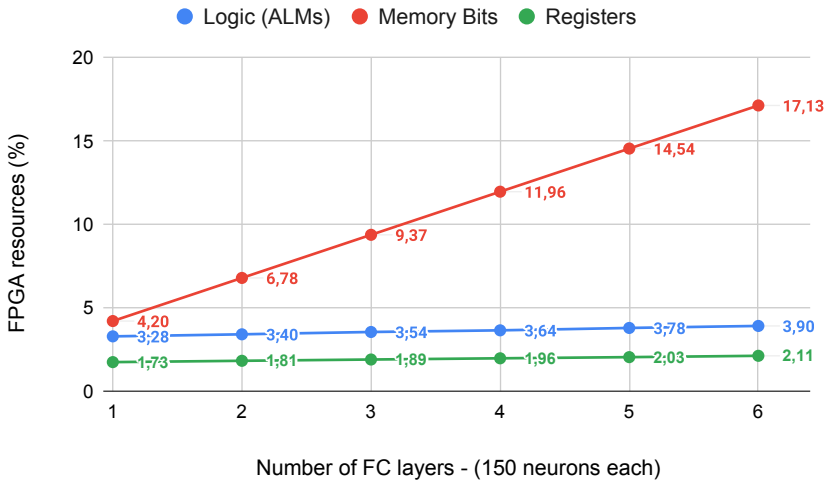
TABLE 6.3: FPGA resources occupation of Spiking CNNs with different number of fully-connected layers. Topology : "28x28 - 6c3s2-6c3s2 - N(FC) - 10" with N the number of FC layers that are composed of 150 or 300 neurons.

FC neurons	FC layers	Logic - ALMs	Memory Bits	Registers
150	1	1852	294846	3917
	2	1921	476546	4096
	3	2000	658246	4269
	4	2056	839946	4434
	5	2137	1021646	4592
	6	2205	1203346	4773
300	1	1856	567296	3938
	2	1951	1290246	4130
	3	2053	2013196	4308
	4	2138	2736146	4492
	5	2238	3459096	4678
	6	2326	4182046	4847
Maximum		56480	7024640	225920

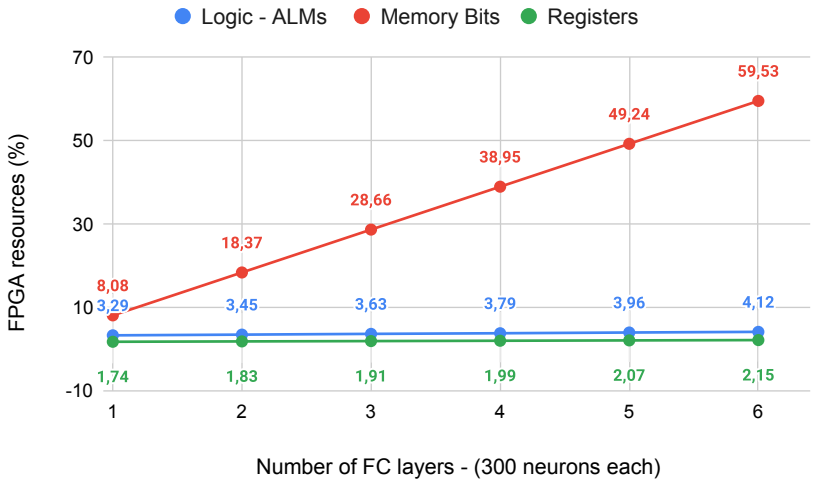
TABLE 6.4: FPGA resources occupation of Spiking CNNs with different number of convolution layers. Topology : "28x28 - N*(6c3) - 80 - 10" with N the number of convolution layers that are composed of 6 kernels of size 3x3 and a stride of 1 or 2 (K=6, F=3, S=1 or 2).

Stride	Conv layers	Logic - ALMs	Memory Bits	Registers
1	1	1862	2644942	3752
	2	1957	2291832	4014
	3	2096	1965026	4303
	4	2214	1664908	4572
	5	2351	1391862	4845
2	1	1756	673626	3708
	2	1838	167676	3911
	3	1927	47830	4070
	4	2012	39200	4193
Maximum		56480	7024640	225920

The figure 6.21 shows the percentages of resources, logic (ALMs), memory blocks and registers, occupied by different spiking CNNs with a 2 convolutional layers and a varying number of FC layers. The data on this figure is represented as the utilization percentage in relation to the number of FC layers. The first sub-figure 6.21a shows spiking CNNs with FC layers of 150 neurons and the other shows other networks with FC layers of 300 neurons. On these graphics, we observe two phenomena: first, the logic and register



(A)



(B)

FIGURE 6.21: FPGA resources occupations versus the number of FC layers.

occupation increases marginally as the number of layers increases; the second point concerns the memory occupation, here, we notice that the memory amount increases considerably with the size of the network. The first phenomenon is due to the fact that the architecture is implemented in a time-multiplexed fashion. Appending a layer in a network is equivalent to adding an NPU, which does not occupy a lot of resources, in the CNN architecture. Nevertheless, in terms of memory (the second phenomenon), adding FC layers is very expensive. Indeed, this type of layers, by the fact that they are fully connected, requires a large memory space to store the synaptic weights. Note that, as you can see, the scales change between the two sub-figures due to the number of neurons in the FC layers, but the tendencies of the curves

do not.

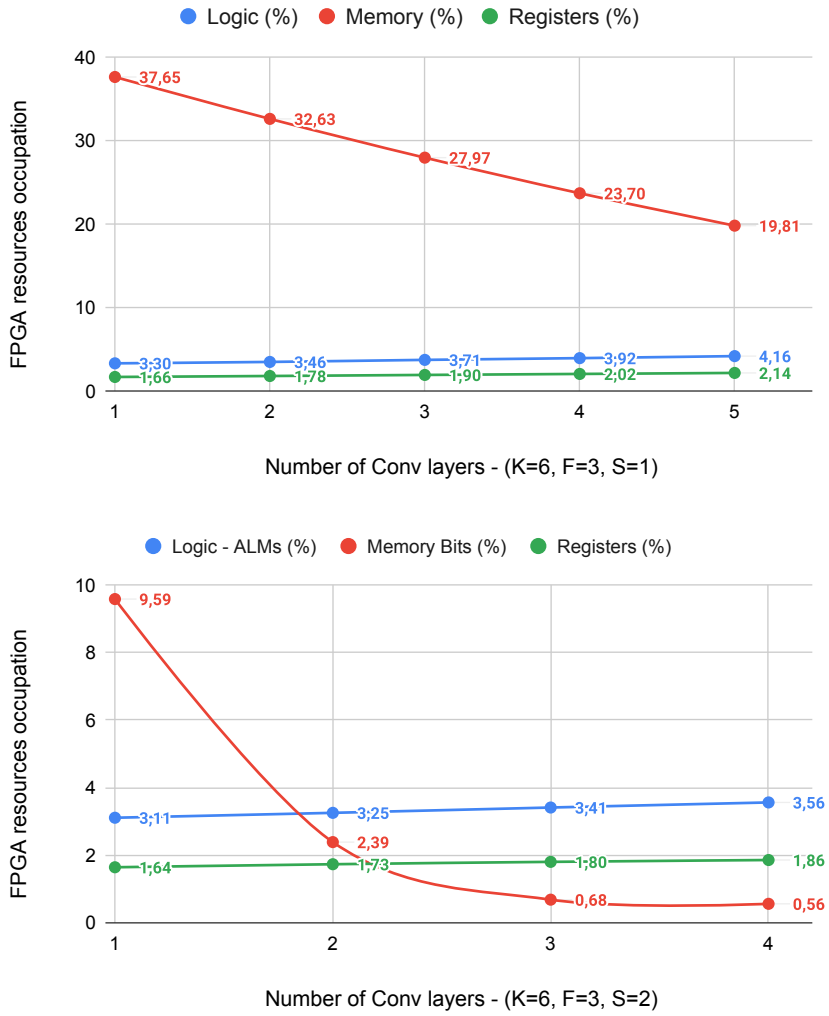


FIGURE 6.22: FPGA resources occupations versus the number of convolutional layers.

The other curves in the figure 6.22 show the same phenomenon concerning occupation in logic and registers. However, we find a completely different scenario concerning the evolution of memory occupation. In this case, the curve plotted to represent memory utilisation (in red) decreases proportionally with the increase in the number of convolutional layers. Indeed, the addition of convolution layers decreases the size of the features maps and due to weight-sharing the memory needed to save the synaptic weights is not increasing a lot. Therefore, the last convolution layer continuously shrinks. Consequently, the number of connections between the last convolutional layer and the FC layer is continuously decreasing. This part of the network is the most memory space demanding due to its fully-connected

aspect to the output FM. For this reason, the memory required to store the synaptic weights decreases incrementally.

6.6 Discussions

Event-based computing

The spiking CNN architecture presented in this chapter is implemented in a completely event-based fashion. This event-based aspect is well-tailored to the spiking data stream found with SNNs. Furthermore, this architecture is conceived in a layer-level pipelined structure with computational resources in each layer. The computational resource used with each layer is a neural processing unit capable of processing a number of logical neurons belonging to a single layer. These two aspects, event-based computing and pipeline structure, when combined together in the same hardware architecture effectively take advantage of the sparsity of spiking data. In addition to the efficiency of the spiking computation found in IF neurons when compared to perceptrons, the amount of operations in the network can be reduced in SNNs due to the sparsity of spiking data. In figure 6.23, the amount of events read per synapse in average per pattern of several SNNs and ANNs with the same topology (with one hidden layer) is shown. For ANNs, obviously one event is read per synapse since analog data is transmitted between neurons. On the other hand, for SNNs, less data is used to do classification due to the sparsity of spiking data.

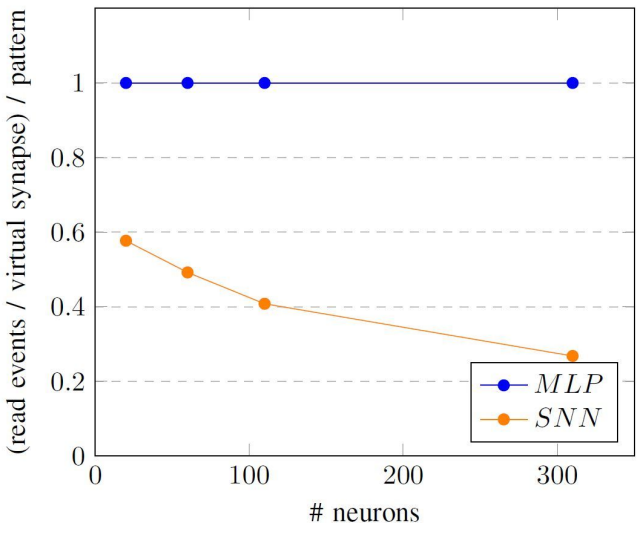


FIGURE 6.23: A comparison between the number of events read per synapse in SNNs and ANNs.

In order to take advantage of this aspect, we must have computing resources in each layer (pipeline) and use an event-based communication protocol to propagate spiking events from one layer to another to the "terminate delta" module in a short time. A spike, if it is sufficiently powerful, is able to propagate from the first to last layer in 3 cycles. Assuming that we have a TD = 4 and that 4 'strong' spikes arrive one after the other, the classification can be established 4 cycles after the arrival of the first spike (3 cycles), which results in a total of 7 cycles. This scenario is only possible with an event-based and pipelined architecture, ensuring a path from input to output for strong spikes and a communication protocol of events from layer to layer.

Spike generation influence : towards EBS

In the current architecture, we use an input spike generator to transform analog data into trains of spikes. This module is essential to process such data provided from the classical sensors found in most applications. As indicated in the chapter 4, this module, in addition to representing an additional hardware cost, influences the accuracy performance and computational efficiency by affecting the spikes propagation. Therefore, clearly this is the part that needs to be adjusted to fully benefit from the sparse nature of spiking data. To do this, two solutions are possible: either , as it is done in this thesis, continue to optimize the generation of spikes to get closer and closer to temporal coding; or by adopting an event-based sensor that is naturally characterized by delivering sparse data (Delbrück et al., 2010). In contrast to classical cameras, which output a succession of discrete frames, an Event-Based Sensor (EBS) emits a continuous flow of events: each pixel outputs a spiking event whenever an edge crosses its receptive field. In other words, an EBS sensor outputs a flow of spikes representing the movement happening in its vision field. Moreover, Farabet et al., 2012 have shown that a fully event-based frame-free processing flow would bring input-to-output pseudo-simultaneity, that is, real-time processing ability. Thus, we expect that SNNs combined with asynchronous sensors would be very well suited to embedded artificial intelligence for real-time video recognition and classification.

Pooling Processing Unit

As we have seen in the results section, the pooling layers were not present in the SNNs that were synthesized. We replaced these layers by using a stride in the convolutional layers. In fact, the PoolPU, presented in the section 6.3,

uses two memory blocks: one to store the number of spikes received by the most active synapse; the other to store the number of spikes received by each synapse. These two parameters are then used to determine the most active synapse to be the maximum input of the pooling node. However, this method is not the most optimal solution and including it in the current architecture would bias the results. It is planned to first implement a simplified version of pooling proposed in Vincent Lorrain's thesis (Lorrain, 2018) which is much less expensive in memory and then synthesize both versions to compare their performances.

6.7 Conclusion

The SNNs are spatio-temporal algorithms that are characterized by event-based computing with a very sparse data processing. In this chapter, based on these two aspects and on the conclusions of the previous chapters, we have designed an architecture dedicated to SNNs (convolutional and Fc-based classifiers) and adapted to event-driven computation of the extremely sparse data. Such an architecture allows a hardware acceleration of the inference phase of spiking CNNs, while being able to be configured from several parameters such as: network topology, neural coding, data encoding precision, etc. For benefiting from the spatio-temporality of SNNs, we have selected an event-based communication and a layer-level pipeline structure. Event-based communication allows for a coherent computation both temporally and spatially, i.e. spikes are processed according to their arrival order (time) and by the appropriate neurons that are located at specific locations in the network (space). The choice of the layer-level pipelined structure is adopted to ensure an input-output data-path that accelerates the classification by performing fewer computations and thus taking advantage of the sparse nature of spiking data. This system could be improved by moving to a fully event-based system with EBS input sensors leading to the removal of the spike generator. Such a solution appears to be very promising because it would provide a fully bio-inspired system.

Chapter 7

Conclusion

7.1 Restatement of the objectives

In this thesis, we have studied spiking neural networks and neuromorphic computing to achieve our objective of designing a neuromorphic hardware architecture for processing embedded AI applications, that respects the following criteria in terms of configurability, programmability, scalability and genericity. Where:

- configurability : no dependency of the architecture on a specific application but can be configured for any application;
- programmability : this means that the SNN hardware architecture is programmable using any machine learning framework;
- scalability : depending on the application and the available hardware resources, the architecture adopts the suitable neural coding schemes and architectural models;
- genericity : the architecture must have the ability to include any feed-forward neural network topology with different sizes and types;

To achieve these objectives, we must also implement a design flow for the exploration and implementation of neuromorphic architectures that are adapted to embedded systems.

7.2 Summary and review of the work done

Now that we have restated the objectives, it is time for a synthesis of the thesis chapters.

First, we started the manuscript with an introduction chapter (chap. 1) that presented the context, the objectives and the contributions that are expected from this thesis.

In chapter 2, we have reviewed the state-of-the-art describing the generalities on neural network models and on the neuromorphic architectures that were used as a basis for selecting some technical options such as the neural network family, the neuron model and the hardware target. In this context we have chosen to use IF neuron-based SNNs because of their computational efficiency and compatibility with embedded systems. For the training of these neural models, we have selected the ANN-SNN conversion approach while keeping the option to switch to another training technique such as off-line spike-based learning or STDP. The reason for this choice was that, when we started the thesis, this conversion approach was the most mature approach and that the "N2D2" tool facilitated the training step using this method. In order to reduce the energy consumption of the SNN during a hardware deployment, we have also explored various methods of neural information coding. For the hardware implementation, we targeted the FPGA because it fits our reconfiguration, programmability and exploration objectives. In addition, we analyzed different architectural settings for computing and memory distribution.

Next, we presented in chapter 3 a design flow framework for the exploration of neuromorphic hardware. In addition, we provided analytical results estimating the hardware cost of some architectural models for implementing the SNNs in hardware. Actually, the results showed that the latency and surface area of SNN architectures are strongly influenced by the computing parallelism and that the target's memory capacity is a limiting factor in the design of such systems.

In chapter 4, different neural coding schemes for spike generation with SNNs have been explored. The aim of this exploration was the selection of an efficient neural coding strategy that is suitable for low-power embedded systems, by performing classification tasks with a state-of-the-art accuracy while minimizing the quantity of spiking events. For this purpose, rate-based coding and time-based coding have been set as references for accuracy and spiking activity. First, rate-based coding is the most widely used technique with SNNs to ensure state-of-the-art accuracy performance, but at the same time, it involves the use of a huge quantity of spiking data. Second, time-based coding is used as the reference for spiking activity since it is characterized by

the use of a smaller amount of spiking events to encode data. From an energy efficiency point of view, this second reference is very interesting, however, in terms of accuracy, it does not reach the state-of-the-art level (using ANN-SNN conversion approach). Therefore, these two references are used to explore other models that aim to meet both activity and accuracy requirements. In this context, we confronted two novel methods, namely "First Spike" and "Spike Select", with the rate coding "Jittered Periodic" method. These two new methods are actually derived from “Jittered Periodic” with the aim of reducing the activity in the SNN. The First Spike method drastically reduces the number of spikes and achieves high accuracy performance on simple topologies and on MNIST dataset. However, when using more complex topologies such as CNNs and on other datasets such as GTSRB, only Spike Select and Jittered Periodic coding schemes reach the state-of-the-art performance of machine learning. In addition, we have analyzed the activity generated by these two methods and we have seen that Spike Select reduces it drastically. Moreover, we observed a regulation in the distribution of the spikes on the SNN layers with most of them generated in the first layers and the remaining ones in the deep layers.

In chapter 5, we have addressed the low-level exploration level of the design flow framework presented in chapter 3. Within this part, we have described at RTL level three different architectural models for SNNs : Fully Parallel Architecture, Time-Multiplexed Architecture and Hybrid Architecture. These architectures have been then synthesised to get hardware cost estimation results in terms of logic, register, memory occupation and latency. These results have been used to determine the best implementation of SNNs for embedded AI applications and serve for selecting the architectural choices for the final accelerator design. The results on deep fully-connected SNNs have shown that the Hybrid architecture makes a better latency and logic occupation trade-off and this is due to its structure taking advantage of the sparsity of spiking data. This architecture is actually combining both FPA and TMA to process the SNN with a first part implementing a fully-parallel manner and another part implemented in a time-multiplexed way. If we consider the neural coding presented earlier, we observe that the distribution of spiking data using Spike Select fits well this architecture, where the parallel part will represent the first layer, where most of the spikes are condensed, and multiplexed part will represent the deeper layers that have less data to process. Therefore, from this chapter we conclude that both parallelism and multiplexing are important to optimally use the hardware for implementing

SNNs. Moreover, to achieve the scalability purpose, it is crucial to use time-multiplexing which is the best solution to deal with large-scale SNNs.

Based on these two aspects, we have designed an architecture dedicated to SNNs adapted to calculation based on very rare data events. This architecture allows hardware acceleration of the inference phase of spiking CNNs, while being parameterizable by: the network topology, the neural coding, the data encoding precision, etc. To benefit from this spatio-temporality, an event-based communication and a layer-level pipeline structure has been selected. The event-based communication allows a coherent computation both temporally and spatially, where spikes are processed according to their arrival order and by the appropriate neurons located at specific positions in the network. In addition, the layer-level pipelined structure ensures an input-output data path that allows for faster classification by performing fewer computations to benefit from the sparse nature of spiking data.

7.3 Limitations and future directions

The conclusions of this thesis were: the use of the Hybrid Architecture combined with Spike Select coding method makes an adequate resource and latency trade-off for embedded AI; second, time-multiplexing is the architectural model that is fitting the implementation of realistic and large-scale spiking neural networks. Thus, in the last part of the thesis, we have implemented an event-based time-multiplexed architecture for spiking CNNs.

Several limitations have been faced during this thesis both in the neural and the architectural models. In the neural coding exploration step, we have seen that the spike generation influences the amount of spiking data but this may decrease the accuracy of the SNN. Where in some cases we must process more spikes to increase the accuracy. Despite the fact that we reduce the number of spikes when using Spike Select instead of Jittered Periodic, we are still emitting high numbers of spikes.The reason for this limitation is probably the unsuitability of the conversion method that is used for training the SNNs. In fact, this method is more adequate with rate-based coding where a lot of data is generated in order to have SNNs as efficient as the ANNs. One possible solution is to investigate the use of other training methods that are spike-based. For example, to build and train the SNNs we can use the S2Net framework (available in open source online at *S2Net*).

Concerning the hardware implementation of the SNNs, two memory-related limitations have been observed: the storage of the neurons' internal potentials and the memory required to implement pooling layers. Due to the temporal aspect of spiking data, the input spikes of a given neuron arrive at different dates, whenever a spike is coming the neuron is updating its internal potential. Therefore, this internal potential must be stored during the whole operation of the architecture and cannot be reused by another neuron, like with ANNs, where the neuron does its computation in one time. Therefore, when we consider spiking CNNs a memory with a size equivalent to the size of the output feature maps of all the layers is permanently required. This limitation must be investigated in future works in order to be able to deal with deep neural networks like ResNet. The other hardware limitation concerns pooling layers, where as mentioned in chapter 6, it is needed to determine the most spiking input synapse of pooling node the activity of each input synapse must be registered because of temporality of the spiking data. To overcome this limitation, Lorrain proposed a simplified pooling node to implement a max pooling with a reduced memory usage (Lorrain, 2018). A short-term perspective concerns the integration of this pooling node to PoolPU presented in chapter 6 to extend the presented results to CNNs with pooling layers instead of just using a stride instead.

An additional perspective would be to upgrade the architecture in order to support different levels of parallelism and not be limited to the current hybrid architecture choice with a one-layer fully parallel layer. The idea is to exploit different degrees of parallelism both at network level, with from one layer to several layers being parallel, and at the layer level, with partial parallelism instead of either time-multiplexed (NPU) or fully-parallel (Neural Core). When this perspective is completed, it would be time to evaluate more accurately the energy consumption of the architecture directly on the target chip. Then finally confront this architecture dedicated to SNNs with the others that are based on ANNs (such as (Carbon et al., 2018)). Moreover, a possible improvement of the proposed hardware architecture would be moving to a fully event-based system with input data coming from an Event-Based Sensor (EBS) and thus remove the spike generator and thus having a complete bio-inspired embedded AI system.

Finally, these different perspectives would allow us to practically evaluate our architecture on realistic application contexts such as autonomous vehicles with Renault or satellite systems with Thales.

Bibliography

Abbott, L. (1999). “Lapicque’s introduction of the integrate-and-fire model neuron (1907)”. In: *Brain Research Bulletin* 50, pp. 303–304.

Abderrahmane, Nassim, Edgar Lemaire, and Benoît Miramond (2020). “Design Space Exploration of Hardware Spiking Neurons for Embedded Artificial Intelligence”. In: *Neural Networks* 121, pp. 366 –386. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.09.024.

Abderrahmane, Nassim and Benoit Miramond (2019). “Information coding and hardware architecture of spiking neural networks”. In: *Euromicro Conference on Digital System Design (DSD)*, "1–8".

– (2020a). “Neural coding: adapting spike generation for embedded hardware classification”. In: *International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8.

Abderrahmane, Nassim and Benoît Miramond (2020b). *Hardware Architecture for Spiking Neural Networks and Method of Operating*. International Patent submitted to "Office européen des brevets".

Ackley, David H., Geoffrey E. Hinton, and Terrence J. Sejnowski (1985). “A learning algorithm for boltzmann machines”. In: *Cognitive Science* 9.1, pp. 147 –169. ISSN: 0364-0213. DOI: https://doi.org/10.1016/S0364-0213(85)80012-4.

Agarap, Abien Fred (2018). “Deep Learning using Rectified Linear Units (ReLU)”. In: *ArXiv* abs/1803.08375.

Akopyan, Filipp et al. (2015). “TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, pp. 1537–1557.

Amir, A. et al. (2013). “Cognitive computing programming paradigm: A Corelet Language for composing networks of neurosynaptic cores”. In: *International Joint Conference on Neural Networks*.

Bebis, G. and M. Georgiopoulos (1994). “Feed-forward neural networks”. In: *IEEE Potentials* 13.4, pp. 27–31. ISSN: 0278-6648. DOI: 10.1109/45.329294.

Behrenbeck, Jan et al. (2018). "Classification and regression of spatio-temporal signals using NeuCube and its realization on SpiNNaker neuromorphic hardware". In: *Journal of neural engineering*.

Bellec, Guillaume Emmanuel Fernand et al. (2018). "Long short-term memory and learning-to-learn in networks of spiking neurons". English. In: *Advances in Neural Information Processing Systems*.

Benjamin, B. V. et al. (2014). "Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations". In: *Proceedings of the IEEE* 102.5, pp. 699–716. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2313565.

Bichler, O. et al. (2017). "N2D2 - Neural Network Design and Deployment". In: https://github.com/CEA-LIST/N2D2.

Bohté, S., J. Kok, and H. L. Poutré (2000). "SpikeProp: backpropagation for networks of spiking neurons". In: *ESANN*.

Borst, A. and F. Theunissen (1999). "Information theory and neural coding". In: *Nature Neuroscience* 2, pp. 947–957.

Brette, Romain (2015). "Philosophy of the Spike: Rate-Based vs. Spike-Based Theories of the Brain". In: *Frontiers in Systems Neuroscience* 9, p. 151. ISSN: 1662-5137. DOI: 10.3389/fnsys.2015.00151.

Brette, Romain et al. (2007). "Simulation of networks of spiking neurons: A review of tools and strategies". In: *Journal of Computational Neuroscience* 23.3, pp. 349–398. ISSN: 1573-6873. DOI: 10.1007/s10827-007-0038-6.

C., J., O. Bichler, and A. Dupret (2019). "SpikeGrad : An ANN-equivalent Computation Model for Implementing Backpropagation with Spikes". In: *CoRR* abs/1906.00851. arXiv: 1906.00851.

Camunas-Mesa, Luis A. et al. (2018). "A Configurable Event-Driven Convolutional Node with Rate Saturation Mechanism for Modular ConvNet Systems Implementation". In: *Frontiers in Neuroscience* 12. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00063.

Cao, Yongqiang, Yang Chen, and Deepak Khosla (2015). "Spiking Deep Convolutional Neural Networks for Energy-Efficient Object Recognition". In: *International Journal of Computer Vision* 113.1, pp. 54–66. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-014-0788-3.

Cao, Yongqiang and Stephen Grossberg (2012). "Stereopsis and 3D surface perception by spiking neurons in laminar cortical circuits: A method for converting neural rate models into spiking models". In: *Neural Networks* 26, pp. 75 –98. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2011.10.010.

Carbon, A. et al. (2018). "PNeuro: A scalable energy-efficient programmable hardware accelerator for neural networks". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1039–1044.

Carpenter, G. A. et al. (1992). "Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps". In: *IEEE Transactions on Neural Networks* 3.5, pp. 698–713.

Cassidy, A. S. et al. (2013). "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores". In: *International Joint Conference on Neural Networks*. DOI: 10.1109/IJCNN.2013.6707077.

Cruz-Albrecht, J. M., M. W. Yung, and N. Srinivasa (2012). "Energy-Efficient Neuron, Synapse and STDP Integrated Circuits". In: *IEEE Trans. Biomed. Circuits Syst.* 6.3, pp. 246–256. ISSN: 1932-4545. DOI: 10.1109/TBCAS.2011.2174152.

Davies, M. et al. (2018). "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning". In: *IEEE Micro* 38.1, pp. 82–99. ISSN: 0272-1732. DOI: 10.1109/MM.2018.112130359.

Davison, Andrew et al. (2007). "PyNN: towards a universal neural simulator API in Python". In: *BMC neuroscience* 8.S2, P2.

Davison, Andrew P et al. (2009). "PyNN: a common interface for neuronal network simulators". In: *Frontiers in neuroinformatics* 2, p. 11.

Delbrück, T. et al. (2010). "Activity-driven, event-based vision sensors". In: *IEEE International Symposium on Circuits and Systems*, pp. 2426–2429.

Diehl, Peter and Matthew Cook (2015). "Unsupervised learning of digit recognition using spike-timing-dependent plasticity". In: *Frontiers in Computational Neuroscience* 9, p. 99. ISSN: 1662-5188. DOI: 10.3389/fncom.2015.00099.

Diehl, Peter U. et al. (2015). "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing". In: *ieee*. IEEE, pp. 1–8. ISBN: 978-1-4799-1960-4. DOI: 10.1109/IJCNN.2015.7280696.

Dong, Xiangyu et al. (2012). "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.7, pp. 994–1007.

Du, Z. et al. (2015). "Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches". In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. DOI: 10.1145/2830772.2830789.

Esser, Steven K. et al. (2016). "Convolutional networks for fast, energy-efficient neuromorphic computing". In: *Proceedings of the National Academy of Sciences* 113.41, pp. 11441–11446. ISSN: 0027-8424. DOI: 10.1073/pnas.1604850113.

Farabet, Cleḿent et al. (2012). "Comparison between Frame-Constrained Fix-Pixel-Value and Frame-Free Spiking-Dynamic-Pixel ConvNets for Visual Processing". In: *Frontiers in Neuroscience* 6. ISSN: 1662-4548. DOI: 10.3389/fnins.2012.00032.

Fukushima, Kunihiko (1980). "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". In: *Biological Cybernetics* 36, pp. 193–202.

Furber, S. B. et al. (2014). "The SpiNNaker Project". In: *Proceedings of the IEEE* 102.5, pp. 652–665.

Gerstner, W. and W. M. Kistler (2002a). "Spiking Neuron Models: Single Neurons, Populations, Plasticity". In:

Gerstner, Wulfram and Werner M. Kistler (2002b). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press. DOI: 10.1017/CBO9780511815706.

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *International conference on artificial intelligence and statistics*, pp. 249–256.

Gu, Jiuxiang et al. (2015). "Recent Advances in Convolutional Neural Networks". In: *CoRR* abs/1512.07108. arXiv: 1512.07108.

Gütig, R. and H. Sompolinsky (2006). "The tempotron: a neuron that learns spike timing–based decisions". In: *Nature Neuroscience* 9, pp. 420–428.

Hahnloser, R. et al. (2000). "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit". In: *Nature* 405, pp. 947–951.

He, Kaiming et al. (2015). "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385. arXiv: 1512.03385.

Hodgkin, A. L. and A. F. Huxley (1952). "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of Physiology* 117.4, pp. 500–544. DOI: 10.1113/jphysiol.1952.sp004764.

Hopfield, J J (1982). "Neural networks and physical systems with emergent collective computational abilities". In: *Proceedings of the National Academy of Sciences* 79.8, pp. 2554–2558. ISSN: 0027-8424. DOI: 10.1073/pnas.79.8.2554.

Hubel, D. H. and T. N. Wiesel (1959). "Receptive fields of single neurones in the cat's striate cortex". In: *The Journal of Physiology* 148.3, pp. 574–591. DOI: 10.1113/jphysiol.1959.sp006308.

Huh, Dongsung and T. Sejnowski (2018). "Gradient Descent for Spiking Neural Networks". In: *ArXiv* abs/1706.04698.

Izhikevich, E. (2003). "Simple model of spiking neurons". In: *IEEE transactions on neural networks* 14 6, pp. 1569–72.

Izhikevich, E.M. (2004). "Which Model to Use for Cortical Spiking Neurons?" In: *IEEE Transactions on Neural Networks* 15.5, pp. 1063–1070. ISSN: 1045-9227. DOI: 10.1109/TNN.2004.832719.

Johansson, R. and I. Birznieks (2004). "First spikes in ensembles of human tactile afferents code complex spatial fingertip events". In: *Nature Neuroscience* 7, pp. 170–177.

Joubert, A. et al. (2012). "Hardware spiking neurons design: Analog or digital?" In: *International Joint Conference on Neural Networks*. DOI: 10.1109/IJCNN.2012.6252600.

Kasabov, Nikola K (2018). *Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence*. Vol. 7. Springer.

Khacef, Lyes, Nassim Abderrahmane, and Benoit Miramond (2018). "Confronting machine-learning with neuroscience for neuromorphic architectures design". In: *International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8.

Kheradpisheh, Saeed Reza and Timothée Masquelier (2020). "Temporal Backpropagation for Spiking Neural Networks with One Spike per Neuron". In: *International Journal of Neural Systems* 30.06. PMID: 32466691, p. 2050027. DOI: 10.1142/S0129065720500276.

Kheradpisheh, Saeed Reza et al. (2018). "STDP-based spiking deep convolutional neural networks for object recognition". en. In: *Neural Networks* 99, pp. 56–67. ISSN: 08936080. DOI: 10.1016/j.neunet.2017.12.005.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (May 2017). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Commun. ACM* 60.6, pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386.

Lecun, Y. et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

LeCun, Yann and Yoshua Bengio (1998). "The Handbook of Brain Theory and Neural Networks". In: ed. by Michael A. Arbib. Cambridge, MA, USA: MIT Press. Chap. Convolutional Networks for Images, Speech, and Time Series, pp. 255–258. ISBN: 0-262-51102-9.

LeCun, Yann et al. (1990). "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems 2*. Ed. by D. S. Touretzky. Morgan-Kaufmann, pp. 396–404.

Lee, Chankyu et al. (2020). "Enabling Spike-Based Backpropagation for Training Deep Neural Network Architectures". In: *Frontiers in Neuroscience* 14, p. 119. ISSN: 1662-453X. DOI: 10.3389/fnins.2020.00119.

Lemaire, E. et al. (2020). "An FPGA-Based Hybrid Neural Network Accelerator for Embedded Satellite Image Classification". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5.

Levi, Timothée et al. (Mar. 2018). "Digital Implementation of Hodgkin—Huxley Neuron Model for Neurological Diseases Studies". In: *Artif. Life Robot.* 23.1, 10–14. ISSN: 1433-5298. DOI: 10.1007/s10015-017-0397-7.

Lichtsteiner, Patrick, Christoph Posch, and Tobi Delbruck (2008). "A 128*128 120 dB 15us Latency Asynchronous Temporal Contrast Vision Sensor". In: *IEEE journal of solid-state circuits* 43.2, pp. 566–576.

Liu, Y. and X. Wang (2004). "Spike-Frequency Adaptation of a Generalized Leaky Integrate-and-Fire Model Neuron". In: *Journal of Computational Neuroscience* 10, pp. 25–45.

Lorrain, Vincent (Jan. 2018). "Etude et conception de circuits innovants exploitant les caractéristiques des nouvelles technologies mémoires résistives". Theses. Université Paris Saclay (COmUE).

Luo, Yuling et al. (2018). "An Efficient, Low-Cost Routing Architecture for Spiking Neural Network Hardware Implementations". In: *Neural Processing Letters* 48.3, pp. 1777–1788. ISSN: 1573-773X. DOI: 10.1007/s11063-018-9797-5.

Markram, Henry et al. (1997). "Regulation of Synaptic Efficacy by Coincidence of Postsynaptic APs and EPSPs". In: *Science* 275.5297, pp. 213–215. ISSN: 0036-8075. DOI: 10.1126/science.275.5297.213.

Mayr, Christian et al. (2015). "A biological-realtime neuromorphic system in 28 nm CMOS using low-leakage switched capacitor circuits". In: *IEEE transactions on biomedical circuits and systems* 10.1, pp. 243–254.

Merolla, P. et al. (2011). "A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm". In: *IEEE Custom Integrated Circuits Conference (CICC)*. DOI: 10.1109/CICC.2011.6055294.

Merolla, Paul A. et al. (2014). "A million spiking-neuron integrated circuit with a scalable communication network and interface". In: *Science* 345.6197, pp. 668–673. ISSN: 0036-8075. DOI: 10.1126/science.1254642.

Moradi, Saber et al. (2017). "A scalable multi-core architecture with heterogeneous memory structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs)". In: *CoRR* abs/1708.04198.

Mostafa, H. (2018). "Supervised Learning Based on Temporal Coding in Spiking Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.7, pp. 3227–3235. ISSN: 2162-237X. DOI: 10.1109/TNNLS.2017.2726060.

Mozafari, Milad et al. (2018). "Combining STDP and Reward-Modulated STDP in Deep Convolutional Spiking Neural Networks for Digit Recognition". In: *ArXiv* abs/1804.00227.

Mozafari, Milad et al. (2019). "SpykeTorch: Efficient Simulation of Convolutional Spiking Neural Networks with at most one Spike per Neuron". In: *CoRR* abs/1903.02440. arXiv: 1903.02440.

N2D2. /urlhttps://github.com/CEA-LIST/N2D2.

Nair, Vinod and Geoffrey E Hinton (2010). "Rectified linear units improve restricted boltzmann machines". In: *International conference on machine learning*, pp. 807–814.

Narayan, Sridhar (1997). "The generalized sigmoid activation function: Competitive supervised learning". In: *Information Sciences* 99.1, pp. 69 –82. ISSN: 0020-0255. DOI: https://doi.org/10.1016/S0020-0255(96)00200-9.

Navabi, Z. (1998). *VHDL: Analysis and Modeling of Digital Systems*. Electrical and Electronic Technology Series. McGraw-Hill. ISBN: 9780070464797.

Neftci, E., Hesham Mostafa, and Friedemann Zenke (2019). "Surrogate Gradient Learning in Spiking Neural Networks". In: *ArXiv* abs/1901.09948.

Neil, Daniel and Shih-Chii Liu (2014). "Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, pp. 2621–2628.

Orchard, Garrick et al. (2015). "HFirst: a temporal approach to object recognition". In: *IEEE transactions on pattern analysis and machine intelligence* 37.10, pp. 2028–2040.

Painkras, E. et al. (2013). "SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation". In: *IEEE Journal of Solid-State Circuits* 48, pp. 1943–1953.

Panda, Preeti Ranjan (2001). "SystemC: A Modeling Platform Supporting Multiple Design Abstractions". In: *Int. Symposium on Systems Synthesis*. ISSS '01. ACM, pp. 75–80. ISBN: 1-58113-418-5. DOI: 10.1145/500001.500018.

Pani, Danilo et al. (2017). "An FPGA Platform for Real-Time Simulation of Spiking Neuronal Networks". In: *Frontiers in Neuroscience* 11, p. 90. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00090.

Parvat, A. et al. (2017). "A survey of deep-learning frameworks". In: *Int. Conf. on Inventive Systems and Control*. DOI: 10.1109/ICISC.2017.8068684.

Perez-Carrasco, J. A. et al. (2013). "Mapping from Frame-Driven to Frame-Free Event-Driven Vision Systems by Low-Rate Rate Coding and Coincidence Processing–Application to Feedforward ConvNets". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.11, pp. 2706–2719. ISSN: 0162-8828, 2160-9292. DOI: 10.1109/TPAMI.2013.71.

Pfeiffer, Michael and Thomas Pfeil (2018). "Deep Learning With Spiking Neurons: Opportunities and Challenges". In: *Frontiers in Neuroscience* 12, p. 774. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00774.

Philippe, J. et al. (2015). "Exploration and design of embedded systems including neural algorithms". In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 986–991.

Polikar, Robi et al. (2001). "Learn++: An incremental learning algorithm for supervised neural networks". In: *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)* 31.4, pp. 497–508.

Ponulak, F. and A. Kasinski (2011). "Introduction to spiking neural networks: Information processing, learning and applications." In: *Acta neurobiologiae experimentalis* 71 4, pp. 409–33.

Ponulak, Filip and Andrzej Kasiński (Oct. 2009). "Supervised Learning in Spiking Neural Networks with ReSuMe: Sequence Learning, Classification, and Spike Shifting". In: *Neural computation* 22, pp. 467–510. DOI: 10.1162/neco.2009.11-08-901.

Rotermund, David and Klaus R. Pawelzik (2018). "Massively Parallel FPGA Hardware for Spike-By-Spike Networks". In: *bioRxiv*. DOI: 10.1101/500280.

– (2019). "Back-Propagation Learning in Deep Spike-By-Spike Networks". In: *Frontiers in Computational Neuroscience* 13, p. 55. ISSN: 1662-5188. DOI: 10.3389/fncom.2019.00055.

Rueckauer, Bodo et al. (2016). "Theory and tools for the conversion of analog to spiking convolutional neural networks". In: *arXiv preprint arXiv:1612.04052*.

Rueckauer, Bodo et al. (2017). "Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification". In:

Frontiers in Neuroscience 11, p. 682. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00682.

Rumelhart, David E., James L. McClelland, and CORPORATE PDP Research Group, eds. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-68053-X.

S2Net. /urlhttps://github.com/romainzimmer/s2net.

Schemmel, Johannes et al. (2010). "A wafer-scale neuromorphic hardware system for large-scale neural modeling". In: *IEEE International Symposium on Circuits and Systems*. IEEE, pp. 1947–1950.

Schmitt, S. et al. (2017). "Neuromorphic hardware in the loop: Training a deep spiking network on the BrainScaleS wafer-scale system". In: *International Joint Conference on Neural Networks (IJCNN)*, pp. 2227–2234.

Schuman, Catherine D. et al. (2017). "A Survey of Neuromorphic Computing and Neural Networks in Hardware". In: *CoRR* abs/1705.06963.

Simonyan, Karen and Andrew Zisserman (2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556. arXiv: 1409.1556.

Song, S., K. Miller, and L. Abbott (2000). "Competitive Hebbian learning through spike-timing-dependent synaptic plasticity". In: *Nature Neuroscience* 3, pp. 919–926.

Stallkamp, J. et al. (2012). "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition". In: *Neural Networks* 32. Selected Papers from IJCNN 2011, pp. 323 –332. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2012.02.016.

Sze, V. et al. (2017). "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12, pp. 2295–2329. ISSN: 0018-9219. DOI: 10.1109/JPROC.2017.2761740.

Szegedy, Christian et al. (2015). "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9.

Tavanaei, Amirhossein et al. (2019). "Deep learning in spiking neural networks". In: *Neural Networks* 111, pp. 47 –63. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2018.12.002.

Thiele, J. C., O. Bichler, and A. Dupret (2018). "Event-Based, Timescale Invariant Unsupervised Online Deep Learning With STDP". In: *Frontiers in Computational Neuroscience* 12, p. 46. ISSN: 1662-5188. DOI: 10.3389/fncom.2018.00046.

Thorpe, S., A. Delorme, and Rufin van Rullen (2001). "Spike-based strategies for rapid processing". In: *Neural networks : the official journal of the International Neural Network Society* 14 6-7, pp. 715–25.

Thorpe, Simon and Jacques Gautrais (1998). "Rank order coding". In: *Computational neuroscience*. Springer, pp. 113–118.

Vigneron, A. and J. Martinet (2020). "A critical survey of STDP in Spiking Neural Networks for Pattern Recognition". In: *International Joint Conference on Neural Networks (IJCNN)*, pp. 1–9.

Wu, Yujie et al. (2018). "Spatio-Temporal Backpropagation for Training High-Performance Spiking Neural Networks". In: *Frontiers in Neuroscience* 12, p. 331. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00331.

Wu, Yujie et al. (2019). "Direct Training for Spiking Neural Networks: Faster, Larger, Better". In: *AAAI*.

Yang, Wei-Jong et al. (2020). "Simplified Neural Networks with Smart Detection for Road Traffic Sign Recognition". In: *Advances in Information and Communication*. Ed. by Kohei Arai and Rahul Bhatia. Cham: Springer International Publishing, pp. 237–249. ISBN: 978-3-030-12388-8.

Yousefzadeh, A, T Serrano-Gotarredona, and B Linares-Barranco (2015). "Fast pipeline 128× 128 pixel spiking convolution core for event-driven vision processing in FPGAs". In: *International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP)*. IEEE, pp. 1–8.

Yu, Qiang et al. (2014). "A brain-inspired spiking neural network model with temporal encoding and learning". In: *Neurocomputing* 138, pp. 3–13. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2013.06.052.

Zaklouta, Fatin, B. Stanciulescu, and O. Hamdoun (2011). "Traffic sign classification using K-d trees and Random Forests". In: *International Joint Conference on Neural Networks*, pp. 2151–2155.

Zeiler, Matthew D. and Rob Fergus (2014). "Visualizing and Understanding Convolutional Networks". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, pp. 818–833. ISBN: 978-3-319-10590-1.

Zhang, G. P. (2000). "Neural networks for classification: a survey". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 30.4, pp. 451–462. ISSN: 1094-6977. DOI: 10.1109/5326.897072.

Zhang, M. et al. (2020). "Efficient Spiking Neural Networks With Logarithmic Temporal Coding". In: *IEEE Access* 8, pp. 98156–98167.