Eindhoven University of Technology

MASTER

Gyro
an Event-Driven Digital Architecture for Spiking Deep Belief Networks

Adriaans, Guido L.A.

Award date:
2020

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems Group

Gyro: an Event-Driven Digital Architecture for Spiking Deep Belief Networks

Master's Thesis

G.L.A. Adriaans

Committee members:

Federico Corradi	-	Supervisor IMEC
Sander Stuijk	-	Supervisor TU/e
Henk Corporaal	-	TU/e
Gijs Dubbelman	-	TU/e

Eindhoven, August 2020

Abstract

This work presents Gyro, an architecture to deploy spiking Deep Belief Networks (DBNs) in digital hardware in an energy and power efficient manner. The high computation, power and energy requirements of deep neural networks are addressed by the use of a spiking neural network that is compatible with event-driven neuromorphic sensors. Both feed-forward and feed-back networks are facilitated by presenting a method to map the weight connections to on-chip memories. Furthermore, a simplified LIF neuron implementation is used to reduce the hardware resources. Gyro has a strong focus on efficiency and configurability and can deploy arbitrary sized fully-connected DBNs with a configurable trade-off between hardware resources and spike throughput. The architecture is implemented on Xilinx FPGAs. A recurrently connected network of 1274 neurons and 248.160 synapses achieves a peak throughput of 6,29 giga synaptic operations per second (GSOPS) and dissipates only 180 pJ per synaptic operation. Moreover, a feed-forward network of the same size achieves a peak throughput 40,83 GSOPS. The architecture is evaluated using three case studies: handwritten digit recognition, speech command recognition and mine detection using sonar. An accuracy of 99,3% is achieved on the MNIST dataset using a network of 2954 neurons.

Acknowledgements

Hereby I want to show my appreciation to all parties that made this project possible. Thanks to stichting IMEC Nederland for giving me the opportunity to do this project. I thank Federico Corradi for the supervision from IMEC. We complement each other due to different technical backgrounds which makes us stronger as a team. It has been a great pleasure to work with you on a daily basis. I am grateful that I got the chance to work with many experts from IMEC which has enriched me technically. I thank Sander Stuijk for the supervision from the TU/e. By guiding me at a higher level you ensured the scientific interests and the right focus. Lastly, I thank my family, Veerle and family van den Biggelaar for the continuous support and facilities.

Contents

Chapter 1

Introduction

The continuous advances in the field of machine learning and deep learning demonstrate their increasing value and meaningful contributions to our world. Both training and inference of deep neural networks require vast amounts of computing, memory, power and energy resources as a result of the growing number of neurons and interconnections. Spiking neural networks (SNNs) are a subset of the deep learning domain which mimic biological spiking behavior by encoding and processing event information using spikes. The large power and energy requirements of SNNs make their deployment on low-power mobile devices challenging or even infeasible. In contrast with the use of a remote computational server, SNNs running on edge devices offer lower latency responses and are able to operate in areas where a connection to other devices is non-existing. Therefore, numerous novel applications can be realized if SNNs run efficiently on low-power mobile devices.

Even though SNN implementations and the human brain differ fundamentally, inspiration can be taken from biology since the human brain achieve orders of magnitude better power efficiency. The human brain consists of billions of neurons that make trillions of connections. Meanwhile, the average power consumption of a human brain is as little as 20 W [1], similar to a single light bulb. While the neurons are noisy, imprecise and operate on a frequency around 40 Hz [8], this neural network is capable of solving very complex tasks with high precision.

Neuromorphic devices, such as a silicon retina [2] or a silicon cochlea [14], are inspired by human vision and hearing respectively and send out spikes that indicate events. Event-driven computation is an approach that can reduce the amount of unnecessary computation. An event-driven system processes data in a frame-free manner, which is in clear contrast to, for example, traditional computer vision where a frame is processed at every time step regardless of its content. A vast amount of computation and power is wasted on processing stationary parts of images. An SNN implicitly supports event-driven computation as it only updates neurons upon spiking activity as opposed to updating neurons at a fixed time interval. The amount of computation in an event-driven system scales with the amount of activity rather than network size, so there are no computations when there is no spiking activity. Additionally, event-driven computation enables lower latency responses. Instead of waiting for all pixels in a frame, spikes can be processed directly upon arrival and flow through the SNN.

Whilst a large number of neural network topologies exist, this work focuses on the Deep Belief Network (DBN) which is formed by stacking multiple recurrently connected Restricted Boltzmann Machines (RBMs). This topology is elaborated in detail in Chapter 2. The combination of both feed-forward and feed-back streams is an important deviation from traditional feed-forward only models and is an integral aspect of this work. The recurrent connections enable predictive error correction and multisensory integration. DBNs have proven themselves in various applications such as emotion recognition, seizure detection and sleep stage classification using electroencephalography (EEG) [18], estimating the remaining useful life [31], speech recognition [24] or handwritten character recognition [5].

DBNs can consist of many neurons with even more connections. All neurons operate concurrently which requires a massive amount of computation. Even though the implementation of a SNN may be the easiest and quickest on a general CPU architecture, the concurrent layout of an SNN does not suit the sequential nature of a CPU. As a result, CPUs are expected to be unable to meet both computation demands and energy constraints of mobile devices. Alternatively, GPUs contain a parallel computing architecture. Since they commonly offload a CPU by computing with a batch of data, they do not fit the event-driven approach nor the continuously streaming spikes in a SNN. Furthermore, the power consumption of GPUs does not meet the requirements of mobile applications.

A custom digital hardware design is able to provide a parallel computing architecture that is optimized for event-driven computation, a spiking neuron model and configurable bit widths. It inherently supports massive parallelism and can contain local memory to facilitate the storage of the weights present in a DBN. Custom hardware is application specific and optimized to run inference in an efficient manner in terms of time, power and energy. A digital, clock based system does not fully adhere to the even-driven approach as the clock continues to oscillate regardless of spiking activity. Nevertheless, by implementing the design event-driven, a drastic reduction in switching activity can be achieved which enables a lower dynamic power consumption. The design is initially developed and verified on an FPGA. For low volumes these are low-cost devices and are reconfigurable to allow updating of the DBN. As the state-of-the-art of deep learning changes quickly, the design can be adapted in limited time and the FPGA can be reconfigured without additional hardware costs. A design that has proven itself on an FPGA can optionally be implemented in an ASIC.

This work presents Gyro, an event-driven architecture to deploy spiking DBNs in digital hardware. The high computation, power and energy requirements of deep neural networks are addressed by the use of a spiking neural network that is compatible with event-driven neuromorphic sensors. Both feed-forward and feed-back connections are facilitated by presenting a method to map the weight connections to on-chip memories. Furthermore, the LIF neuron implementation is simplified to reduce the hardware resources. Gyro has a strong focus on configurability and can deploy arbitrary sized SNNs with a configurable trade-off between hardware resources and performance.

The remainder of this thesis is structured as follows. The background material and relevant literature are described in Chapters 2 and 3 respectively. Chapter 4 presents the design of Gyro, and Chapter 5 discusses the results and compares with related work. Chapter 6 contains conclusions and lastly recommendations for future work are given in Chapter 7.

Chapter 2

Background material

2.1 Spiking neural networks

An SNN is formed by interconnecting spiking neurons. In an SNN information is contained in the relative timing of spikes, i.e. temporal coding. Figure 2.1 shows a spiking neuron with n inputs and one output. The neuron integrates input spikes at the inputs into its membrane potential. When the membrane potential exceeds its threshold value it generates an output spike. The neurons are interconnected using weighted synapses. The output of the neuron is

$$y(t) = \phi\left(\sum_{i=1}^{n} x_i(t) \cdot w_i\right) \tag{2.1}$$

where x_i represents incoming spikes, and w_i the corresponding weights. The activation function ϕ ensures a non-linear function between inputs and output which enables the network to learn non-linear dependencies.

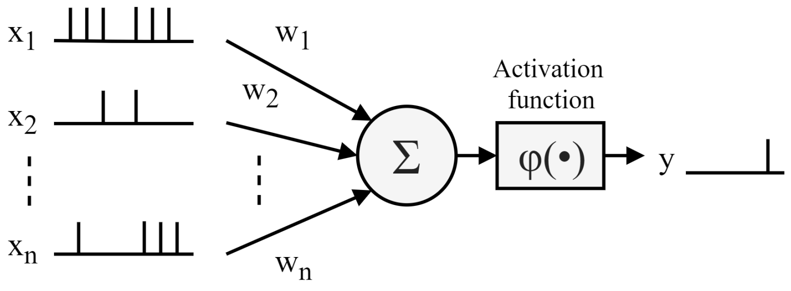

Figure 2.1: A spiking neuron with n inputs.

2.2 Leaky integrate-and-fire model

The spiking neuron model used in this work is the Leaky Integrate-and-Fire (LIF) model. This model is used due to its simplicity compared to other neuron models. The neuron is modeled using a resistor and capacitor in parallel, and the dynamic behavior of the membrane potential is described by

$$\tau_m \frac{dV(t)}{dt} = R_m I(t) - V(t) \tag{2.2}$$

where R_m the membrane resistance, C_m the membrane capacitance, $\tau_m = R_m C_m$ is the membrane time constant, V the membrane potential and I the total input current. As the model name implies, the neuron integrates the input current in its membrane potential V. Once the membrane potential exceeds its threshold potential V_{thr}, an output spike is generated and the membrane potential is reset to V_{rst} and remains V_{rst} during the refractory period t_{refrac}. The neuron model has a time-dependent memory as the membrane potential leaks away exponentially over time. An exemplary LIF membrane potential behavior is shown in Figure 2.2.

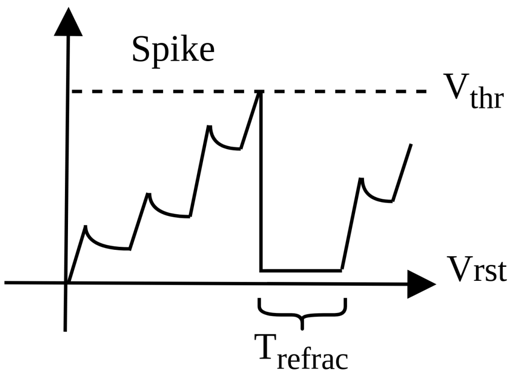

Figure 2.2: Exemplary LIF membrane potential.

2.3 Restricted Boltzmann machines

Among the many different SNN topologies that exist, this work focuses on the Restricted Boltzmann Machine (RBM). RBMs are generative stochastic artificial neural networks which is made up of two layers of neurons, a hidden and a visible layer, and is shown in Figure 2.3. The two layers are fully and symmetrically connected, but neurons are not connected within a layer.

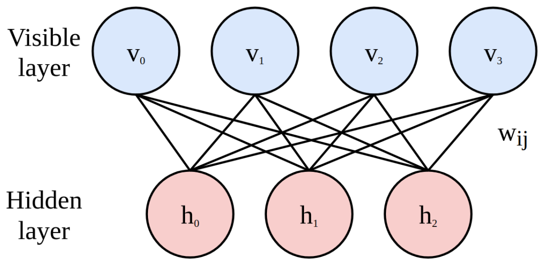

Figure 2.3: An example of a Restricted Boltzmann machine.

The energy of the joint configuration of the visible and hidden units is given by

$$E(\boldsymbol{v},\boldsymbol{h}) = -\sum_{i\in visible} a_i v_i - \sum_{j\in hidden} b_j h_j - \sum_{i,j} v_i h_j w_{ij} \tag{2.3}$$

where v_i, h_j are the binary states of visible unit i and hidden unit j, a_i, b_j are the visible and hidden biases respectively and w_{ij} are the interconnecting weights. Note that in this work the biases a_i and b_j are always zero, and therefore not shown in Figure 2.3. The encoded joint probability can be written as

$$p(\boldsymbol{v},\boldsymbol{h}) = \frac{1}{Z}\exp(-E(\boldsymbol{v},\boldsymbol{h})) = \frac{\exp(-E(\boldsymbol{v},\boldsymbol{h}))}{\sum_{v'}\sum_{h'}\exp(-E(\boldsymbol{v'},\boldsymbol{h'}))}. \tag{2.4}$$

From Equations 2.3 and 2.4 the following rules are derived for the states of the units, such that on average every network update results in a lower energy state, ultimately resulting in an equilibrium.

Given a visible input vector $\boldsymbol{v}$, the binary state h_j of hidden unit j is set to 1 with probability

$$p(h_j = 1 \mid \boldsymbol{v}) = \sigma\left(b_j + \sum_i w_{ij} v_i\right) \tag{2.5}$$

where $\sigma(x)$ is the sigmoid function. Similarly, given a hidden vector $\boldsymbol{h}$, the binary state v_i of visible unit i is set to 1 with probability

$$p(v_i = 1 \mid \boldsymbol{h}) = \sigma\left(a_i + \sum_j w_{ij} h_j\right). \tag{2.6}$$

When the network runs freely it generates samples over all possible states according to the joint probability distribution in Equation 2.4.

2.4 Deep belief networks

A DBN is a deep learning architecture consisting of recurrently connected RBMs, as shown in Figure 2.4. A DBN is created by stacking multiple RBMs onto each other, such that the hidden layer of one RBM acts as the visible layer for the next RBM. DBNs are a subset of artificial neural networks that span the state-of-the-art in the area of machine learning with applications in, for example, computer vision and speech recognition. They can extract more abstract relevant features compared to shallow neural networks. An DBN is trained using unsupervised learning to predict the activity of the visible layer from the activity of the hidden layer. The higher layers the DBN are capable of learning more abstract features of the sensory input. DBNs can be trained layer-by-layer using unlabeled data sets. This is very useful since labeling data sets is very expensive and large unlabeled data sets can be used for training.

DBNs can be used in generation mode where neurons in the top label layer are activated and cause spiking activity to propagate through the network. The resulting spiking activity in generation mode shows what the network has learned. Moreover, these feed-back connections can facilitate in applications such as error correction and sensor fusion.

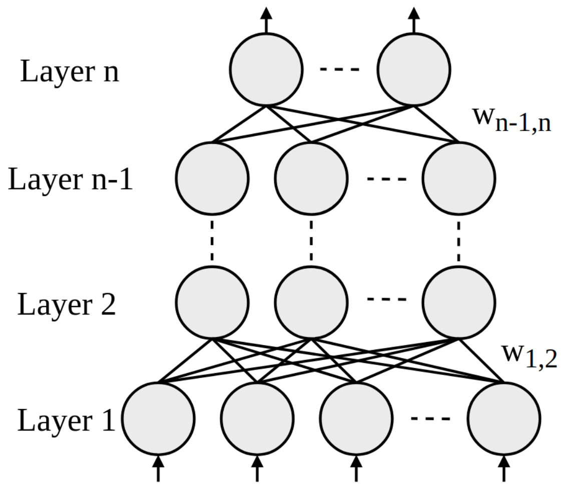

Figure 2.4: n-layered DBN where w_{ij} denote the set of weights between layers i and j.

2.5 Training

Training of an RBM is done by adapting the parameters $\boldsymbol{w}$, $\boldsymbol{a}$ and $\boldsymbol{b}$ using gradient-ascent updates. Updating of the weights is described by

$$\Delta w_{ij} = \eta(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}) \tag{2.7}$$

where η is the learning rate, $\langle . \rangle_{data}$ denotes the expectation of the input data and $\langle . \rangle_{model}$ denotes the expectation of the model distribution. The gradient ascent on the log-likelihood with respect to the weights can be approximated by a Gibbs-sampling procedure. A Gibbs sampling iteration consists of updating all hidden units in parallel using Equation 2.5 followed by updating all visible units in parallel using Equation 2.6.

Ideally the Gibbs sampling procedure is executed an infinite amount of times to ensure that the chain converges to an equilibrium. Since this is infeasible, contrastive divergence (CD) is commonly used to estimate the log-likelihood gradient. k-step contrastive divergence (CD-k) means that k iterations of Gibbs sampling are performed. It has been shown that only a few Gibbs samples are sufficient to train an RBM, usually $k = 1$ or $k = 2$. The learning rule now becomes

$$\Delta w_{ij} = \eta(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{recon}). \tag{2.8}$$

where $\langle . \rangle_{recon}$ is the approximation of $\langle v_i h_j \rangle_{model}$ using CD [10].

Furthermore, in [21] is described how the parameters of a DBN that is trained using Siegert neurons can be converted into an functionally equivalent network of spiking LIF neurons, in which event-driven real-time inference is performed. This avoids the necessity of training using spiking neurons.

Chapter 3

Related work

Numerous spiking neural network architectures have been demonstrated in neuromorphic implementations. First, there are some large-scale academic and industrial architectures. The Spiking Neural Network Architecture (SpiNNaker) [7] from the University of Manchester is a massively parallel architecture interconnecting processing nodes where each node consists of 18 ARM cores, memory and a packet router. The nodes communicate via interconnecting fabric that routes small packets efficiently. The largest SpiNNaker system comprises over a million ARM cores requiring up to 75 kW of power. TrueNorth [16] from IBM incorporates 4096 neurosynaptic cores interconnected via an intra-chip network. Each core consists of 256 programmable spiking neurons with each 256 synapses, leading to a total of one million neurons and 256 million synapses. It is an hybrid asynchronous-synchronous chip where control and communication are event-driven asynchronous, i.e. do not require a clock, and the neuron is synchronous. The neuron operates time-driven, i.e. each core updates its neurons at discrete time steps. Loihi [3] from Intel is a chip consisting of a mesh network of 128 cores that incorporate 131.072 neurons and 130 million synapses. The chip also has an on-chip learning engine to adapt weights dynamically. At each time step spikes are routed on the network and all neuron states are updated.

Since the proposed design is evaluated by means of an FPGA in Chapter 5, the related works listed in Table 3.2 are all implemented on FPGA. The Bluehive project [17] from Cambridge University implemented 256.000 Izhikevich neurons with 1.000 synapses each by interconnecting four FPGAs that each contain 64.000 neurons. Minitaur [20] and its improved version n-Minitaur [12] build upon [21] and operate event-driven. They both time-multiplex 32 physical Leaky-Integrate-and-Fire (LIF) neurons to emulate at maximum 65.536 neurons. Note that this is the only purely event-driven implementation int his selection. In [13] the Efficient Neuron Architecture (ENA) is proposed that consists of layers of neurons that communicate using packets. Using the LIF model and 32 bit precision it promises to emulate up to 3982 neurons and 400 k synapses. The authors implemented only 3 neurons though. In [28] an FPGA Design Framework (FDF) is proposed that time-multiplexes up to 200.000 neurons with one physical conductance-based neuron. Furthermore, a network is presented that consists of 1.5 M neurons and 92 G synapses on an FPGA. To reduce memory usage only the most significant bits of the exponential decay are stored and the least significant bits are stochastically generated. The same authors emulate 20 M to 2.6 B neurons in a so called Neuromorphic Cortex Simulator (NCS) [29] using only one FPGA. There is a large drop in performance, which is likely a result of using off-chip memory which increases the supported network size.

ASIC implementations support very large networks and can be extremely power and energy efficient, but are expensive to build and not easily accessible to other people. It is acknowledged that it is infeasible to outperform ASIC implementations that are made by industry and academics in terms of performance or efficiency. Therefore, this work will focus on small to medium scale event-driven spiking DBNs. The event-driven algorithm used by Minitaur [20] and n-Minitaur [12]

Table 3.1: Spiking neural network MNIST accuracies [27].

Model	**Year**	**Architecture**	**Learning method**	**Accuracy**
Neftci [19]	2014	RBM	Contrastive divergence in LIF neurons	91,9%
Neil [12]	2014	DBN	Offline learning, hardware	92,0%
Merolla [15]	2011	RBM	Offline learning, hardware	94,0%
O'Connor [21]	2013	DBN	Offline learning, conversion	94,1%
Neil [12]	2016	DBN	Offline learning, hardware	94,1%
Stromatias [26]	2015	DBN	Offline learning, conversion	94,9%
Stromatias [25]	2015	DBN	Offline learning, hardware	95,0%
Han [9]	2020	SNN	Offline learning, hardware	97,1%
Esser [6]	2015	Deep SNN	Offline learning, conversion	99,4%
Rueckauer [23]	2017	Spiking CNN	Offline learning, conversion	99,4%

is reused. As a result of reduced weight bit precision, weights are stored in on-chip memories and use of off-chip memories can be eliminated. Even though this directly limits the network size, this is very beneficial for performance, and power and energy consumption.

In Table 3.1 the MNIST accuracy results are shown for a number of spiking neural networks. All spiking RBMs and DBNs from [27] are listed, complemented with the highest accuracies of different architectures. Conversion learning method indicates that a neural network is converted to a SNN after training. The highest achieved accuracy among the RBM and DBN architectures is 95,0%, while overall the highest accuracies is 99,4% achieved by a spiking CNN and a deep SNN.

Table 3.2: FPGA-based spiking neural network implementations in chronological order.

Name	**Bluehive**	**FDF**	**n-Minitaur**	**Pani**	**NCS**	**Tsinghua**	**Gyro**
Reference	[17]	[28]	[12]	[22]	[29]	[9]	N/A
Year	2012	2014	2016	2017	2018	2020	2020
FPGA	Stratix IV	Virtex 6	Spartan 6	Virtex 6	Stratix V	Zynq 7000	Zynq Ultrascale+
Clock (MHz)	200	266	105	100	200	200	250
Neuron model	Izhikevich	Conductance	LIF	Izhikevich	LIF	LIF	LIF
Network	Unknown	Unknown	Feed-forward	Recurrent	Unknown	Feed-forward	Recurrent
Driven	Time-driven	Time-driven	Event-driven	Time-driven	Time-driven	Hybrid	Event-driven
Weight storage	Off-chip	On-chip	Off-chip	On-chip	Off-chip	Off-chip	On-chip
Weight bit-width	12 bits	12 bits	16 bits	7 bits	4 bits	16 bits	6 bits
Cores	16	23	32	8	200 k	Unknown	Same as neurons
Number of neurons	256 k	1,5 M	1794	1440	100 M	2842	2954
Time resolution	1 ms	0.32 ms	N/A	0,1 ms	1 ms	N/A	N/A
Peak throughput (GSOPS)	0,256	1200	0,0535	0,0144	20	0,67	22,85
Power dissipation (W)	Unknown	Unknown	1.5	8,5	32,4	0,477	2,586
Energy efficiency (nJ/SO)	Unknown	Unknown	28	590	1,62	712	0,109
MNIST accuracy	Unknown	Unknown	94,1%	Unknown	Unknown	97,1%	99,3%

Chapter 4

Design

This chapter presents the design of Gyro. First, the design of the neuron model is described with the event-driven network update algorithm. Thereafter, the architecture is presented in a top-down manner.

4.1 Neuron model

The LIF neuron model is described in Section 2.2. The implemented LIF neuron has instantaneous dynamics. When neuron i spikes to neuron j, the membrane potential of neuron j is increased or decreased with a step of size $W^{i,j}$. An adapted version of the event-driven network update algorithm from [20] is used as a basis for implementation. Since the layers are fully-connected, all neurons in one layer are updated simultaneously and the previous spike times are always identical for all neurons in a layer. To save storage, the algorithm is adapted by replacing the individual previous spike times by a single previous spike time that is shared over all neurons in a layer. Besides, an 'else if' statement is added which sets the membrane potential to zero if it becomes negative. The resulting optimized algorithm is shown in Algorithm 1. While a time-driven algorithm loops through all time steps, this event-driven algorithm loops through spike events which indicates that computations only happen upon spiking activity.
Algorithm 1 shows that the membrane potential of neuron i decays with

$$V_m^i = V_m^i \cdot e^{-(t-t_{prev}^i)/\tau_m} \tag{4.1}$$

which requires the computation of the exponential function. Calculating the exponential function, for example using Tailor series, is resource and time expensive. To simplify this computation, the exponential function is approximated using bit-shifts. By shifting the membrane potential by n bits to the right, its value is divided by 2^n and the least significant bits are eliminated. This is a resource efficient operation in digital hardware. Figure 4.1 shows the dataflow for one neuron derived from Algorithm 1, except for subtraction $t - t_{prev}$ which is shared over all neurons in one layer.

By moving the computation of the previous spike time outside the neuron and simplifying the computation of the exponential function, the neuron is implemented such that it processes an input spike in two clock cycles without pipelining. Figure 4.2 shows the neuron behavior using RTL simulation waveforms. The event-driven approach appears from the observation that the membrane potential, indicated as ´voltage´, remains unchanged when input spikes are absent. The refractory period causes the membrane potential to remain zero after an output spike even though new input spikes arrive. Furthermore, every now and then the membrane potential leaks away by a noticeable amount. In the period where the weight has value 1, the membrane potential gets into an equilibrium as the potential increases as much as it leaks away.

Algorithm 1 Event-driven LIF neuron update

```
Require: set of sorted spike times $S_t$
Require: set of corresponding source neurons $S_{src} \in 1..N$
Require: set of corresponding destination layers $l_{dst} \in 1..L$
  with neurons $S^l_{dst} \in 1..M$
Require: set of synaptic weights $W \in R^{N \times M}$
  $t^{1..M}_{re} \leftarrow 0$
  $t^{1..L}_{prev} \leftarrow 0$
  for t in $S_t$ do
      for s in $S_{src}$ do
          for l in $l_{dst}$ do
              $t_\delta \leftarrow t - t^l_{prev}$
              $t^l_{prev} \leftarrow t$
              for d in $S^l_{dst}$ do
                  $V^i_m \leftarrow V^i_m \cdot e^{-t_\delta/\tau_m}$
                  if $t > t^i_{re}$ then
                      $V^i_m \leftarrow V^i_m + W^{S^k_{src},i}$
                  end if
                  if $V^i_m > V_{thr}$ then
                      Spike()
                      $V^i_m \leftarrow V^i_{reset}$
                      $t^i_{re} \leftarrow t + t_{refrac}$
                  else if $V^i_m < 0$ then
                      $V^i_m \leftarrow 0$
                  end if
              end for
          end for
      end for
  end for
```

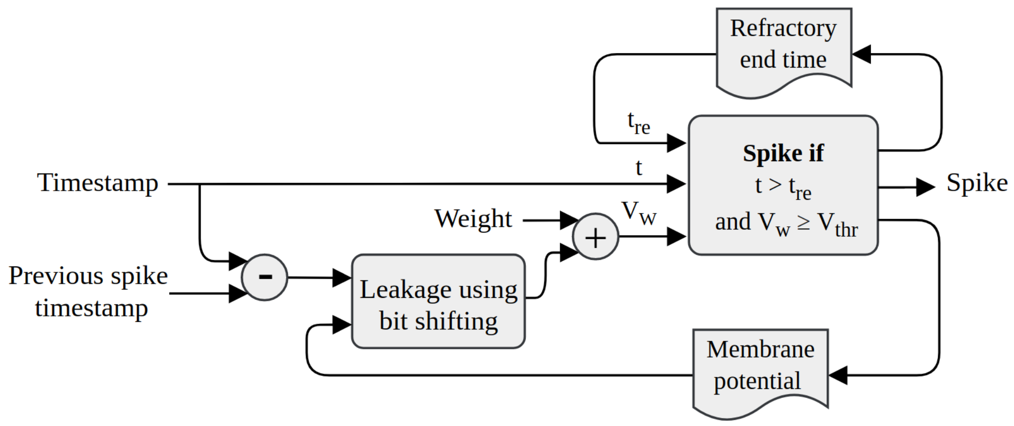

Figure 4.1: LIF neuron behavior.

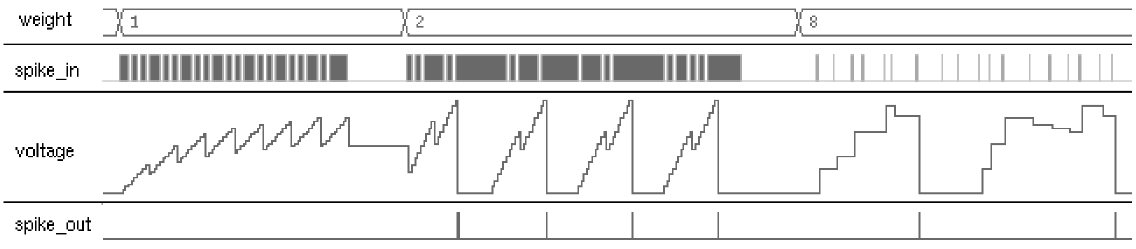

Figure 4.2: LIF neuron RTL simulation.

4.2 Architecture

The way in which connection weights are stored determines the memory bandwidth and ultimately the performance of the network. It is chosen to store all the connection weights in on-chip memory in contrast to off-chip memory for multiple reasons. First, based on the related work in Section 3 it is concluded that memory bandwidth is a major issue for the real-time implementation of DBNs. The use of multiple on-chip memories that are close to computations achieves higher memory bandwidth and lower access latency compared to off-chip weight storage. This exploits the massive parallelism of digital hardware and fits the layout of the DBN. Second, the use of on-chip memory renders a self-contained, vendor independent and external hardware independent design. This preserves the possibility to implement the design in an ASIC without significant modifications. Additionally, the lack of large off-chip storage capacity is partly compensated by the recent advances in weight bit precision reductions [11][26]. Representing weights using lower bit precision requires less storage and therefore provides the ability to implement larger neural networks with on-chip memory. In conclusion, the performance and independence advantages of on-chip memory is favoured over the large capacity of off-chip memory.

The design of Gyro is based on the layered structure of the DBN and is shown in Figure 4.3. Because the network is fully-connected, each spike is sent to all neurons of another layer and therefore all neurons in a layer are always updated simultaneously. The assumption that all layers are fully-connected has two advantages. Firstly, a fully-connected network is the worst-case scenario in terms of connection quantity, thus an connection scheme that requires less connections is inherently supported. Secondly, by updating all neurons in a layer simultaneously upon an input spike, the implementation and behaviour is simple and predictable. Apart from interfacing modules, the design consists of two main modules: the layer and the weight memory. There is a layer module for each layer in the DBN except for the very first layer as the input layer does not require computations. Between every two layers there is a weight memory module that stores the weights that interconnect the two layers. Hence, for a DBN of N layers, the design consists of $N-1$ layers and $N-1$ weight memories. The layer module consists of a spike queue that receives one or two streams of spikes. The weight controller receives spikes from the spike queue and retrieves the corresponding weights from the weight memories depending on the spike source. The neuron wrapper contains the LIF neurons along with spike buffering and arbitration. The layer module is further elaborated in Section 4.5. The timer module, described in Section 4.4, creates a notion of time which is required for the calculations in the neurons. A set of registers is used to dynamically set parameters via an AXI4-lite interface.

The main input of the network are spikes that stream into the spike queue of the first layer module, as shown in the top-left of Figure 4.3. The spikes can either be streamed in directly or they can be generated using spike generators. The output of the network are the spikes at the output layer. There are three ways to monitor the behaviour of the output layer. The literal spikes can be streamed through a buffer and the Inter-Spike Interval (ISI) and low-pass filtered membrane potential is computed for each neuron at the output layer. These modules are elaborated in Section 4.6.

In our design, each weight memory contains one or more true dual-port on-chip memories. Being true dual-port ensures that two layers can access the same weight memory simultaneously, so each layer has independent access to weights. Furthermore, although it is out of scope of this work, weights can be updated after synthesis at run time since the memories are writable. This avoids a potentially long synthesis and implementation time. The mapping of the weights to the on-chip memories, the associated difficulties and a proposed solution are elaborated in Section 4.3.

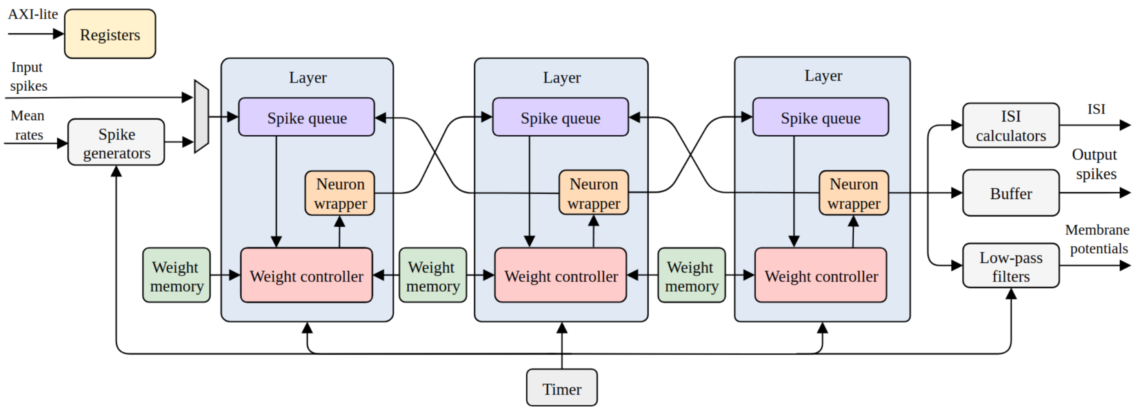

Figure 4.3: Exemplary DBN instance of four layers.

4.3 Weight memory mapping

Each of the weight memory modules stores a weight matrix that represents all the weight values of the connections between two layers. The major challenge of implementing the DBN on digital hardware is to efficiently map the connection matrices to the on-chip memories. The way the weights are distributed over the memories determines the memory bandwidth and ultimately the system throughput. Additionally, this mapping determines the implementation of the weight controller.

To visualize the impact of a weight mapping on memory bandwidth, an example weight set for a four by four network is shown in Figure 4.4. The corresponding weight matrix is shown in Figure 4.5. When this structure is also the way it is mapped to memory, one row can be read per clock cycle. When a neuron v_n spikes, each of the four neurons h_m require to be updated and the corresponding, equally colored, weights can be read in one clock cycle. The resulting throughput is four neuron updates per clock cycle. On the contrary, when a neuron h_n spikes, each of the four neurons v_m require to be updated and the corresponding weights, which contain the same shape, can be read in four consecutive clock cycles. As it effectively reads one weight per clock cycle, the resulting throughput is only one neuron update per clock cycle.

To make a more fair trade-off between forward and backward weight accesses, the weights can be mapped differently to the memory as shown in Figure 4.6. In this case, the weights corresponding to any neuron v_n or h_n are on two different lines. Thus, both a forward and backward spike require two consecutive reads and have a throughput of two neuron updates per clock cycle.

To generalize the mapping of any weight matrix between V visible neurons and H hidden neurons to a set of memories, a three dimensional x, y, z coordinate system is introduced. Variables x and y represent the memory width and depth respectively and z represents the distinctive memories. All variables x, y and z are integers. The memory width is represented as a number of weights,

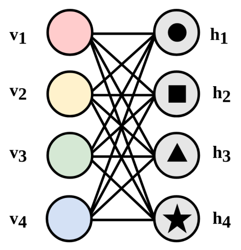

Figure 4.4: Example network of two layers.

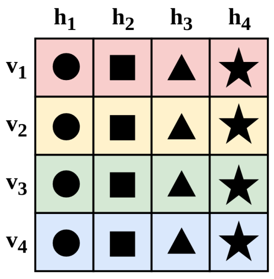

Figure 4.5: Weight mapping requiring either one or four memory accesses.

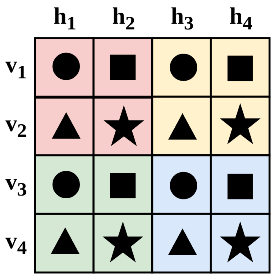

Figure 4.6: Weight mapping always requiring two memory accesses.

so the memory width in bits is obtained by multiplying x by the weight bit width. The core idea of the memory mapping is that the set of H weights that correspond to a visible neuron v_n are grouped into a cube of size x_1 by y_1 by z_1. The volume of this cube equals the amount of hidden neurons, so $x_1 \cdot y_1 \cdot z_1 = H$. This cube is repeated x_2 times in the x dimension, y_2 times in the y dimension and z_2 times in the z dimension. As there is a cube for each visible neuron, the amount of repetitions equals the amount of visible neurons, so $x_2 \cdot y_2 \cdot z_2 = V$. To update the set of hidden neurons upon a forward spike, x_1 by y_1 weights are read from z_1 memories. This is visualized in Figure 4.7 where the weights belonging to a hidden neuron are colored red. Similarly, to update the set of visible neurons upon a backward spike, x_2 by y_2 weights are read from z_2 memories. This is visualized in Figure 4.8 where the weights belonging to a visible neuron are colored green.

The maximum throughput in terms of weights per clock cycle can be derived solely from the weight mapping parameters $(x_1, y_1, z_1, x_2, y_2, z_2)$. The parameters (x_1, y_1, z_1) correspond to the weights for a forward spike, and (x_2, y_2, z_2) to the weights for a backward spike. As x_n indicates the amount of effective weights per row and z_n indicates the amount of memories accessed in parallel, $x_n \cdot z_n$ is the amount of effective weights that are read per clock cycle. The amount of effective weights per clock cycle equals the amount of neurons that are simultaneously updated, i.e. the neuron cluster size $C_1 = x_1 \cdot z_1$ and $C_2 = x_2 \cdot z_2$. Updating all neurons in a layer upon a forward and backward spike takes y_1 and y_2 clock cycles, respectively.

All variables related to the weight memory mapping are listed in Table 4.1. Moreover, the weight memory mapping objective and constraints are written in Equations 4.2a to 4.3b. Even though it is written as an optimization problem, formally solving is out of scope of this work. The objective in Equation 4.2a is to maximize performance which includes both forward and backward throughput and priority. Variable F in Equation 4.2b defines the forward throughput priority compared to backward throughput. Equations 4.2c and 4.2d ensure that the product of the parameters is sufficient to storage all weights. The width and depth of the memories are constrained in Equations 4.2e and 4.2f respectively and Equation 4.2g constrains the amount of memories.

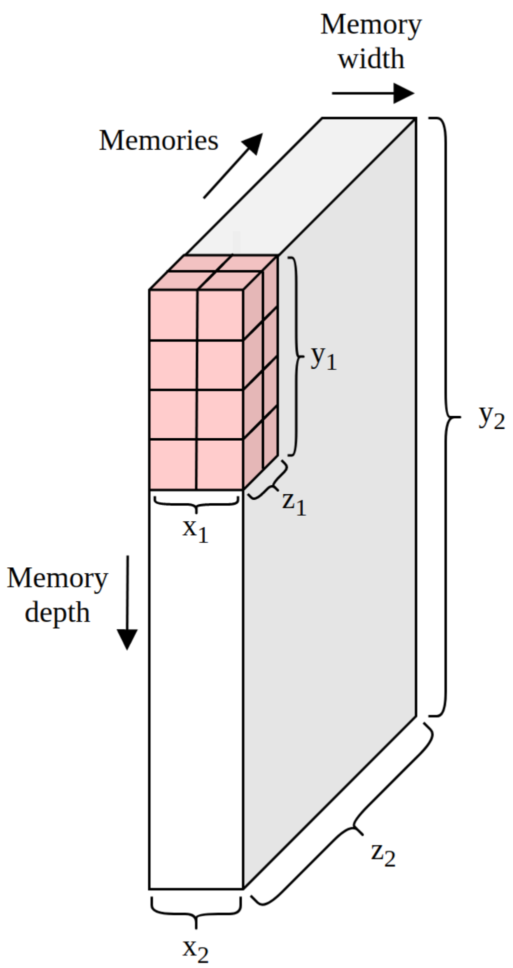

Figure 4.7: Set of $x_1 \cdot y_1 \cdot z_1$ red colored weights for a forward spike.

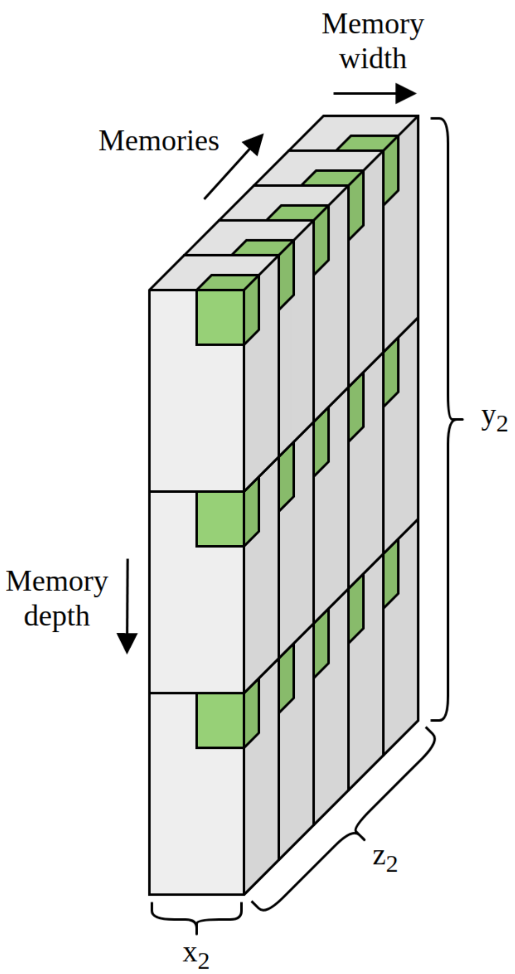

Figure 4.8: Set of $x_2 \cdot y_2 \cdot z_2$ green colored weights for a backward spike.

Variable	Type	Description
$x_1, y_1, z_1, x_2, y_2, z_2$	Integer	Memory mapping parameters
C_1	Integer	Forward neuron cluster size, i.e. $x_1 \cdot z_1$
C_2	Integer	Backward neuron cluster size, i.e. $x_2 \cdot z_2$
M_x	Integer	Maximum memory width in bits
M_y	Integer	Maximum memory depth
M_z	Integer	Maximum amount of memories
V	Integer	Number of neuron in visible layer
H	Integer	Number of neuron in hidden layer
W	Integer	Weight bit width
F	Real	Relative importance forward performance

Table 4.1: Variables related to the weight memory mapping

$$\begin{aligned}
\max_{x_1,y_1,z_1,x_2,y_2,z_2 \in \mathbb{N}} \quad & (F \cdot C_1) + (1 - F) \cdot C_2 && \text{(4.2a)}\\
\text{subject to} \quad & 0 \leq F \leq 1 && \text{(4.2b)}\\
& x_1 \cdot y_1 \cdot z_1 \geq H, && \text{(4.2c)}\\
& x_2 \cdot y_2 \cdot z_2 \geq V, && \text{(4.2d)}\\
& x_1 \cdot x_2 \leq \frac{M_x}{W}, && \text{(4.2e)}\\
& y_1 \cdot y_2 \leq M_y, && \text{(4.2f)}\\
& z_1 \cdot z_2 \leq M_z && \text{(4.2g)}
\end{aligned}$$

4.4 Timer

The neurons require a notion of time for the membrane potential leakage and the refractory period as shown in Algorithm 1. The timer module, shown in the bottom of Figure 4.3, introduces a notion of time by generating a time value and upon a time increment it generates a trigger. The timer period is configurable and depends on network requirements and the clock frequency. For example, for a resolution of 1 µs and clock period of 10 ns, the timer period is 1 µs / 10 ns = 100 clock cycles.

4.5 Layer

The layer is the computational core of the design as shown in Figure 4.3. The inner structure of the layer is shown in Figure 4.9 along with the timer and weight memories to show the external interfaces. The layer is split into three submodules: the spike queue, the weight controller and the neuron wrapper. Each of the submodules is elaborated in the following sections.

4.5.1 Spike queue

The spike queue is the purple module at the top of Figure 4.9. It receives two streams of spikes, one for forward spikes generated by the previous layer and one for backward spikes generated by the next layer. A spike is represented by the address of the neuron that generated the spike which is abbreviated to S_{src}. Each stream is separately buffered using a FIFO. The FIFO outputs are arbitrated using a round robin scheme.

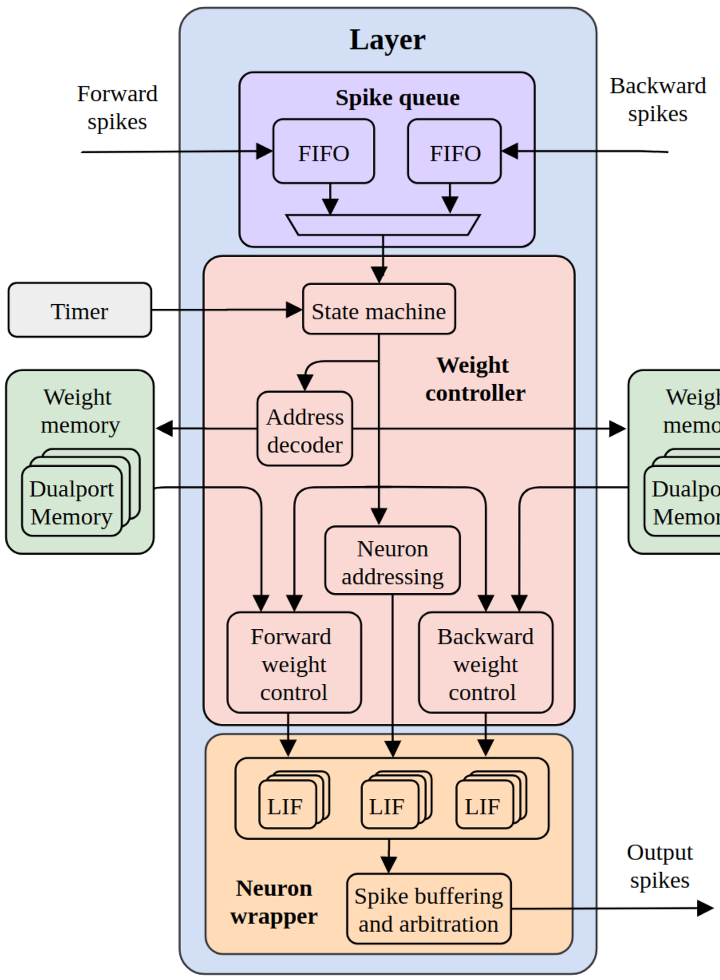

Figure 4.9: Layer diagram.

4.5.2 Weight controller

The weight controller retrieves the weights from the memories and controls the neurons. Figure 4.9 shows the weight controller in red including its submodules and external interfaces. Additionally, Figure 4.11 shows the behavior of the weight controller in more detail. The state machine, which behavior is shown in Figure 4.10, synchronizes the other modules in the weight controller. Initially, the weight controller latches the spike source address S_{src} and spike direction from the spike queue. During states decode 1 and decode 2, the address decoder module generates the memory addresses derived from S_{src} and the spike direction. During the forward or backward active states, the state machine generates indices y_f and y_b for the forward and backward weight control modules respectively to multiplex the weights that are read from the memories.

On top of the weight memory mapping Equations 4.2a to 4.2g, there are a few additional constraints in our design which are shown in Equations 4.3a to 4.3c. Firstly, parameter x_2 is fixated to 1 because this resulted in an optimal performance for all weight memory mappings in this work and this simplifies the implementation of the weight controller. This simplification is shown in the forward weight control in Figure 4.11, since it does not multiplex weights in the x dimension. Secondly, since a neuron requires two clock cycles to update its membrane potential, the weight controller must wait for a neuron to finish before it can initiate another update. This is expressed in Equations 4.3b and 4.3c, where N_{OH} represents the amount of clock cycles overhead of the weight controller. In the current implementation the overhead is two clock cycles, i.e. $N_{OH} = 2$, which allows any value for y_1 and y_2.

$$x_2 = 1, \tag{4.3a}$$
$$y_1 + N_{OH} \geq 2, \tag{4.3b}$$
$$y_2 + N_{OH} \geq 2, \tag{4.3c}$$

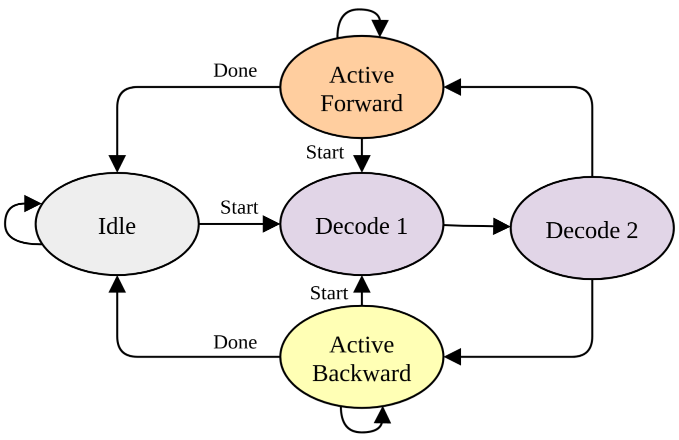

Figure 4.10: Weight controller state machine diagram.

The weights corresponding to a forward spike are all weights within a virtual cube. The surface of the cube of C_1 weights are read in parallel, so the address decoder module generates memory addresses by iterating over the rows of the concerned cube. Index y_f from the state machine counts from 0 to $y_n - 1$ with steps of size 1 which represents the index of the row within the cube. To get the memory address, this index is incremented by the row offset of the cube. Since parameter x_2 is fixed to one, dividing S_{src} by z_2 obtains the cube offset in the y dimension, and this is multiplied by cube height y_1 to get the row offset. Thus, the forward address is generated by

$$\text{Forward memory address} = y_f + \frac{S_{src} \cdot y_1}{z_2}. \tag{4.4}$$

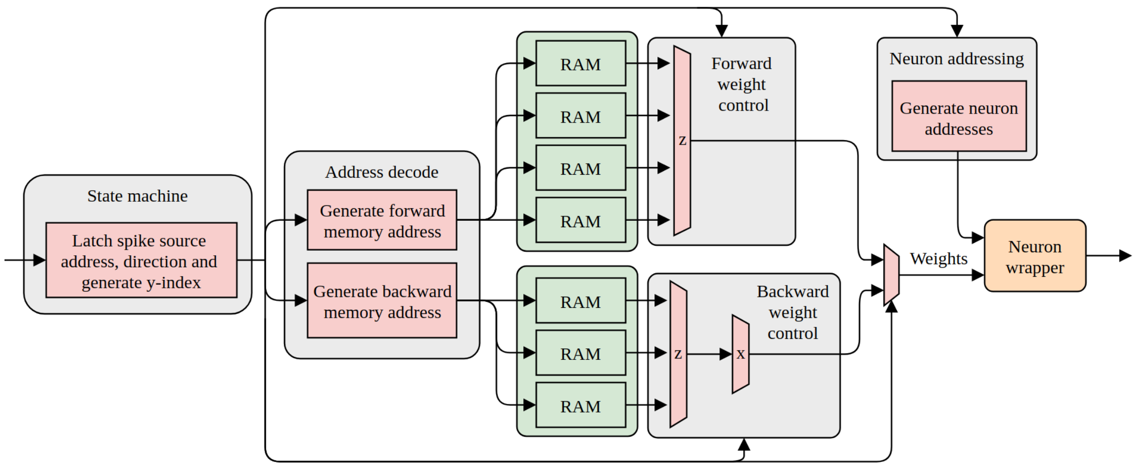
State machine
Latch spike source address, direction and generate y-index
Address decode
Generate forward memory address
Generate backward memory address
RAM
RAM
RAM
RAM
RAM
RAM
RAM
Forward weight control
z
Backward weight control
z
x
Neuron addressing
Generate neuron addresses
Weights
Neuron wrapper

Figure 4.11: Weight controller diagram.

For a backward spike, one weight is retrieved per virtual cube, and C_2 weights are read in parallel for each y_b index. The vertical position within the cube of the C_2 weights is obtained by dividing S_{src} by the surface size, i.e. C_1. The vertical cube offset is given by index y_b which counts from 0 to $(y_2 - 1) \cdot y_1$ with steps of size y_1. Thus, the backward address is generated by

$$\text{Backward memory address} = y_b + \frac{S_{src}}{C_1} \tag{4.5}$$

which concludes the calculations of the address decoder module.

In the forward and backward weight control modules receive the data from the memories and multiplex the data to filter out the relevant weights. In forward weight control module, the cube offset in z dimension is obtained by

$$z_f = S_{src} \bmod z_2. \tag{4.6}$$

The relevant weights are from the memories related to this cube, so memories $z_f \cdot z_1$ up to and including $((z_f + 1) \cdot z_1) - 1$. There is no multiplexing in the x dimension because x_2 is fixed to 1. In the backward weight control module, the offset in the z dimension within the cube is obtained by

$$z_b = \frac{S_{src}}{x_1} \bmod z_1. \tag{4.7}$$

The corresponding weights are read from memories $z_b + i \cdot z_1$ for $i = 0..z_2 - 1$. The weight offset in the x dimension is obtained by

$$x = S_{src} \bmod x_1 \tag{4.8}$$

which is used to select the valid bits from each weight vector.

The last module in the weight controller is the neuron addressing module which generates weight valid signals to control the neurons and source addresses neurons that possibly spike. The weight valid signals are generated from a counter y which goes from 0 to $\max(y_1, y_2)$ with step size 1. Clusters of C_1 or C_2 neurons are subsequently activated using the weight valid signals. The addresses of neurons in a cluster are generated to store the spike source addresses. The spike source address is generated for each neuron in the forward cluster by

$$S^i_{src} = y \cdot C_1 + i \text{ for } 0 < i < C_1 - 1. \tag{4.9}$$

Similarly, for each neuron in the backward cluster the spike source address is generated by

$$S^i_{src} = y \cdot C_2 + i \text{ for } 0 < i < C_2 - 1. \tag{4.10}$$

These addresses are used in the neuron module to store the spike source addresses if neurons spike which is shown in Figure 4.12 and described in Section 4.5.3.

4.5.3 Neuron wrapper

The neuron wrapper module in the layer consists of a set of neurons which are described in Section 4.1 and the processing of the output spikes. All neurons exist in parallel, but the output of the layer is a serial stream of spike sources addresses, identical to the spike queue input streams.

To serialize the output spikes from all neurons in the layer, a round-robin arbiter can be used to grant exclusive access to the output stream. However, this solution introduces two issues. Firstly, when N is large, the amount resources to implement the arbiter and the amount of logic levels in the arbiter increases. This leads to high resource utilization and a low maximum clock frequency. Secondly, the serial stream processes at maximum one spike per clock cycle, while the neurons can generate many spikes in a single clock cycle. This difference in throughput is inevitable for this design and the throughput of the serial stream must be at least the average amount of generated spikes. However, it can be beneficial or even necessary to buffer a burst of spikes to avoid losing them which requires spike buffering between the neurons and the serial stream. Note that this argument is not valid when a sufficiently large refractory period is used since that can guarantee that the arbiter grants a neuron access before the next spike is generated which eliminates the need of a spike buffer.

Due to the fact that a subset of C_1 or C_2 neurons is activated in parallel and therefore at maximum $\max(C_1, C_2)$ spikes are generated in parallel, only $\max(C_1, C_2)$ buffers are needed, one buffer for each neuron in a cluster. Depending on the cluster sizes C_1 and C_2, this can save a significant amount of buffers and therefore hardware resources. Figure 4.12 shows the sharing of spike buffers for three clusters of two neurons. The spike signals are fed through a logical or gate to generate the buffer write signal. As spikes are represented using their source address, the information from the neuron addressing module is used. In this example, instead of six buffers, one buffer for each neuron, only two buffers are required since at maximum two neurons spike simultaneously. Lastly, the arbiter arbitrates between the buffers to ensure fair and mutual access to the serial stream.

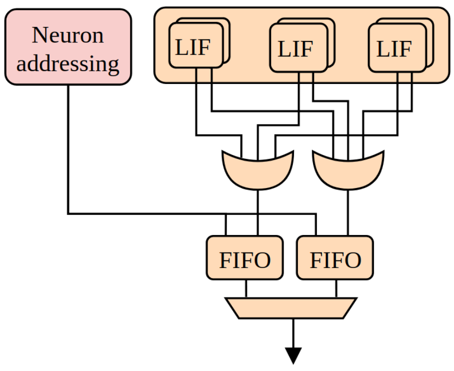

Figure 4.12: Exemplary post-neuron spike buffer sharing.

4.6 External interfaces

4.6.1 Network input

The input spikes of the first layer can either be temporally coded or mean rate (MR) coded. In case of temporal coding the spikes are, just like the streams between the layers, encoded as the spike source address. Since time is implicit in temporal coding, communication latency and jitter must be limited. Firstly, limited jitter avoids unacceptable differences between the spike arrival times and the time they are processed. If the jitter is too large, the time related calculations become erroneous. For instance, neurons might leak more than they should or it is incorrectly decided that the refractory period is expired. Secondly, in principle latency has no influence on functional correctness as it does not affect the relative spike timing. However, latency affects the feasibility of real-time requirements. When the communication latency increases, the response time of the whole system increases to a potentially unacceptable amount.

An example use case that uses temporal coding has been implemented and is shown in Figure 4.13 where a Xilinx DMA IP is used to outsource the communication to and from an external memory.

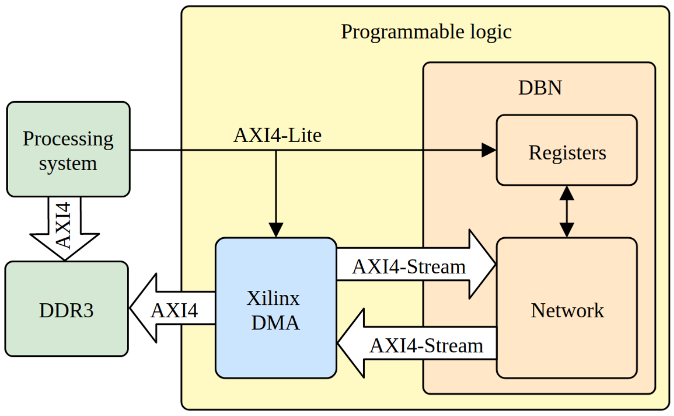

Figure 4.13: Example system that streams spikes via DMA.

Alternatively, the spike inputs can be supplied using rate coding. The advantage of this approach is that there is no dependence on communication latency or jitter. On the other hand, it requires additional logic to convert the MR for each input neuron into a sequence of spikes. To achieve this, a spike generator module is used which is shown in Figure 4.14. This module uses the MR which represents the probability that a neuron spikes and is set via registers. This stochasticity is implemented using a Linear-Feedback Shift Register (LFSR) which is triggered once every timer period. A spike is generated if the pseudorandom number is greater than the mean rate. The value of the MR is relative to its maximum value of $2^M - 1$ for M bits. An MR of zero results in zero spikes and an MR of $2^M - 1$ results in one spike every timer period. The probability that a neuron spikes upon a timer trigger for a MR of M bits is

$$P(spike) = \frac{MR}{2^M - 1}. \tag{4.11}$$

A spike generator is time-multiplexed over multiple input neurons. Each neuron cannot spike more than once per time resolution. Therefore, the amount of neurons that can share a spike generator is $N_{tm} = T_{res}/T_{clk}$. For example, a time resolution of 1 μs and clock period of 10 ns results in $10^{-6}/(10 \cdot 10^{-9}) = 100$ neurons per spike generator. Thus, an input layer of 784 neurons requires only $\lceil 784/100 \rceil = 8$ spike generators at 100 MHz.

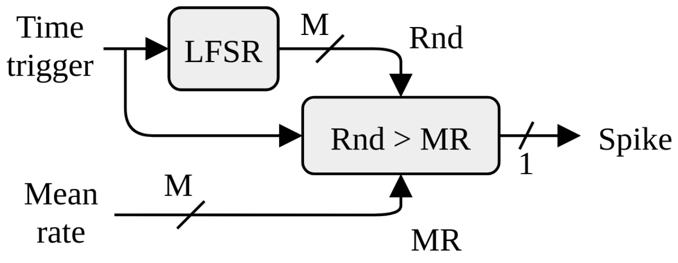

Figure 4.14: Spike generation with stochastic ISI.

4.6.2 Network output

Determining the network inference result requires monitoring of the neurons of the last layer. To increase coverage of the monitors, two different methods are applied. Firstly, for higher spike rates the Interspike Interval (ISI) is a reliable measure. The ISI is measured for each neuron at the last layer by computing the interval between the two last spikes. The resulting values are fed into registers and can be read via the AXI-lite interface. As the unit of the ISI is clock cycles, the highest supported frequency is equal to the clock frequency. The bit width of the ISI measurement determines the largest ISI and therefore the lowest possible frequency. The frequency range of the ISI measurement is

$$\frac{1}{2^{ISI\ bits} \cdot T_{clk}} \leq f_{ISI} \leq \frac{1}{T_{clk}}. \tag{4.12}$$

Secondly, the membrane potential is used to monitor the neurons when spike rates are below f_{ISI}. The membrane potentials are readily available from the neurons, but the values are low-pass filtered to obtain an averaged potential without high frequency variations. A first-order Infinite Impulse Response (IIR) filter as shown in Figure 4.15 is used since it is resource efficient. Since low frequencies are of interest, the data is sampled on the, relatively low frequency, timer trigger. To further simplify the implementation, α is chosen to be $1/32$ such that the multiplication can be implemented by bit shifts. Consequently, $1 - \alpha = 31/32$ which results in a cut-off frequency of

$$\frac{31}{32} = e^{-2\pi \frac{f_c}{f_s}} \tag{4.13}$$

which is rewritten to obtain

$$\frac{f_c}{f_s} = \frac{-\ln \frac{31}{32}}{2\pi} = 0.005 \tag{4.14}$$

where the sampling frequency f_s is derived from the timer period parameter $f_s = 1/T_{res}$.

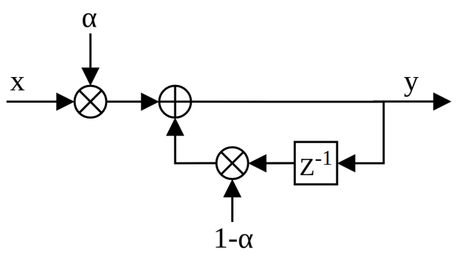

Figure 4.15: First order IIR filter.

Chapter 5

Results

The assessment of Gyro is divided into three sections. First, the characterization presents quantitative results in terms of throughput, latency, resource utilization and power dissipation. Second, three case studies evaluate the inference accuracy and above all they empirically verify the functional correctness. Thereafter, the results are discussed by comparing with related works.

5.1 Characterization

The characterization of Gyro consists of a theoretical analysis on throughput and latency and the results of an FPGA implementation of the design which reports FPGA resource utilization, power dissipation and energy efficiency. Networks are represented using format $L_1 - L_2 - .. - L_N$ format where For instance, 784-10 is a network of two layers where the first layer has 784 neurons and the second layer 10.

5.1.1 Throughput

The throughput is represented as the amount of synaptic operations per second (SOPS). One neuron that generates a spike to N other neurons causes N synaptic operations. The peak throughput depends on the weight mapping parameters and the clock frequency and assumes there is sufficient spiking activity. As described in detail in Section 4.3, the weight mapping parameters define the memory sizes, layer size and the amount of concurrency inside the layer. To distinguish the weight mapping parameters of different weight sets, notation $y^m_{1,2}$ is used where m is the weight set. Layer l uses parameters $(x_1^{l-1}, y_1^{l-1}, z_1^{l-1})$ from the previous weight memory and parameters (x_2^l, y_2^l, z_2^l) of the next weight memory.

Upon a forward spike the layer does $x_1 \cdot y_1$ synaptic operations per clock cycle. The layer requires y_1 clock cycles plus two clock cycles overhead to update all neurons in a layer. The maximum amount of forward spikes that a layer processes per time unit is

$$\frac{1}{t_{clk}(y_1 + 2)}. \tag{5.1}$$

Consequently, the peak feed-forward throughput in synaptic operations per time unit for layer l is

$$\frac{x_1^{l-1} \cdot y_1^{l-1} \cdot z_1^{l-1}}{t_{clk}(y_1^{l-1} + 2)}. \tag{5.2}$$

The peak throughput of the whole DBN is obtained by summing the peak throughput of the individual layers. The input layer is excluded since the input layer does not require computations, resulting in

$$\sum_{l=2}^{L} \frac{x_1^{l-1} \cdot y_1^{l-1} \cdot z_1^{l-1}}{t_{clk}(y_1^{l-1}+2)} \tag{5.3}$$

for a network of L layers.

The calculation of the peak feed-back throughput is similar to feed-forward apart from using parameters with suffix 2, i.e. x_2, y_2 and z_2, from the next weight memory. Upon a backward spike the layer does $x_2 \cdot y_2$ synaptic operations per clock cycle and it takes $y_2 + 2$ clock cycles to update all neurons in a layer. The peak feed-back throughput in neuron updates per time unit for layer l is

$$\frac{x_2^l \cdot y_2^l \cdot z_2^l}{t_{clk}(y_2^l+2)}. \tag{5.4}$$

For the peak throughput of the whole DBN, both the first and last layer are excluded as these layers do not receive nor compute backward spikes. The total peak feed-back throughput is

$$\sum_{l=2}^{L-1} \frac{x_2^l \cdot y_2^l \cdot z_2^l}{t_{clk}(y_2^l+2)}. \tag{5.5}$$

As shown in Equation 5.3 and 5.5, the throughput is maximized by minimizing y_1 and y_2. As a result of storing the weights in on-chip memories, it might be necessary to fill the available memories virtually completely which translates to maximizing $y_1 \cdot y_2$. However, as Equation 4.2f indicates, $y_1 \cdot y_2$ cannot exceed the amount of rows in a memory, which creates a trade-off between y_1, forward throughput, and y_2, backward throughput, and the amount of memories $y_1 \cdot y_2$. When choosing a low value for y_1, y_2 must be high to sufficiently fill the memories and vice versa.

Example

Two examples of network throughput are given: one with recurrency taken into account and one with feed-forward only. First, a four-layered recurrent network of 784-240-240-10 neurons is considered that runs on a clock frequency of 250 MHz. To have a fair trade-off between forward and backward throughput for layer l, y_1^{l-1} and y_2^l are chosen to be similar. This does not hold for $m = 1$ since weight set 1 is only accessed by layer 2 for forward spikes. Therefore, y_1^1 can be chosen low without having a backward throughput penalty. The chosen memory mapping parameters are shown in Table 5.1.

Weight memory	$\mathbf{x_1}$	$\mathbf{y_1}$	$\mathbf{z_1}$	$\mathbf{x_2}$	$\mathbf{y_2}$	$\mathbf{z_2}$
1	6	10	4	1	98	8
2	6	20	2	1	24	10
3	5	2	1	1	24	10

Table 5.1: Example weight memory mapping for a recurrent 784-240-240-10 network.

The throughput for the input layer 1 is derived from Equation 5.1 with unit spikes per second. The forward and backward throughput is computed for each layer and shown in Table 5.3. For instance, the throughput of layer 2 is computed using Equation 5.2

$$\frac{x_1^1 \cdot y_1^1 \cdot z_1^1}{t_{clk}(y_1^l+2)} = \frac{240}{4 \cdot 10^{-9}(10+2)} = 5 \cdot 10^9 = 5 \text{ GSOPS}. \tag{5.6}$$

Since there cannot be spikes towards to the input layer nor to the output layer, there is no backward throughput for layer 1 and 4. The throughput can be combined with the knowledge that forward and backward spikes are processed one after the other due to the round-robin arbiter.

Both directions update the same amount of neurons, but with a different throughput. Therefore, the combined throughput is

$$\frac{2 \cdot FW \cdot BW}{FW + BW}. \tag{5.7}$$

Next, the network throughput is presented for the same network size, 784-240-240-10 neurons, but feed-forward only. In this case y_1^m can be minimized for all weight memories without being disadvantaged on backward throughput. By using more memories, in this case 82 in total, the throughput is further increased which is shown in Table 5.3. The throughput of input layer increases to $83,3 \cdot 10^6$ spikes per second. The throughput of both layer 2 and 3 increase to 20 GSOPS and with the little increase at layer 4 it results in a total throughput of 40,83 GSOPS. In this case the two clock cycles overhead become a major factor on the throughput.

Weight memory	$\mathbf{x_1}$	$\mathbf{y_1}$	$\mathbf{z_1}$	$\mathbf{x_2}$	$\mathbf{y_2}$	$\mathbf{z_2}$
1	6	1	40	1	784	1
2	6	1	40	1	240	1
3	6	1	2	1	240	1

Table 5.2: Example weight memory mapping for a non-recurrent 784-240-240-10 network.

Network	**Direction**	**Layer 1**	**Layer 2**	**Layer 3**	**Layer 4**	**Total**
	Forward	20,8 MSPS	5	2,73	0,63	8,35
Recurrent	Backward	N/A	2,31	2,31	N/A	4,62
	Combined	20,8 MSPS	3,16	2,50	0,63	6,29
Feed-forward	Forward	83,3 MSPS	20	20	0,83	40,83

Table 5.3: Peak throughput for 784-240-240-10 networks on 250 MHz. The unit is GSOPS unless stated otherwise.

5.1.2 Latency

Latency can have various meanings in a DBN, such as the time interval between an input and output spike or the time it takes to obtain a reliable inference result. The latter cannot be analyzed without having information about the input and internal spiking activity, network size and the use case. Therefore, this section gives insight in the input spike to output spike latency for a feed-forward network.

As the design is based on the layers of the DBN, the input layer to output layer latency is obtained by summing the latency of each layer plus any overhead. The latency of a layer consists of a number of elements, which correspond to the submodules of the layer, the spike queue, weight controller and the neuron wrapper. The best-case latency of the spike queue is one clock cycle when the FIFO is empty. The worst-case latency is when the FIFO is full, which purely depends on the buffer size S_{SQ}. The weight controller has a clearly defined latency. As described in Section 5.1.1, it takes $y_1 + 2$ clock cycles to update all neurons in a layer. On top of this, a neuron update requires 2 clock cycles, so the latency of a layer is between $2+2 = 4$ and $2+(y_1+2) = y_1+4$ clock cycles. The best-case latency of the spike buffering and arbitration in the neuron wrapper is one clock cycle when all FIFOs are empty. The worst-case latency depends on the buffer size S_{NW} but also on the amount of buffers. As described in Section 4.5.3, the amount of buffers is $\max(C_1, C_2)$, resulting in a worst-case latency of $S_{NW} \cdot \max(C_1, C_2)$.

Accumulating the derived latencies, the best-case latency in clock cycles of a layer is

$$T_{layer}^{bc} = 6 \tag{5.8}$$

and worst-case latency in clock cycles of a layer is

$$T_{layer}^{wc} = S_{SQ} + y_1 + 4 + S_{NW} \cdot \max(C_1, C_2). \tag{5.9}$$

To provide a theoretical example, a network of 784-240-240-10 neurons is considered with weight memory mapping parameters as shown in Table 5.1 and spike buffer sizes $S_{SQ} = 16$ and $S_{NW} = 8$. By using Equations 5.9 and 5.8, this network has a best-case latency of $3 \cdot 6 = 18$ clock cycles and a worst-case latency in clock cycles of

$$\begin{aligned} T_{L2}^{wc} + T_{L3}^{wc} + T_{L4}^{wc} = 3S_{SQ} + y_1^2 + y_1^3 + y_1^4 + S_{NW}(\max(C_1^2, C_2^2) + \max(C_1^3, C_2^3) + \max(C_1^4, C_2^4)) \\ = 408. \end{aligned} \tag{5.10}$$

5.1.3 Implementation

The design is implemented in hardware description language VHDL and evaluated on the Xilinx FPGAs which specifications are listed in Table 5.4. As the project evolved, the implemented network size increased which led to the use of different boards. Using a bottom-up approach, each module is separately written and verified in block-level simulation. Thereafter, the modules are integrated and verified using system-level simulations. The synthesis and implementation of the FPGA using Vivado is fully automated using tcl scripting and a Makefile. The whole design and tool scripting is developed by Guido Adriaans, except for the AXI-slave module which is developed by Jan Stuyt.

Board	**FPGA**	**LUTs**	**FFs**	**DSPs**	**BRAM**	**UltraRAM**
Trenz TE0720-03-1QFA	7Z020	53.200	106.400	220	5 Mb	0 Mb
Trenz TE0820-03-4DE21FA	ZU4EV	87.840	175.680	728	4,5 Mb	13,5 Mb
Trenz TE0808-04-09EG-1EE	ZU9EG	274.080	548.160	2520	32,1 Mb	0 Mb

Table 5.4: FPGA board specifications.

The parameters shown in Table 5.5 along with the memory mapping parameters are specified using a VHDL package. In the following sections, the default values are used unless stated otherwise.

Parameter	**Data type**	**Default**
Amount of layers	Integer $\in \{1, 2, 3\}$	-
Feedback connections	Boolean	-
Input spike encoding	Mean rate or temporal	-
Enable ISI monitors and potential filters	Boolean	-
Amount of neurons at the output layer	Integer	-
Depth of spike buffers at layer input	Integer	32
Depth of spike buffers after neurons	Integer	32
Depth of spike buffer after the last layer	Integer	32
Period of the timer in clock cycles	Integer	T_{res}/T_{clk}
Bit width of input mean rate	Integer	8
Bit width of ISI output	Integer	19
Bit width of neuron address	Integer	10
Bit width of neuron membrane potential (V)	Integer	9
Bit width of the timer	Integer	24
Bit width of weights	Integer	6
Neuron membrane potential threshold	Integer	$2^V - 1$
Neuron refractory period in timer period	Integer	0

Table 5.5: Network parameters.

The weight values are specified using ASCII text files containing binary data. These files are parsed by Vivado during synthesis which initializes the BRAM memories accordingly. As it is impractical to create or maintain such a format, a Python script is created that converts a weight matrix with memory mapping parameters into a set of memory initialization files. Additionally, this script verifies whether the resulting memory sizes do not exceed the actual BRAM sizes as stated by Equations 4.2e and 4.2f.

5.1.4 Resource utilization

A main measure of the design is the amount of FPGA resources that are required to implement it. The resource utilization in terms of Lookup Tables (LUTs), Flip-Flops (FFs), BRAMs and DSPs is shown in Table 5.6 for different network sizes. The numbers only include the resources for the DBN, so external logic such as reset and AXI infrastructure logic is excluded. For the 60-60 network, which is a network of two layers that each have 60 neurons, the tool decided to implement the weight memories using FFs instead of BRAMs, resulting in an utilization of zero BRAMs. The networks of size 528-528-462-30 and 528-396-396-396-30 use 30 DSP blocks because the tool implemented the IIR filter of each neuron at the output layer using a DSP block. Furthermore, the additional cost of recurrency is small for LUTs and virtually zero for the other resources. No additional memory is required, because the same weights are used for both forward and backward spikes. The increase in LUTs is caused by the second FIFO in the spike queue, additional logic in the address decoder and the backward weight control.

Network		**FPGA**	**LUTs**	**FFs**	**BRAMs**	**DSPs**
Feed-forward	60-60	7Z020	9246	8392	0	0
Feed-forward	784-240-240-10	7Z020	35.578	33.586	62	0
Recurrent	784-240-240-10	7Z020	37.832	33.746	62	0
Feed-forward	528-528-462-30	ZU4EV	69.490	65.915	127	30
Feed-forward	528-396-396-396-30	ZU4EV	81.415	77.357	126	30
Feed-forward	784-720-720-720-10	ZU9EG	140.206	131.977	306	10

Table 5.6: FPGA resource utilization for different networks.

Figure 5.1 shows the resource utilization distribution of a network of size 784-240-240-10. The number of BRAMs correlates to the number of connections between the layers. The resources of layer 2 and 3 slightly differ while they both contain 240 neurons. It can be concluded that the number of neurons determines the number of FFs which is virtually equal. The number of LUTs slightly differs, because layer 2 multiplexes weight memory 1 which contains more BRAMs. A striking measure is the resource utilization of the registers. This large amount is caused by the mean rate coding of the inputs. One mean rate input register exists for each of the 784 input neurons, resulting in many FFs and even more LUTs to connect the AXI-lite interface to all the registers. The resource utilization of spike generators, ISI calculation and IIR filters are relatively low. This is a merit of the resource optimizations described in Section 4.6, which are time-multiplexing of spike generators, simple ISI calculation and using first order IIR filter.

Table 5.7 provides insight in the resource utilization within a layer of 240 neurons. This shows that the neurons use the majority of both LUTs and FFs, followed by the weight controller. Dividing the resources by 240 reveals that one LIF neuron uses 28 LUTs and 51 FFs, no BRAMs nor DSP blocks. The relatively large amount of FFs is caused by the storage of the refractory period end time and the membrane potential. The weight controller contains many LUTs compared to FFs, because the module mostly contains combinational logic to generate memory and neuron addresses and multiplex data which requires little memory.

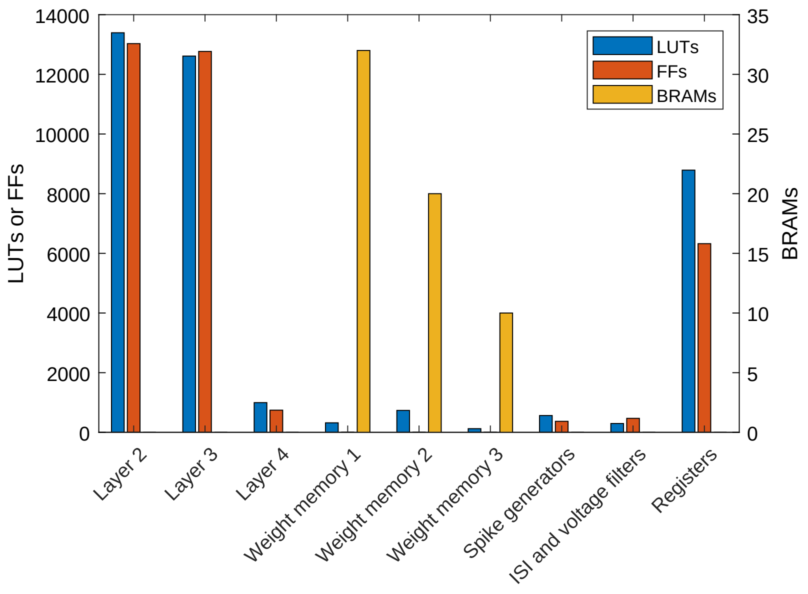

Figure 5.1: FPGA resource distribution of a recurrent DBN of size 784-240-240-10.

Module	**LUTs**	**FFs**
Spike queue	82	42
Weight controller	5721	262
LIF neurons	6730	12.240
Spike buffers and arbitration	1116	486

Table 5.7: FPGA resource distribution of a layer with 240 neurons.

Device floorplan

Figure 5.2 and 5.3 show the FPGA device floor plans for a recurrent and a non-recurrent DBN of size 784-240-240-10. Besides giving a quick overview of the resource distribution it also shows the relative placement of logic and memories. In both figures weight memory 1 is surrounded by layer 2, since this is the only layer accessing these memories. In a recurrent network, some weights are accessed by different layers. This is expressed in Figure 5.2 as weight memory 2 is placed between layer 2 and 3, while this does not apply to Figure 5.3. It is rather surprising that weight memory 3 is relatively far away from layer 4 in Figure 5.2. It is expected that the high LUT utilization causes the tool to move memories away. Lastly, it is in line with expectation that the registers are close to the PS, since the PS is the external interface of the registers.

5.1.5 Power

For a measure of the power dissipation of the design, the estimation from Vivado is used. Due to the lack of simulated signal toggle information, the default activity settings are used. The toggle rate is 12,5% indicating a toggle every 8 clock cycles. The static probability is 50%. The distribution of the power consumption is listed for different network sizes in Table 5.8. The design is implemented on a SoC that includes an embedded PS. However, because the DBN is implemented in PL while the PS is only used to interface with the DBN, the power consumption of the PS is excluded.

The power consumption positively correlates with the network size and the amount of resources

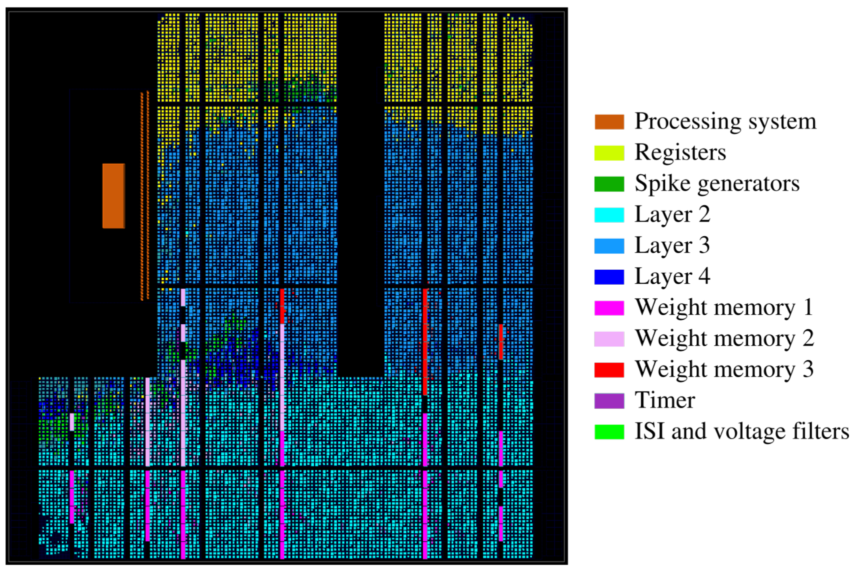

Figure 5.2: FPGA floor plan for a recurrent DBN of size 784-240-240-10.

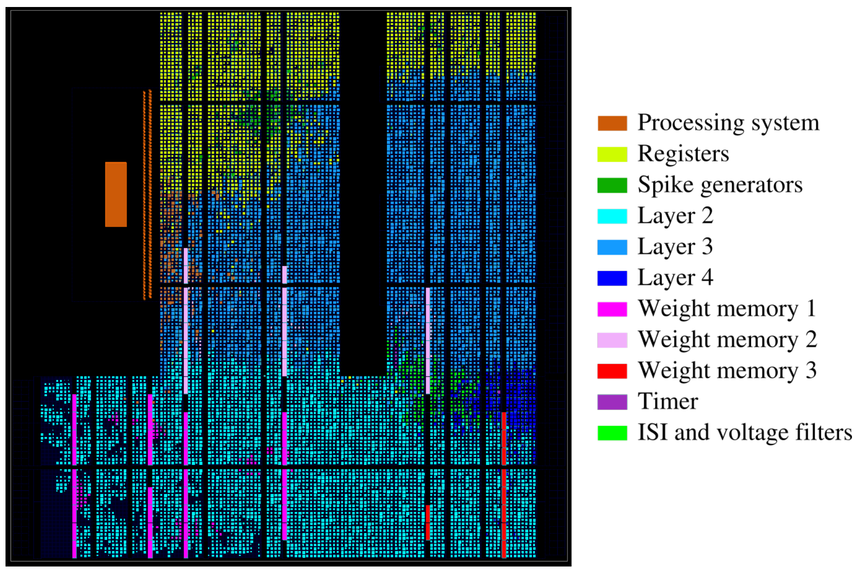

Figure 5.3: FPGA floor plan for a non-recurrent DBN of size 784-240-240-10.

used as shown in Table 5.6. Though, for the 7Z020 FPGA the static power dissipation has a significant share of the total. Therefore, the power dissipation does not scale well when implementing even smaller networks.

Networks 528-396-396-396-30 and 784-240-240-10 contain 1218 and 490 neurons respectively which is approximately a factor three differing. Nevertheless, their difference in power dissipation differs more than a factor four. This is explained by the difference in clock frequency, since a higher frequency results in more switching activity and therefore more power dissipation.

Network size	**528-396-396-396-30**	**784-240-240-10**	**60-60**
FPGA	ZU4EV	7Z020	7Z020
Clock frequency (MHz)	250	100	100
Clocks (mW)	309	62	27
Signals (mW)	701	123	27
Logic (mW)	359	66	18
BRAM (mW)	590	191	0
DSP (mW)	47	0	0
Static (mW)	479	159	143
Total (mW)	2486	601	215

Table 5.8: Estimated PL power dissipation for non-recurrent networks.

5.1.6 Energy efficiency

The absolute resource utilization, throughput or power dissipation are impractical for comparison with other implementations since they highly depend on the used network size. Therefore, the peak throughput and power dissipation are combined to create a measure of the energy efficiency in unit nanojoule per synaptic operation (nJ/SO). The energy efficiency is listed for different network sizes and clock frequencies in Table 5.9.

The results in energy efficiency show that, regardless of network size, a higher clock frequency results in more energy efficiency. The reason is provided by the results of the 784-330-330-10 network, where the clock frequency is increased by 5.5 times. Even though the throughput scales identically, the power dissipation only increases by a factor two. This shows that the dynamic power dissipation, which depends on clock frequency, is not the dominant factor in the total power dissipation in the FPGA. Therefore, it can be concluded that an FPGA is more energy efficient when it runs on high frequency and achieves the highest possibly throughput.

Network size	**528-396-396-396-30**	**784-330-330-10**		**784-240-240-10**
Clock frequency (MHz)	250	250	45	100
Peak throughput (GSOPS)	22,85	13,32	2,40	3,34
Power dissipation (mW)	2486	1890	927	601
Energy efficiency (nJ/OP)	0,109	0,142	0,254	0,180

Table 5.9: Best-case energy efficiency for different non-recurrent networks at 250 MHz.

5.2 Case studies

Three case studies are performed to evaluate the design. This empirically verifies functional correctness of the implementation and enables a comparison with related works. The case studies are handwritten digit recognition, speech command recognition and mine detection using sonar. The training is done using Python. The obtained floating-point weight values are converted to fixed-point and truncated to the desired bit with, which is 6 bits for all case studies in this work.

Training of the networks and obtaining the inference accuracies is done by Federico Corradi from IMEC. Interpreting the results and writing the text is done by Guido Adriaans.

5.2.1 MNIST

The handwritten digits dataset MNIST provides a basis for accuracy comparison with related work. The MNIST dataset contains 60.000 training images and 10.000 test images. The images are black and white 28 by 28 pixels, which causes the input layer to be $28 \cdot 28 = 784$ neurons. An exploration is done to obtain the relation between network size and classification accuracy for a weight resolution of 6 bits. The results are listed in Table 4.1, which shows a positive correlation between the network size and classification accuracy. The highest accuracy achieved is 99,3% using a 784-720-720-720-10 network. It is concluded that the relatively low bit precision of 6 bits is not a blocking factor in achieving a high accuracy. However, having sufficient number of neurons in the hidden layers is key to reduce the error rate for low weight bit precision.

Network	Neurons	Accuracy
784-120-120-10	1034	75%
784-196-196-10	1186	88%
784-280-280-10	1354	96%
784-288-288-10	1370	96%
784-330-330-10	1454	97%
784-720-720-720-10	2954	99,3%

Table 5.10: MNIST accuracy using 6 bits weight resolution.

5.2.2 Speech command recognition

The second case study uses the Google speech recognition dataset [30] which consists of 64.727 audio samples of 30 different words. In this case, the ten spoken commands are used which are yes, no, up, down, left, right, on, off, stop and go. The input of the network uses Mel-frequency cepstral coefficients (MFCC) to create 12 by 44 input features which results in an input layer of 528 neurons. The Mel scale relates the actual measured frequency to the perceived frequency. As humans are sensitive to little pitch changes on low frequencies, this scale ensures that input features match more closely to human hearing.

The inference accuracy results are shown in Table 5.11 for two networks of different sizes. This indicates that a higher accuracy is achieved by a larger second layer even though the third layer size is smaller.

Network	Neurons	Accuracy
528-528-462-10	1528	78%
528-648-360-10	1546	80%

Table 5.11: Speech command recognition accuracy for different networks.

5.2.3 Mine detection

The third case study comprises underwater mine detection using Sound Navigation and Ranging (sonar). The dataset [4] has 208 spectograms which are obtained by emitting frequency modulated sonar signals on either a metal cylinder or rocks. Each spectogram contains 60 values representing different frequency bands over a fixed period of time. Therefore, the input layer has 60 neurons, and the output layer has 2 neurons indicating either 'rock' or 'mine'.

The inference accuracy results are shown in Table 5.12 for two networks with different sizes.

For this case study the same conclusion is drawn as for the MNIST dataset, a larger number of neurons in the second and third layer results in a higher accuracy.

Network	Neurons	Accuracy
60-120-60-2	242	82%
60-240-120-2	422	90%

Table 5.12: Sonar mine detection inference accuracy for different networks.

5.3 Discussion

In this section the presented results are discussed and the key results are compared to the related works listed in Table 3.2. All related work time-multiplex (TM) neurons to reduce the neuron resource utilization and consequently support larger network sizes with fewer resources. Based on this observation, it can be concluded that TM is an effective method to increase the network size. Gyro does not TM neurons which causes the neurons to be the major contributor to LUT usage as shown in Table 5.7. When TM is implemented by storing the neuron states in memories, it will reduce LUT and FF usage at the cost of memory utilization. As synaptic weights are stored in on-chip memories, there may already be a shortage of memory. It is application and device dependent whether TM is recommended. TM is further elaborated in Section 7.3.

A benefit of Gyro is that it supports a range of layer and network sizes. When sufficient resources are available, many layers can be stacked on top of each other. Moreover, layers can be distributed over multiple FPGAs to further increase the network size. On top of configurable network size, the weight mapping parameters provide the ability to make a trade-off between resources and throughput for each layer. The optimal trade-off is application specific and heavily depends on the device, requirements and its context. Network scales cannot easily be compared, since Gyro refers to the architecture rather than a specific implementation. Nevertheless, the largest network implemented using Gyro consists of 2954 neurons which is rather small compared to related works.

The peak throughput of Gyro is heavily depending on network size, but also on whether recurrency is taken into account. When including recurrency, a trade-off is made between forward and backward throughput using the weight memory mapping parameters. For a network of 784-240-240-10 neurons, a peak throughput of 6,29 GSOPS is achieved which includes both forward and backward connections. When there are no backward connections, the weights can be mapped optimally for forward throughput without being penalized on backward throughput. For the same network of 784-240-240-10 neurons a peak throughput of 40,83 GSOPS is achieved.

Since the amount of spikes per second and the power and energy dissipation depend on network size, a comparison with related work is difficult. Instead, the energy efficiency with unit nanojoule per synaptic operation is used for comparison. The energy efficiency of Gyro for different networks is shown in Table 5.9 and varies from 0,109 to 0,254 nJ/OP. This is superior to all related works in Table 3.2 which range from 1,62 to 712 nJ/OP. Gyro achieves a factor 15 higher energy efficiency compared to NCS which is the most energy efficient among the related works. Though, it must be noted that Gyro uses newer FPGA technology compared to all related works.

Lastly, the MNIST dataset enables a comparison of classification accuracy. A number of accuracies of related works are listed in Table 3.1. Table 5.10 shows that Gyro achieves 99.3% on MNIST using a network of 2954 neurons, which exceeds all other spiking RBMs and DBNs. Gyro has only 0,1 percentage point less accuracy than the highest among the related works which has 99,4% accuracy.

Chapter 6

Conclusion

This work presents Gyro, an architecture to deploy spiking DBNs on digital hardware. On-chip memories are used to store the synaptic weights to ensure a high memory bandwidth as opposed to off-chip memory. A method is presented to map the synaptic weights to on-chip memories facilitating both feed-forward and feed-back networks. The weight memory mapping parameters offer a configurable trade-off between forward and backward throughput and hardware resource utilization. When more memories can be used, a higher memory bandwidth and ultimately a higher spiking throughput can be achieved. Furthermore, a hardware efficient LIF neuron model has been implemented to avoid the necessity of neuron time-multiplexing.

The architecture is implemented and evaluated on various Xilinx FPGAs and is characterized using theoretical and empirical analyses. The theoretical analysis on throughput shows that a recurrently connected network of 1274 neurons achieves a peak throughput of 6,29 GSOPS. Moreover, a feed-forward network of the same size achieves a peak throughput of 40,83 GSOPS. This achievement is a merit of using on-chip memory which exploits co-localization of computation and memory. The energy efficiency of Gyro varies for various network sizes and clock frequencies from 0,109 nJ/OP to 0,254 nJ/OP and turns out to be superior to all related works.

Three case studies are performed, handwritten digit recognition, speech command recognition and mine detection using sonar. These prove the applicability and functional correctness of the architecture. Gyro achieved a classification accuracy of 99,3% on the MNIST dataset using a network of 2954 neurons which is among the highest of the related SNN implementations. Research and development in the area of deep neural networks is nowhere close to and end. Nevertheless, Gyro continues to pave the way for the deployment of spiking DBNs on power and energy constrained systems.

Chapter 7

Future work

As a result of a tight time constraint, before and during execution of the project the list of desired features is prioritized and only a selection is implemented. The proposed architecture is a first working version that can be improved in many ways. This chapter describes a number of methods that further enhance functionality, performance, area, power and energy.

7.1 Reconfigurable weights

The weights of the network are set by initializing the BRAM contents during synthesis and their values are not adapted afterwards. However, since the weights are stored in true dual-port BRAMs which have both read and write ports, their values can be changed dynamically while the DBN is running on the FPGA. This avoids the potentially long synthesis and implementation times between updates of weight values where nothing but the memory contents change.

The FPGA implementation in this work uses BRAM memories. The Xilinx Ultrascale+ architecture contains UltraRAM memory blocks which have larger width and depth compared to BRAM, enabling storage of more weights. A key difference is that UltraRAM memory contents are initialized to zeros at power up, and therefore weight reconfiguration functionality is required to use them.

7.2 Pipelining

Currently the state machine in the weight controller finishes fetching all weights before it starts decoding the next spike source as shown in Figure 4.10. The weight controller can be pipelined to already start the address decoder for the next spike before it finishes fetching the weights. This would eliminate the two clock cycles overhead for each spike source address that is processed. The percentage improvement is relative to parameter y_1. To illustrate the impact, the first layer of the two examples given in Section 5.1.4 are considered. For the recurrent network shown in Table 5.1 the peak performance would increase by 20% to $240/(10 \cdot 4 \cdot 10^{-9}) = 6$ GSOPS. For the non-recurrent network shown in Table 5.1, the overhead elimination leads to a neuron update interval of a single clock cycle which violates Equation 4.3b. Still, with one clock cycle overhead the peak performance would increase by 50% to $240/(2 \cdot 4 \cdot 10^{-9}) = 30$ GSOPS.

7.3 Neuron time-multiplexing

The resource utilization in Table 5.7 shows that the neurons are the major contributor to LUT usage. Therefore, time-multiplexing (TM) has been implemented in an attempt to reduce the amount of physical neurons. Since memory bandwidth is commonly the bottleneck for performance, having a subset of neurons suffices to achieve maximum performance. However, the amount of LUTs increased significantly after implementing TM. It turned out that the amount of LUTs required to implement TM exceeded the decrease in LUTs that are saved in physical neurons. The vast majority of LUTs are used to multiplex the voltages and refractory end times which are stored in flip-flops. For a network of two layers that each have 500 neurons and use 9 bits to represent membrane potential and 32 bits to represent time, the TM module multiplexes 500 vectors of each 41 bits wide. Consequently, implementing this network required more than 220.000 LUTs.

A potential solution is to store this data in dedicated memories, for example BRAMs in FPGAs, to drastically reduce the amount of multiplexing logic. By using true dual-port memories the weight controller can simultaneously read and write data which is required to achieve maximum performance.

7.4 Timer overflow

All timer related signals use the bit-width of the timer, such as the refractory end time and the previous spike time. The refractory end time is stored in each neuron and is used to determine the membrane voltage leakage. Therefore, this bit-width parameter significantly affects the amount of resources. Currently, the timer is implemented such that it wraps-around when it overflows and as a result neurons behave incorrectly. There are two causes, leakage and the refractory period. As leakage depends on the difference between the current time and the previous input spike time, the resulting leakage becomes erroneous after a timer overflow. Secondly, once a neuron spikes it stores the refractory end time which is defined as the current time incremented with the refractory period. As long as the refractory end time is larger than the current time, the neuron is in its refractory period. When subsequently the timer is reset to 0, the refractory period is unintentionally extended. After some time, virtually all neurons are in the refractory period.

A possible solution is to introduce a reset signal that is triggered upon a timer overflow. This signal resets the previous spike time of each layer and the refractory end time of each neuron. Due to the loss of the refractory end times, neurons may violate the refractory period and spike too soon. If this is not acceptable, a safer approach is to set the refractory period of all neurons to a non-zero value. Similarly, the loss of previous spike times may cause neurons to decay less then they should. A safe approach to solve this is to reset the timer to a non-zero value to force decay for all neurons.

7.5 Buffer optimization

Resources can be saved when buffer sizes are chosen as small as possible as a result of analyzing the DBN performance and the input data. The maximum throughput of each layer can be derived from the weight memory mapping parameters, this derivation is described in Section 4.3. The number of spikes in the network and the variation in number of input spikes can be approximated from training results and knowledge of the spike source. When the input of the DBN uses mean input rates, they can be scaled such that the cumulative mean input rate is below the throughput of the first layer to avoid buffer overflows. Furthermore, when the variation in the number of input spikes is known, the buffers can be scaled accordingly.

In case the buffer sizes are chosen too small and an overflow occurs, the buffer contents do not change and therefore drop the newest spike. As time is implicit for each spike, the mismatch between the spike arrival time and when it is processed increases as time elapses. Consequently, the implementation should be adapted such that a buffer drops the oldest spike and stores the newest spike.

7.6 Memory addressing

The weight controller inside each layer outputs the memory addresses to fetch the weights from the memories. Currently, a single address is shared over a set of memories while the resulting data of only a subset may be used as shown in Figure 7.1. The toggling of signals in and around memories whose data is unused in that clock cycle causes unnecessary power and energy consumption. To avoid this, the weight controller should generate a separate address for each memory, as shown in Figure 7.2, such that only the addresses of memories whose data is used toggle and addresses of unused memories remain stable.

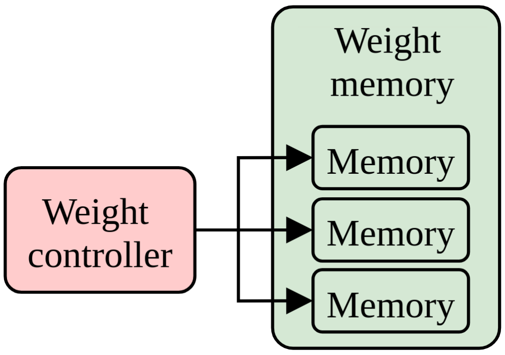

Figure 7.1: Shared memory addressing.

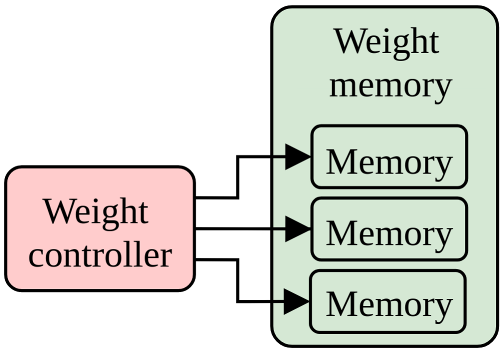

Figure 7.2: Individual memory addressing.

Bibliography

[1] D. Attwell and S. B. Laughlin. An energy budget for signaling in the grey matter of the brain. *Journal of cerebral blood flow and metabolism : official journal of the International Society of Cerebral Blood Flow and Metabolism*, 21 10:1133–45, 2001. 2

[2] C. Brandli, R. Berner, M. Yang, S. Liu, and T. Delbruck. A 240 180 130 db 3 µs latency global shutter spatiotemporal vision sensor. *IEEE Journal of Solid-State Circuits*, 49(10):2333–2341, Oct 2014. 2

[3] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, January 2018. 8

[4] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. 32

[5] M. Elleuch, N. Tagougui, and M. Kherallah. Arabic handwritten characters recognition using deep belief neural networks. In *2015 IEEE 12th International Multi-Conference on Systems, Signals Devices (SSD15)*, pages 1–5, 2015. 3

[6] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S Modha. Backpropagation for energy-efficient neuromorphic computing. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1117–1125. Curran Associates, Inc., 2015. 9

[7] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, May 2014. 8

[8] Ian Gold. Does 40-hz oscillation play a role in visual consciousness? *Consciousness and Cognition*, 8(2):186–195, 1999. 2

[9] J. Han, Z. Li, W. Zheng, and Y. Zhang. Hardware implementation of spiking neural networks on fpga. *Tsinghua Science and Technology*, 25(4):479–486, 2020. 9, 10

[10] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, July 2006. 7

[11] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *J. Mach. Learn. Res.*, 18(1):6869–6898, January 2017. 13

[12] I. Kiselev, D. Neil, and S. Liu. Event-driven deep neural network hardware system for sensor fusion. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2495–2498, May 2016. 8, 9, 10

[13] Lei Wan, Yuling Luo, Shuxiang Song, J. Harkin, and Junxiu Liu. Efficient neuron architecture for fpga-based spiking neural networks. In *2016 27th Irish Signals and Systems Conference (ISSC)*, pages 1–6, June 2016. 8

[14] S. Liu, A. van Schaik, B. A. Minch, and T. Delbruck. Asynchronous binaural spatial audition sensor with $2 \times 64 \times 4$ channel output. *IEEE Transactions on Biomedical Circuits and Systems*, 8(4):453–464, Aug 2014. 2

[15] Stephane Loiselle, Jean Rouat, Daniel Pressnitzer, and Simon Thorpe. Exploration of rank order coding with spiking neural networks for speech recognition. volume 4, pages 2076 – 2080 vol. 4, 01 2005. 9

[16] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, aug 2014. 8

[17] S. W. Moore, P. J. Fox, S. J. T. Marsh, A. T. Markettos, and A. Mujumdar. Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 133–140, April 2012. 8, 10

[18] F. Movahedi, J. L. Coyle, and E. Sejdić. Deep belief networks for electroencephalography: A review of recent contributions and future outlooks. *IEEE Journal of Biomedical and Health Informatics*, 22(3):642–652, 2018. 3

[19] Emre Neftci, Srinjoy Das, Bruno Pedroni, Kenneth Kreutz-Delgado, and Gert Cauwenberghs. Event-driven contrastive divergence for spiking neuromorphic systems. *Frontiers in Neuroscience*, 7:272, 2014. 9

[20] D. Neil and S. Liu. Minitaur, an event-driven fpga-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2621–2628, Dec 2014. 8, 11

[21] P. O'Connor, D. Neil, S. Liu, T. Delbruck, and M. Pfeiffer. Real-time classification and sensor fusion with a spiking deep belief network. *Frontiers in Neuroscience*, 7:178, 2013. 7, 8, 9

[22] Danilo Pani, Paolo Meloni, Giuseppe Tuveri, Francesca Palumbo, Paolo Massobrio, and Luigi Raffo. An fpga platform for real-time simulation of spiking neuronal networks. *Frontiers in Neuroscience*, 11:90, 2017. 10

[23] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience*, 11:682, 2017. 9

[24] T. N. Sainath, B. Kingsbury, B. Ramabhadran, P. Fousek, P. Novak, and A. Mohamed. Making deep belief networks effective for large vocabulary continuous speech recognition. In *2011 IEEE Workshop on Automatic Speech Recognition Understanding*, pages 30–35, 2011. 3

[25] Evangelos Stromatias, Dan Neil, Francesco Galluppi, Michael Pfeiffer, Shih-Chii Liu, and Steve Furber. Scalable energy-efficient, low-latency implementations of trained spiking deep belief networks on spinnaker. 07 2015. 9

[26] Evangelos Stromatias, Daniel Neil, Michael Pfeiffer, Francesco Galluppi, Steve B. Furber, and Shih-Chii Liu. Robustness of spiking deep belief networks to noise and reduced bit precision of neuro-inspired hardware platforms. *Frontiers in Neuroscience*, 9:222, 2015. 9, 13

[27] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, Mar 2019. 9

[28] R. Wang, T. J. Hamilton, J. Tapson, and A. van Schaik. An fpga design framework for large-scale spiking neural networks. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 457–460, June 2014. 8, 10

[29] Runchun M. Wang, Chetan S. Thakur, and André van Schaik. An fpga-based massively parallel neuromorphic cortex simulator. *Frontiers in Neuroscience*, 12:213, 2018. 8, 10

[30] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition, 2018. 32

[31] C. Zhang, P. Lim, A. K. Qin, and K. C. Tan. Multiobjective deep belief networks ensemble for remaining useful life estimation in prognostics. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2306–2318, 2017. 3