Implementation of a SNN model on an SBC-GPU and on a workstation in order to compare their efficiency

Jonathan Ferney Gómez Hurtado

Research work presented as a requirement to obtain the degree of:
Master in Engineering

Director
PhD. Ricardo Andrés Velásquez Vélez

Co-Director
PhD. Sebastián Isaza Ramírez

University of Antioquia
Faculty of Engineering
Master's degree in engineering
Medellin, Antioquia, Colombia

2023

Master's degree in Engineering

Research group: Sistemas Embebidos e Inteligencia Computacional (SISTEMIC).

Library Carlos Gaviria Díaz

Institutional Repository: http://bibliotecadigital.udea.edu.co

University of Antioquia - www.udea.edu.co

Contents

List of Figures

List of Tables

Abstract

Researchers using Spiking Neural Networks to deploy its applications on Servers and Workstations with graphics processing units because of the restricted access the specialized neuromorphic platforms. Moreover, using such conventional systems imply high energy and acquisition costs.

Recently, we have seen the popularization of computing platforms with small form factors, low energy consumption, and the ability to perform artificial intelligence. These platforms, known as single-board computers, often integrate graphics processing units and other hardware accelerators; thus, they are feasible alternatives to traditional computer systems in critical energy consumption applications.

This work presents our insights into implementing a 2-layer Spiking Neural Networks inference algorithm for handwritten digit recognition. We implemented the network on a GPU MALI included on the VIM3 and a workstation GPU using the C++ language and openCL. Our experimental results show that while single-board computer inference is 6x slower compared to a workstation, it is 7x more efficient in energy consumption.

Keywords: Spiking Neural Networks, embedded GPU, Single Board Computer

Chapter 1

Introduction

For many years, the brain and its learning mechanisms have attracted the scientific community's interest. An example of this are Artificial Neural Networks (ANNs). ANNs are inspired by the biological nervous system and have become one widely used tool to implement Artificial Intelligence. On the other hand, Spiking Neural Networks (SNNs) have emerged as a promising computational paradigm, because they are capable of modeling the information processing observed in biological neural networks [52]. Moreover, SNNs can naturally capture space and time dimensions, including frequency and phase of the the input signals, and are suitable for implementation on low-power hardware [76].

One feature differentiating SNNs from ANNs is how information travels inside the network. That is, using spikes as the elementary unit of information transmitted between neurons. A spike is a short electrical pulse with amplitude in the order of millivolts and durations of a few milliseconds [29]. In line with this, academia and industry are recently developing specialized neuromorphic hardware to take advantage of the Spatio-temporal characteristics of SNNs in engineering applications. The most outstanding projects in this area are SpiNNaker [27], Loihi [17], TrueNorth [4], which are digital platforms explicitly created to simulate SNNs. However, the use of these systems is limited, and the technology in this field has not reached sufficient maturity to make these hardware platforms commercially available.

Today, most SNNs are implemented on conventional computer systems, often helped by Graphics Processing Units (GPUs) in order to accelerate both training and inference

stages [3, 59, 45]. GPUs can provide large-scale parallel computations at a cost in developing effort and power, depending on the type of computer systems they are part of.

Single-board computers (SBCs) are low-cost and low-power alternatives that can be used to run neural network applications. Some of those SBCs feature embedded GPUs to accelerate compute intensive applications such as SNNs, achieving higher efficiency when compared to expensive and power hungry workstations. Running SNNs on GPU powered SBCs can be very valuable for building a simulation cluster infrastructure for neuroscientists or for edge computing applications.

To the best of our knowledge there are no works where SNNs are mapped to an embedded GPU and hence, we implemented a 2-layer Spiking Neural Network using the MALI GPU of a VIM3 SBC and Nvidia GPU of a workstation which were programmed using C++ and openCL, taking as an example of application a handwritten digit recognition system. Our objective with this work is provide insights into the speed and energy efficiency levels of SBCs and workstations featuring GPUs.

1.1 Objectives

To implement a SNN model on an off-the-shelf heterogeneous MPSoC and on a workstation in order to compare their efficiency.

1.1.1 Specific objectives

- To select the SNN model type and input encoding techniques for reducing the computational cost on an off-the-shelf heterogeneous MPSoC, using a pre-trained SNN.
- To design a suitable partitioning and mapping strategies for the trained SNN onto both, the heterogeneous MPSoC and the workstation.
- To implement an SNN in the heterogeneous MPSoC and workstation, considering an application that allows efficient use of heterogeneous hardware components.
- To evaluate the performance of the SNN implementation in the heterogeneous MPSoC and workstation.

1.2 Document Organization

This work is organized as follows:

Chapter 2: This chapter describes fundamental concepts necessary for the project development. We will be presenting the definition of neuron models from the electrical point of view, solution forms of the neural model based on numerical methods, types of encoding for the information communication between neurons, network topologies commonly used with SNNs, SNNs software, frameworks for parallel computing and finally, we will refer to computer systems and hardware accelerators.

Chapter 3: This chapter presents the literature review regarding the computer systems where SNNs are deployed, including the use of digital or analog/digital systems called neuromorphic. We discuss the use of general purpose computers, as well as the use of embedded GPUs for applications related to Neural Networks and Artificial Intelligence.

Chapter 4: This chapter presents the methodology used for the implementation of the SNN, explaining the procedures for achieving the project objectives and the technical aspects of the computer systems.

Chapter 5: This chapter presents the network architecture, the training process in BindsNET, the considerations regarding the handling of the information to reduce the memory footprint and also the algorithms to be executed in both the CPU and GPU.

Chapter 6: This chapter discusses the experimental results obtained in the project. The reduction of the analysis time window for obtaining both, accuracy and execution speed. We also present the results of running the base algorithm in its sequential implementation and the notable difference in throughput with the parallel implementation.

Chapter 7: In this chapter we present the conclusions obtained from the experimental component and the future work alternatives to improve the understanding of SNNs deployed on embedded GPUs.

1.3 Contributions

Based on the literature review, it can be concluded that research on spiking neural networks has been mainly conducted on GPU-enabled computing systems. The implementation of these networks is restricted to the use of libraries such as BindsNET, CARLsim, Nengo, among others, which are based on libraries with support mainly for CUDA, deployed predominantly in conventional computing systems.
Thus, we were able to define the following contributions: (1) this study presents the first work that integrates a Spiking Neural Network algorithm on a Single Board Computer through the use of OpenCL as an embedded GPU programming tool for parallel computation, (2) we found that single-board computer inference process is 6x slower compared to a workstation, but, it is 7x more efficient in energy consumption, this means that for projects that do not require high processing power, single-board computers can be a more cost-effective and eco-friendly option, (3) we performed an exploration of the simulation window to fine tune this parameter to the ideal point that would allow us to obtain the best performance without suffering significantly degradation of accuracy in the network inference process.

Other contributions resulting from the project are highlighted below.

Conference paper
Gómez Hurtado, J. F., Isaza Ramírez, S., Velásquez Vélez, R. A. (2022). On the Efficiency of Embedded GPUs for Spiking Neural Networks. In N. Callaos, N. Lace, B. Sánchez, M. Savoie (Eds.), Proceedings of the 26th World Multi-Conference on Systemics, Cybernetics and Informatics: WMSCI 2022, Vol. I, pp. 23-29. International Institute of Informatics and Cybernetics. https://doi.org/10.54808/WMSCI2022.01.23

Source code
Github repository with the source code containing the implementation of the sequential algorithm and also the parallel implementation using openCL:
https://github.com/JonathanProf/thesisCode

Chapter 2

Background

This chapter presents essential concepts used throughout the document. These concepts serve as a reference for developing our network and executing the experiments. We discuss the neuron models used in SNNs, information coding techniques, and differential equation solution methods. Moreover, we also present the most important programming models, parallel computing platforms and, SNN simulation tools.

2.1 Neuron models used in SNNs

The primary purpose of developing SNNs was to model how the brain works. The level of detail used for this modeling has generated different ways to represent spiking neurons. Some neuron models are more biologically plausible than others but with high computational complexity. At the same time, some of them relegate biological plausibility slightly in favor of easy implementation and computational efficiency. We will start presenting the most biologically plausible models, such as the Hodgkin-Huxley. Then, we finish this section by discussing the Leaky Integrate-and-Fire model, a biologically plausible model with lower computational complexity.

2.1.1 Hodgkin-Huxley neuron model (HH)

Hodgkin and Huxley proposed their spiking neuron model in 1952. They experimented with the giant squid's axon and found three ionic currents (sodium, potassium, and leak) [36]. Thus, a neuron can be represented through an electrical circuit (Figure 2.1). Here, an input current $I_{input}(t)$ is injected into the cell. The capacitor represents the

semipermeable cell membrane that separates the cell interior from the extracellular liquid. R_L, R_K, and R_{Na} model the leakage, potassium, and sodium ionic currents. Resistor R_K and R_{Na} are variable since their values depend on whether the ion channels are open or closed. V_K, V_{Na} and V_L model the Nernst potential [35] for each ion channel [30].

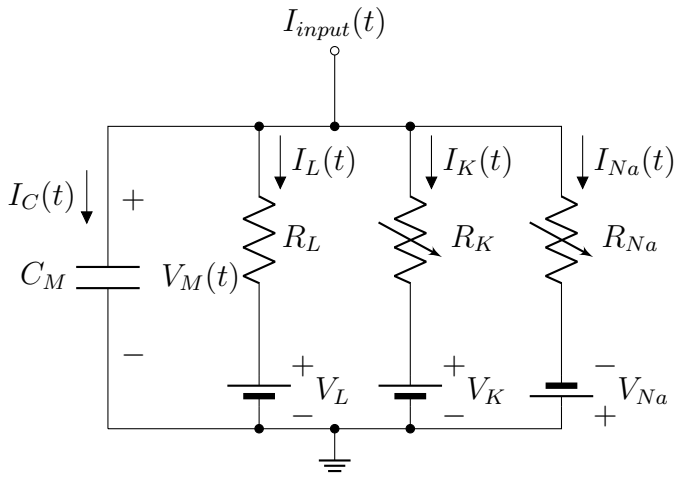

Figure 2.1: Electrical diagram of Hodgkin-Huxley neuron model [36].

We must use electrical circuit solution techniques to derive a mathematical representation. For this case, we apply the Kirchhoff's current law [75] on the top node to have the mathematical representation of this neuron model, where:

$$I_{input}(t) = I_{C_M}(t) + I_L(t) + I_K(t) + I_{Na}(t)$$

$$I_{C_M}(t) = I_{input}(t) - I_L(t) - I_K(t) - I_{Na}(t)$$

$$C_M * \frac{dV_M(t)}{dt} = I_{input}(t) - \frac{[V_M(t) - V_L]}{R_L} - \frac{[V_M(t) - V_K]}{R_K} - \frac{[V_M(t) - V_{Na}]}{R_{Na}} \tag{2.1}$$

The computational complexity and biological plausibility of the model presented in Equation 2.1 comes from modeling the time dynamics of the opening and closing of ion channels, because the resistors are not a constant value but depend on other variables associated with probabilistic events as shown in the following equations.

$$R_K = \frac{1}{g_K * n^4} \tag{2.2}$$

$$R_{Na} = \frac{1}{g_{Na} * m^3 * h} \tag{2.3}$$

$$R_L = \frac{1}{g_L} \tag{2.4}$$

$$\frac{dn}{dt} = \alpha_n(V_M(t))(1-n) - \beta_n(V_M(t))n \tag{2.5}$$

$$\frac{dm}{dt} = \alpha_m(V_M(t))(1-m) - \beta_m(V_M(t))m \tag{2.6}$$

$$\frac{dh}{dt} = \alpha_h(V_M(t))(1-h) - \beta_h(V_M(t))h \tag{2.7}$$

The variables m, n, and h represent the probability of a channel opening at a given time [30]. Thus, the combined action of m and h controls the sodium channel, while n controls the potassium channel.

The Hodgkin-Huxley model offers excellent biological accuracy, but it is computationally prohibitive for many real-life applications. However, the research community has proposed models that reduce the computational complexity compared to HH while maintaining biological plausibility.

2.1.2 Izhikevich neuron model (IZH)

Izhikevich proposed in [37] a new model which reduces the Hodgkin-Huxley neural model to a two-dimensional system of ordinary differential equations. Equations (2.8), (2.9) and (2.10) present the Izhikevich neural model.

$$\frac{dV_M(t)}{dt} = 0.04V_M^2(t) + 5V_M(t) + 140 - U(t) + I_{input}(t) \tag{2.8}$$

$$\frac{dU(t)}{dt} = a(bV_M(t) - U(t)) \tag{2.9}$$

with the auxiliary after-spike resetting equation:

$$\textit{if } V_M(t) \geq 30mV, \textit{ then } = \begin{cases} V_M(t) \leftarrow \boldsymbol{c} \\ U(t) \leftarrow \boldsymbol{U(t)+d} \end{cases} \tag{2.10}$$

Unlike the HH model, this one does not have a circuit representation, but it is used in the literature to design spiking networks that can emulate in a similar way the cortex of

a mammal [37].
Table 2.1 shows the typical values presented by Izhikevich for the different parameters considered in his model.

Table 2.1: Izhikevich neuron model parameters acquired from [37].

Parameter or Variable	**Feature**
Membrane potential ($V_M(t)$)	This variable is in the order of mV
Resting potential (V_{rest})	Depending of the b value is between $-70mV$ and $-60mV$
Threshold potential (V_{th})	This parameter is between $-55mV$ and $-40mV$ depending on the $V_M(t)$ history
Membrane recovery variable ($U(t)$)	Accounts for the the inactivation of sodium ionic currents and the activation of potassium ionic currents
time (t)	This variable is in the order of ms
a	Describes the time scale of $U(t)$. A typical value for this parameter is 0.02
b	Describes the sensitivity of $U(t)$ to $V_M(t)$ subthreshold fluctuations. A typical value for this parameter is 0.2
c	Describes the reset value in mV of $V_M(t)$ after the spike has occurred. A typical value for this parameter is $-65mV$
d	Describes the reset value of $U(t)$ after the spike has occurred. A typical value for this parameter is 2

2.1.3 Leaky Integrate-and-Fire neuron model (LIF)

While Izhikevich neuron model reduces complexity compared with HH model, a simpler neural model that preserves biological plausibility but with a lower computational cost is desirable. The leaky integrate-and-fire (LIF) [29] is an efficient neuron model whose differential equation solution is one-dimensional, and thus it requires less computational operations than the two previously presented models. Figure 2.2 shows the equivalent electrical RC circuit for this neuron model. A current $I_{input}(t)$ feds the circuit, R_M is

the membrane resistor, C_M is the membrane capacitor, and $V_M(t)$ is the membrane potential.

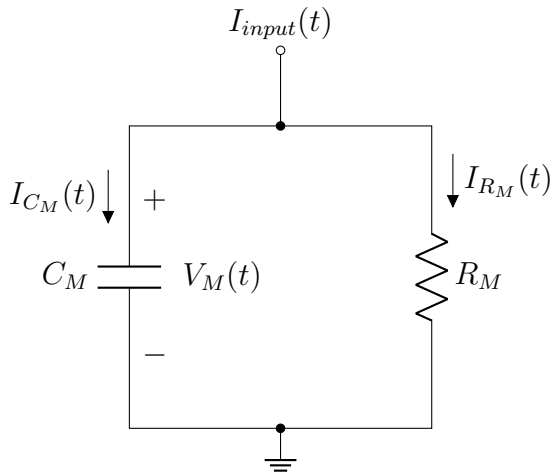

Figure 2.2: Electrical diagram of Leaky Integrate-and-Fire neuron model [29].

We can obtain the mathematical model of this electrical circuit by applying the Kirchhoff's current law [75] on the top node

$$I_{input}(t) = I_{R_{M(t)}} + I_{C_{M(t)}} \tag{2.11}$$

Now, let us replace $I_{R_M}(t) = V_M(t)/R_M$ and $I_{C_M}(t) = C_M * d[V_M(t)]/dt$ in Equation 2.11

$$I_{input}(t) = \frac{V_M(t)}{R_M} + C_M * \frac{d[V_M(t)]}{dt}$$

Finally, let us define $\tau_m = R_M * C_M$, the neuron's leaky integrator constant or membrane time constant.

$$\tau_m \frac{d[V_M(t)]}{dt} = R_M * I_{input}(t) - V_M(t) \tag{2.12}$$

The Equation 2.12 is known as a differential equation (field of mathematics), leaky integrator or RC-circuit (electrical engineering field), or passive membrane equation (neuroscience field) [29]. In addition to Equation 2.12, the model define that a spike occurs at the output when $V_M(t) \geq V_{th}$(threshold voltage).

The LIF model with current-based synapse and leakage can help reduce the computational complexity compared to IZH or HH models. Therefore, many researchers

on SNNs often select LIF as its neural model. Among the applications and works carried out with this neuron model we can highlight the following: Energy-Efficient Neuron, Synapse and STDP Integrated Circuits [16], Image Classification [71], Real-Time Convolution-Based Nonlinear Feature Extraction [58].

2.1.4 Synaptic Model

In the three neural models, the input $I_{input}(t)$ is the total synaptic current or the sum of the input synapses. We can express mathematically $I_{input}(t)$ as the dot product of the weights vector $\mathbf{W} = [w_1, w_2, ..., w_N]$ and the spatiotemporal input spike patterns $\mathbf{S}(t) = [s_1(t); s_2(t); ...; s_N(t)]$ at an instant t [76].

$$I_{input}(t) = \mathbf{W} \cdot \mathbf{S}(t) \tag{2.13}$$

The amplitude and sign of the synaptic current depend on $\mathbf{W}$, where for each synapse i, if $w_i > 0$ we have excitatory synapses while for $w_i < 0$ we have inhibitory synapses[20]. Therefore, we can express $I_{input}(t)$ as the sum of an excitatory synaptic current (positive current) and an inhibitory synaptic current (negative current) as a function of time [28]. We redefine the total current as:

$$I_{input}(t) = \mathbf{W} \cdot \mathbf{S}(t) = I_{exc} - I_{inh} \tag{2.14}$$

$\mathbf{S}$(t) is known as a Poisson process with rate $N\nu$, where N is the synapse number and ν is the activation rate of one synapse [12]. This method is used to transform real values into spike trains to feed the input layer [76].

2.2 Numerical methods for differential equations solution

Once we have studied the neuron models, a question emerges: Which numerical methods are used to solve the neuron models' differential equations? In this section, we review the methods commonly used to solve differential equations, based on the work presented by Skocik et al. in [70], who studies the computational cost of the Forward Euler, Exponential Euler and Fourth-Order Runge-Kutta methods.

2.2.1 Forward Euler

Euler proposed this method in [24] as:

$$y_{i+1} = y_i + \Delta t * f(t_i, y_i) \tag{2.15}$$

Where Δt is a fixed step size in the time domain during the entire simulation and the error is proportional to this parameter. An important consideration is that the model is simple, and the differential equation can be solved only with Equation 2.15. Hence, the performance in terms of solution accuracy is not as good if we compare it to the Fourth-Order Runge-Kutta method, which is more accurate.

2.2.2 Exponential Euler

Let us consider $d[y(t)]/dt = f(t, y) = -A(t) * y(t) + B(t)$, , where $A(t)$ and $B(t)$ are arbitrary functions. This method is defined in [54] and [80] as:

$$y_{i+1} = \left(y_i - \frac{B_i}{A_i}\right) e^{-A_i \Delta t} + \frac{B_i}{A_i} \tag{2.16}$$

If $A(t)$ and $B(t)$ are constants in the interval Δt, we obtain the exact solution. However, if these functions vary rapidly over time or depend on y, the method may require iterations over each step. Skocik et al. in [70] indicate that the LIF neuron model is appropriate for this method by the nature of the differential equation.

2.2.3 Fourth-Order Runge-Kutta

Kutta defines this method in [47] as:

$$\begin{aligned} y_{i+1} &= y_i + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ k_1 &= f(t_i, y_i) \\ k_2 &= f(t_i + 0.5\Delta t, y_i + 0.5\Delta t * k_1) \\ k_3 &= f(t_i + 0.5\Delta t, y_i + 0.5\Delta t * k_2) \\ k_4 &= f(t_i + \Delta t, y_i + \Delta t * k_3) \end{aligned} \tag{2.17}$$

Although this method is computationally more intensive than the two previous ones due to the number of equations required, it provides more accurate solutions to differential

equations.

A study was carried out in [80] to measure the computational cost of the three neuron models LIF, IZH, and HH for each numerical method presented. Table 2.2 shows the results. It is possible to conclude that the LIF neuron model needs fewer operations than the other models with the three numerical methods considered. Forward Euler and Exponential Euler methods produce similar FLOPS when computing the differential equation of the LIF model. These results confirm that LIF model is computationally more efficient than the IZH and HH models, but it still retain the biological plausibility required to preserve the structure of the spiking network. The latter being one of the main reasons to consider neuron models based on biological experimentation and not only on mathematical models that only seek the highest performance in the network response.

Table 2.2: FLOPS per step update [80].

Spiking Neuron	**Forward Euler**	**Exponential Euler**	**Fourth-Order Runge-Kutta**
LIF	4	4	23
IZH	14	23	57
HH	114	158	472

Another important study was conducted by Capra et al. [13]. The research work is important because it highlights the trade-off between biological plausibility and computational efficiency in neuron models (Figure 2.3), which validates the information presented above in Table 2.2, as it identifies which neuron models are significantly more biologically plausible and which ones allow for efficient computation.

2.3 Spikes and encoding techniques for SNNs

Let us now introduce how information is conveyed in the synaptic process. We will first define the spike representation from the electrical point of view. Subsequently, we will present how a spike train allows encoding information to feed the neurons.

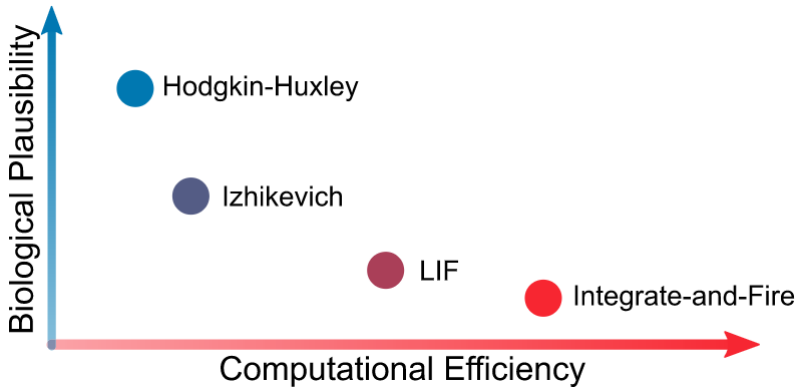

Figure 2.3: Comparison of different neuron models in terms of biological plausibility versus computational efficiency [13].

2.3.1 Action potentials or Spikes

The neurons mainly communicate using short electrical pulses (action potential) or spikes. A spike typically has a duration between 1ms and 2ms and a voltage amplitude of around 100mV [30]. Figure 2.4 shows the shape of the electrical signal for a single spike.

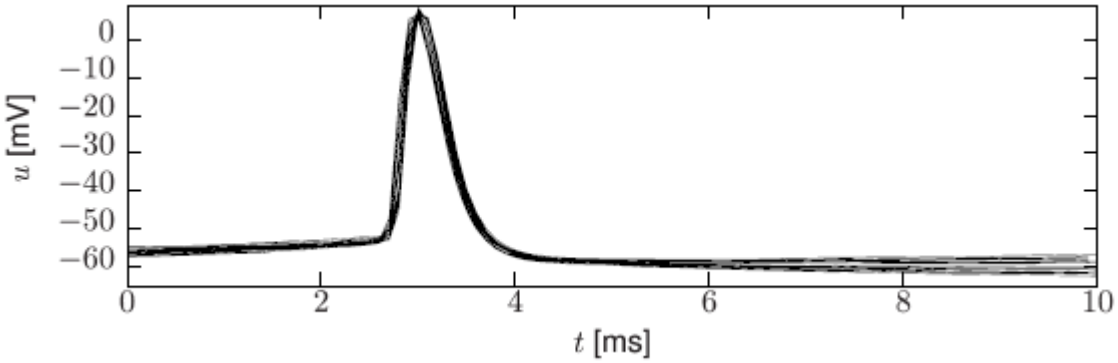

Figure 2.4: Representation of the action potential or spike [55]. The vertical axis represents the signal voltage (u), and the horizontal axis the evolution of the spike over time (t).

When neurons produce several spikes over time, we call this spikes train. The spikes are well separated and may occur at regular or irregular time intervals [30]. Mathematically

a spike train for a neuron i can be expressed as:

$$S_i(t) = \sum_f \delta(t - t_i^f) \tag{2.18}$$

Where $\delta(t)$ is the Dirac function and f is the spike label [30]. Next, we will study spikes train but from the perspective of the most commonly used encoding techniques for SNNs.

2.3.2 Encoding techniques

How spikes encode information is one of the critical aspects studied by neuroscientists, there are several ways to represent them but we will illustrate the most common ones, rate coding and temporal coding [76].

Rate coding

In this method, the input data is transformed into spike trains through the neuron's mean firing rate [1]. This encoding approach is the most popular and, mathematically we can express the mean firing rate (ν) as follows:

$$\nu = \frac{n_{sp}(T)}{T} \tag{2.19}$$

Where $n_{sp}(T)$ is the number of spikes in a time window T, and ν has units of Hz. One advantage of this technique is its low encoding and decoding process complexity and its wide use for SNNs [85]. Figure 2.5 presents a rate coding example for encoding an image's pixels into spikes trains.

Temporal coding

This technique generates fewer spikes than rate coding, reducing computational resources in the inference process [2]. If we compare against the example for rate coding, only one spike per pixel is used (Figure 2.6).

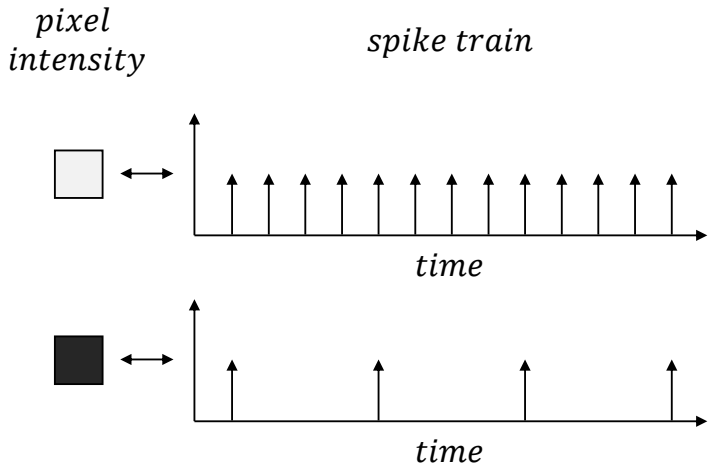

Figure 2.5: Rate coding example, based on [2].

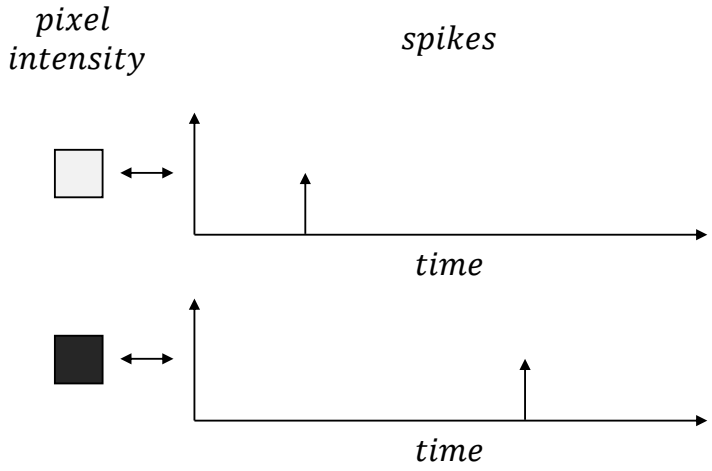

Figure 2.6: Temporal coding example, based on [2].

2.4 Network topologies

The way in which the neurons are connected among them is known as the topology of a Neural Network. Generally, a complete network is constructed using a set of neurons that are distributed in layers [21]. Feedforward networks and recurrent networks are a common classification of SNNs topologies [76]. We review some of these topologies below:

2.4.1 Feedforward networks

The feedforward structure seems to be an accepted model for information processing, supported by the response of inferotemporal cortex cells [69]. This model can be represented in a general way by three layers as can be seen in Figure 2.7. The first layer handles the input stimuli and performs the encoding of the information into spike trains applying one of the techniques described in section (2.3). Then, all the units that perform the encoding are fully connected to the neurons in learning (processing) layers, this intermediate layers are where the neuron models described in section (2.1) are used as the fundamental information processing units. Finally, there is an output layer where information is retrieved from the responses generated in the processing layer and the classification process is carried out [84].

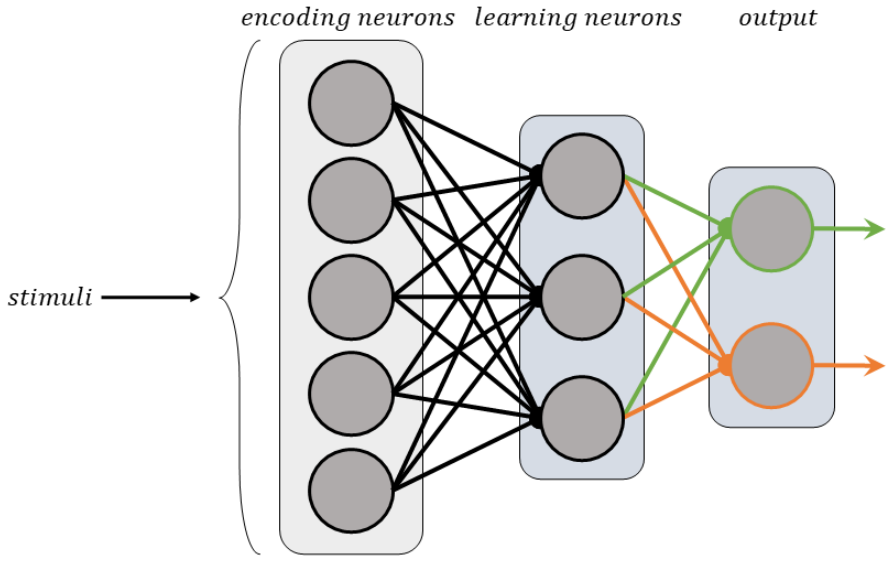

Figure 2.7: Feedforward network.

2.4.2 Recurrent networks

Recurrent Neural Networks (RNNs) differ from feedforward networks, because its internal state evolves over time. RNNs simulate memory functions and it has been evidenced that play an important role in the visual cortex [20]. In studies such as those conducted by Chance et al. [14] have shown that complex cell responses could be generated by recurrent networks. So, this type of networks can be useful for pattern recognition

applications [23, 76]. In Figure 2.8 we can see the representation of a typical RNN.

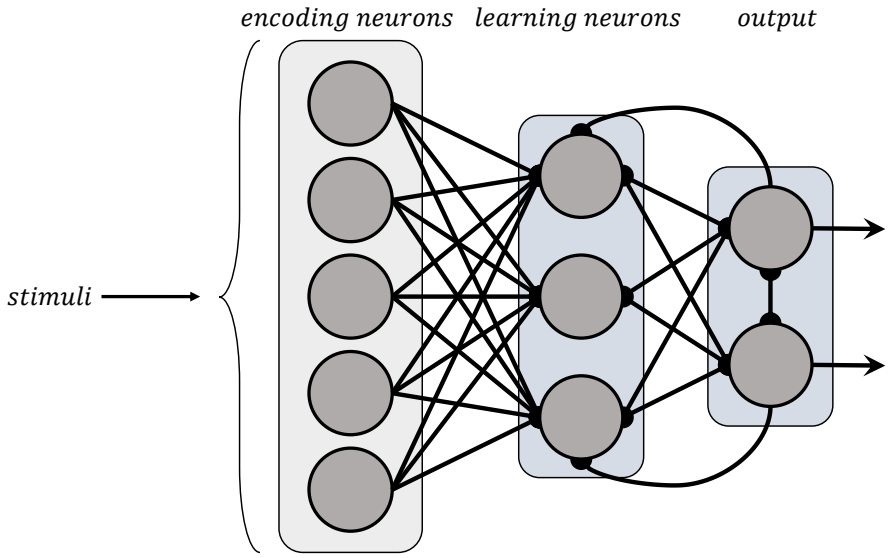

Figure 2.8: Recurrent network [20].

In the field of SNNs, the implementation of competitive recurrent architectures which are based on RNNs are very common, because the accuracy of the network can be improved, as well as making better the convergence speed during the training stage [48]. Winner-Take-All (WTA) stands out as one of the most widely used systems for this competitive process. WTA, besides being a brain-inspired mechanism, consists of making the neurons of the same layer compete with each other. Therefore as soon as a neuron generates a spike to the output, an inhibition process to the other neurons in the same layer is generated, preventing more than one neuron from generating spikes at the output simultaneously [11, 53].

From the studies carried out in [22, 67, 38], we conclude that competitive learning is relevant for digit/vowels classification. Additionaly, it is necessary to cluster layers of neurons with excitatory and inhibitory synapses to facilitate recurrence.

2.5 SNNs software packages

There are many software packages that can simulate SNNs. Several of these have a specific application domain with different approaches. The differences are in the context of whether it is desired to represent the neural model with sufficient features or better target a more functional design to obtain the best performance of the network.

For example, NEST [31], BRIAN [73], and ANNarchy [81] concentrate on precise biological simulation from subcellular components and biochemical reactions to complex neurons models, and the programmer can specify the neuron dynamics via differential equations [33]. These simulation libraries run on desktop computers or high-performance platforms, representing a significant advantage. However, a drawback of these tools is that due to the detailed definitions required for each component, they are not suitable for quick prototyping.

Nengo [9], CARLsim [15], BindsNET [33] and NeuCube [39] allow the user to work with the behavior of the SNNs at a high level, and they target AI applications. Nengo is an open-source project developed in Python, and it has options for deploying neural models on dedicated hardware platforms such as SpiNNaker [33]. CARLsim was designed to run simulations on heterogeneous computing cluster using several CPUs and GPUs [15]. BindsNET uses Pytorch [65] for efficient computation of SNNs and can be connected to some systems such as FPGA, ASIC, DSP. NeuCube, on the other hand, is not an open-source project and therefore has limited usability. It is developed in MATLAB and supports simulation at the level of spikes or firing rates [33].

Table 2.3 provides a summary description of relevant aspects in addition to those mentioned above. The table shows that the majority of simulation tools have implemented the most popular neuron models (LIF, IZH and HH). Additionally, they have implemented one or more of the numerical methods presented in section (2.2), which allows us to delimit the design space for the parameters selection in the construction and simulation of our neural network.

Table 2.3: Review of numerical methods used to solve differential equations of spiking neuron models and other relevant features of SNNs libraries and simulators.

Tool	**Numerical methods**	**Neuron implemented**
NEST [31]	Forward Euler, Fourth-Order Runge-Kutta	Integrate-and-fire (IF), HH, IZH and LIF
Brian 2 [72, 73]	Exponential Euler, Forward Euler, Second-Order Runge-Kutta, Fourth-Order Runge-Kutta, Stochastic Heun	User-defined by means of symbolic equations
ANNarchy [81]	Forward Euler, Backward Euler, Exponential Euler, Second-Order Runge-Kutta	HH, IF, IZH, user defined e.g. LIF
Nengo [9]	Forward Euler	LIF and IZH
CARLsim [15]	Forward Euler, Fourth-Order Runge-Kutta	IZH and LIF
BindsNET [33]	Euler	McCulloch-Pitts, IF, LIF, IZH
NeuCube [39]	-	LIF and IZH

2.6 Computer systems

A complete computer is defined as a system that includes a processor, memory, input, and output ports, built to receive user actions with input peripherals, process information and present results on its output peripherals [78]. Electronic vendors design these systems following a general model: input, storage, processing, and output. However, there are differences in how a computer system can be classified, built, and used. There are three categories of computer systems: high-performance, general-purpose, and embedded systems [66].

2.6.1 General-Purpose Computer (GPC)

General-Purpose computer systems can be programmed to carry out many tasks. These systems are versatile in the number of activities since the software can be added, updated, and removed without altering the functionality [66]. Desktops and laptops are examples of GPC systems since they have multiple input ports (USB, PS2, Serial Connector), output ports (HDMI, VGA), processing units, memories, and storage units. In addition,

they run an operating system and several applications simultaneously, mainly for daily use.

2.6.2 Embedded Systems (ES)

Embedded systems carry out fewer tasks than GPCs since manufacturers focus on dedicated functions that the system must perform efficiently and optimally [66]. An ES can be, for example, a pacemaker, a device that aims to monitor and control the heartbeat to ensure its regularity, where low power and low resource consumption implementations are required. Among the advantages of ESs are high efficiency in executing tasks, reliability, low production cost, compact size, and low power consumption.

2.6.3 High Performance Computers (HPC)

High Performance Computers are systems with a significantly higher degree of performance than the previous ones. They are custom-built to perform computationally intensive tasks, which is why they are widely used in research and industry. With these systems it is possible to perform climate simulations, large-scale modeling for the generation of new materials, cryptographic analysis, among others. They are mainly used to get results in a shorter time than a conventional computer, since they make use of hundreds of CPUs to implement parallel and distributed programming methodologies.

2.7 Domain-specific accelerators

Hardware accelerators are devices that increase the performance of applications that require parallel or concurrent computing. There is a wide range of accelerators but the most widely used and implemented in computer systems are graphics processing units and neural processing units.

2.7.1 Neural Processing Unit Module (NPU)

A neural processing unit (NPU) is a dedicated circuit that has all the control and arithmetic logic components necessary to execute machine learning algorithms. NPUs' primary purpose is to accelerate the performance of machine learning tasks such as image classification, machine translation, object detection, and various other predictive models.

2.7.2 Graphics Processing Unit Module (GPU)

Early GPUs were initially developed and produced for computer graphics, particularly for video processing and computer gaming. GPUs try to maximize the computing throughput by exploiting data parallelism. GPUs can speed up computations by simultaneously running a single instruction on multiple data points (SIMD) [40]. Some SNN simulation libraries use GPUs as vector processors to speed up large-scale SNN simulations, such as NeMo [26] or BindsNet [32]. There are some differences of these accelerators depending on the system that integrates them. For instance, embedded GPUs have less number of cores compared to server GPUs and, embedded GPU-CPU share the physical memory because they are built on the same chip [83].

2.8 Parallel computing development frameworks

In this section we discuss the programming models that make possible the implementation of SNN networks on computer systems. Here, we mainly discuss the programming interfaces to efficiently operate the CPU and GPU resources on computer systems.

2.8.1 OpenMP

OpenMP is not a programming language but an extension for languages like C, C++, or Fortran. OpenMP is considered an Application Programming Interface (API) used to program multithreaded applications, allowing running programs on shared memory systems. The API comprises compiler directives, supporting functions, and shell variables [79]. This programming model, beyond being portable and scalable, eases the programmer to use the Fork-Join model, where the aim is to divide a sequential task performed by a single process into several processes working together to perform the total work.

2.8.2 MPI

Message **P**assing **I**nterface or **MPI** is a standardized parallel programming model for programming distributed memory computer systems. Similar to OpenMP, MPI is not a programming language. The MPI standard defines a helpful API for various programming languages like C, C++, and Fortran. There are several open-source MPI implementations [79]. MPI is generally used to try to optimize the execution time

of a program that needs to perform massive operations but cannot be executed on a single machine, instead it requires the cooperative work of several computer systems to perform a job.

2.8.3 CUDA

CUDA or **C**ompute **U**nified **D**evice **A**rchitecture is a parallel programming model for developing general-purpose applications on GPUs. CUDA provides language extensions for programming languages such as C, C++, Fortran, Python, and MATLAB. CUDA users develop applications for desktops, data centers, and embedded systems that use NVIDIA technology [62]. This programming model has become very popular in recent years, since it has allowed the development of several areas of knowledge, such as: animation and modeling (mainly in video games), astronomy and astrophysics, big data, data mining, bioinformatics, climate modeling, among other applications.

2.8.4 OpenCL

Open **C**omputing **L**ibrary or **OpenCL** is a widely used and standardized parallel programming model. Unlike the previous programming models, OpenCL programs can execute on different computing devices without modifications. This feature provides an advantage over other programming models whose programs require customization for each computing device. OpenCL runs on various computing devices such as CPUs, GPUs, or hardware accelerators [79]. OpenCL users develop parallel programs for supercomputers, servers, personal computers, mobile devices, and embedded systems [42]. A developer must understand three concepts when working with this programming model: platform, execution, and memory models [79].

Platform model

The OpenCL platform model (Figure 2.9), consists of a single host (e.g. CPU) connected to one or more accelerator devices known as Compute Device (e.g. GPU, DSP, FPGA) [64]. An OpenCL program consists of two components, the first component which is the main program that runs on the host, and the second called kernels that will run on the devices [79].

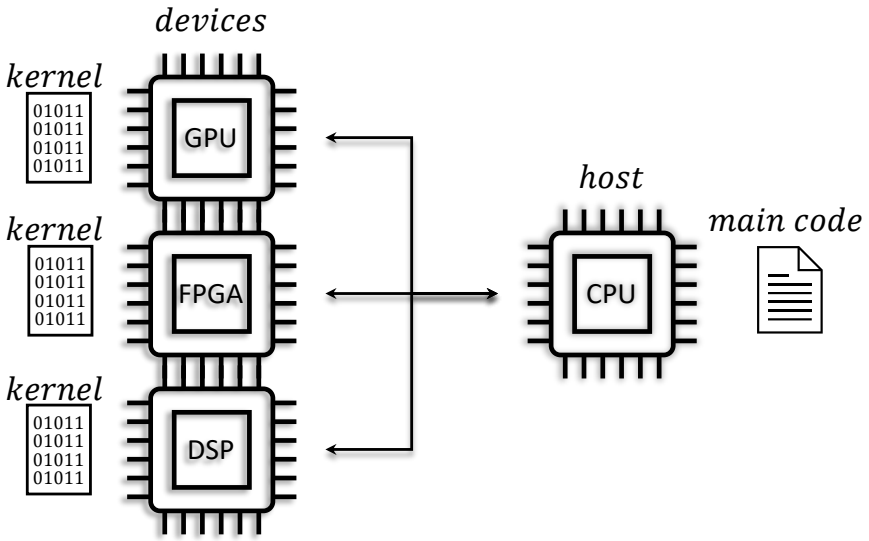

Figure 2.9: Platform model or heterogeneous system.

Execution model

The execution model define the way in which kernels will be executed. The main program (host program) is responsible for preparing the environment to launch the kernels to the devices [79]. There are some concepts needed to understand the execution model. A *work-item* is the basic unit of work on a device, work-items are grouped into *work-groups*. A *kernel* is the code to be executed by one or several work-items, the *program* is the set of kernels. Finally, the *context* is the environment in which the kernels will run [43].

Memory model

The memory model defines 4 address spaces (Figure 2.10): private (per work-item), local (shared by work-group), constant (read function only and visible to all work-groups) and global (read and write functions and visible to all work-groups) [79, 64]. When kernels are defined, the location where the data will be placed must also be explicitly indicated: *__global* if the data will be sent to the global memory region of the device, *__constant* if the data will be sent to the constant memory region of the device, *__local* if the data will be sent to the local memory region of the device. By default variables are private within a kernel. It is also important to note that according to the location where

the data is stored, the access speed and storage capacity will be inversely proportional. Private memory is the fastest among the four, but has the smallest capacity, on the other hand, the global memory has the largest capacity but the access speed is slower [7].

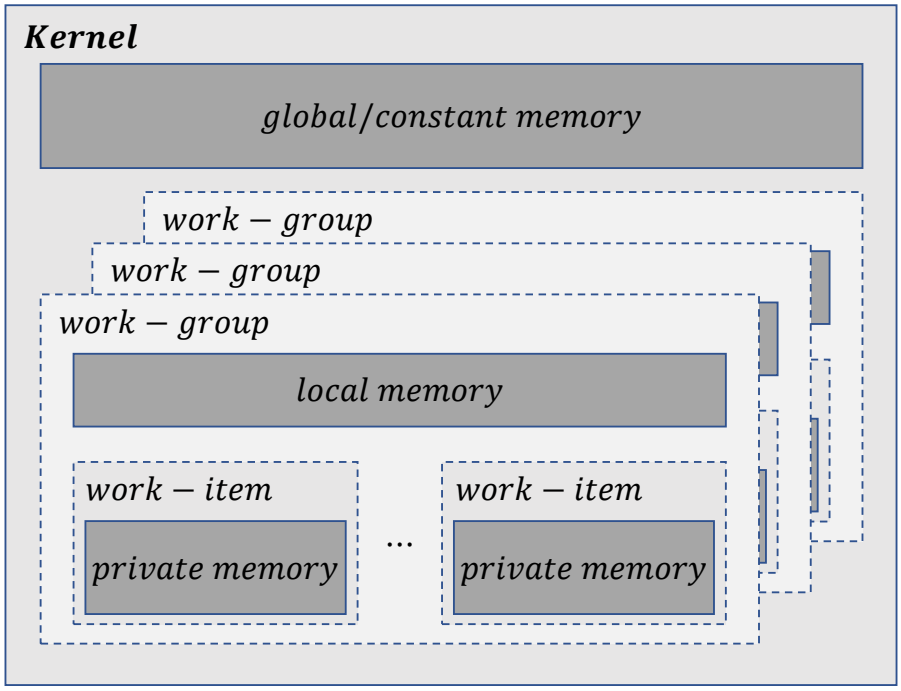

Figure 2.10: OpenCL memory model.

As already mentioned, the first step to develop an application with OpenCL is to code the program that will be executed on the host side. Considering the information presented in [79] the following steps must be performed:

1. Discover and initialize the available device(s) in the heterogeneous system.
2. Test the device(s) to verify that the designed kernel(s) is(are) adapted to its characteristics.
3. Create a context and a command queue.
4. Read and build the source code which defines the kernel(s).
5. Set the memory objects to be configured on the device(s).
6. Write host data to the device(s) memory object(s).

7. Create and compile the kernel(s).
8. Set the kernel arguments.
9. Run the kernel(s) on the device(s).
10. Retrieve the values returned by the device(s).

Chapter 3

Related Work

Differential equations model SNN's neurons, which differ from each other only in the synaptic weights. A neuron model does not require much computation, but hundred, thousand, or millions of neurons do. Consequently, the research community and the semiconductor companies have started developing specialized hardware to train and simulate SNNs. These hardware platforms are known as neuromorphic and enable the efficient training and execution of SNNs. Since these platforms are still under development, they are not available for retail.

Without access to neuromorphic hardware, some researchers utilize conventional computer systems such as general-purpose and high-performance computers with GPUs to perform their studies and developments with SNNs. Moreover, embedded computers like SBCs have gained importance for AI applications because they possess powerful multicore processors and include GPUs. In particular, embedded GPUs' computational power has increased and enables the acceleration of AI applications like SNNs.

In this section, we review the state-of-the-art, mainly the works related to the deployment of SNNs on computing systems; we detail SNN characteristics such as network architecture, type of application, type of neuron, and software tools used for the implementation. In addition, we will review the use of SBC in several works as an alternative platform for AI application deployment to support its use and utility in these applications.

3.1 Neuromorphic hardware

SNNs have attracted neuroscientists and engineers interested in algorithms and network architectures. Furthermore, hardware designers have shown interest in developing neuromorphic hardware to make SNN's computation as efficient as possible. Those hardware systems consist of ASIC or FPGA chips, and their primary purpose is to simulate large numbers of spiking neurons. The following are some of the most outstanding projects, which we will classify into two categories: digital and mixed-signal (analog and digital) systems.

3.1.1 Digital Systems

SpiNNaker [27] is a fully digital system aiming to simulate massive spiking networks in real-time. SpinNNaker uses an event-driven simulation technique. It comprises 864 ARM9 cores, divided into 48 chips containing 18 cores each. The main feature of SpiNNaker is its efficient communication system. A high-throughput network interconnects all the nodes. The communication network routes small packets, which contain Address Event Representation (AER) spikes, i.e., the address of the transmitter neuron, the date of the emission, and the destination neuron [27]. SpiNNaker is programmable using the PyNN library. PyNN is a Python library for SNN simulation [18, 19], which provides various neuron models (LIF, Izhikevich, etc.) and synaptic plasticity rules such as Spike-Time-Dependent Plasticity (STDP).

TrueNorth [4] is a fully digital system capable of simulating up to 1 million spiking neurons. It consists of 4096 Neurosynaptic cores dedicated to LIF neuron emulation [2]. TrueNorth can perform 46 billion synaptic operations per second (SOPS) per Watt. The system is programmable thanks to the Corelet programming language [6], allowing to tune neuron parameters, synapse connectivity, and inter-core connectivity. Similar to SpiNNaker, the communication scheme in TrueNorth is asynchronous, event-based, and able to tolerate a very high level of parallelism [2].

Loihi [17] is a fully-digital chip containing 128 cores. Each core can simulate up to 1024 different neurons. It supports up to 130.000 neurons and 130 million synapses, and it can perform online learning [17]. The chip can be programmed to implement various learning rules, notably STDP, and simulate up to 30 billion SOPS [2].

SpiNNaker, TrueNorth, and Loihi are neuromorphic systems built from application-specific integrated circuits (ASICs). This fact represents an advantage over SNN implementation on GPC, HPC or SBCs concerning power efficiency and performance.

FPGAs are an alternative technology for implementing SNNs accessible to anyone able to program them. Minitaur [61] and HFirst [63] are neuromorphic platforms based on FPGAs. Minitaur is an event-driven neural network accelerator dedicated to high performance and low power consumption. This SNN accelerator implemented on the Xilinx Spartan 6 FPGA board has an architecture consisting of 32 LIF-based cores dedicated to parallel processing spikes [61]. HFirst is based on a frame-free paradigm, as it takes inputs from a Dynamic Vision Sensor (DVS) [50] and is used for temporal pattern recognition. It runs on Xilinx's Spartan 6 FPGA with a 100MHz clock and consumes between 150mW and 200mW [63].

3.1.2 Mixed-signal systems

BrainScaleS [68] is a mixed-signal system. The processing units (neuron cores) are analog circuits, whereas the communication units are digital. It implements the adaptive exponential Integrate-and-Fire (EIF) neuron model, which can reproduce many biological firing patterns [68]. It is composed of HiCANN (High-Input Count Analog Neural Network) chips, which can simulate 224 spiking neurons and 15.000 synapses [68]. The PyNN interface allows users to program the network similarly to SpiNNaker.

NeuroGrid [10] is a mixed-signal system that targets real-time simulation of large SNNs. The board comprises 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 can simulate up to 1 million neurons and billions of synaptic connections [10].

We can highlight some common facts among the neuromorphic platforms reviewed. Only NeuroGrid, BrainScaleS, and SpiNNaker can perform online learning, and none of the FPGA-based platforms can carry out this type of learning. FPGA-based works support large-scale simulations and generally support integrate and fire neuron models. SpiNNaker exhibits lower power efficiency, mainly when it simulates the more complex

neuron and synapse models [11]. Table 3.1 summarizes neuromorphic hardware architectures and their most essential features.

Table 3.1: Hardware platforms to simulate SNNs.

Device	**Electronics**	**Neuron Model**	**Application Domain**
SpiNNaker [27]	Digital	LIF, IZH, HH	NeuroSciences
TrueNorth [4]	Digital	LIF	Multi-object detection
HFirst [63]	Digital	Complex IF	DVS-based object recognition
Minitaur [61]	Digital	LIF	Classification
Loihi [17]	Digital	CUBA LIF	LASSO/classification
BrainScaleS [68]	Analog/Digital	exp IF	NeuroSciences/ Classification
NeuroGrid [10]	Analog/Digital	Dimensionless Model	NeuroSciences

Regarding the power consumption in Figure 3.1 [13], it is clear that neuromorphic systems consume several times less power than the human brain. It is precisely the feature that makes them such suitable for SNNs. On the other hand, conventional computing systems equipped with accelerators such as GPUs or TPUs consume higher power than neuromorphic systems. Although programs with equivalent functionality to be deployed on neuromorphic hardware can run in conventional computer systems plus accelerators, these are slower and less energy efficient, but until neuromorphic hardware becomes commercially available, they should be used to continue the software component development of SNNs.

3.2 Conventional computer systems plus accelerators

As we have stated before, neuromorphic hardware is not commercially available yet. FPGA alternative is not the first choice for deploying SNNs due to its programming complexity. Hence, SNNs researchers use conventional computer systems in their studies.

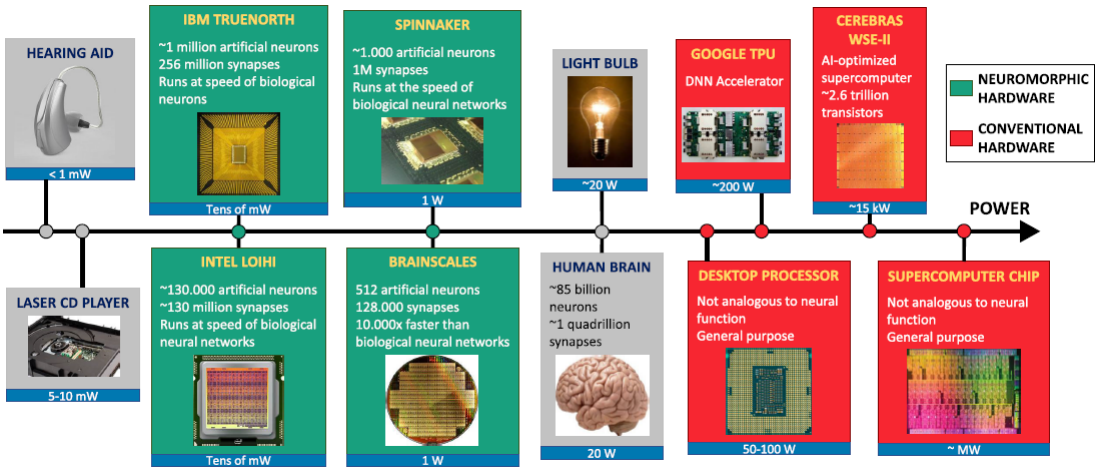

Figure 3.1: Power consumption of neuromorphic hardware versus conventional computing systems [13].

There are many works where GPCs and HPCs with GPUs serve as computing platforms for implementing SNNs. This section presents these works.

Ahmadi et al. [3] implemented an Izhikevich [37] neuron-based simulator using a multilayer SNN structure partially connected, where the number of neurons in the network was scaled up to 1.000.000 to evaluate GPU's performance. The GPU-SNN model was deployed on three different systems: a laptop with NVIDIA GeForce GT 325M with 48 GPU CUDA cores, a desktop computer with NVIDIA GeForce GT 9500 with 32 GPU CUDA Cores, and a desktop computer with NVIDIA GeForce GT 9400 with 16 GPU CUDA Cores. This work concludes that the simulator has excellent flexibility and performance in large-scale simulations. They also identify the GPU's memory as the main bottleneck for the simulator.

Nageswaran et al. [59] implemented an Izhikevich neuron-based large-scale SNN simulator, considering a network with random connections between excitatory neurons and inhibitory neurons. The network size comprises 100.000 to 220.000 neurons. The work's purpose was to identify the performance of SNNs using GPU by measuring fidelity, memory usage, and scalability. A workstation with an Nvidia GTX 280 GPU was used as the platform for carrying out the trials. The authors reported 24 times higher performance in the GPU simulations compared to the CPU simulation for 100,000 neurons with 50 million synaptic connections.

Kulkarni et al. [45] implemented an SNN for classifying handwritten digits in real-time. They used a three-layer topology with LIF neurons. The hidden layer performed feature extraction and the output layer classification. The former had 8112 neurons and used the NormAD gradient-descent supervised learning algorithm to adjust the SNN synaptic weights. The output layer had ten neurons (one neuron for each digit) and used fixed synaptic weight maps to extract the key features of the image. The network was implemented using an Nvidia GTX 860M GPU with 640 CUDA cores.

It is important to note that the works mentioned previously used the Compute Unified Device Architecture(CUDA) framework as the development tool for the algorithms' implementation on the GPUs. LIF or IZH neuron models are used to perform the experiments both for their biological plausibility and computational complexity. Depending on the objective to be achieved with the work, biological plausibility can be sacrificed to gain performance in the execution of the application, or performance can be improved by sacrificing accuracy in the task performed by the network.

3.3 GPGPUs applications on SBCs

Works such as Nazir et al. [60] show the current interest in developing AI applications employing SBCs to take advantage of their energy efficiency and reduced form factor. The authors developed a car detection system using computer vision and applying techniques like Histograms of Oriented Gradients (HOG) and Support Vector Machines (SVM). They implemented this system on a Raspberry Pi 3 and an Odroid C2. However, they only used the CPUs because these two SBCs do not have GPUs for general-purpose computing. Embedded GPUs for general-purpose computing have been available only in the last decade and were restricted generally to graphic processing given the lack of support of hardware and software [49]. We present now some works that use GPUs in applications that require energy or computational efficiency on SBCs.

Lindner et al. [51] conducted studies for implementing a face recognition system on a Raspberry Pi 3B+, an NVIDIA Jetson Nano, and a Banana Pi M3. They aim to evaluate the possibility of carrying face recognition and detection systems in small, low-power devices. The Jetson Nano exhibits higher performance than its competitors,

mainly because it is the only one with a GPU. The study concludes that although the application can be performed on all the computer systems tested, the Jetson is the most efficient in terms of execution time, but if very fast inference times are not required, the Raspberry Pi offers a greater benefit in production costs.

In [44], Kim et al. developed a driver status monitoring system utilizing Convolutional Neural Networks (CNNs). They implemented the system on an NVIDIA Jetson Nano and its GPU. The results show that MobileNetV2 can execute in real-time on the Jetson Nano. They also highlighted the importance of the system memory size and how this affects the overall system budget.

Matthews and Leger in [56] used an NVIDIA Jetson Nano SBC for developing an anomaly detection system, aiming to improve computing speed. They performed an energy consumption analysis and reported that the system's peak power consumption using the GPU reaches up to $9.4 Watts$.

Haogang et al. in [25] investigated the performance of the You Only Look Once (YOLO) network using different SBCs: Raspberry Pi 4B with Intel Neural Compute Stick 2, NVIDIA Jetson Nano and NVIDIA Jetson Xavier NX. The results showed that the Raspberry with the accelerator is better to lightweight models. The Jetson Xavier NX obtained the best results in execution time for all the models analyzed, but consumed twice as much energy as the other systems. The study therefore concludes that the Jetson Nano offers the best performance and cost.

According to the works, as mentioned earlier, the use of GPUs for general-purpose computing has been evidenced only in the NVIDIA Jetson boards, limiting the applications developed only to the NVIDIA software and hardware components installed on the mentioned SBC. Furthermore, there is no evidence in the literature review on using SBCs for SNN applications.

Chapter 4

Methodology

This chapter will cover the aspects considered for the development of the project. Firstly, we will discuss the selection of the architecture, the selection of the dataset, the training process carried out, and the deployment of the algorithm to perform inference on GPUs. We will also describe the performance evaluation mechanism considered. Finally, we will provide details of the platforms and software used for the experimental and deployment stage.

4.1 Methodology for Network Deployment

The primary purpose of this work is to implement SNNs in a GPU-enabled SBC efficiently. To reach this goal, we divided our work into four steps (see Figure 4.1):

- We selected an SNN architecture and its baseline implementation in the BindsNET simulator [32].
- We used the MNIST dataset [57] to train the SNN for handwritten digit recognition and obtain the network parameters and reference output.
- We developed a version of the SNN inference algorithm for the GPU-enabled SBC and Workstation deployment.
- We evaluated the performance of the SNN inference in both platforms and the energy efficiency.

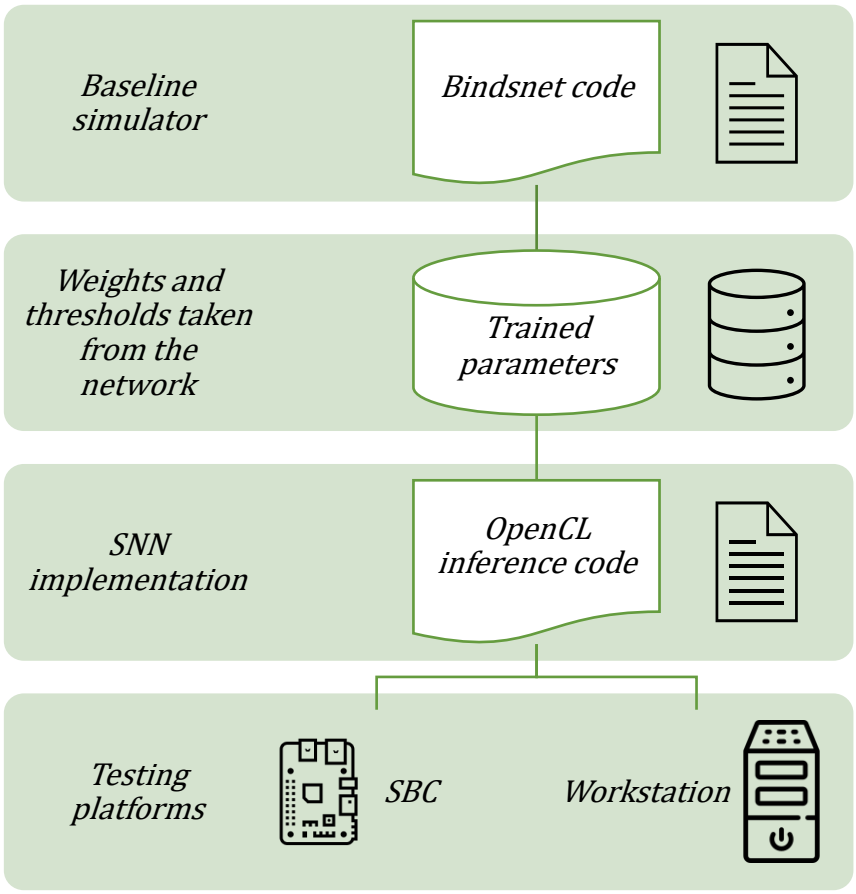

Figure 4.1: Block diagram of the project development process.

4.1.1 Selection of the network architecture

The first step in the project was to carry out a literature review to investigate the fundamental components to assemble a spiking network to proceed with the implementation at the software level. Therefore, our project was based on the architecture worked by [22, 67] as the baseline for training the SNN's parameters and as a reference to validate the output of the inference algorithm that interests us. In the next chapter, we will give more details since it is required to specify aspects such as the type of neuron, coding technique, topology, differential equation solution method, etc.

4.1.2 Dataset selection and network training

Several related papers seeking to explore the response of spiking networks use the MNIST Dataset to perform handwritten digit recognition [22, 46, 67, 84]. At this stage,

after choosing the network parameters and architecture training was performed using BindsNET. For network training, we explored different network sizes and the optimal time window, and subsequently we extracted the parameters required by the inference algorithm: weights and firing thresholds of excitatory neurons. This process will be explained in more detail in the following chapter.

4.1.3 Algorithm deployment to perform network inference

In this part, we first performed a sequential implementation of the algorithm to validate its suitability for the classification process using the parameters obtained after the training stage. These validations will be explained in chapter 5. After this, some optimizations were made to improve the performance of the application not only from the parallelization of the key points, but also from the data input to the network with the compression of the input signals to the first layer.

4.1.4 Performance evaluation of deployment on SBC-GPU and Workstation-GPU

Last but not least, we validate this implementation against the BindsNET simulator outputs. We evaluated the performance of SNN inference in SBCs against the workstation and their energy efficiency. For this purpose, we define 3 different network sizes and used 10000 images from the MNIST database as the set of tests. We used the *time* Linux command for timing measurements, covering the whole application running time, not only the GPU compute part. We consider this command reliable enough given that the time measurements were in the order of tens of seconds. We used a Uni-T Professional Digital Multimeter UT70A [74] to measure both systems' voltage and current consumed at the power outlet. For these measurements we proceeded to set up the instrument in AC configuration as shown in the Figure 4.2 and Figure 4.3, that are described in the user's manual of the instrument [74]. We calculated the total energy consumed with these parameters and the runtime measured in each experiment. We performed 10 repetitions of the experiments and took the nominal value of each metric which are considered the final values to be reported in the experimental results chapter.

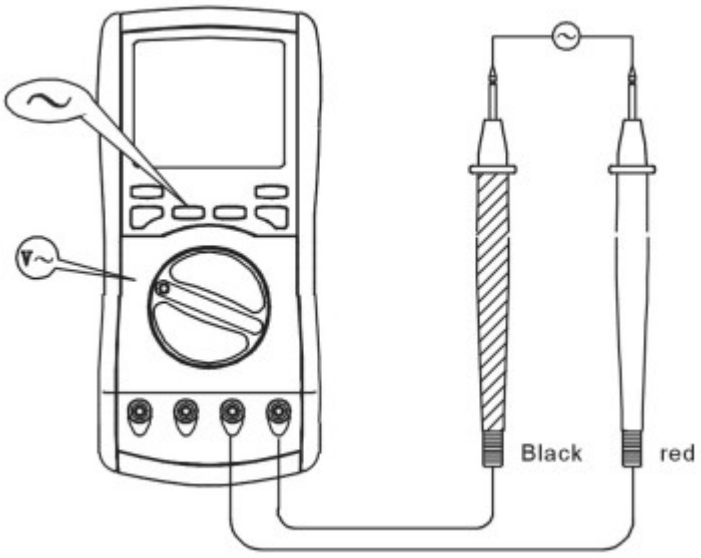

Figure 4.2: Setup procedure for using the instrument for voltage measurement [74].

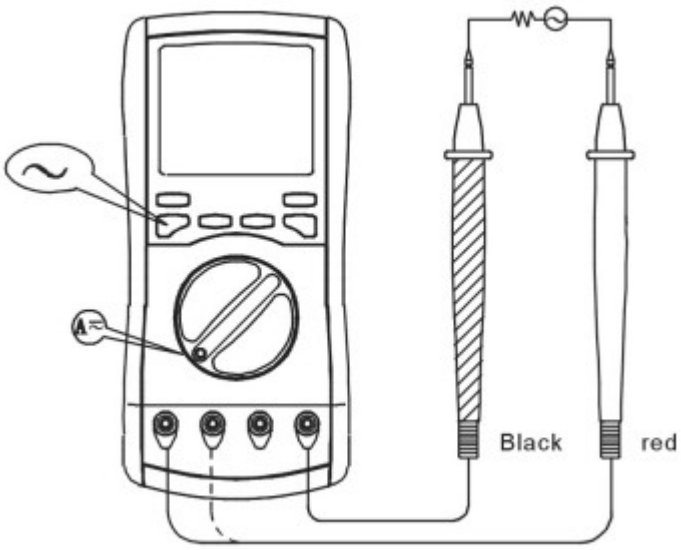

Figure 4.3: Setup procedure for using the instrument for current measurement [74].

4.2 Target Platforms

We used two computer systems for the experiments: a Dell Precision 5820 Workstation and a VIM3 Pro. Table 4.1 shows the most relevant features of the platforms selected for the experiments. It should be noted that the difference in hardware resources between

the platforms is considerable, so for our work it is important to highlight the convenience of using SBCs for inference tasks deploying SNNs with the graphics unit, to be considered as an alternative for mobile computing or edge computing applications. Among the remarkable differences is the considerable price difference that the two systems represent, on the one hand the Workstation GPU has a cost of 829.00 USD [5] while the entire SBC VIM3 PRO has a cost of 159.99 USD [41] at the date of this document.

Table 4.1: Workstation and SBC features.

Feature	**Workstation**	**SBC**
Reference system	Dell Precision 5820 Workstation	VIM3 Pro
CPU	6-Core Intel Xeon W-2135@3.7GHz	2xCortex-A52@1.8GHz + 4xCortex-A73@2.2GHz
Operating System	Ubuntu 18.04.6	Ubuntu 20.04
G++ Compiler	Version 7.5.0	Version 9.4.0
OpenCL version	2.2	2.2
GPU		
Device name	Nvidia Quadro P5000	ARM Mali-G52
Max. Compute units	20	2
Max. Clock frequency	1733MHz	750MHz
Max. Work item sizes	1024x1024x64	384x384x384
Max. Work group size	1024	384
Preferred work group size multiple	32	8
Global memory size	15.89GiB	3.621GiB
Global memory cache size	960KiB	128KiB
Local memory size	48KiB	32KiB

4.3 Performance metrics

In order to compare the performance and energy efficiency of SNN's inference algorithm against the workstation counterpart, we used the following metrics:

- Digits per second: the number of digits available in MNIST (10000) divided by the total execution time measured.
- Digits per Watt-hour: the number of digits available in MNIST (10000) divided by the total energy measured.
- SOPS: Synaptic Operations per Second. This metric is calculated using the following equation:

$$SOPS = (N * (I + N - 1) + N)/((1/DR)/ST) \tag{4.1}$$

  Where N is the number of neurons, I the number of inputs of the network, DR the number of digits processed per second or throughput, and ST the number of simulation steps to obtain the digit classification.
- SOPS/W: It is the number of Synaptic Operations Per Second per Watt.

Chapter 5

SNN Implementation

In this chapter, we will describe the implementation process carried out for the network simulation. First, we will review the main components of the network architecture, then we will describe how the information that is transported through the network was represented. We will continue with the details of the implementation of the sequential program describing the verification processes that were carried out to validate the correct implementation, and finally, we will provide details of both the sequential and parallel programs.

We developed a C version of the SNN for handwritten digit recognition and selected part of the algorithm for offloading to the GPU. To run the compute intensive parts on the GPUs, we used the OpenCL library, which allows us to run the same code both on the embedded GPU and on the workstation GPU. Furthermore, we applied some strategies in order to achieve an efficient implementation.

It is important to note that we train the SNN with BindsNET in the workstation because we are focused on optimizing the performance of the network inference process. In addition, BindsNET was used exploratorily in the SBC to perform the inference but we found two limitations that highlight the relevance of our project and that we will detail below:

- When we tried to perform inference with BindsNET, it was not possible to use the SBC-GPU for handling the tensors required in Pytorch for the simulation. This is because currently there is no PyTorch support for SBC-GPU architectures.

- In this context, the simulation was performed using the SBC-CPU but when trying to scale the neurons, an execution error occured again because of PyTorch's limitation with the handling of tensors for SBCs for 800 and 1600 neuron networks. So, this second limitation leads us to desist from testing with BindsNET on the SBC due to the constraints that PyTorch introduces in our experiments.

For the reasons previously described, we chose to perform the algorithm deployment on the SBC with OpenCL.

5.1 Network Architecture

Diehl and Cook in [22] proposed the SNN architecture used in their research for handwritten digit recognition using an unsupervised learning technique, and other authors [77, 82] have used this work as reference for their experiments with SNNs. A LIF neuron model [29] has been selected for this architecture considering the results presented by Valadez-Godínez et al. in [80](Table 2.2), in which the LIF neuron model is referenced as the most computationally efficient using Euler methods (Forward Euler or Exponential Euler), and the work done by Capra et al. [13] where they compare the biological plausibility versus the computational efficiency of LIF, IZH, and HH neuron models (Figure 2.3).

A two-layer topology with one excitatory and one inhibitory layer symmetrically distributed was implemented using the Winner-Take-All competitive system for the inference process. Poisson rate coding for encoding the input pixel into spike trains is used to feed the input layer of the network. Figure 5.1 shows the network architecture. The circles in the graph represent the LIF neurons; excitatory neurons are labeled exc_n; while the inhibitory neurons are labeled inh_n. The small circles with the plus (+) symbol that lead to the excitatory neurons represent the synapses. Each network layer has K neurons. During our experiments, we explore three values for the K parameter (see chapter 6).

The SNN's input is the digit images of 28x28 pixels. The 784 pixels are encoded into spike trains through a Poisson rate coding strategy using a time window of 64 ms. Figure 5.2 presents the encoding process step by step. Thus, spike-trains for each pixel become a binary signal where we represent spikes with 1's and its absence with 0's. Instead of computing this encoding process every time an image feeds the network, we

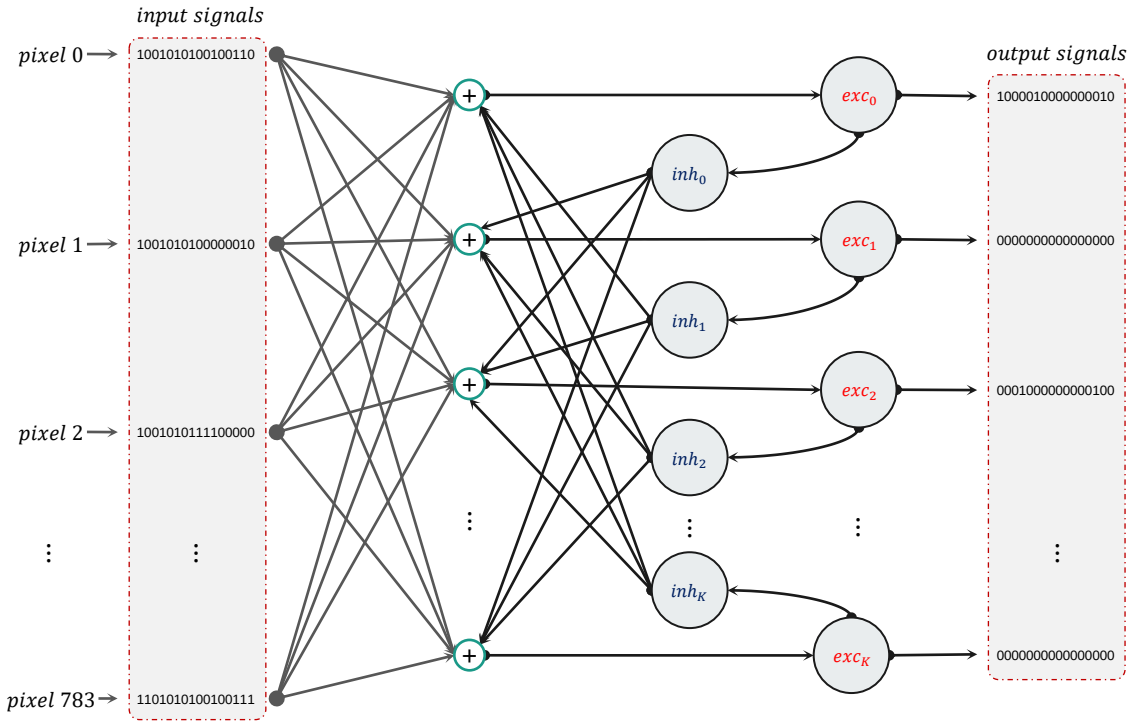

Figure 5.1: Two-layer SNN architecture using LIF neurons.

saved the complete encoding for each image in the dataset and reused it. Hence, we stored each digit on the MINST dataset as a 784 64-bit integer matrix. As a result, the SNN can work directly with spike encoded images, and we can focus on the actual SNN's inference performance. In Figure 5.3 we can see an example of the zero digit converted from pixels to a spike train. The vertical axis represents each pixel of the image and the x-axis the 64ms that represent 64 simulation steps in our system.

A synapse integrates multiple input signals into a single excitatory current to the neuron. Each synapse on the left side of the excitatory neurons aggregates all input signals into each excitatory neuron. Moreover, the synapses on the right side of the excitatory neurons aggregate the signals from all inhibitory neurons but themselves. Equation 2.13 defines how to compute the synapse. In summary, every excitatory neuron receives the 784 input signals and $K - 1$ inhibitory signals through its synapse and produces an output.

Every neuron in the inhibition layer takes the output of one excitatory neuron. In

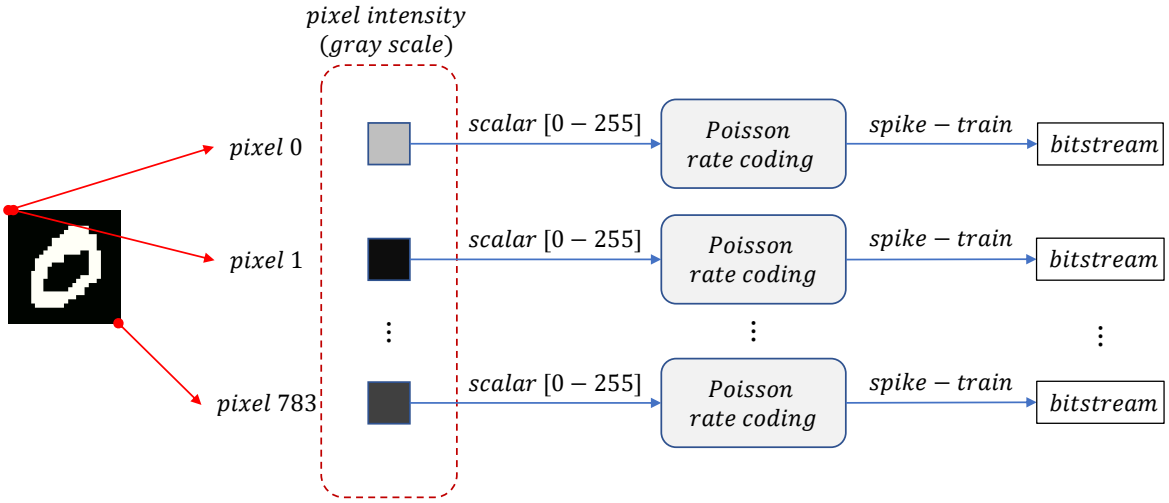

Figure 5.2: Image encoding process step by step.

the case of a spike, the inhibitory neuron sends an inhibitory spike back to all other excitatory neurons. Thus, the excitation layer recognizes specific patterns in the input image, and the inhibition layer helps the network select the most active excitatory neurons to produce the encoded output.

Finally, the first excitatory neuron firing will determine the recognized digit by applying the soft Winner-Take-All mechanism [53].

5.2 Input data compression

In the baseline implementation, the input spike-trains are generated as vectors containing boolean types, each requiring one byte (8 bits) to store each sample. We modified the code in such a way that the spike-train was encoded with bits instead of bytes. In such a scheme we managed to compress the input data 8×. This optimizations allows us to reduce the memory footprint of the application which allows for a faster data upload to the network per iteration, as there is an improvement in the execution of the complete database. This procedure is illustrated in the Figure 5.4.

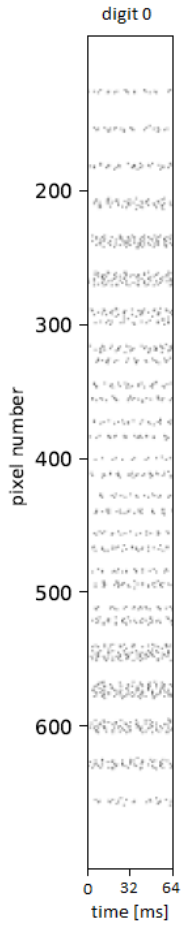

Figure 5.3: Representation of digit 0 after being transformed from pixels to a poisson signal.

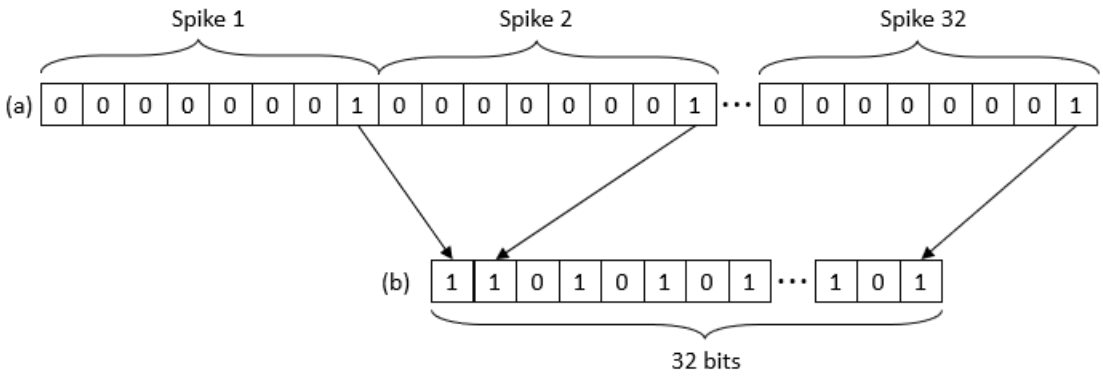

Figure 5.4: Compression of spikes in bits: (a) Pytorch spike representation using 8 bits, (b) Representation of the spikes in our system.

5.3 Sequential implementation

Based on the selected architecture, a C implementation of the inference algorithm was developed. In this implementation, the objective was to represent as accurately as possible both, the process and the outputs generated by BindsNET. In this section, we will give details of the implementation of the sequential program and the validations carried out with the data.

5.4 Sequential program

We present in Figure 5.5 the implementation of the sequential program for which the previously described validations were made. In the first stage of the execution is where the loading of the parameters trained with BindsNET is performed and also the initialization of the vectors that store the total input synapses, the membrane voltage vectors, the spike vectors and the refractory time vectors necessary for the correct simulation of the network. Subsequently, there are two nested loops, the first one that performs the counting for each of the 10000 samples that will be entered into the network, and the second inner loop that corresponds to the evolution of the membrane voltage signals over time. The inner loop is where the computation of synapses for excitatory and inhibitory neurons is performed, and it is selected from the output spikes that excitatory neurons produce the winner in case of several firing at the same time by applying the WTA mechanism.

5.4.1 Validation of data integrity resulting from training

Considering that training was performed using tensors from the PyTorch framework and also computed on the Workstation GPU, it was necessary to validate that the information would remain consistent both for testing with the Workstation CPU and for transferring it to the SBC for the inference process.

To make the transfer process uniform and avoid losing decimal precision with the floating point units, the information was transferred in binary format for both the trained weights and the neuron firing thresholds of the network. In Figure 5.6 we can see a graphical representation of the weights obtained from the training process for the neural network with 400 excitation neurons would look like. In the image we can see

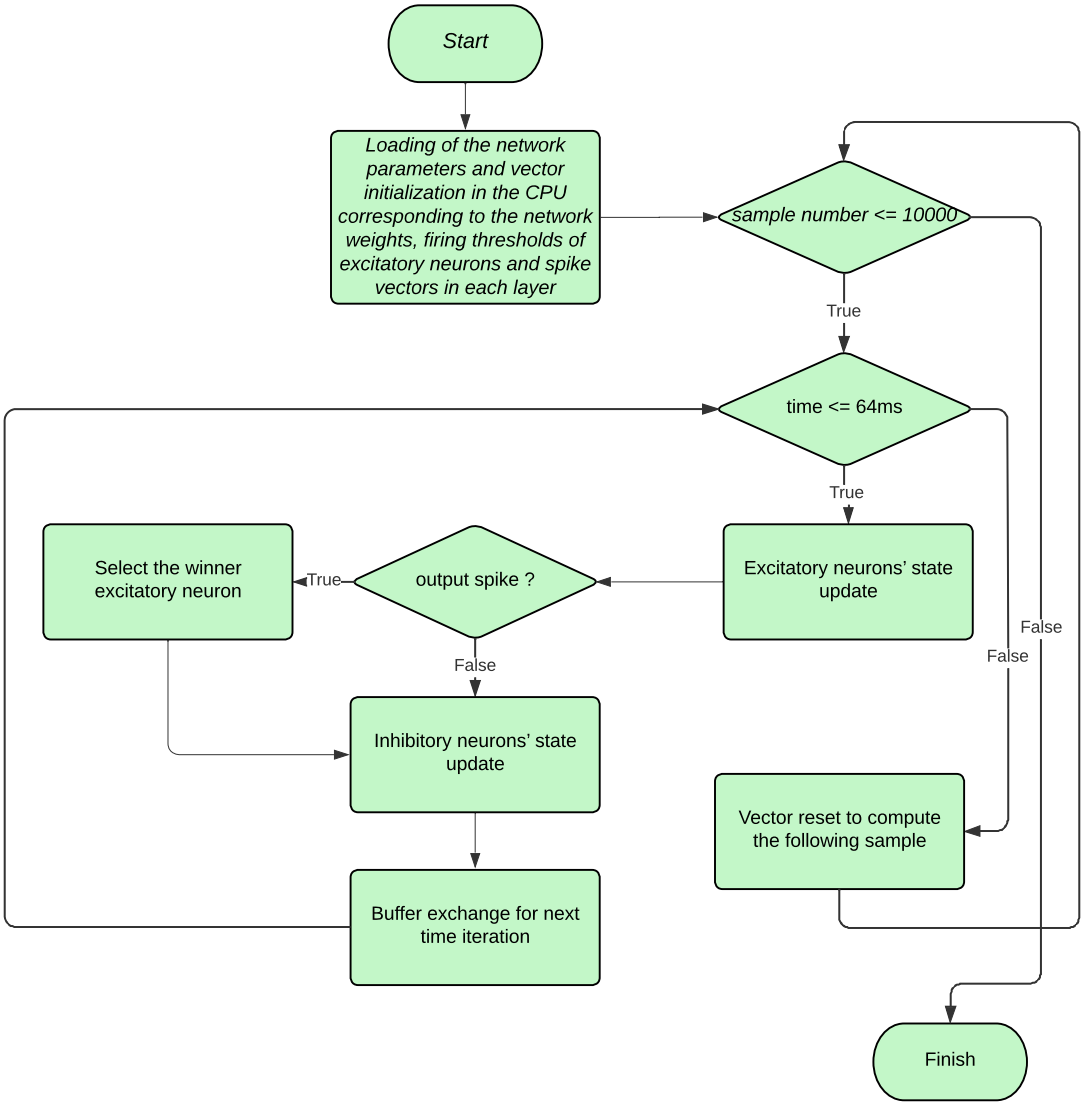

Figure 5.5: Flow diagram for sequential implementation of the algorithm.

400 digits arranged in 20 rows with 20 columns. Each digit corresponds to the filter associated to each excitation neuron with the trained weights. For example, the upper left corner represents the filter associated with the first excitation neuron that will be in charge of classifying patterns similar to digit 9. This filter receives synaptic connections from the input layer so it has 784. The color scale represents the strength of the synaptic connection, where values close to 1 reflect a strong synaptic connection and values close to 0 reflect a weak synaptic connection which prevents it from being sensitive to input signal stimuli.

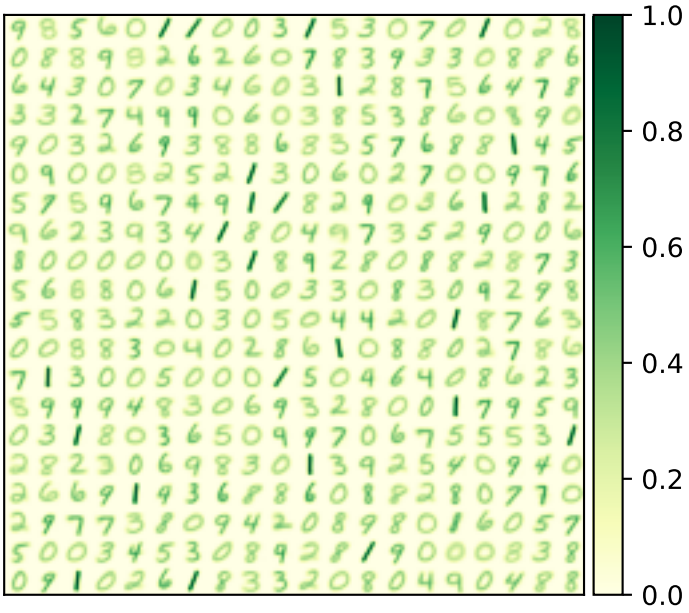

Figure 5.6: Representation of the weights corresponding to the synaptic connections of the input layer with the layer of excitatory neurons.

5.4.2 Validation of the input synapse and membrane voltage in the excitatory layer

For this case, the tests were performed considering the training data for the 400-neuron network in the excitation layer. The procedure consisted of capturing the input synapse and membrane voltage of excitatory neurons. For synapse, information from both the input layer and inhibitory neurons was stored for the input synapse. These synapses were subsequently stored in binary form on the hard disk. In the case of membrane voltage, we considered storing this variable for both excitatory and inhibitory neurons. The above procedure was followed in the same way in BindsNET to verify the suitability of the data generated with our implementation.

To verify the consistency of the data, a Python program was developed to load the data and compare the information to verify both, the similarity of the data and the decimal accuracy. It is important to clarify that the information generated in BindsNET (using

PyTorch) stores the data for the synapse with 32-bit float data types and therefore in our case we use the same data type.

5.4.3 Validation of spike train vectors in the time domain

Because the spikes occur at a given instant of time, a capture of the spikes generated by both the input layer, excitation neurons and inhibition neurons is performed both, in BindsNET and in our implementation to validate that the equations defining the firing thresholds are satisfied for all t. Figure 5.7 shows how for any layer of the network, a vector of spikes converted to binary is obtained, showing both the neuron and the time instant where the neuron generates the spikes.

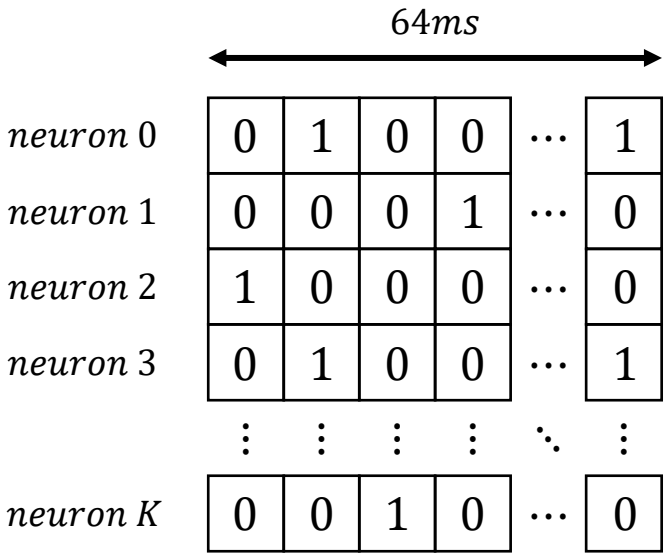

Figure 5.7: Matrix of spikes generated for the validation process in the layers of the network.

5.4.4 Validation of digit classification

For this case the output of the classification process, i.e. the digit predicted by the network is stored in a file. The same process was followed with BindsNET and finally the files were cross-checked to verify the correspondence in the classification process.

The validations previously performed allowed us to ensure that the serial implementation

was carried out properly and, the required adaptations for the parallel implementation were made.

5.5 Parallel Program

In this section, we will discuss the fundamental aspects of the program implementation developed in OpenCL to be executed in the two selected computer systems. In the first instance, the implementation executed by the CPU is considered, since it is in charge of executing the instructions corresponding to the loading of the trained network parameters and the initialization of the vectors that allow the evolution of the network over time. After initialization, the communication interface with the GPU must be configured in the main program, and the buffers for transporting data vectors to/from the GPU must be initialized. Next, the kernel containing the code that allows the computation of synapses of the network layers and the selection of excitatory neurons using WTA must be launched. Subsequently, the vectors that perform the spike counting are retrieved from the GPU to finalize the classification process in the main program. Finally, the vectors are reinitialized to process the next input sample in the network. In the Figure 5.8 we can see the flowchart corresponding to the components implemented by CPU and GPU respectively for the execution of the network.

5.5.1 CPU code

The CPU code is the part that manages the overall behavior of the application and the interaction with the GPU accelerator. In the code delivered to the CPU is where the loading of data into the program is performed, the link to the GPU is configured, data is sent and received to/from the GPU and in case of having several associated devices such as multiple GPUs, FPGAs or other hardware accelerators, this code performs the entire process of control and management. In the following, we will explore the steps required to deploy our program on the CPU:

1. Loading of the network parameters that were obtained in the training stage using BindsNET.

2. Vector initialization in the CPU corresponding to the network weights, firing thresholds of excitatory neurons and spike vectors in each layer. In addition, the

number of Work Groups (WG) and Work Items (WI) for the GPU kernel execution are defined.

3. Platform and context creation using OpenCL commands to configure the GPU as an associated computing device.

4. Kernel loading onto the GPU.

5. Definition of arguments to be sent to the kernel and writing of buffers containing the data structures to be sent to GPU.

6. External loop where each of the 10000 samples of the test set are read and sent to the kernel.

7. Internal loop where the runtime window of the spiking network is defined.

8. Kernel launching for a fixed number of neurons.

9. Reading of spikes in excitatory neurons to identify which digit has been classified. According to the one with the highest number of spikes on the output.

10. After passing all the samples to the network, a validation of the number of network hits is carried out, taking as a reference the classification accuracy performed by BindsNET.

5.5.2 GPU code

The following are the main steps of the algorithm that performs the computations required at each layer of the SNN architecture. These part of the program runs on the GPU accelerator:

1. *Synaptic currents update:* the calculation of the input synaptic current is performed in this step for all excitatory neurons. Here, both the pre-synaptic connections of input layer and inhibitory neurons are considered to find the total synaptic current.

2. *Excitatory neurons' state update:* in this step, the differential equations characterizing the LIF excitation neurons are computed and, the output spikes are obtained if the activation threshold defined for each neuron has been exceeded to be evaluated at time $t + 1$.

3. *The winning neuron is selected with WTA:* the soft Winner-Take-All mechanism is applied to select only one of the excitatory neurons if more than one has generated a spike at time t.

4. *Synaptic current is updated for inhibitory neurons:* synaptic current is updated for the inhibition neuron group for time t. For this case, we consider that an inhibitory neuron j has only one pre-synaptic connection from an excitatory neuron i.

5. *Update the state of inhibitory neurons:* the differential equation characterizing the LIF inhibitory neurons are computed and, the output spikes are obtained if the activation threshold defined for each neuron has been exceeded to be evaluated at time $t + 1$.

6. *Update the vectors for time t+1:* in this step is where the data corresponding to the vectors of state t are transferred to the vectors of state $t + 1$.

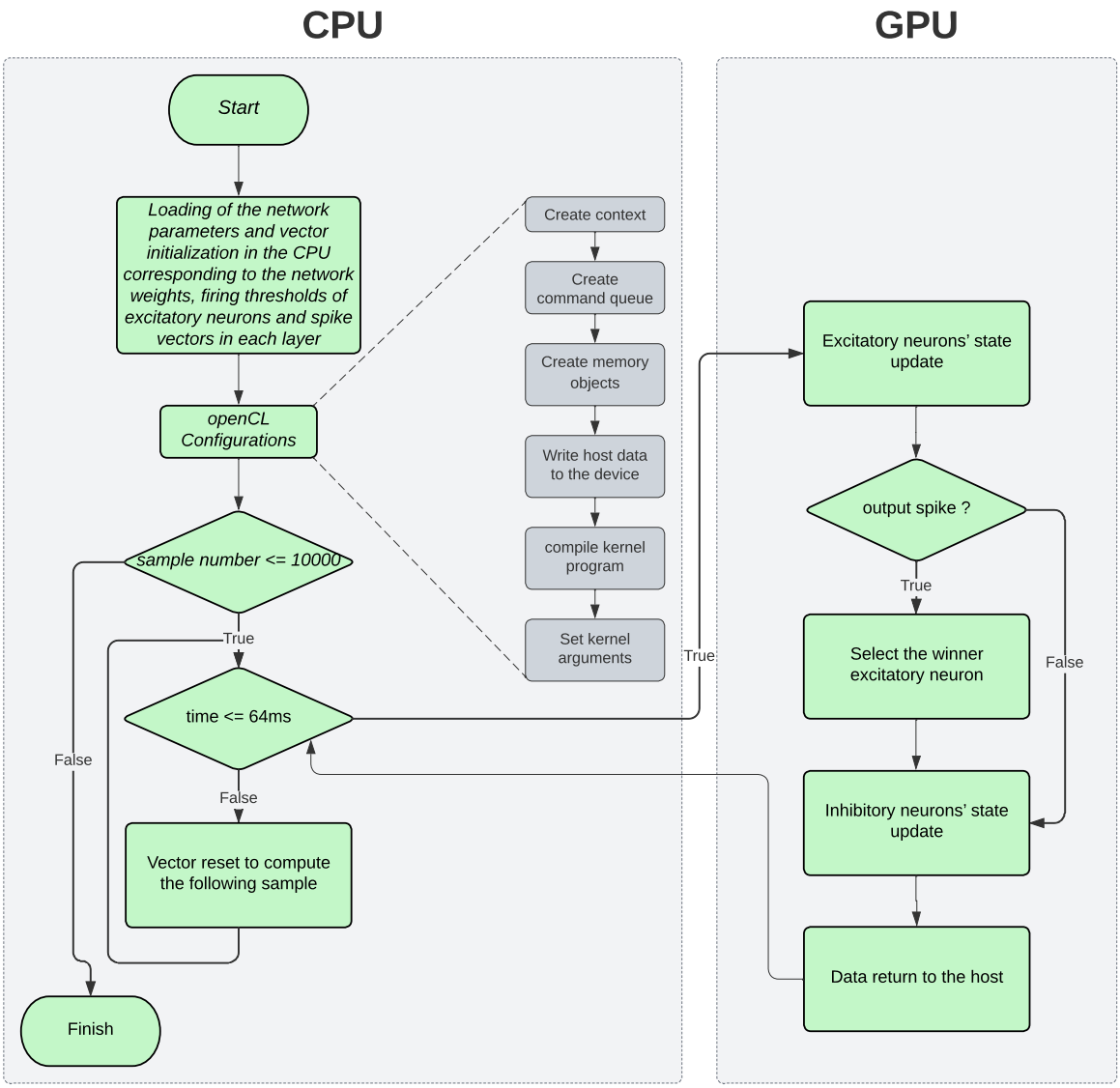

Figure 5.8: Flowchart corresponding to the parallel implementation of the algorithm.

Chapter 6

Experimental Results

For our experiments we first identified the optimal temporal window length to perform the inference process using the BindsNET simulator, since as we will see later on, generating an inference using large window lengths directly affects the execution time but the whole network must be computed per simulation step, and it does not improve the accuracy of classification significantly. We will then evaluate the OpenCL implementation to measure both the system performance in terms of throughput and power.

6.1 Time window tuning for simulations

The Poisson-based generation of the input spike-trains in the baseline implementation used a window size of 250 ms, which translates into 250 simulation steps. We performed experiments with the length of this window since there is no evidence in the literature of an optimal window size and that the accuracy of the pattern recognition process depends on it. Figure 6.1 presents the results after performing the simulation window tuning experiments. We obtained that by reducing the window progressively, an approximately constant accuracy level was maintained until reaching the 64ms simulation window, when reducing to 32ms there was a significant degradation of the performance of the network for the classification task, so it was decided to keep all subsequent experiments with the window length at 64ms. An aspect to highlight also in the selection of the simulation window, is that by having 64ms we will be having 64 simulation steps so that each spike train can be simulated with 64 bits by having a binary signal or in terms of variables of 8 bytes.

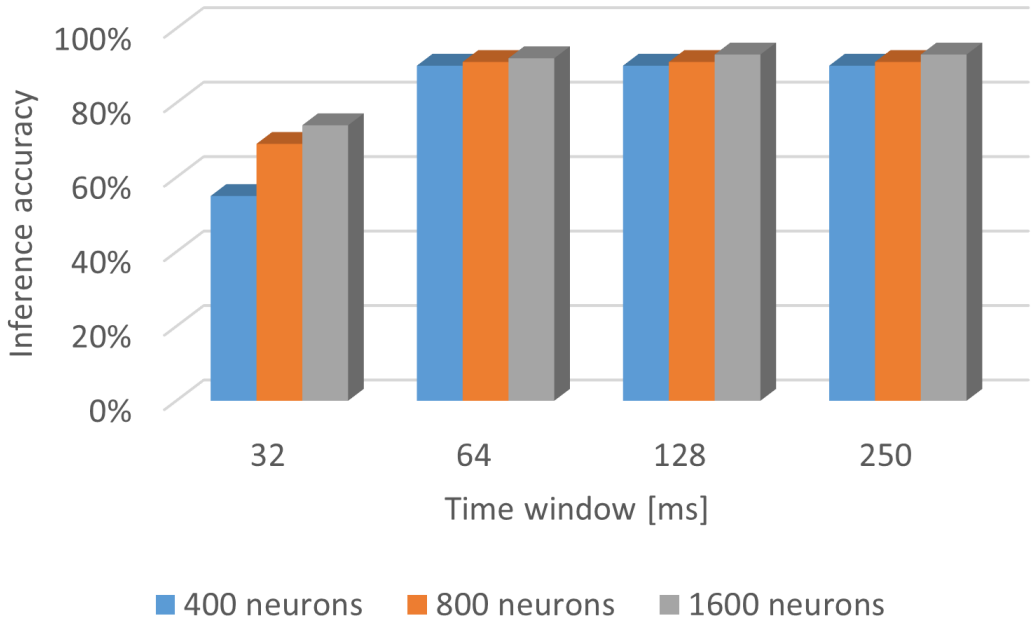

Figure 6.1: Analysis window over time vs. algorithm accuracy in the inference process using BindsNET on Workstation.

6.2 Network training

After selecting the simulation window and leaving it fixed at 64ms. We performed experiments with BindsNET in order to identify the number of neurons needed to obtain the accuracy in the training and inference. The results can be seen in Figure 6.2, Figure 6.3, Table 6.1 and Table 6.2 . In which we can see that for network sizes considering the number of excitatory neurons between 100 and 6400, the time used by the Workstation GPU to perform the inference remains practically constant. But, for training, a slight increase in time is perceived considering that the network is scaling.

A considerable increase in the number of neurons from 1600 to 6400 does not present a significant change in the network's accuracy percentage, neither for training nor for inference. But there is a negative effect on performance considering that training time increases by 4.5× and the time to perform inference increases by 3×. From this evaluation we can see that scaling the number of neurons, does not allow us to obtain substantial improvements that would allow us to accept the excessive time required to perform both, the training of the network and the inference.

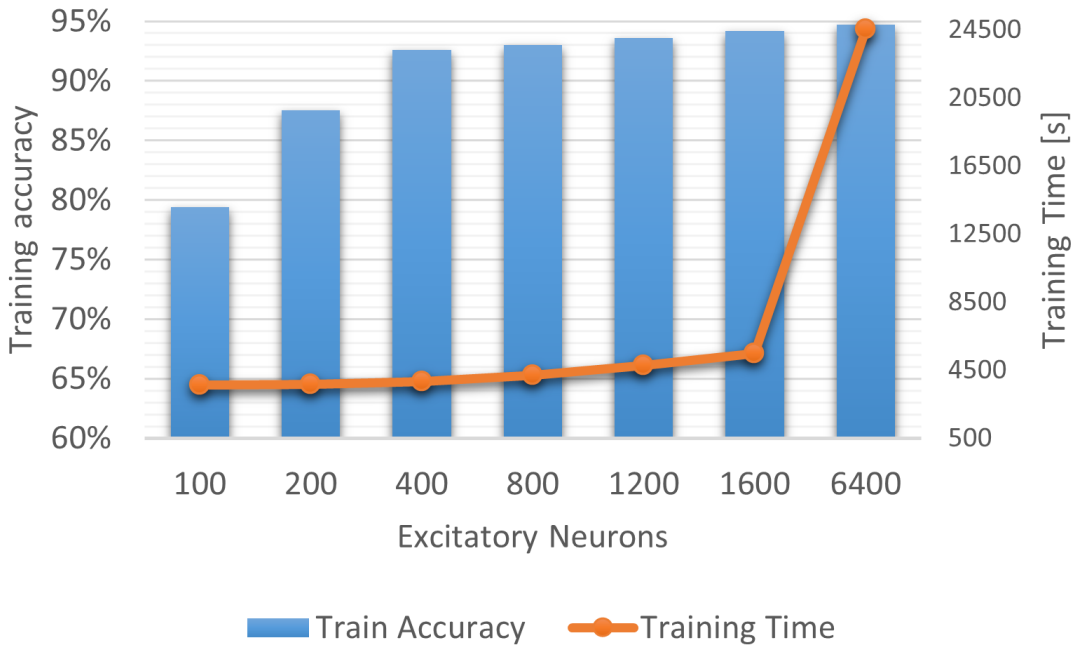

Figure 6.2: Training accuracy vs. Training time using BindsNET on Workstation.

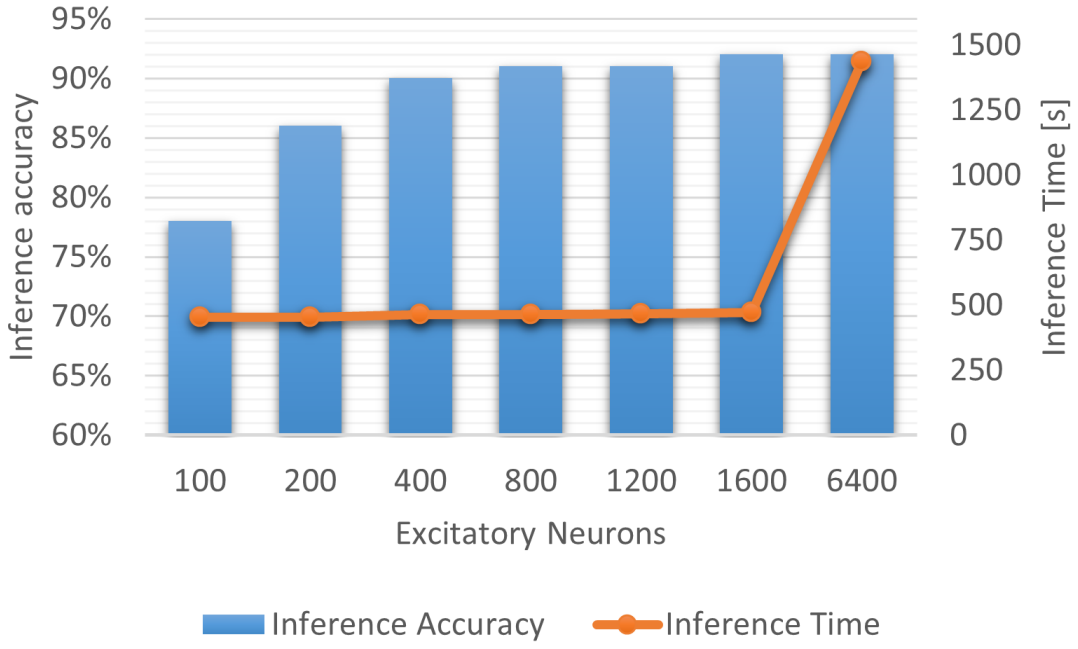

Figure 6.3: Inference accuracy vs. Inference time using BindsNET on Workstation.

Table 6.1: Accuracy of the network for the training stage

Number of neurons	100	200	400	800	1200	1600	6400
Train Accuracy	79%	88%	93%	93%	94%	94%	95%

Table 6.2: Accuracy of the network for the inference stage

Number of neurons	100	200	400	800	1200	1600	6400
Inference Accuracy	78%	86%	90%	91%	91%	92%	92%

6.3 Inference sequential code

After the process carried out with BindsNET in the training stage, the characteristics to be considered in the network for the inference process were obtained, as shown in Table (6.3).

In Figure 6.4 we can see the throughput of the SNN in VIM3 and workstation with the sequential code processing the 10,000 samples from the testing set of the database. As the network scales, performance degrades considerably, which allows us to later compare the optimizations made and the execution performed on the GPU to identify the feasibility of implementing applications that involve SNNs and SBC-GPUs.

6.4 Performance and energy efficiency

Our primary purpose is to evaluate the feasibility of SBCs for implementing applications with SNNs. We evaluated the performance of our SNN's inference algorithm for the three network sizes (400, 800, and 1600), the two hardware platforms (SBC+GPU and Workstation+GPU), and nine work group values (2, 4, 5, 8, 10, 16, 20, 40, 100). Figure 6.7 presents the performance of each configuration in digits per second. As expected, the more powerful and costly Workstation+GPU achieves a higher performance. When having 400 and 800 neuron networks, the performance gap between the two hardware systems for the best Work Group configuration is 6.2× and 6.5×, respectively. For 1600 neurons, the gap grows significantly to 7.8× because at that point, the parallel resources

Table 6.3: Simulation features

Feature	Value	
Number of neurons (excitatory and inhibitory)	400, 800, 1600	
Strength of synapse weights from excitatory to inhibitory layer	22.5	
Strength of synapse weights from inhibitory to excitatory layer	-120	
Simulation time step	1ms	
Simulation time	64ms	
	Excitatory neurons	**Inhibitory neurons**
Rest voltage	-65mV	-60mV
Post-spike reset voltage	-60mV	-45mV
Spike threshold voltage	variable	-40mV
Post-spike refractory period	5ms	2ms

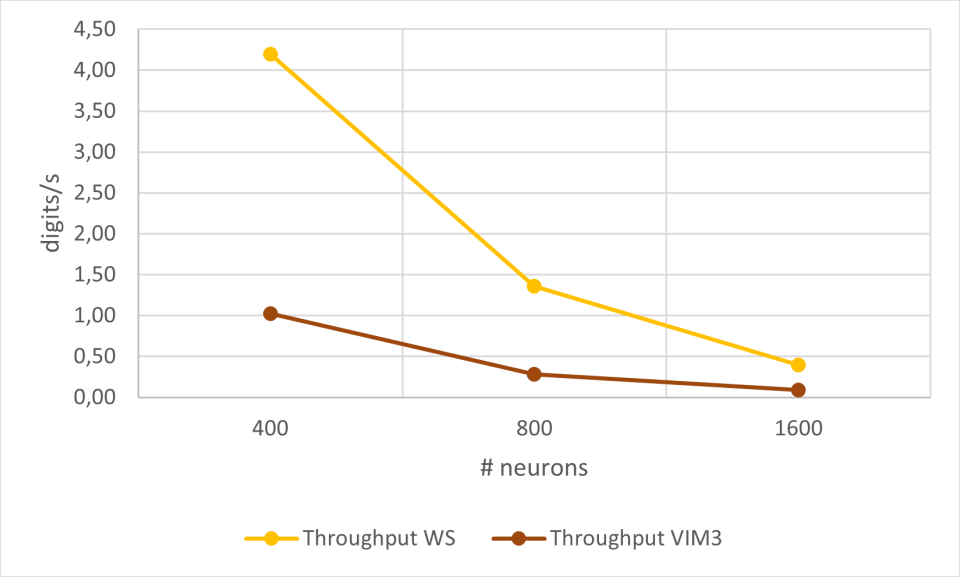

Figure 6.4: Serial implementation throughput VIM3 and Workstation.

of the embedded SBC-GPU are exhausted, while the larger workstation GPU's resources are not.

Concerning the Work Group exploration, we can observe that its impact on performance for the SBC+GPU platform is negligible. On the other hand, the Work Groups have a notable impact on the performance of the Workstation+GPU platform. Thus, a 400 neuron SNN achieved maximum performance with 20 Work Groups, 8 Work Groups for 800 neurons, and 20 Work Groups for 1600 neurons. OpenCL uses Work Groups and Work Items abstractions to distribute the work among the GPU computing units. The number of Work Items is limited to 384 in the SBC+GPU and 1024 in the Workstation+GPU. That is why there are missing bars in Figure 6.7 for some SBC+GPU configurations.

Regarding the number of neurons, the networks with 400 neurons achieved the highest performance in all configurations since they require fewer computations. Of course, this translates into slightly lower accuracy at digit recognition. In Figures (6.5, 6.6) we can see the difference in performance achieved in the network inference process after performing the optimizations in the base algorithm.

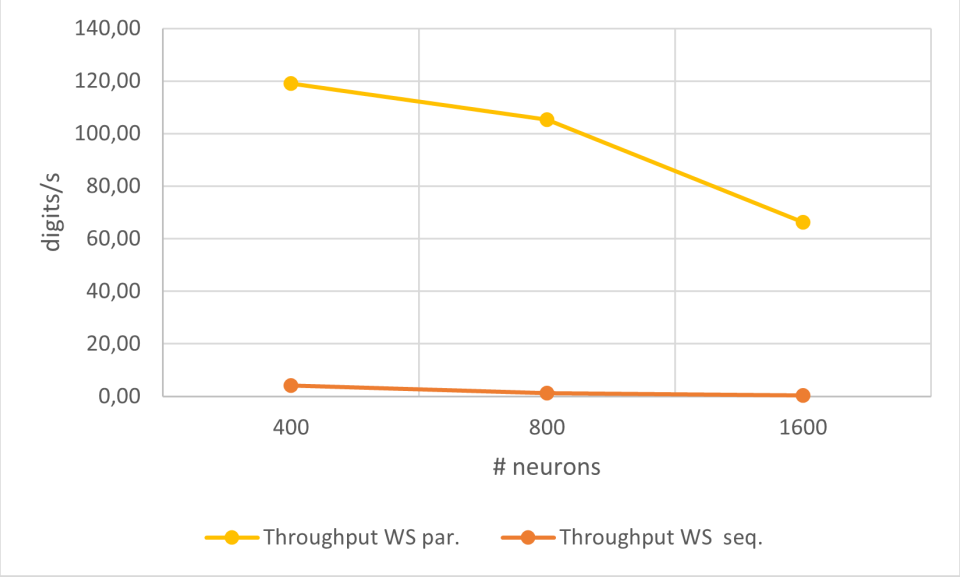

Figure 6.5: Performance comparison between sequential and parallel implementation on the Worsktation.

Although the inference process is faster on the Workstation GPU, the SBC-GPU is more power-efficient. Table 6.4 puts together a summary of the speed results next to

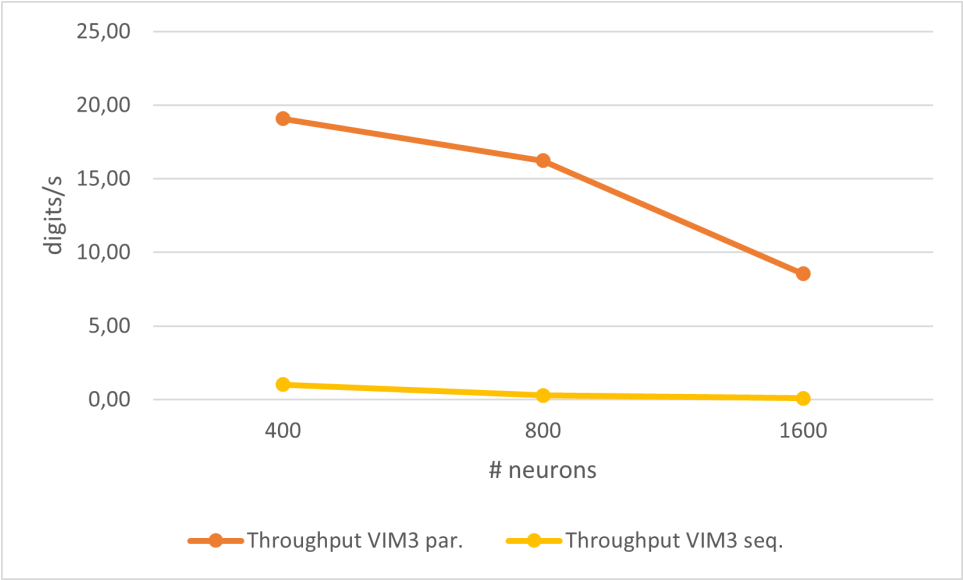

Figure 6.6: Performance comparison between sequential and parallel implementation on the VIM3.

the energy efficiency metric. We can see the maximum achieved values of throughput and energy of the inference process with the network of 400 excitation neurons using the GPUs.

Table 6.4: Speed and energy efficiency comparison.

System	Excitatory Neurons	Throughput $[digits/s]$	Kdigits/Wh	GSOPS	GSOPS/W
SBC	400	19.08	22.80	0.58	0.19
	800	16.23	19.40	1.32	0.44
	1600	8.55	10.22	2.09	0.69
Workstation	400	119.05	3.29	3.61	0.03
	800	105.26	2.91	8.54	0.07
	1600	68.97	1.91	16.84	0.13

We can identify that there is a profit margin with respect to SOPS and SOPS/W that our system achieves with respect to neuromorphic systems such as SpiNNaker (0.064 GSOPS; 0.064 GSOPS/W [8]), which is von-Neumann-based. However, in systems such as the Truenorth (58 GSOPS; 46 GSOPS/W [8]), which is a neurosynaptic processor with 4096 cores, the results illustrate the real performance gaps between a system designed and built to compute SNNs and our platform.

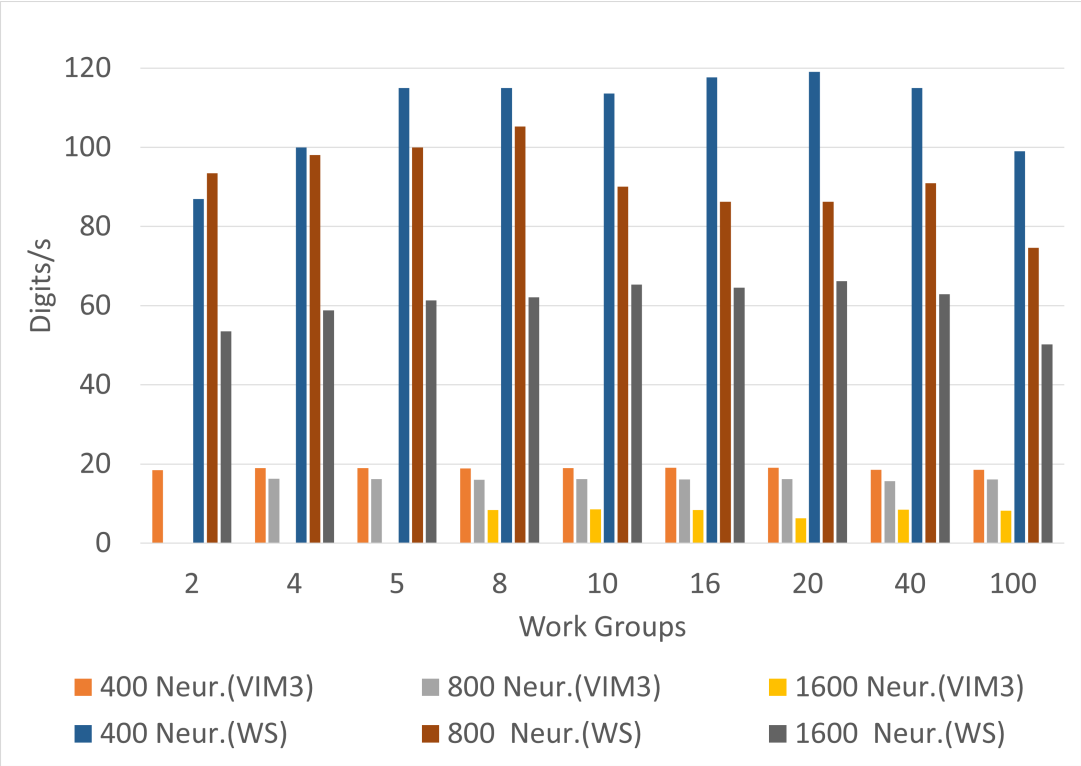

Figure 6.7: Throughput achieved in the experimental stage with SBC-GPU and Workstation GPU.

With respect to the results achieved, we can see an advantage in terms of power consumption of embedded GPUs, because as we saw in Figure (3.1) the power is comparable to that used for the brain but keeping the distance in terms of the amount of synaptic operations and also that the brain has many areas dedicated to other processes such as the motor process, which not only perceives the environment but also must generate motor control actions.

The workstation has an advantage in terms of computation due to its advanced and powerful GPU for both the training stage of the network and for performing inference, however in terms of consumption and portability the embedded system is a great alternative for deploying applications that require edge computing, and even for applications in the area of Internet of things that the large dimensions and weight of the workstation would make it impossible to deploy.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this work we presented the implementation and performance evaluation of a GPU-enabled Vim3 Pro card and a workstation for Spiking Neural Networks. We tailored the SNN for handwritten digital recognition and measure the execution time and energy consumption in both systems.

Experimental results showed that despite the significant processing speed advantage of the workstation, the energy efficiency of the SBC creates an opportunity to build cluster systems based on SBCs to compete with workstations or conventional computing clusters for brain simulation research, for instance.

On the other hand, these results show that SBC cards deployed on edge devices can be used efficiently to run Artificial Intelligence applications based on SNNs, where the power budget is typically highly constrained.

As for the software tools involved, it's important to notice that the use of flexible programming libraries such as OpenCL, allows engineers to easily migrate codes from a high-end computing platform such as a workstation to budget systems such as the Vim3 Pro.

The window reduction technique showed in this research, is an example of optimizations that are required when porting neural network applications to constrained systems

such as the SBCs. There is a small loss of accuracy but this can be balanced by using more complex neuron models or by using a more sophisticated inhibition scheme that increases the performance of the competitive system.

Finally, the results provided us with a great opportunity to develop applications that are focused on portable and edge computing, for example for autonomous robotic systems or for offline applications for pattern recognition. In raw computational terms it is evident the gap that exists between the SBC and the Workstation, for this reason it is not recommended to use the SBC to train this network, since the main objective of the work is in the inference process which is where the model deployment works properly.

7.2 Future Work

Different architectures could be evaluated to determine whether a numerical method that provides a more accurate solution of the differential equations like Fourth-Order Runge-Kutta, could make it possible to reduce the number of neurons needed to have the same accuracy for pattern recognition applications such as handwritten digit recognition. Also, it could be considered to choose a more biologically complex neuron model combined with a numerical method to identify the effects on the scalability of the network to achieve the same level of accuracy in the classification task, regarding a reduction in the number of neurons which are the fundamental processing unit.

Further work could also consider giving a greater role to the inhibitory layer to experiment with its effects on the network response, since in articles such as [34] it is concluded that modifying the inhibitory layer schemes can lead to greater biological plausibility of the network, and thus compensate for the characteristics of the LIF neuron model that allowed us to have good performance measures.

It would be interesting to test the algorithm implemented on neuromorphic hardware in order to verify the efficiency of the algorithm implemented on embedded GPUs. This would largely depend on whether one of the more advanced systems such as Intel's LOIHI is released for the market for easy access or consider the option of developing on an FPGA.

References

[1] Nassim Abderrahmane, Edgar Lemaire, and Benoît Miramond. "Design Space Exploration of Hardware Spiking Neurons for Embedded Artificial Intelligence". In: *Neural Networks* 121 (Oct. 2019), pp. 366–386. DOI: 10.1016/j.neunet.2019.09.024. arXiv: 1910.01010.

[2] Nassim Abderrahmane, Edgar Lemaire, and Benoît Miramond. "Design Space Exploration of Hardware Spiking Neurons for Embedded Artificial Intelligence". In: *Neural Networks* 121 (Jan. 2020), pp. 366–386. ISSN: 18792782. DOI: 10.1016/j.neunet.2019.09.024.

[3] Arash Ahmadi and Hamid Soleimani. "A GPU based simulation of multilayer spiking neural networks". In: *2011 19th Iranian Conference on Electrical Engineering*. 2011, pp. 1–5.

[4] F. Akopyan et al. "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* 34.10 (2015), pp. 1537–1557.

[5] Amazon. *Quadro P5000*. Available at https://www.amazon.com/PNY-Quadro-P5000-VCQP5000-PB-estaci%C3%B3n/dp/B01N6W4CVB?ref_=ast_sto_dp [Accessed 5 Nov 2022].

[6] A. Amir et al. "Cognitive computing programming paradigm: A Corelet Language for composing networks of neurosynaptic cores". In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. 2013, pp. 1–10.

[7] Jason D Bakos. *Embedded Systems: ARM Programming and Optimization*. Morgan Kaufmann, 2015.

[8] Arindam Basu et al. "Spiking Neural Network Integrated Circuits: A Review of Trends and Future Directions". In: *2022 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE. 2022, pp. 1–8.

[9] Trevor Bekolay et al. "Nengo: a Python tool for building large-scale functional brain models". In: *Frontiers in Neuroinformatics* 7 (2014), p. 48. ISSN: 1662-5196. DOI: 10.3389/fninf.2013.00048. URL: https://www.frontiersin.org/article/10.3389/fninf.2013.00048.

[10] B. V. Benjamin et al. "Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations". In: *Proceedings of the IEEE* 102.5 (2014), pp. 699–716.

[11] Maxence Bouvier et al. "Spiking neural networks hardware implementations and challenges: A survey". In: *ACM Journal on Emerging Technologies in Computing Systems* 15 (2 2019). ISSN: 15504840. DOI: 10.1145/3304103.

[12] Nicolas Brunel and Simone Sergi. "Firing Frequency of Leaky Intergrate-and-fire Neurons with Synaptic Current Dynamics". In: *Journal of Theoretical Biology* 195.1 (1998), pp. 87–95. ISSN: 0022-5193. DOI: https://doi.org/10.1006/jtbi.1998.0782. URL: https://www.sciencedirect.com/science/article/pii/S0022519398907822.

[13] Maurizio Capra et al. "Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead". In: *IEEE Access* 8 (2020), pp. 225134–225180.

[14] Frances S Chance, Sacha B Nelson, and Larry F Abbott. "Complex cells as cortically amplified simple cells". In: *Nature neuroscience* 2.3 (1999), pp. 277–282.

[15] Ting Shuo Chou et al. "CARLsim 4: An Open Source Library for Large Scale, Biologically Detailed Spiking Neural Network Simulation using Heterogeneous Clusters". In: *Proceedings of the International Joint Conference on Neural Networks* 2018-July (2018), pp. 1158–1165. DOI: 10.1109/IJCNN.2018.8489326.

[16] Jose M. Cruz-Albrecht, Michael W. Yung, and Narayan Srinivasa. "Energy-efficient neuron, synapse and STDP integrated circuits". In: *IEEE Transactions on Biomedical Circuits and Systems* 6.3 (2012), pp. 246–256. ISSN: 19324545.

[17] M. Davies et al. "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning". In: *IEEE Micro* 38.1 (2018), pp. 82–99.

[18] Andrew Davison et al. "PyNN: a common interface for neuronal network simulators". In: *Frontiers in Neuroinformatics* 2 (2009), p. 11. ISSN: 1662-5196.

[19] Andrew Davison et al. “PyNN: towards a a universal neural simulator API in python”. In: *BMC Neuroscience* 8.S2 (2007), P2. ISSN: 1662-5196.

[20] Peter Dayan and Laurence F Abbott. *Theoretical neuroscience: computational and mathematical modeling of neural systems*. MIT press, 2005.

[21] Lei Deng et al. “Rethinking the performance comparison between SNNS and ANNS”. In: *Neural networks* 121 (2020), pp. 294–307.

[22] Peter U. Diehl and Matthew Cook. “Unsupervised learning of digit recognition using spike-timing-dependent plasticity”. In: *Frontiers in Computational Neuroscience* 9.August (2015), pp. 1–9. ISSN: 16625188. DOI: 10.3389/fncom.2015.00099.

[23] Pangao Du et al. “An Unsupervised Learning Algorithm for Deep Recurrent Spiking Neural Networks”. In: *2020 11th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*. 2020, pp. 0603–0607. DOI: 10.1109/UEMCON51285.2020.9298074.

[24] Leonhard Euler. *Institutionum calculi integralis volumen primum.* Vol. 2. Academic Press, 1769.

[25] Haogang Feng et al. “Benchmark Analysis of YOLO Performance on Edge Intelligence Devices”. In: *Cryptography* 6.2 (2022). ISSN: 2410-387X. DOI: 10.3390/cryptography6020016. URL: https://www.mdpi.com/2410-387X/6/2/16.

[26] Andreas K Fidjeland and Murray P Shanahan. “Accelerated simulation of spiking neural networks using GPUs”. In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2010, pp. 1–8.

[27] S.B. Furber et al. “The SpiNNaker project”. In: *Proceedings of the IEEE* 102.5 (2014), pp. 652–665.

[28] Andrew A George et al. “A diversity of synaptic filters are created by temporal summation of excitation and inhibition”. In: *Journal of Neuroscience* 31.41 (2011), pp. 14721–14734.

[29] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, Aug. 2002. ISBN: 97805-21813846. DOI: 10.1017/cbo9780511815706.

[30] Wulfram Gerstner et al. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014, pp. 1–577. ISBN: 9781107447615. DOI: 10.1017/CBO9781107447615.

[31] Marc-Oliver Gewaltig and Markus Diesmann. "Nest (neural simulation tool)". In: *Scholarpedia* 2.4 (2007), p. 1430.

[32] Hananel Hazan et al. "BindsNET: A Machine Learning-Oriented Spiking Neural Networks Library in Python". In: *Frontiers in Neuroinformatics* 12 (2018), p. 89. ISSN: 1662-5196. DOI: 10.3389/fninf.2018.00089.

[33] Hananel Hazan et al. "BindsNET: A machine learning-oriented spiking neural networks library in python". In: *Frontiers in Neuroinformatics* 12.December (2018), pp. 1–18. ISSN: 16625196.

[34] Hananel Hazan et al. "Unsupervised learning with self-organizing spiking neural networks". In: *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2018, pp. 1–6.

[35] B. Hille. *Ion Channels of Excitable Membranes*. 3rd. Sinauer Associates, INC., 2001. ISBN: 0-87893-321-2.

[36] A. L. Hodgkin and A. F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of Physiology* 117.4 (1952), pp. 500–544. ISSN: 0022-3751.

[37] E. M. Izhikevich. "Simple model of spiking neurons". In: *IEEE Transactions on Neural Networks* 14.6 (2003), pp. 1569–1572.

[38] Hyeryung Jang, Nicolas Skatchkovsky, and Osvaldo Simeone. "VOWEL: A local online learning rule for recurrent networks of probabilistic spiking winner-take-all circuits". In: *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE. 2021, pp. 4597–4604.

[39] Nikola K. Kasabov. "NeuCube: A spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data". In: *Neural Networks* 52 (2014), pp. 62–76. ISSN: 18792782. DOI: 10.1016/j.neunet.2014.01.006. URL: http://dx.doi.org/10.1016/j.neunet.2014.01.006.

[40] Bahadir Kasap and A. John van Opstal. "Dynamic parallelism for synaptic updating in GPU-accelerated spiking neural network simulations". In: *Neurocomputing* 302 (2018), pp. 55–65. ISSN: 18728286. DOI: 10.1016/j.neucom.2018.04.007. URL: https://doi.org/10.1016/j.neucom.2018.04.007.

[41] KHADAS. *Khadas Shop: Vim3 Pro Edition*. Available at https://www.khadas.com/product-page/vim3 [Accessed 5 Nov 2022].

[42] KHRONOS Group. *OPEN STANDARD FOR PARALLEL PROGRAMMING OF HETEROGENEOUS SYSTEMS*. Available at https://www.khronos.org/opencl/ [Accessed 9 Mar 2022].

[43] KHRONOS Group. *OpenCL Details*. Available at https://www.khronos.org/assets\/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details\-Taiwan_June-2012.pdf. [Accessed 18 May 2022].

[44] W. Kim et al. "An adaptive batch-image based driver status monitoring system on a lightweight GPU-equipped SBC". English. In: *IEEE Access* 8 (2020). Cited By :5, pp. 206074–206087.

[45] Shruti R Kulkarni, John M Alexiades, and Bipin Rajendran. "Learning and real-time classification of hand-written digits with spiking neural networks". In: *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2017, pp. 128–131.

[46] Shruti R. Kulkarni and Bipin Rajendran. "Spiking neural networks for handwritten digit recognition—Supervised learning and network optimization". In: *Neural Networks* 103 (2018), pp. 118–127. ISSN: 18792782. DOI: 10.1016/j.neunet.2018.03.019.

[47] Wilhelm Kutta. "Beitrag zur näherungsweisen integration totaler differentialgleichungen". In: *Z. Math. Phys.* 46 (1901), pp. 435–453.

[48] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. "Training Deep Spiking Neural Networks Using Backpropagation". In: *Frontiers in Neuroscience* 10 (2016). ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00508. URL: https://www.frontiersin.org/article/10.3389/fnins.2016.00508.

[49] W. K. Lee, Raphael C.W. Phan, and B. M. Goi. "Fast and Energy-Efficient Block Ciphers Implementations in ARM Processors and Mali GPU". In: *IETE Journal of Research* 0 (0 2020), pp. 1–8. ISSN: 0974780X. DOI: 10.1080/03772063.2020.1725656.

[50] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. "A 128×128 120dB 15us Latency Asynchronous Temporal Contrast Vision Sensor". In: *Solid State Circuit* 43.2 (2008), pp. 566–576.

[51] T. Lindner et al. "Face recognition system based on a single-board computer". English. In: *15th International Conference Mechatronic Systems and Materials, MSM 2020*. Cited By :2. 2020.

[52] Jesus L Lobo et al. "Spiking Neural Networks and online learning : An overview and perspectives". In: *Neural Networks* 121 (2020), pp. 88–100. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2019.09.004.

[53] Wolfgang Maass. "On the computational power of winner-take-all". In: *Neural computation* 12.11 (2000), pp. 2519–2535.

[54] Ronald J. MacGregor. *Neural and Brain Modeling*. Ed. by Richard F. Thompson. Neuroscience: A series of monographs and texts. Academic Press, 1987.

[55] Henry Markram et al. "Interneurons of the neocortical inhibitory system". In: *Nature reviews neuroscience* 5.10 (2004), pp. 793–807.

[56] Suzanne J. Matthews and Aaron St. Leger. "Energy-Efficient Analysis of Synchrophasor Data using the NVIDIA Jetson Nano". In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 2020, pp. 1–7. DOI: 10.1109/HPEC43674.2020.9286226.

[57] *MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges*. Available at http://yann.lecun.com/exdb/mnist/. [Accessed 28 Apr 2022].

[58] Jayram Moorkanikara Nageswaran et al. "Computing spike-based convolutions on GPUs". In: *Proceedings - IEEE International Symposium on Circuits and Systems* (2009), pp. 1917–1920. ISSN: 02714310.

[59] Jayram Moorkanikara Nageswaran et al. "Efficient simulation of large-scale Spiking Neural Networks using CUDA graphics processors". In: *2009 International Joint Conference on Neural Networks*. 2009, pp. 2145–2152. DOI: 10.1109/IJCNN.2009.5179043.

[60] Danish Nazir et al. "Vehicle Detection on Embedded Single Board Computers". In: *2018 7th International Conference on Computer and Communication Engineering (ICCCE)*. 2018, pp. 480–485. DOI: 10.1109/ICCCE.2018.8539298.

[61] D. Neil and S. Liu. "Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.12 (2014), pp. 2621–2628.

[62] NVIDIA. *CUDA Zone*. Available at https://developer.nvidia.com/cuda-zone [Accessed 8 May 2022].

[63] G. Orchard et al. "HFirst: A Temporal Approach to Object Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37.10 (2015), pp. 2028–2040.

[64] Sang-Soo Park, Jung-Hyun Hong, and Ki-Seok Chung. "Modified convolution neural network for highly effective parallel processing". In: *2017 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE. 2017, pp. 325–331.

[65] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035.

[66] Raspberry Pi Foundation. *Embedded and general-purpose computer systems*. [Online]. Available: https://www.futurelearn.com/courses/embedded-systems/1/todo/58820. Accessed: June 2, 2020.

[67] Daniel J Saunders, Hava T Siegelmann, Robert Kozma, et al. "STDP learning of image patches with convolutional spiking neural networks". In: *2018 international joint conference on neural networks (IJCNN)*. IEEE. 2018, pp. 1–7.

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

[69] Thomas Serre, Aude Oliva, and Tomaso Poggio. "A feedforward architecture accounts for rapid categorization". In: *Proceedings of the national academy of sciences* 104.15 (2007), pp. 6424–6429.

[70] Michael J Skocik and Lyle N Long. "On the capabilities and computational costs of neuron models". In: *IEEE Transactions on neural networks and learning systems* 25.8 (2014), pp. 1474–1483.

[71] Shiming Song et al. "A Matching Pursuit Approach for Image Classification with Spiking Neural Networks". In: *2019 IEEE Symposium Series on Computational Intelligence, SSCI 2019* (2019), pp. 2354–2359.

[72] Marcel Stimberg, Romain Brette, and Dan FM Goodman. "Brian 2, an intuitive and efficient neural simulator". In: *eLife* 8 (Aug. 2019). Ed. by Frances K Skinner, e47314. ISSN: 2050-084X. DOI: 10.7554/eLife.47314.

[73] Marcel Stimberg et al. "Equation-oriented specification of neural models for simulations". In: *Frontiers in Neuroinformatics* 8 (2014), p. 6. ISSN: 1662-5196. DOI: 10.3389/fninf.2014.00006. URL: https://www.frontiersin.org/article/10.3389/fninf.2014.00006.

[74] Suconel. *Operating Manual Digital Multimeter Unit UT70A*. Available at https://drive.google.com/file/d/1UJLt5N3PmMxvm_Oe0iw71ohC8w7MQG_n/view [Accessed 5 Nov 2022].

[75] James A. Svoboda and Richard C. Dorf. *Introduction to electric circuits*. 9th. Wiley, p. 900. ISBN: 9781118477502.

[76] Aboozar Taherkhani et al. "A review of learning in biologically plausible spiking neural networks". In: *Neural Networks* 122 (2020), pp. 253–272. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.09.036.

[77] Hoyoung Tang et al. "Spike counts based low complexity snn architecture with binary synapse". In: *IEEE Transactions on Biomedical Circuits and Systems* 13.6 (2019), pp. 1664–1677.

[78] Joan L. Tomsic. *Dictionary of Materials and Testing (2nd Edition)*. SAE International, 2000. ISBN: 978-0-7680-0531-8. URL: https://app.knovel.com/hotlink/toc/id:kpDMTE0001/dictionary-materials/dictionary-materials.

[79] Roman Trobec et al. *Introduction to parallel computing: from algorithms to programming on state-of-the-art platforms*. Springer, 2018.

[80] Valadez-Godínez, Sergio and Sossa, Humberto and Santiago-Montero, Raúl. "On the accuracy and computational cost of spiking neuron implementation". In: *Neural Networks* 122 (2020), pp. 196–217.

[81] Julien Vitay, Helge Ülo Dinkelbach, and Fred H Hamker. "ANNarchy: a code generation approach to neural simulations on parallel hardware". In: *Frontiers in neuroinformatics* 9 (2015), p. 19.

[82] Jiajun Wu et al. "Efficient design of spiking neural network with STDP learning based on fast CORDIC". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.6 (2021), pp. 2522–2534.

[83] Saehanseul Yi et al. "Real-time integrated face detection and recognition on embedded GPGPUs". In: *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. 2014, pp. 98–107. DOI: 10.1109/ESTIMedia.2014.6962350.

[84] Qiang Yu et al. "Rapid Feedforward Computation by Temporal Encoding and Learning With Spiking Neurons". In: *IEEE Transactions on Neural Networks and Learning Systems* 24.10 (2013), pp. 1539–1552. DOI: 10.1109/TNNLS.2013.2245677.

[85] Nan Zheng and Pinaki Mazumder. "Operational Principles and Learning in Spiking Neural Networks". In: *Learning in Energy-Efficient Neuromorphic Computing: Algorithm and Architecture Co-Design*. 2020, pp. 119–171. DOI: 10.1002/9781119507369.ch4.