UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

Implémentation matérielle d'un réseau de neurones à décharges pour synchronisation rapide

Hardware implementation of a spiking neural network for fast synchronization

Mémoire de maîtrise
Specialité : génie électrique

Louis-Charles CARON

Jury: Jean ROUAT (codirecteur)
Frédéric MAILHOT (codirecteur)
Benjamin SCHRAUWEN (codirecteur)
Réjean FONTAINE (rapporteur)
Benoit GOSSELIN (évaluateur externe)

Sherbrooke (Québec) Canada Novembre 2011

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-83686-6

Our file Notre référence
ISBN: 978-0-494-83686-6

RÉSUMÉ

Pour ce mémoire, un réseau de neurones à décharges basé sur le "Oscillatory Dynamic Link Matcher" (ODLM) a été implémenté sur une matrice de portes logiques programmable ou "Field Programmable Gate Array" (FPGA). Le projet vise à identifier les caractéristiques et les choix de conception clés dans l'implémentation efficace de l'algorithme ODLM sur une plate-forme embarquée. Deux systèmes ont été développés et testés sur FPGA avec des tâches de segmentation et de comparaison d'images.

L'architecture du premier système est de type "bit-slice" et il utilise la stratégie "time-driven" où l'état du réseau est entièrement mis à jour à intervalle régulier. Le système utilise une architecture "bit-slice" avec un grand nombre d'unités de calcul. Ces unités prennent la forme de colonnes et sont connectées de façon locale à leurs deux colonnes voisines. Les poids synaptiques sont stockés sur le FPGA et le système peut réaliser n'importe quelle topologie de réseau. La fonction mathématique qui gouverne l'évolution temporelle du potentiel membranaire des neurones est approximée par une fonction définie par segments de droite. Avec cette architecture, on peut implémenter un réseau de 648 neurones sur un FPGA Xilinx XC5VSX50T cadencé à 100 MHz. Le système atteint une vitesse de calculs maximale de 6 millions de décharges par seconde. Il est en mesure de segmenter une image de 23 par 23 pixels en 2 secondes. Il peut réaliser une tâche d'appariement sur deux images préalablement segmentées de taille 90 par 30 pixels en 550 ms.

Le second système utilise le mode de fonctionnement évènementiel. Il utilise une seule unité de calcul qui traite les décharges une à une. L'unité de calcul prend la forme d'un pipeline à 5 étages qui peut traiter une synapse en 7 coups d'horloge. Ce système détermine quels neurones sont affectés par chaque décharge au moment où elles se produisent. Il calcule ensuite les poids synaptiques qui doivent être additionnés au potentiel de ces neurones. Avec cette stratégie, le système peut implémenter des réseaux de neurones ayant des topologies régulières. Également, le système utilise une table de correspondance ("look-up table") pour calculer de façon précise l'évolution temporelle du potentiel membranaire des neurones. Sur le même FPGA il est possible d'implémenter 65 536 neurones et plus de 4 millions de synapses. Ce système permet de segmenter des images de 406 par 158 pixels en 200 ms, le FPGA étant cadencé à 100 MHz.

La plupart des caractéristiques du second système sont mieux adaptées à une implémentation embarquée de l'algorithme ODLM. Les poids synaptiques dans l'ODLM sont souvent fonction d'une seule variable, ils peuvent donc facilement être calculés à la volée pour économiser de la mémoire. Le traitement évènementiel réduit considérablement la quantité de calculs à effectuer. Enfin, l'utilisation d'une fonction précise pour le calcul de l'évolution temporelle du potentiel membranaire des neurones favorise une convergence rapide du réseau.

Mots-clés : Réseau de neurones à décharges, synchronisation, système embarqué, traitement d'images

ABSTRACT

In this master thesis, we present two different hardware implementations of the Oscillatory Dynamic Link Matcher (ODLM). The ODLM is an algorithm which uses the synchronization in a network of spiking neurons to realize different signal processing tasks. The main objective of this work is to identify the key design choices leading to the efficient implementation of an embedded version of the ODLM. The resulting systems have been tested with image segmentation and image matching tasks.

The first system is bit-slice and time-driven. The state of the whole network is updated at regular time intervals. The system uses a bit-slice architecture with a large number of processing elements. Each processing element, or slice, implements one neuron of the network and takes the form of a column on the hardware. The columns are placed side by side and they are locally connected to their 2 neighbors. This local hardware connection scheme makes the system scalable, which means that columns can be easily added to increase the capacity of the system. Each column consists of a weight vector, a synapse model unit and a membrane model unit. The system can implement any network topology, making it very flexible. The function governing the time evolution of the neurons' membrane potential is approximated by a piece-wise linear function to reduce the amount of logical resources required. With this system, a fully-connected network of 648 neurons can be implemented on a Virtex-5 Xilinx XC5VSX50T FPGA clocked at 100 MHz. The system is designed to process simultaneous spikes in parallel, reaching a maximum processing speed of 6 Mspikes/s. It can segment a 23×23 pixel image in 2 seconds and match two pre-segmented 90×30 pixel images in 550 ms.

The second system is event-driven. A single processing element sequentially processes the spikes. This processing element is a 5-stage pipeline which can process an average of 1 synapse per 7 clock cycles. The synaptic weights are not stored in memory in this system, they are computed on-the-fly as spikes are processed. The topology of the network is also resolved during operation, and the system supports various regular topologies like 8-neighbor and fully-connected. The membrane potential time evolution function is computed with high precision using a look-up table. On the Virtex-5 FPGA, a network of 65 536 neurons can be implemented and a 406×158 pixel image can be segmented in 200 ms. The FPGA can be clocked at 100 MHz.

Most of the design choices made for the second system are well adapted to the hardware implementation of the ODLM. In the original ODLM, the weight values do not change over time and usually depend on a single variable. It is therefore beneficial to compute the weights on the fly rather than saving them in a huge memory bank. The event-driven approach is a very efficient strategy. It reduces the amount of computations required to run the network and the amount of data moved in and out of memory. Finally, the precise computation of the neurons' membrane potential increases the convergence speed of the network.

Keywords: Spiking neural network, synchronization, embedded system, image processing

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Project context

Artificial spiking neural networks (ASNNs) are considered more computationally powerful than previous generations of artificial neural networks [Maass, 1997]. The characteristic temporal aspect of spiking neurons takes them closer to their biological counterpart and makes them very attractive in several fields of application. Many models have been proposed to tackle different signal processing tasks. Some models are simple and only consist of a single differential equation. Others are designed to closely mimic the behavior of biological neurons and involve several equations and parameters which allow them to exhibit very complex dynamics. Furthermore, the exact topology of a network, the use of learning rules and of different encoding schemes are other factors that influence the computational capacities of an ASNN.

With the use of phase coding, for instance, one can easily solve the binding problem [Milner, 1974; Von Der Malsburg, 1981] and perform complex computations such as feature labelling. This kind of processing can be used for different applications like image processing, sound processing and clustering. In this master thesis, we focus on the efficient implementation of ASNNs which can achieve this kind of signal processing by exploiting temporal binding through spike synchronization.

1.1.1 Conventional computers and ASNNs

The conventional computer is a very popular implementation platform for signal processing systems and ASNNs. Yet computers offer limited performance when simulating large networks of spiking neurons because these networks are inherently parallel in their functioning. As a serial machine, a computer has to handle each neuron separately, repeating the same operations over the entire population. On the other hand, a parallel Single Instruction Multiple Data (SIMD) platform could efficiently process the network in batches of several neurons. Also, computers are not dedicated to simulating neural networks and therefore are bound to offer lower performance than specialized simulators: an update of the internal state of a neuron can require tens of instructions on a computer while a more

adapted system might accomplish this in a few clock cycles. Lastly, the work presented in this paper has been realized with a focus on signal processing applications, a field where an embedded solution is an undeniable asset. A computer cannot be used as the implementation platform of an embedded ASNN.

1.2 Project definition

The project aims at designing an efficient embedded ASNN used in real signal processing tasks. The Oscillatory Dynamic Link Matcher (ODLM), proposed by Pichevar et Rouat [Pichevar *et al.*, 2006], is a versatile algorithm based on an ASNN for image segmentation, image matching [Pichevar *et al.*, 2006], and monophonic sound source separation [Pichevar and Rouat, 2007]. The work presented here will be based on the ODLM algorithm. The project has two main objectives:

1. Realize an embedded implementation of the ODLM algorithm.
2. Test different designs and determine which ones lead to the most efficient implementation of the ODLM.

To fulfill these objectives, the ODLM algorithm is implemented on a hardware platform. Two different hardware spiking neural networks (HSNN) are designed. Both HSNNs feature very different architectures and demonstrate which design choices lead to a more efficient system. The performance of the design in terms of spikes processed per second is important, but both systems are also tested with real signal processing tasks: image segmentation and image matching.

1.3 Design tools and experimental setup

The work presented in this master thesis consists in the design of 2 digital hardware systems. Both systems were coded in VHDL-2008 and synthesized using Xilinx ISE Design Suite 10.1. Preliminary testing was done in simulation using ModelSim XE III 6.2g. All image processing experiments and performance measures took place on real hardware by implementing the systems on a ML506 development board featuring a Xilinx Virtex-5 XC5VSX50T FPGA. For the signal processing experiments, a personal computer, connected to the board with a serial cable, was used to pre-process and analyse the data. Figure 1.1 shows a typical experimental setup. Some experiments we also done using the Brian spiking neural networks simulator version 1.1.1 (http://www.briansimulator.org/).

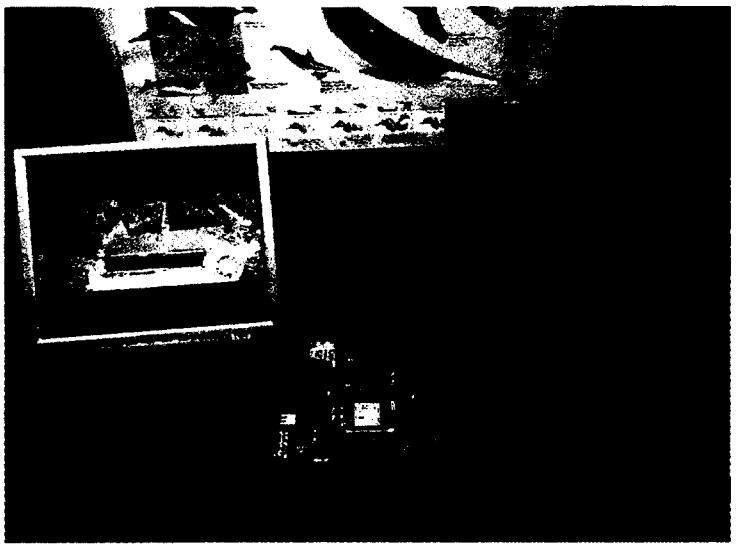

Figure 1.1 The physical setup for image processing experiments. The FPGA is connected to the computer by a serial cable. Pre-processing of the images takes place on the computer which then initializes the hardware spiking neural network on the FPGA. The system is run for a given amount of time and the final state of the network is transfered back to the computer for interpretation and display.

1.4 Outline

In the remaining of this document, state of the art design of spiking neural networks is first presented. The review covers different aspects of the hardware implementation of ASNNs like the neuron model, the communication of spikes on the hardware and the synchronization capabilities of ASNNs. Some existing implementations of hardware spiking neural networks are also described. A short description of a simplified version of the ODLM then follows. The simplified ODLM makes for easier integration on hardware and presents the overall same capabilities as the original algorithm. Chapter 4 describes the first design realized for the project, the bit-slice, time-driven implementation. Also discussed in this chapter is the impact of the approximation of the dynamical equations of the neuron model. In chapter 5, the event-driven implementation of the ODLM is described. This second system uses a specialy adapted hardware heap queue sorting algorithm which is also detailed. Finally, an evaluation and comparison of both systems is made and a short discussion concludes the document.

CHAPTER 2

LITTERATURE ON SPIKING NEURAL NETWORKS

This chapter presents a global view of the literature relating to the design of hardware spiking neural networks. Some of the existing work regarding the efficient simulation of ASNNs is first introduced. A review of some fundamental work on synchronization in neural networks follows. Then, a number of embedded HSNN are presented with a focus on systems dedicated to signal processing. Finally, an overview of some interesting sorting algorithms for the implementation of event-driven ASNNs is presented.

2.1 Efficient design of ASNNs

The design of an HSNN involves several aspects which have a huge impact on its processing speed. The next sections will give an overview the most important ones, including the neuron models, the implemation platform and the communication scheme on the hardware.

2.1.1 Neuron model

The choice of a neuron model influences the capabilities of a neural network. Some of the most popular neuron models are the Hodgkin-Huxley [Hodgkin and Huxley, 1952], the Izhikevich [Izhikevich, 2004], the Eckhorn [Eckhorn *et al.*, 1989] and the leaky integrate and fire (LIF) models.

The Hodgkin-Huxley is highly complex and is the most biologically accurate model currently used. It is one of the most influential models in computational neuroscience. It is based on a set of 4 differential equations and involves several biologically meaningful parameters which can be tuned to produce different behaviors.

The Izhikevich model comprises 2 differential and a resetting equations. It retains a certain level of complexity but lacks the biological meaningfulness found in the Hodgkin-Huxley model. By tuning 4 parameters, Izhikevich neurons can exhibit different behaviors observed in biological neurons. The 6-stage pipelined implementation of the Izhikevich model proposed by Cheung et al. [Cheung *et al.*, 2009] requires 4 hardware multipliers.

The Eckhorn model was proposed to explain the behavior of neurons found in the visual cortex of cats. The inputs of the neuron are grouped into dendrites of two types: feeding or linking. Linking inputs serve as a modulation factor for the feeding ones. The model is especially popular for image processing.

The LIF model, in its most simple form, only requires the resolution of a single differential equation which models the decaying of the membrane potential over time. Tuning of the decay time constants produce different dynamics. Upegui et al. [Upegui *et al.*, 2003] implement LIF neurons with a relative refractory period in hardware using a 2-state Moore finite state machine (FSM).

Other models exist, like the one presented by Mihalas and Niebur [Mihalas and Niebur, 2009] which is specially designed to be hardware-friendly. It can exhibit several of the behaviors produced by the Izhikevich model. The model consists of a number of state variables governed by a linear differential equation and an update rule. More state variables can be used to increase the complexity of the response of the neuron. The model is well suited for event-driven simulation. The power of the model comes from the complex update rules of the variables rather than from nonlinear differential equations.

2.1.2 Implementation platform

ASNNs can be implemented on a wide variety of platforms, including conventional computers, digital signal processors (DSPs), supercomputers, specialized neuro-computers, etc. Jahnke et al. [Jahnke *et al.*, 1997] state that among these solutions, only very expensive supercomputers can deliver sufficient computational power to simulate large neural networks. Seiffert [Seiffert, 2004] suggests using computer clusters to improve simulation performance. The SpiNNaker project [Furber *et al.*, 2006] demonstrates that arrays of hundreds of digital signal processors (DSPs) are a viable option for the implementation of ASNNs. Systems based on graphics processing units (GPUs) have also recently been proposed [Nageswaran *et al.*, 2009]. None of these platforms, though, can be considered to implement an embedded ASNN, as they are either very large in size or require too much power.

Remaining solutions are the application-specific integrated circuits (ASICs), both analog and digital, and the field-programmable gate arrays (FPGAs). All three can be designed to include an arbitrary level of parallelism, be specialized for the implementation of an ASNN and are embedded systems. While an analog ASIC would likely lead to the most compact and energy-efficient solution, it is also likely to produce a very inflexible final

product [Furber and Temple, 2007]. In contrast, digital ASICs are more power-hungry, but offer a certain level of programmability. The digital solution also has the advantage that the same hardware description language (HDL) code can be used to program both ASICs and FPGAs. A FPGA is a very flexible platform. It allows for fast prototyping and early hardware validation of the design.

2.1.3 Implementation strategy

The design of an ASNN involves several choices. An ASNN can be time- or event-driven, use a matrix of memory to store each synaptic weight, involve parallelism at different levels, etc. A time-driven implementation is simple and can be adapted to almost any situation. It is realized by updating the state of the neurons at regular time intervals, processing the spikes as they occur. In an event-driven ASNN, the equations of the model are inverted to predict the time of the next event produced by each neuron. The ASNN jumps to the time of occurrence of the next event and processes it. This technique is typically more complex but lowers the computational burden in certain contexts. Event-driven ASNNs have also been proposed both in software [Delorme and Thorpe, 2003; Watts, 1994] and hardware [Ros *et al.*, 2006; Agís *et al.*, 2007]. In an event-driven ASNN, the firing time of the neurons have to be predicting. This might require the use of approximation techniques. D'Haene et al. [D'Haene *et al.*, 2009] present a solution for the prediction of events when using a neuron model involving several different time constants.

Schaefer et al. [Schaefer *et al.*, 2002] present a series of considerations for the design of a HSNN. Using an Eckhorn neuron model and a basic network setup, they introduce several principles and optimizations that reduce the computational requirements of HSNNs. For example, they state that the use of a connection matrix is inefficient with networks of sparse connectivity. The use of connection lists with adapted size leads to huge memory savings.

When designing a digital system, one has to choose between fixed-point and floating-point representations. Roth et al. [Roth *et al.*, 1995] state that floating point precision is not required and, in fact, a fixed-point representation with variable widths as small as 8 bits is enough to simulate a simple network of Eckhorn neurons.

2.1.4 Communications scheme

A good communication scheme allows for fast and reliable events transmission. This is especially useful in a multi-chip hardware implementation, where local computations must

take into account the spikes generated by neurons processed by remote processing elements. A communication scheme is not required if a fixed local connectivity is used. In this case, direct wired connections can be used to distribute the spikes.

According to Boahen [Boahen, 2000], only two pieces of information are needed to describe an event in a neural network: the emitting neuron and the time of occurrence. Only the address, or identification number, of the pre-synaptic neuron is transmitted to post-synaptic neurons and the events are processed as soon as they are acknowledged. This communication scheme is called address-event representation AER. The technique requires a certain arbitration to ensure that 2 simultaneous or closely timed events do not collide and are propagated in the correct order.

Networks on chips (NoCs) are also being used for the management of events in HSNNs. They use similar principles to what can be found in standard computer networks. The SpiNNaker system uses a NoC to manage the communications between several DSPs [Rast *et al.*, 2008].

2.2 Synchronization in a network of oscillators

The neuron model used in the ODLM is the LIF model which approximates a relaxation oscillator. Networks of relaxation oscillators have been thoroughly studied in order to determine the factors influencing the capabilities of the individual oscillators to synchronize with one another. The internal dynamics and the coupling between the oscillators are critical elements to consider.

2.2.1 Internal dynamics of an oscillator

In his study of the cardiac pacemaker, Peskin introduces a simple model of an oscillatory LIF neuron [Peskin, 1975]. He discusses the behavior of a population of these neurons coupled through weak excitatory connections. He demonstrates that for any initial conditions, two of these neurons will eventually synchronize. Furthermore, he states that the synchronization is guaranteed as soon as the coupling between both neurons is weak and the model involves a form of dissipation. Mirollo and Strogatz [Mirollo and Strogatz, 1990] generalize Peskin's proof for all dissipation level, coupling strength and number of coupled neurons. Their proof also applies to any neuron model in which the membrane potential follows a monotonically increasing, concave down function. They stress the importance of the excitatory coupling between the neurons. Their result also shows that if the time

course of the membrane potential is linear, for instance in a non-dissipative integrate and fire neuron, the neurons do not synchronize.

Hopfield and Herz [Hopfield and Herz, 1995] use a planar network of neurons with a 4-neighbor connectivity. They study neuron models with and without leakage and with two different resetting strategies. Resetting takes place when a neuron emits a spike and can be hard or soft. The hard reset sets the neuron's potential to zero. The soft reset subtract the threshold value to the membrane potential of the neuron. These variations lead to four different neuron models which are all studied in term of long term synchronization. Hopfield and Herz conclude that local synchronization occurs in most models, but global synchronization only happens in the ones which include leakage. The soft reset leads to total synchronization whereas with the hard reset, the network does not quite reach a state of perfect synchrony.

2.2.2 Coupling between oscillators

Nischwitz and Glünder [Nischwitz and Glunder, 1995] compare the use of instantaneous excitation and delayed inhibition to produce synchrony in a network of LIF neurons. They state that instantaneous excitation quickly leads to the synchronization of neurons. Furthermore, the more in phase two neurons are, the more they are pulled together by their interactions. Delayed inhibition also induces synchrony, but the synchronizing force diminishes as the neurons are pulled closer together. Although it is a slower mechanism, delayed inhibition leads to more robust synchrony in the network.

Instead of using propagation delays, van Vreeswijk, Abbott and Ermentrout [Van Vreeswijk *et al.*, 1994] study inhibitory synapses with slow dynamics. The same effect as delayed inhibition is observed.

2.3 Embedded hardware spiking neural networks

This section gives a brief overview of embedded HSNNs. Each system was designed for specific purposes and none of them is directly usable to implement the ODLM. They all present interesting features and are a good source of inspiration for the design of the hardware ODLM.

Pearson et al. [Pearson *et al.*, 2007] explore the use of spiking neural networks for the real-time control of mobile robots. They propose an FPGA implementation of a network of augmented leaky integrate and fire neurons. Their model includes propagation delays,

refractory periods, several decay constants, and noisy synaptic integration and spike generation. They implemented a network of 1 120 neurons and 9 120 synapses on a Xilinx XC2V1000 Virtex-II FPGA. The system comprises 10 processing elements (PEs) and uses a time-driven simulation strategy. Each PE takes care of updating 112 neurons. At the end of each time step, all the PEs communicate the state of their neurons to the rest of the circuit. Each PE stores a local copy of the state of the whole network. Using this information, a PE can update the state of its own neurons for the next time step. Each PE also locally stores the parameters of its neurons. Simulation of a time step is divided into two phases, the computation of the current state of the neurons and the sharing of the spike vector for the next time step.

[Cheung *et al.*, 2009] study the high speed simulation of Izhikevich neurons on hardware. They propose an event-driven systolic architecture implemented on a Xilinx XC5VLX155T FPGA. Each processing element takes care of a number of neurons and shares their state using a ring buffer. The Izhikevich neuron model is implemented as a 6-stage pipeline. A matrix of on-chip memory stores the weights of a totally connected network. This way, any topology can be implemented by zeroing the weights between two unconnected neurons. This however affects the size of the implemented network since on-chip memory is limited and the weight matrix's size increases with the square of the number of neurons. A network of 800 Izhikevich neurons and up to 640 000 synapses is implemented and clocked at 110 MHz.

Thomas and Luk [Thomas and Luk, 2009] explore the fast simulation of small, fully connected networks of biologically plausible artificial spiking neurons on FPGA. They implemented a network of 1 024 Izhikevich neurons and 1 048 576 synapses on a Xilinx Virtex5 XC5VLX330 FPGA. The circuit provides 2.26 GFLOPS, updating the state of the whole network every 8.42 μs. To accomplish this, a synapse unit is instantiated in hardware for every implemented neuron. Each one is fed with one pre-synaptic weight of a given neuron, and the unit passes the weight through if the neuron it is associated to fires at the current time step. A 10-stage pipelined adder tree sums these weights. The result is used by the neuron model unit along with the current state of the neuron to compute its new state. Every neuron is updated in turn using a single neuron model unit. This computation scheme makes the processing speed of the circuit independent of the spiking activity of the network. Simulation time can thus be precisely predicted. However, the utilization of hardware resources is typically very low. Depending on spiking activity, a number of weights are fetched from the memory only to be zeroed inside a synapse unit.

Agis et al. [Agís *et al.*, 2007] implement an event-driven HSNN on FPGA using a pipelined searching algorithm to find the next event in an unsorted event list. The size of the event list changes according to the activity in the network and their search algorithm can handle lists of less than 2 048 events without degrading the performance of the system. With a larger list, the search takes from 10 up to 69 clock cycles, and the performance of the system goes down from 2.5 Mspikes/s to 0.478 Mspikes/s. The search algorithm also requires a considerable amount of logic and memory resources to implement large comparators and buffers for the pipeline.

2.3.1 Signal processing systems

[Schrauwen *et al.*, 2008] present a HSNN which implements a Liquid state machine (LSM) as the core of an isolated spoken digit recognition system. They evaluate different possible implementations which process the LIF neurons, synapses and arithmetic either in a serial or parallel fashion. They conclude that the designs involving a lot of parallelism in the main processing unit result in a large system and faster than real-time execution. They then introduce a very compact hardware unit which processes the synapses serially, also using serial arithmetic. This last implementation is more scalable then the other ones and still allows real-time processing of the spoken digits. One of their serial processing units fits in 4 4-input look-up tables (LUTs) of their Xilinx XC4VLX25 FPGA.

[Girau and Torres-Huitzil, 2007] design a hardware version of the LEGION network [Wang and Terman, 1997] for the segmentation of small images. They use a matrix of hardware neuron model units with hardwired synapses to their 8 nearest neighbors. They do not test the design on hardware, but they estimate that it would be possible to segment 50×50 pixel images on the largest Xilinx Virtex FPGA available at that time. According to their simulation results, the design can be clocked at 77 MHz and can segment a 16×16 pixel image in about 3 μs. However, their design is limited to networks with an 8-neighbor connectivity.

In [Vega-Pineda *et al.*, 2006], Vega-Pineda et al. describe the implementation of a hardware pulse coupled neural network for image segmentation. The neuron model they use resembles the Eckhorn model. Not much detail is given about the exact implementation, but clearly, the design topology of the network is fixed to an 8-neighbor connectivity. As this is a local connectivity scheme, no complex communication or spike propagation system must be used. They can simply deliver spikes physically connecting the hardware neurons according to the topology. The implementation on a Stratix EP1S10F484C5 FPGA is clocked at 250 MHz and processes a 30 images/s stream of 320×240 pixel images, reach-

ing a performance of 230 Mpixel/s. Just like the HSNN in [Girau and Torres-Huitzil, 2007], their design is limited to locally connected networks.

CAVIAR [Serrano-Gotarredona *et al.*, 2009] is a very complex and sophisticated neuron based object recognition and tracking hardware system. The system comprises a total of 45 000 neurons and 5 000 000 synapses. They use the AER to propagate spiking information among the different parts of the system. They also implement Spike-timing-dependent plasticity (STDP) and rate-based Hebbian learning in order to track specific objects. CAVIAR is a very exciting example of the application of artificial neural networks to signal processing. However, all the stages of the processing are fixed and cannot be easily adapted to tasks other than object recognition and tracking.

2.4 Sorting algorithms

An efficient event-driven hardware spiking neural networks needs to sort the firing time of the neurons in order to process the events in the order of their occurrence. The sorting algorithm must identify the next neuron to fire and allow the dynamic modification of the state of any neuron in the queue. In this section, an overview of existing sorting algorithms is presented.

Typically, a sorting algorithm allows for the insertion of numbers in arbitrary order and their removal from the biggest (smallest) to the smallest (biggest). In software, the design of a sorting algorithm basically consists in a compromise between the ease of insertion and that of removal of the elements. For example, one extreme solution would be the use of an unsorted list in which new elements are simply added at the end of the list. This leads to a complexity of O(1) for insertion. For the removal, the algorithm must cycle through all the elements inside the list in order to identify the biggest (smallest) element. The complexity of the removal part is thus O(N), where N is the number of elements in the list. Most sophisticated software sorting algorithms try to reduce the global complexity of the algorithm by doing part of the job at insertion and the rest at removal, for example by using a tree structure instead of a list. In hardware, the level of parallelism involved and the amount of logical resources dedicated to the sorting algorithm are additionnal considerations. A sorting algorithm must be able to discriminate several elements having the same value or priority.

The following lists and briefly describes some sorting algorithms. Some of these and other algorithms are discussed in [Thompson, 1983].

- Pipelined comparator tree [Moon *et al.*, 2000]. A comparator tree is used to identify the highest element in an unsorted array.
- Array of shift registers [Moon *et al.*, 2000]. A pipelined comparator tree is used to determine the position of new elements to insert in the array. All the elements with a lower priority are shifted down the array to make room for the new one.
- Systolic array of shift registers [Moon *et al.*, 2000]. New elements are compared with the top element and the one with a lower priority is sent down the array for comparison with the next element.
- Hybrid systolic/shift registers [Moon *et al.*, 2000]. A systolic array of shift registers arrays (a mix of the last two methods). As an element moves down the systolic array, it is compared with several elements using an array of shift-registers.
- Priority FIFOs [Moon *et al.*, 2000]. Using one FIFO per priority level, new elements are sent to the FIFO corresponding to their priority.
- Pipelined heap queue [Bhagwan and Lin, 2000; Ioannou and Katevenis, 2007]. A binary tree where each level of the tree composes one stage of a comparison pipeline. New elements are compared to the highest priority element and the one with a lower priority is sent down the tree for insertion in a lower level.
- van Emde Boas queue [van Emde Boas *et al.*, 1976; Wang and Lin, 2007]. Recursive algorithm which sorts the list of priority levels currently existing inside the queue instead of sorting the elements themselves (useful if there is a small number of possible priority levels).
- Calendar queue [Brown, 1988], lazy queue [Rönngren and Ayani, 1997], ladder queue [Tang *et al.*, 2005]. Separate the list of elements to sort in small bins and only sort the highest priority bin.

The last three items in the list, the calendar, lazy and ladder queues, are rejected because they are partial sorting algorithms. That is, they are sorting algorithms in which only a small portion of the list of elements is sorted. The sorting is done by using a full sorting algorithm such as the other ones listed above. The three stated partial sorting algorithms use different mechanisms to determine whether each data point should be sorted or not. These mechanisms perform more or less well depending on the form of the distribution of the dataset to sort. In our case, the form of the distribution is not fixed and can change drastically from a uniform distribution to a Dirac delta function. Because of this, cases might occur where the algorithms would consider that the whole dataset should be fully sorted. This means that the full sorting algorithm used in conjunction with the partial sorting one has to be designed such that it can handle the whole dataset. This renders the use of a partial sorting algorithm totally pointless.

2.4.1 Evaluation

Most sorting algorithms on hardware have a $O(1)$ complexity in time. However, none of them provides a means to easily locate a specific element already inserted in the list. In most cases, the only way to fix this would be to use a second list containing pointers to the elements in the sorting list. While this gives good result in software, it is hardly manageable in hardware because a large number of read and write operations would be required for this solution to work. For example, upon insertion of a new element in an array of shift registers, a certain number of elements will be pushed downwards, calling for an update of their pointers. This would require that the list of pointers supports an intractable number of simultaneous reads and writes.

Among the considered algorithms, the pipelined heap queue presents the best overall complexity [Thompson, 1983; Bhagwan and Lin, 2000; Ioannou and Katevenis, 2007]. However, it does not offer the possibility to easily locate an arbitrary element inside the queue. This issue will be dealt with by modifying the basic pipelined heap queue algorithm. The resulting algorithm, the structured heap queue, is described in section 5.4.3.

CHAPTER 3

THE SIMPLIFIED OSCILLATORY DYNAMIC LINK MATCHER

The HSNNs presented in this work implements a simplified version of the ODLM. Some of the features originally included in the ODLM were not implemented on the FPGA for simplicity's sake. These features include global inhibition and normalization of the spikes received by each neuron. The simplified ODLM is still able to perform interesting signal processing. This chapter describes the neuron model and the ways it can be used to accomplish image segmentation and image matching.

3.1 Neuron model

A LIF neuron model is used to approximate the behavior of relaxation oscillators. In the absence of pre-synaptic spikes, the membrane potential of an isolated neuron with initial potential zero follows

$$p_i(t) = \frac{I_0}{\tau}\left(1 - e^{-t/\tau}\right), \tag{3.1}$$

where $p_i(t)$ is the membrane potential of neuron i, and I_0 and τ are parameters. When the membrane potential of a neuron reaches a constant value $v^{threshold}$, it is reset and a spike is emitted. A neuron is reset by subtracting the value $v^{threshold}$ from its membrane potential. A neuron defined by equation 3.1 fires periodically if $v^{threshold}$ is smaller than $\frac{I_0}{\tau}$.

When the membrane potential of neuron i reaches the threshold, the neuron emits a spike. This spike will affect all post-synaptic neurons j to which the pre-synaptic neuron i is connected. The neurons j will have their membrane potential instantly increased by an amount w_{ij} according to equation 3.2, where t_s is the time of occurrence of the spike and w_{ij} is the synaptic weight between neuron i and neuron j.

$$p_j(t_s) \leftarrow p_j(t_s) + w_{ij} \tag{3.2}$$

The neuron model does not comprise any particular biologically inspired behavior such as propagation delays, refractory periods or complex synapse models. Synapses are defined by

a scalar weight computed using equation 3.3, where w_{ij} is the synaptic weight connecting neuron i to neuron j, w_{max} is the maximum value for a weight, f_i is a feature of neuron i, and α and δ are parameters which must be adapted to the processing task.

$$w_{ij} = w_{max} \times \left(1 - \frac{1}{1 + e^{-(\alpha|f_i - f_j| + \delta)}}\right) \tag{3.3}$$

3.2 Processing using the ODLM

The membrane potential of the neurons is initialized randomly and the network is run until convergence is reached. As they interact, neurons connected by strong synaptic weights will tend to synchronize, firing at the same instant. When the groups of synchronous neurons remain unchanged for a number of neuron oscillations, the network is said to have reached convergence. The final membrane potential value of the neurons can be analyzed knowing that neurons with a similar potential belong to the same group.

The topology of the network, the value of w_{max} and the features used depend on the signal processing task to accomplish. In any case, the instantaneous excitatory interactions lead to the synchronization of strongly connected neurons.

3.2.1 Image segmentation

To segment an image, each of its pixels is associated with one neuron of the network. The neurons are locally connected to their 8 neighbors. The features used to compute the synaptic weights with equation 3.3 can be various. In the present work the level of grey of the pixels is used for image segmentation experiments.

3.2.2 Image matching

For image matching, a multi-layer neural network is used, each layer representing one of the images to compare. Inside a layer, neurons are associated with each segment of the corresponding image. Synaptic connections exist between each neuron and all of the neurons in the other layers of the network. The number of connections thus increases very quickly with image size, or more precisely with the number of segments in the images to compare [Bergeron, 2008]. Again, various features can be used to characterize the segments and compute the synaptic weights. These features can involve the position of the segment in the image, its shape, its size, or any other interesting property. For the image matching

experiment presented in this work, the feature used to compute the weights is the mean grey level of the pixels in a segment [Bergeron, 2008].

3.3 Implementation of the ODLM

The neuron model only requires two arithmetic operations to be performed by the hardware: additions for the synapses and the exponential function of equation 3.1. The computation of an exponential on hardware can be relatively costly. Several implementations exist which are compact and offer good performance. However, it is also interesting to note that the implementation of an exact exponential function is not necessary for the ODLM. Indeed, for synchronization to occur in the ODLM, the charging curve of the neurons only has to be concave down and monotonically increasing. This fact is exploited in our time-driven implementation which uses a piece-wise linear approximation of the exponential function. In the event-driven version, a precise exponential function is used to evaluate the difference in processing speed.

Both image segmentation and image matching use a regular network topology. This means that the post-synaptic neurons can easily be computed on-the-fly. Nevertheless, the time-driven implementation stores a list of post-synaptic connections for each neuron in the form of a matrix of synaptic weights. The event-driven system implements on-the-fly resolution of the topology and computation of the synaptic weights.

The image matching task requires global connectivity between the neurons of the network. Both implementations will thus have to be able to handle a large number of synaptic connections.

CHAPTER 4

BIT-SLICE HARDWARE IMPLEMENTATION

4.1 Introduction

This chapter presents a bit-slice, time-driven hardware implementation of the ODLM as published in our conference paper [Caron *et al.*, 2011]. In this system, the state of the network is entirely recalculated at each time step. Spikes are processed as they are detected, with a temporal precision which depends on the size of the time steps. The system alternates between two steps: time evolution and spike propagation. The first step is the computation of the time evolution of the membrane potential of all neurons until any neuron reaches the threshold value. The computation of the potentials is then paused and a spike propagation step starts. Synaptic weights are fetched from memory and added to the membrane potential of the neurons receiving a spike. If this causes new spikes to happen, subsequent spike propagations will take place. Otherwise, computation of the time evolution is resumed. This algorithm is summurized as a finite state machine in Fig. 4.1. The most characteristic feature of this design is the full-fleged parallel processing of the neurons. In the time evolution phase, the membrane potential of every neuron in the network is updated in parallel. Conversely, in a spike propagation phase, all the spikes registered for this time step are distributed in parallel. That is, it takes one spike propagation phase to distribute any number of simultaneously occuring spikes. This means that this implementation processes spikes faster as the level of synchrony in the network increases.

On the hardware system, a membrane model unit (MMU) stores the membrane potential of a neuron and updates it during the time evolution step. The synapse model unit (SMU) adds the synaptic weights to a neuron's membrane potential during the spike propagation. A neuron model unit (NMU) is composed of a MMU and a SMU (see Fig. 4.2).

The remainder of this chapter will present the high level design of the system. Then, the global architecture of the system and the actual hardware implementation will be described. The performance of the system and results of both an image segmentation and image matching experiments will follow.

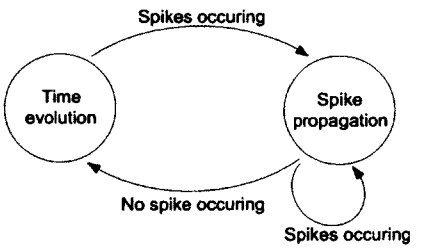

Figure 4.1 Finite state machine of the bit-slice implementation. The system alternates between two phases. During the time evolution phase, the potential of all the neurons is increased until any one of them reaches the threshold and fires. The spike propagation phase then takes place. The propagation of a spike can cause other spikes to occur, calling for subsequent spike propagations to occur before the system transitions back to the time evolution phase.

4.2 High level design

4.2.1 Levels of parallelism

A hardware simulator for a system like the ODLM can involve parallelism at different levels. Each added level of parallelism reduces the processing time but requires more hardware resources.

Neuron level

Neuron level parallelism is achieved by implementing multiple NMUs on the hardware. A fully parallel implementation uses one NMU for each neuron in the network. On the other hand, one single NMU could be used to serially process all the neurons. The neuron model in the ODLM is simple and a fairly compact hardware NMU can be designed. Neuron level parallelism becomes advantageous because the duplication of the small NMUs is not very expensive and yields an interesting gain in processing speed. Consequently, one NMU will be implemented on the FPGA for every simulated neuron. This allows full paralellism during the time evolution and spike propagation phases.

Synapse level

Inside the SMUs, synaptic weights from incoming spikes can be added in parallel by using an adder tree. Conversely, SMUs could contain one single adder and process each synapse one at a time. Considering the choice of using a large number of NMUs, the use of a small hardware SMU is preferred. A SMU will therefore consist of a single adder which will receive and add synaptic weights serially. This causes spike propagation phases to vary in duration depending on the number of synapses the neurons have.

Arithmetic

Lastly, serial arithmetic is preferred for the compact implementation it provides. The combination of serial synapse processing with serial arithmetic yields a small NMU. The parallelism lost in these stages is balanced by the large amount of NMUs working simultaneously.

4.2.2 Bit Slice Architecture

The previous design choices translate well into a bit slice architecture. Each slice, or column, contains the necessary functional blocks to instantiate a single neuron: a weight memory, a synapse model unit (SMU) and a membrane model unit (MMU). All of the slices are identical and interconnect mostly through local, neighbor to neighbor links. There is only one global signal: the spike detection signal. A high level view of the hardware system and that of a slice are shown in Fig. 4.2. Most of the interfaces between the components of a slice are 1-bit wide.

4.2.3 Hardware algorithm

During the time evolution step, the ever-growing membrane potential of every neuron is computed by the MMUs. When the potential of one or more neurons reaches the threshold, the "Spike" signal in the corresponding MMU goes high. This causes the "Spike detection" signal to also go high, as this signal is the result of an OR operation on the "Spike" signal of all the MMUs. The HSNN controller monitors the "Spike detection" signal to transistion between time evolution and spike propagation phases (see Fig. 4.1). When a spike is detected, the MMUs are disabled, stopping the time evolution phase, and a spike propagation phase starts. The spiking bit in the SMU of each column is updated to reflect its state at this time step: firing or not firing. All of the weights stored in the weight memories will then be read as the spiking bits will cycle through the whole network. The spiking bits are used to activate the SMUs when the weight read from the memory is to be added to a neuron's potential. Since each column initially stores its own spiking bit, the first weights read from memory are feedback weights. Then, each column transfers its spiking bit to the column to its right and receives the bit from the left, forming a ring buffer. Each column now holds the spiking bit of its left neighbor, and reads the corresponding synaptic weight from the weight memory. This process goes on until the spiking bits return to their initial position. During the spike propagation phase, the neurons that just fired are reset, and their "Spike" signal goes low. The propagation of their spike could however

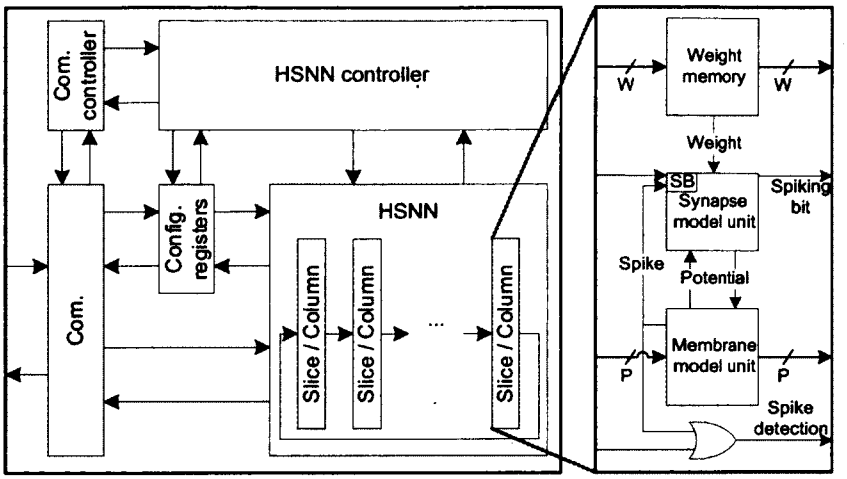

Figure 4.2 High level view of the hardware spiking neural network (HSNN) and that of a slice. **Left.** The HSNN. The "Com." and "Com. controller" take care of the communications with the host. The "Config. registers" hold the network state information and settings from the host. The "HSNN controller" is a finite state machine interacting with the HSNN. The HSNN uses a bit slice architecture, it is composed of several 1-bit wide column; **Right.** A slice. The weight memory contains all the pre-synaptic weights to the neuron instantiated by the column. The synapse model unit (SMU) adds the synaptic weights to the neurons' potential during spike propagation. The shift register noted "SB" holds the so called spiking bit. The SB shift register of every column is connected to its left and right neighbor to form a ring buffer. The membrane model unit (MMU) stores the neuron's membrane potential in a register. When the MMU is enabled, it computes the time evolution of the membrane potential. The "Spike detection" signal is the result of a OR operation on the "Spike" signal of all the columns. The buses entering and leaving the weight memory and MMUs are used for loading and unloading the potential and weight values.

cause other neurons to fire, meaning that the "Spike detection" signal would stay high and another spike propagation phase would take place. Otherwise, the system switches back to time evolution by enabling the MMUs again.

As a result of this algorithm, the weight memory of each slice must contain one weight value for every other slice in the design. If no connection exists between two given neurons, the weight has a value of zero. This is required so that the correct weight is read from memory for every spiking bit processed by a slice during spike propagation.

During operation, only two pieces of information must be transferred from one column to the others: the spiking bits and the spike detection signal. The latter is the result of an OR operation over the spiking bit of every column. It indicates that at least one neuron is firing.

4.3 Implementation

4.3.1 Weight memory

The weight memories are implemented as a whole using an array of Block RAMs (BRAMs), dedicated memory units in the FPGA. Each column of the array implements the weight memory of a slice. The organization and the data ports of the weight memories are shown in Fig. 4.3. A counter is used to generate the address for the weight memories during a spike propagation step. The weights are arranged so they fit with the cycling of the spiking bits as the memory is read.

4.3.2 Synapse model unit

Synapses are implemented as serial adders. These adders are designed to perform serial arithmetic with two different width inputs. The spiking bits circulating among the columns are held in a shift register and used to enable the adder when a weight must be added to the neuron's potential.

4.3.3 Membrane model unit

The MMU stores the membrane potential of a neuron and implements the neuron model. It updates the membrane potential according to the neuron model in the time evolution phase. The potential value is read and modified by the SMU during spike propagations. The MMU outputs the "Spike" signal, a flag indicating if the neuron is currently firing, that is, if its membrane potential is above the threshold value. The signal is used to update the "Spiking bit" in the SMU when a spike propagation phase starts and to generate the global "Spike detection" signal.

The charging curve and synchronization

The function defining the evolution of the membrane potential is called the charging curve. The exact form of the charging curve influences the neurons' capacity to synchronize. If the curve is a straight line, for example, two neurons will not tend to synchronize, even if they are strongly connected. As noted in section 2.2.1, Peskin [Peskin, 1975] and Mirollo and Strogatz [Mirollo and Strogatz, 1990] have demonstrated that the curve must be monotonically increasing and concave down for synchronization to occur in a network of oscillators. In the simplified ODLM, the charging curve takes the form of an exponential

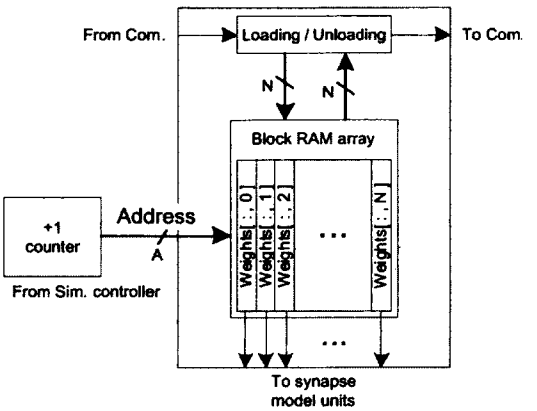

Figure 4.3 The weight memories, implemented altogether as an array of block RAMs. Each column stores the weights for the synapses connecting each neuron of the network to the neuron instantiated by the column. An address in memory, received from the HSNN controller block, points to a single bit of a weight value from each column. During spike propagation, this address increments until it reaches the most significant bit of the last weights and is then reset. The Loading/unloading block on top of the memory is a serial to parallel register used during initialization and read back.

function. However, this is not the only possibility, and a more hardware friendly solution can be used.

Let us first see why the exponential charging curve leads to the synchronization of strongly connected neurons. Consider a neuron defined by equation 3.1. The effect of incoming spikes on the membrane potential of the neuron is shown in Fig. 4.4. In the figure, the neuron receives two spikes which both increase its potential by the amount of the synaptic weight w. This increase in potential can be translated into a change in the neuron's phase. Depending of the timing of the spike, the same weight will have a different influence on the receiving neuron's phase. The closer the neuron is to the threshold when it receives the spike, the bigger the increase of its phase.

Consider now two strongly connected neurons. The neurons' phase is 0 when their potential is 0 and 2π when they reach the threshold. The initial phase difference between the neurons is $\Delta\phi \leq \pi$. When neuron A reaches the threshold, neuron B's phase is $2\pi - \Delta\phi$. The spike will increase the phase of neuron B by an amount c_1. Neuron B will eventually reach its threshold also, spiking on neuron A. At that moment, neuron A's phase is $\Delta\phi - c_1$, which is smaller than $2\pi - \Delta\phi$, the phase at which neuron B was when it received the first spike. The corresponding influence on neuron A's phase c_2 is thus smaller than c_1. After both neurons have spiked, the initial phase difference $\Delta\phi$ is reduced by an amount $c_1 - c_2$. This is due to the form of the charging curve, which translates the same weight w into different

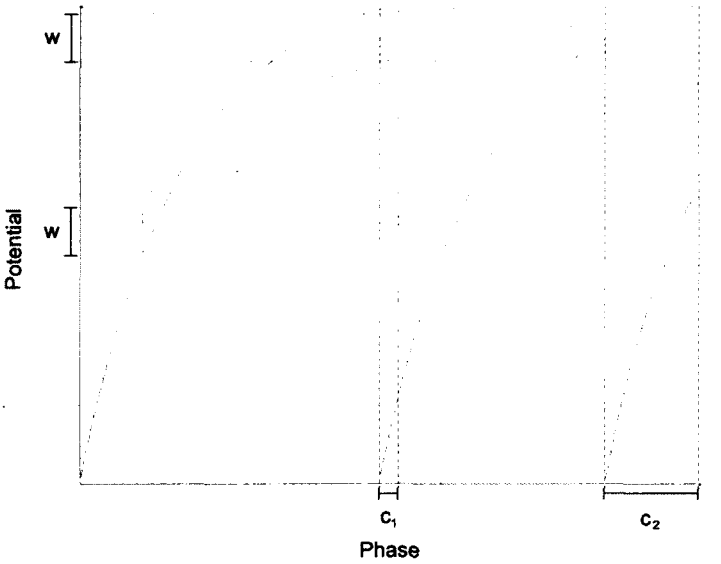

Figure 4.4 The effect of spikes on a neuron's phase. The figure shows the membrane potential of a neuron as it receives two spikes of synaptic weights w. The figure shows both the neuron's actual membrane potential (blue curve) and it's potential if it hadn't received the spikes (red curve). The difference between the blue and a red curve represents the influence c of a spike on the neuron's phase. The closer the neuron is to the threshold as it receives a spike, the larger the resulting increase of its phase.

increases in phase. If the charging curve was a straight line, the increases in phase c_1 and c_2 would be equal and the final phase difference between the neurons would remain $\Delta\phi$. In the example, when neuron B received the first spike, it was in the second half of the charging curve, that is, its phase was larger than π. Since both neurons are defined by the same charging curve, when neuron B reaches the threshold, neuron A will necessarily be in the first half of the charging curve. Thus, if the slope of the first half of the curve, between phases 0 and π, is different than the slope in the second half, from π to 2π, the interaction between the neurons will lead to their synchronization. This is the most basic requirement on the charging curve to obtain synchronization.

4.4 Implementation of a piece-wise linear charging curve

It is possible to approximate equation 3.1 using a piece-wise linear function. In a time-driven implementation, the membrane potential of the neurons is increased at each time step. Depending on the current state or phase of the neuron, one time step will correspond to a different increase of its membrane potential (the reasoning is the same as for when a

neuron receives a spike). The slope of the segments in the piece-wise linear approximation define these increases to apply at each time step. To get the concave down shape required for synchronization to occur, the slope of a segment must be less than that of the previous segment. The exact slope of each segment as well as the position of the turning points between the segments must be chosen carefully.

In hardware, equally spaced turning points are easy to implement. For example, to detect 3 equally spaced turning points, the 2 MSBs of the membrane potential can be monitored. Furthermore, it is simple in hardware to add values which are powers of 2, since they correspond to a single bit. Taking these facts into consideration, the values to add, based on the value of the state variable, can be calculated using a simple decoder. For instance, the 2 MSBs of the state variable are fed to the encoder which outputs a one-hot 4-bit word. This word is then shifted to the left appropriately and added to the state variable. If the word is not shifted, the state variable will increase very slowly in time. Conversely, if the word is shifted by several bits, the dynamics of the state variable will be faster. An example of a 4-segment piece-wise linear approximation using this method is given in figure 4.5.

The number of segments in the piece-wise linear approximation influences the capacity of neurons to synchornize. If the approximation is too poor, with only two segments for example, two strongly connected neurons might have to exchange several spikes before they synchronize. The use of a larger number of segments might result in the two same neurons synchronizing after a single spike. To assess the exact impact of the number of segments in the linear approximation, experiments with the Brian simulator were done. An equivalent of the ODLM neural network was implemented in Brian and three different 23×23 pixel images were segmented using a 4- and an 8-segment piexe-wise linear approximation. The second image was also segmented with a 16-segment approximation. The three images are shown in figure 4.6 and table 4.1 summarize the results of the experiments. For each image and approximation, the mean time required for a good segmentation to be obtained, over 20 trials, is reported. This simulation time is directly related to the number of spikes required for strongly connected neurons to synchronize.

Table 4.1 Mean time, over 20 trials, for the network to segment three different images using a 4- and an 8-segment piece-wise linear approximation in the Brian simulator

	IMAGE 1		**IMAGE 2**			**IMAGE 3**	
Number of segments	**4**	**8**	**4**	**8**	**16**	**4**	**8**
Mean time (s)	50.9	21.1	71.8	30.3	7.0	136.8	58.2

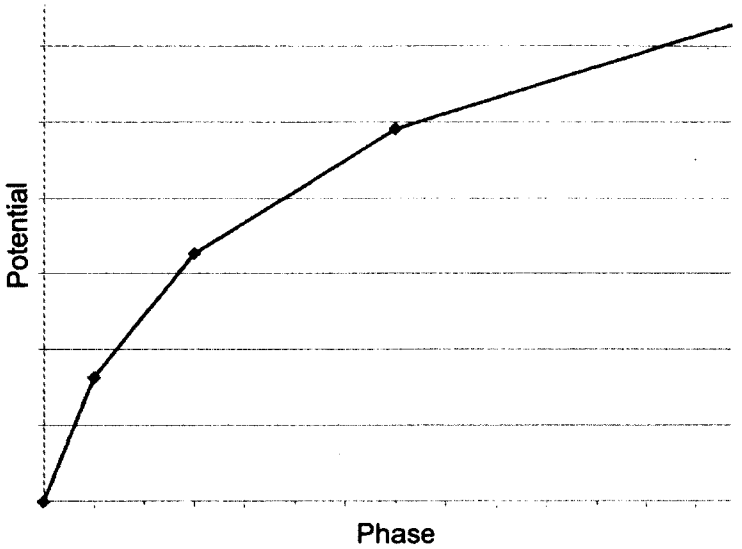

Figure 4.5 A 4-segment piece-wise linear charging curve generated using the described hardware method.

The results shown in table 4.1 show that an increase in the number of segments results in much quicker synchronization of the neurons. The use of an 8-segment approximation reduces the time required to segment a simple image by 60% over a 4-segment approximation. Using the 16-segment approximation further reduces the time by 75%, for a total 10× improvement over the 4-segment approximation. Experiments with the original ODLM algorithm and the event-driven system described in chapter 5 show that increasing the number of segments past 16 does not result in a significant improvement of performance. The limited logic resources on the FPGA prevent the use of more than 4 segments for the piece-wise approximation of the neuron model.

4.5 Results

Using 16 bits to code the membrane potential values and 11 bits for the weights, a fully connected network of up to 648 LIF neurons and 419 904 synapses can be implemented on a single Xilinx XC5VSX50T device. The system is clocked at 100 MHz.

4.5.1 Resource usage

Table 4.2 summarizes the resources used by the whole system in terms of 6-input look-up tables (LUTs), flip-flops (FFs), slices and block RAMs (BRAMs). It also shows how these

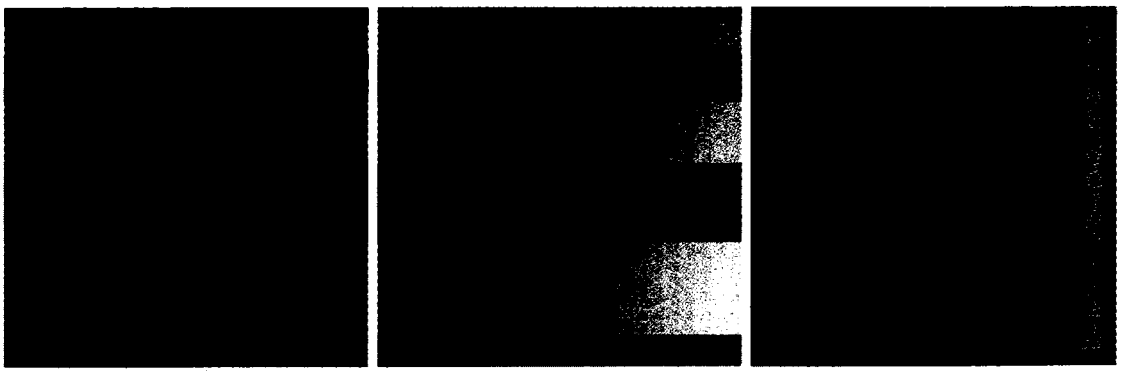

Figure 4.6 Images for the experiment on the number of semgents in the piecewise linear approximation. **Left.** Image 1, an 23×23 pixel grayscale image of three rectangles. **Center.** Image 2, the same 23×23 pixel image with a slight gradient from left to right. **Right.** Image 3, an 23×23 pixel grayscale image of nested rectangles with a slight gradient from left to right.

numbers relate to the number of neurons (N), and the width of weights (W) and potentials (P). Table 4.3 presents the resource usage for each functional block.

4.5.2 Performance

Table 4.4 summarizes the processing power of the HSNN both in terms of the number of clock cycles required to compute one neuron oscillation period and the number of spikes processed per second. The performance depends on the number of simultaneously spiking neurons. In the best case scenario, all neurons fire at the same time step, and the performance is proportional to $P \times N + M$, with M the number of time steps in an oscillation period. On the other hand, if all neurons fire one at a time, the spikes per second count will be lower (depending on $P \times \mathrm{N}^2 + M$). In the current implementation, M is set to 448.

4.5.3 Image segmentation task

The segmentation of an image starts with the computation of the synaptic weights with equation 3.3 using the pixels' level of grey as features. This computation is done on a personnal computer and the weight values are sent to the HSNN using a serial cable. The random initial value of the neurons' membrane potential is also computed on the computer. The computer then sends a command to enable the HSNN for a given time. The final membrane potentials are uploaded to the computer and translated into a pixel color value for vizualization. The results of an image segmentation experiment are presented in

Table 4.2 Resource usage for the whole system

RESOURCE	USED	AVAILABLE	% USED	DEPENDENCE
FFs	14 098	32 640	43%	P, N
LUTs	22 815	32 640	69%	P, N
Slices	8 106	8 160	99%	N/A
BRAMs	126	132	95%	W, N^2

Table 4.3 Resource usage for each functional block

FUNCTION	FFs	LUTs	BRAMs
Weight memory	663	1 031	126
Synapse model	1 286	2 026	0
Membrane model	12 050	19 531	0
FSM	99	227	0

Fig. 4.7. The simulation lasted 2 seconds, not accounting for the loading of the weights and initial potentials, and 50 kspikes were propagated.

4.5.4 Image comparison task

The image matching task represents the comparison of a set of previously segmented images, according to the methodology in [Pichevar *et al.*, 2006]. For this simple example, the features used are the pixel values of different regions of the images. Synaptic weights are computed according to equation 3.3 where parameters w_{max}, α and δ are respectively set to $v^{threshold}/32$, 100 and 4. As for the segmentation task, the synaptic weights and initial potential values are calculated on a personnal computer and loaded into the HSNN using a serial cable. When the processing is done, the final potentials are uploaded to the computer for visualization.

The result of the task of a positive and a negative matching task are shown in Fig. 4.8. The figure presents the final potential values of the neurons to give a general idea of the synchronized neurons. The result of the comparison is a score based on the correlation between the spike phases of the neurons of both images [Pichevar *et al.*, 2006]. The score for the positive matching was of 0.992 and it was 0.278 for the negative task. In both cases, the HSNN ran for 550 ms, not accounting for the loading time and approximately 45 kspikes were processed.

Table 4.4 Performance for a fully synchronized (best case) and desynchronized (worst case) network

SCENARIO	CLOCK CYCLES	SPEED (AT 100 MHZ)
Best case	11 k	6 Mspikes/s
Worst case	6.7 M	9.6 kspikes/s

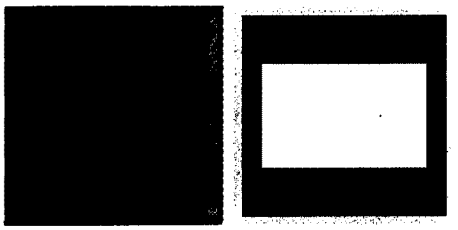

Figure 4.7 Image segmentation experiment. **Left.** 23×23 pixel input image. The input is a set of nested rectangles with a slight variation of gray level from left to right; **Right.** Segmented image. The output image is built by defining the gray level of each pixel according to the membrane potential of its associated neuron. The potential is used as an indicator of the phase of the oscillator. As expected, the different segments on the output are easily discriminated based on their gray level.

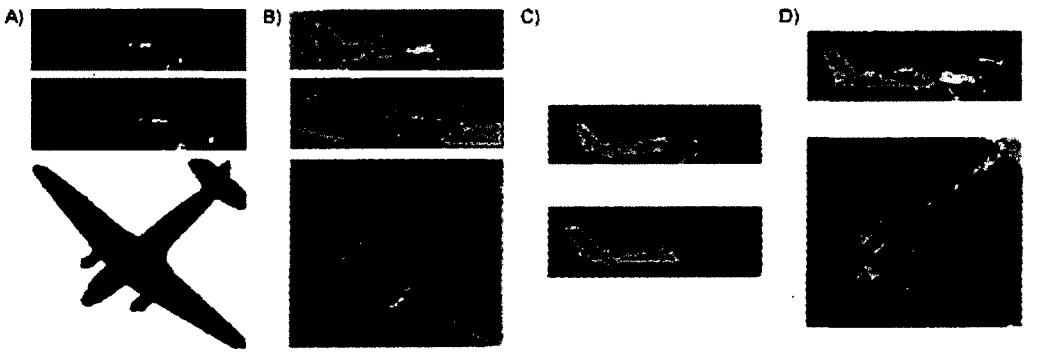

Figure 4.8 Image comparison experiment. **A.** Original images. The two top images are different views of the same airplane. The third image is a different airplane. Images size are, from top to bottom, 90×26, 89×30 and 89×79 pixels; **B.** Segmented images used as input; **C.** Result of a positive matching. Most of the segments from the two images are synchronized (have the same color). Comparison score: 0.992; **D.** Result of a negative matching. Comparison score: 0.278. The colors of images C and D represent the final value of the membrane potential of each neuron. Synchronized neurons have similar final membrane potentials and thus show the same color on the output images. The choice of the colors is arbitrary.

CHAPTER 5

EVENT-DRIVEN HARDWARE IMPLEMENTATION

5.1 Introduction

This section presents a second FPGA implementation of the ODLM. This second system is event-driven, which reduces the number of computation compared to the time-driven approach. Also, a weight matrix is not saved in memory. Instead, the post-synaptic neurons are identified when required. The processing element consists of a 5-stage pipeline and most of the calculations are memory based. An event queue is used to keep track of the next neuron to fire. It is implemented using a structured heap, a hardware sorting algorithm developed for this specific implementation. The system consists of a single processing element and the event queue, but this design can be enhanced by adding more processing elements. The results of an image segmentation experiment are presented.

The system is inspired by the work done at the Reservoir Lab of Gent University in Belgium. They developed an FPGA implementation of a liquid state machine (a special type of ASNN) and used it for isolated spoken digit recognition [Schrauwen *et al.*, 2008]. They use a different neuron model and other features specific to reservoir computing, but the global idea behind the system remains the same. The main difference between their work and the one presented here is the introduction of the structured heap queue algorithm, which greatly simplefies the hardware implementation of the ASNN. The structured heap queue is described in section 5.4.3.

The remaining of this chapter will introduce the concept of event-driven simulation. Then, the structured heap queue will be described. The hardware architecture will then be presented followed by the performance of the system and results of the image segmentation experiment.

5.2 Event-driven approach

According to equation 3.1, the potential of an isolated neuron increases continuously. Using this equation, it is possible to calculate how much time is left before a neuron's membrane

potential reaches the threshold on its own. This calculation is voided if the neuron receives a spike in the meantime, since it will fire earlier than expected. However, the firing time of the very first neuron to fire cannot be wrong, since no other neuron can possibly fire before that time. The event-driven simulator can safely jump to the time of occurrence of this event. It is then processed and the next event to happen is identified and resolved similarly. In a time-driven implementation, the time steps are usually much smaller than the average time interval between two events. The membrane potential of the neurons is updated regularly, even though no significant change of the network's state happenned. Conversely, in our event-driven HSNN, a neuron's state is only modified when it is involved in an event. Much less computation is required and the amount of data moved in and out of memory is significantly reduced. Our event-driven HSNN uses a sorted list of firing times, the event queue, to keep track of the next neuron to fire. The HSNN loops over the following steps:

Algorithm 1 - Event-driven simulation

```
1. Sort the event queue
2. Pick the neuron on top of the queue
3. Jump to the firing time of this neuron
4. Process the event
```

5.3 Processing of an event

When a spike is processed, the pre-synaptic neuron is reset and a synaptic weight is added to the membrane potential of each of the post-synaptic neurons. These two operations are very similar, as they both consist in the modification of the membrane potential of a neuron. Algorithm 2 is used to apply the effect of an incoming spike on a post-synaptic neuron, that is, to add a synaptic weight to its membrane potential.

Algorithm 2 - Processing of a spike

```
1. Identify the post-synaptic neuron to process
2. Retrieve the neuron's firing time and pixel value
3. Calculate the neuron's membrane potential
4. Calculate the synaptic weight
5. Calculate the neuron's new membrane potential
6. Calculate the neuron's new next firing time
7. Update the neuron's information in the state memory
```

The same algorithm can be used to the reset the pre-synaptic neuron, with a single difference. Instead of adding a synaptic weight to the membrane potential of the neuron, step 5 of algorithm 2 resets the mmebrane potential (and discards the synaptic weight calculatated in step 4). For the processing of an event, algorithm 2 is executed $N + 1$ times, where N is the number of post-synatic neurons. During one of the executions, the pre-synaptic neuron is reset.

The state of the neurons is stored in memory as the predicted time of their next spike. That way, the next neuron to fire can be easily identified by sorting the state of the neurons. However, when a neuron receives a spike, it's membrane potential, and not its predicted firing time, must be summed with a synaptic weight. Steps 3 and 5 of algorithm 2 are required to translate the predicted firing time into a membrane potential and then back into predicted firing time. Equation 3.1 defines the maping between firing time and membrane potential. To compute a membrane potential from a predicted firing time using equation 3.1, it is necessary to know the current time. Then, the potential can be computed based on the difference between the predicted firing time and the current time. For example, if the difference is small, the neuron should fire shortly and its membrane potential will be large. Otherwise, the potential will be low. Each time the HSNN processes an event, the current time is updated to the time of the present event.

The HSNN identifies the post-synaptic neurons of an event on-the-fly. In the present implementation, a synapse refers to a precalculated offset which is added to the pre-synaptic neuron's identification number (ID) to obtain a post-synaptic neuron ID. The same offset values are used no matter which neuron fires, imposing a regular topology to the network. Different strategies can be used to resolve various regular topologies. For example, the fully-connected network used for image matching can be implemented by using a counter to cycle through the whole set of neuron IDs. For segmentation, an 8-neighbor connectivity is used. The precalculated offsets technique is used for this topologies, but it sometimes produces illegal post-synaptic neuron values for neurons on the edge of the image. To prevent this, a layer of inactive neurons (that is, neurons that cannot fire) must be present all around the network.

The synaptic weights for the segmentation task depend on a single variable: the difference of the pixel values associated with two given neurons. The calculation of the weights is further simplified by the small range of values the pixels can take: from 0 to 255. A small LUT can thus be used to store all the possible synaptic weight values. The weight of every existing synapse doesn't have to be stored, as it is calculated on demand.

5.4 Implementation

The present system is an implementation of algorithms 1 and 2 on a FPGA. An overview of the resulting HSNN is given in figure 5.1. The components of the system are detailed in the following sections.

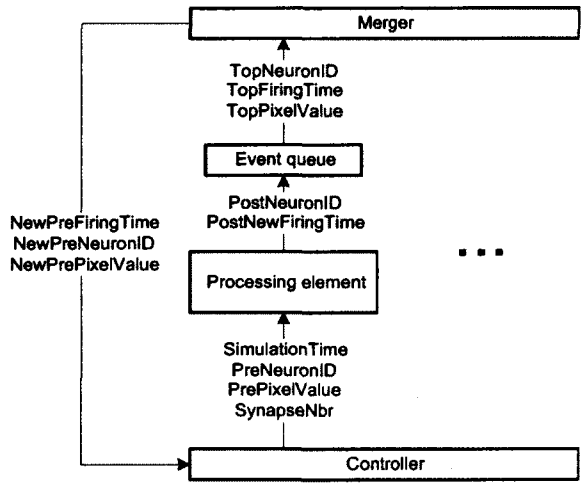

Figure 5.1 High level view of the HSNN. The controller receives the next event's information from the merger and forwards it to the processing element (PE) along with some control signals. The PE identifies the post-synaptic neurons, and computes the required synaptic weights and the new state of the processed neurons. The new state of the post-synaptic neurons is sent to the event queue for update and the queue is sorted on-the-fly. The merger reads the event on top of the event queue and sends it to the controller. If several PEs and event queues are used, the merger chooses the very next event to happen and the controller distributes the processing load among the PEs.

5.4.1 Controller

The controller realizes step 3 of algorithm 1. It receives from the merger the next event to be processed, then updates the simulation time and forwards the event for processing. If several processing elements (PEs) are implemented, the synapses to be processed can be evenly distributed among them. In the present work, the post-synaptic neurons are serially processed by a single PE. The synapse number increments until all synapses of the pre-synaptic neuron have been processed. The controller also takes care of the communications with the computer host.

5.4.2 Processing element

A PE instantiates step 4 of algorithm 1 as detailed in algorithm 2. An overview of a PE is given in figure 5.2. PEs receive a synapse number, a pre-synaptic neuron ID, a weight parameter and the simulation time from the controller. A 5-stage pipeline processes the neurons involved in the event, computing their new state and sending the result to the event queue for update. In the first stage of the pipeline, step 1 of algorithm 2 is executed. Stage 2 executes the 2nd step of the algorithm. Steps 3 and 4 are executed in the 3rd stage of the pipeline. The 4th stage implements steps 5 and 6 and the last stage executes step 7. Each stage of the pipeline takes one clock cycle to execute. The stages of the pipeline are detailed in the following sections.

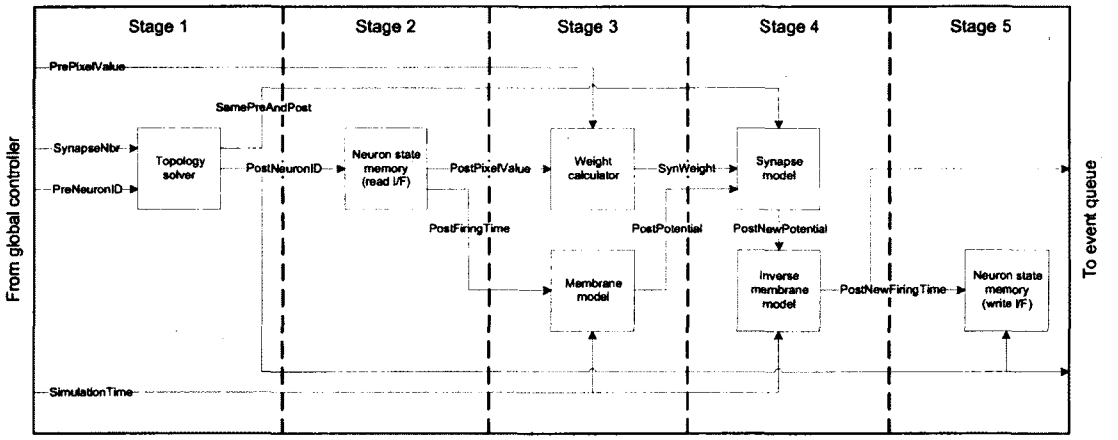

Figure 5.2 A processing element of the hardware event-driven system. The inputs to the pipeline are coming from the controller and the outputs are sent to the event queue. The topology solver identifies the post-synaptic neurons associated with the event. The neuron state memory stores the state of all neurons. It has one read and one write interface, the latter being used for write-back in the last stage of the pipeline. The weight calculator computes a synaptic weight based on the pixel value of the processed neurons. The membrane model translates the firing time of a neuron into a membrane potential value. The synapse model either adds a synaptic weight to the membrane potential of a neuron or resets this neuron. The inverse membrane model translates the new membrane model back into a firing time. Signals crossing dashed lines are delayed appropriately (delay elements not shown).

Topology solver

The topology solver translates a pre-synaptic neuron ID and synapse number into a post-synaptic neuron ID. The block also indicates when the post-synaptic neuron ID is equal to the pre-synaptic neuron ID. In this case, the membrane potential of the neuron will be

reset in a latter stage of the pipeline, as explained in section 5.3. A functional view of the topology solver is shown in figure 5.3.

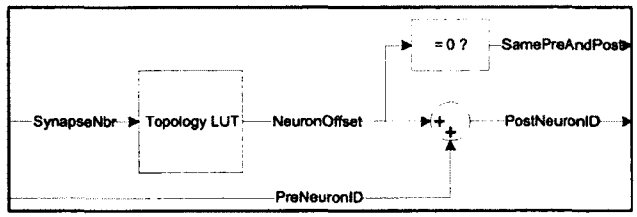

Figure 5.3 The topology solver is the 1st stage of the pipeline in a processing element. The synapse number is used as the address to the topology look-up table which outputs an offset to add to the pre-synaptic neuron ID. If the offset is 0, a flag is set, indicating that the pre-synaptic and post-synaptic neuron IDs are equal.

Neuron state memory

In the second stage of the pipeline, the post-synaptic neuron's information is retrieved from the neuron state memory. This memory stores the state variables and parameters of the neurons. For the image segmentation experiment, a firing time and a pixel value are stored for each neuron. Neuron IDs are used to address the neuron state memory.

Weight calculator

The weight calculator takes two pixel values and computes the corresponding synaptic weight. The calculation of the weight depends on the difference between the pre-synaptic neuron's pixel value and the post-synaptic neuron's pixel value. A functional view of the weight calculator is shown in figure 5.4.

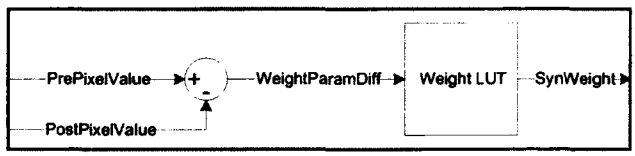

Figure 5.4 The weight calculator composes, with the membrane model, the 3rd stage of the pipeline of a processing element. The pixel difference is used as the address to a look-up table which outputs the synaptic weight.

Membrane model

The membrane model calculates the membrane potential at the current simulation time given a firing time. A functional view of the membrane model is shown in figure 5.5.

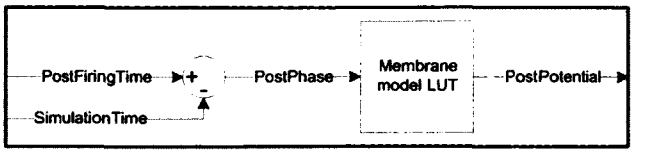

Figure 5.5 The membrane model, along with the weight calculator, is part of the 3rd pipeline stage in a processing element. The subtraction of the simulation time to the firing time of a neuron results in a value which represents how far from the threshold the neuron is. This value is used as the address of a LUT to get the membrane potential of the neuron.

Synapse model

The synapse model has a different role depending on the value of the flag outputted by the topology solver. If the flag is 0, then the post-synaptic neuron's potential must be added to the synaptic weight. Otherwise, the processed neuron is the pre-synaptic neuron, and its potential has to be reset since it just fired. A functional view of the synapse model is shown in figure 5.6.

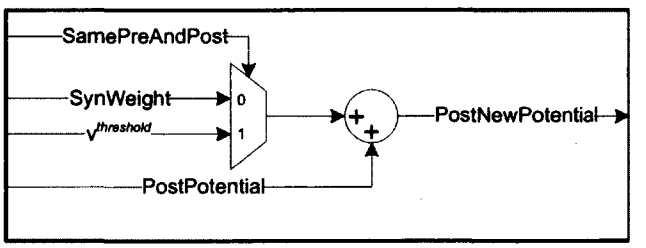

Figure 5.6 The synapse model is combined with the inverse membrane model to form the 4th stage of the pipeline in a processing element. The post-synaptic neurons' membrane potential is added to the synaptic weight if the input flag is 0. Otherwise, the value $v^{threshold}$ is subtracted from the potential to reset the pre-synaptic neuron.

Inverse membrane model

The inverse membrane model computes the new firing time of a neuron based on its new membrane potential. A functional view of the inverse membrane model is shown in figure 5.7.

State memory write back

Once the new firing time is calculated, it is written back into the neuron state memory. This is done in the last stage of the processing pipeline.

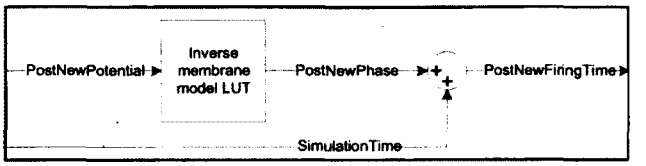

Figure 5.7 The inverse membrane model and the synapse model form the 4^{th} stage of the pipeline. The post-synaptic neuron's membrane potential is the address to a look-up table which outputs the amount of time left until the neuron fires. This value is added to the simulation time to get the new neuron's firing time.

5.4.3 The event queue

The event queue is implemented using a structured heap queue, a sorting algorithm derived from the heap queue. The difference between the standard heap queue and the structured heap queue is that in the latter, elements inside the queue can be located. This is possible because each element can only be found on a specific path down the binary tree as shown in figure 5.8. As for the standard heap queue, the structured heap queue supports the insertion of new elements. In addition, the structured heap queue allows the deletion of arbitrary elements in the tree. It can also be used as a memory, as read operations are supported. The memory-optimized structured heap queue described in section 5.4.4 has a 25% memory overhead when compared to the standard heap queue.

The structured heap queue is used to sort the neurons according to their predicted firing time. The queue takes the form of a binary memory tree. Each node in the tree can store an element, which consists of a neuron ID and its predicted firing time. Elements are moved up (promoted) and down (demoted) the tree to maintain the heap property, that is, to guarantee that any node has a smaller predicted firing time than its 2 children nodes. The next neuron to fire is found in the root node at all time. When an element is moved down the tree during an insertion, it can only follow a specific path determined by its neuron ID. Figure 5.8 shows a schematic representation of the binary memory tree. The structured heap queue does not keep the binary tree balanced. Empty nodes can appear in the tree, but only if all of their children nodes are also empty.

An L-level structured heap queue can sort up to 2^{L-1} elements and is composed of $2^L - 1$ nodes, as shown in figure 5.8. In comparison, an L-level standard heap queue can sort twice this number of elements using the same amount of memory. This is the price to pay to be able to locate and delete arbitrary elements in the tree. The memory overhead can be minimized by the memory-optimized version of the structured heap queue, described

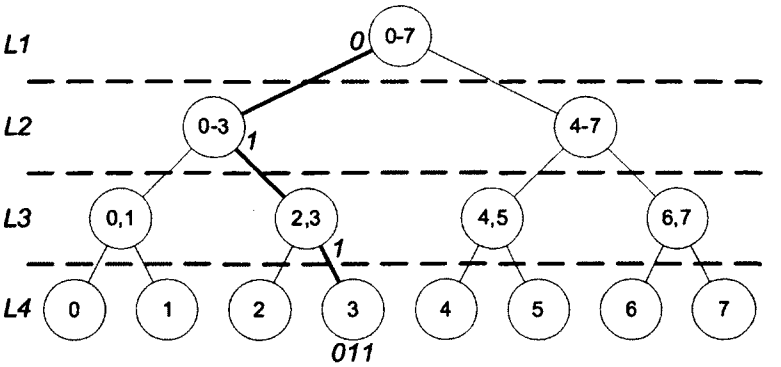

Figure 5.8 Schematic representation of the binary memory tree in a structured heap queue. Each element moves down the tree following a unique path from the root node to one of the leaf nodes. The numbers in a node indicate which elements can possibly occupy this node. Any element can be stored in the root node, but each leaf node is specific to only one element identification number (ID). The path associated with an element is found by branching either left or right depending on the binary representation of the element's ID, starting with the MSB when branching from the root node. The path associated to the element with ID 3 is shown in bold. This 4-level queue can sort up to 8 elements.

in section 5.4.4. This version can handle 2^{L-1} elements with $1.25 \times 2^{L-1}$ nodes, which only represents a 25% increase in memory compared to the standard heap queue.

In software, the complexity in time for delete, insert and read operations in the structured heap queue is $O(L)$. In hardware, the complexity in time is $O(1)$ for delete and insert and $O(L)$ for the read operation. The complexity in logic is $O(L)$, as one comparator exists on each level of the binary tree. Table 5.1 compares the complexity of the standard heap queue and our structured heap queue when implemented on hardware. The values shown in table 5.1 for the standard heap queue are based on [Thompson, 1983; Bhagwan and Lin, 2000; Ioannou and Katevenis, 2007]. The structured heap queue basically has the same performance, with the differences that the standard heap queue does not support the "delete any node" and "read" operations and the structured heap queue has a 25% memory overhead.

Operations in the structured heap queue

All operations in a structured heap queue are issued at the root node and ripple down the tree, one level at a time.

Delete and read operations The delete operation is divided into 2 phases: location and promotion. To locate an element, the nodes on its path are read, starting from the

Table 5.1 Comparison of the hardware implementation of the standard (HQ) and the structured (SHQ) heap queues, with N the number of elements to sort

	HQ	**SHQ**
Complexity in logic	$O(\log(N))$	$O(\log(N))$
Complexity in memory	$O(N)$	$O(N)$
Complexity in time for		
Delete root node	$O(1)$	$O(1)$
Delete any node	-	$O(1)$
Insert	$O(1)$	$O(1)$
Read	-	$O(\log(N))$

root node, until it is found. This scanning of the tree is executed one level at a time. In a delete operation, the element is removed from the tree, creating an empty node. Elements from lower levels of the tree must be promoted to fill this empty node. An example of a delete operation can be seen in figure 5.9. A read operation finishes when the element is found, as its information can be outputted right away.

Insert operation When an element is inserted, it is compared with the elements located on its path. The inserted element is pushed down the tree as long as it encounters elements with a higher priority than its own. When it finds a node with a lower priority element, it takes its place and pushes this element down the tree. An example of an insert operation is shown in figure 5.10.

Pipelining the operations The operations in the structured heap queue can be pipelined. As soon as an operation is passed to a lower level of the tree, another one can be serviced at the current level. The heap property is always satisfied and the highest priority element occupies the root node at all time, even if operations are still going on in lower levels of the tree. Delete operations, since they deal with data on 2 levels of the tree, 1 node and its 2 children nodes, take longer to execute than insert operations. Table 5.2 shows how delete and insert operations can be pipelined if reads, comparisons and write-backs each take one clock cycle to execute. Delete and insert operations can also be interleaved, for example to change the priority of an element. The table shows how to pipeline two consecutive delete operations, two insert operations and finaly how to alternate delete and insert operations (delete-insert operation). If the comparison in the insert operation is allowed to use the result of the comparison of the delete operation, delete-insert operations only require one extra clock cycle over a delete operation. Figure 5.11 shows an example of a delete-insert

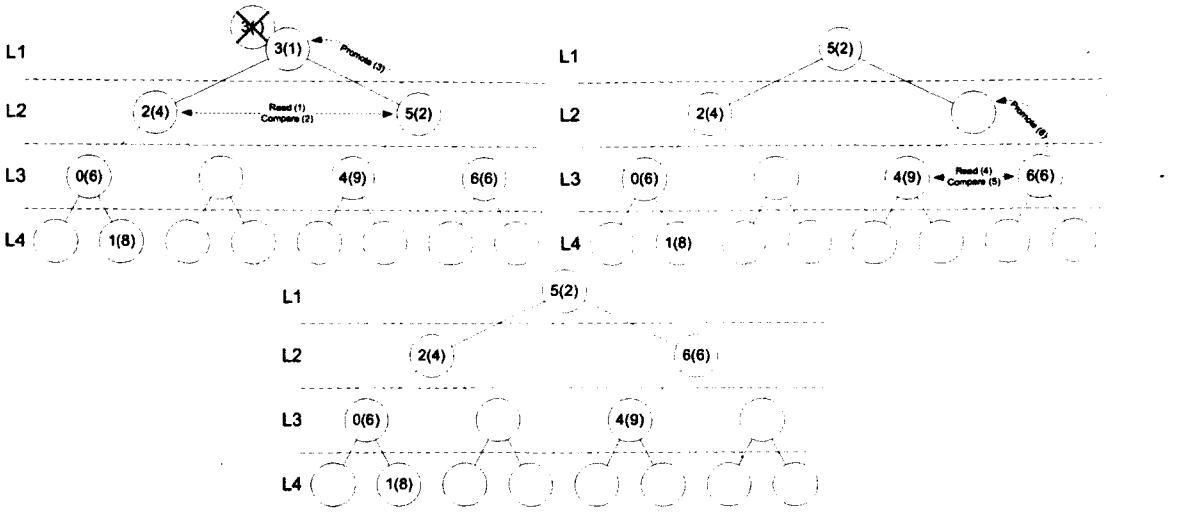

Figure 5.9 Delete operation in the structured heap queue. The pair of numbers in a node indicates the identification number and, inside parentheses, the priority of the element (in this example, the lower the number inside parentheses, the higher the priority). The operation consists in the deletion of the element with identification number 3. **Top left.** Initial state of the queue and first steps of the delete operation. Element 3 is in the root node and thus the locate phase doesn't take place. Elements 2 and 5 are read and compared to determine that element 5 should be promoted to replace the deleted element. **Top right.** Last steps of the delete operation. Elements 4 and 6 are read and compared and the latter is promoted. **Bottom** Final state of the queue after the delete operation.

operation in a 4-level memory-optimized structured heap queue introduced in the next section.

5.4.4 Memory-optimized structured heap queue

An L-level structured heap queue as it was presented in the previous sections comprises $2^L - 1$ nodes and can sort up to 2^{L-1} elements. The queue requires $2^{L-1} - 1$ more memory locations than the number of elements it can actually handle. Actually, the last level of the tree alone provides enough memory to store the entire set of elements to sort. However, empty nodes are not allowed to form in the middle of the tree and it is impossible that the last layer be full, since the rest of the tree would have to be empty. The size of the last level of the tree can thus be reduced. A 3-level non-optimized structured heap queue handles 4 elements and comprises 7 nodes. For an element to be allowed to occupy a node on the last level, all the other elements must fill the 3 nodes located in the upper levels of the tree. Thus, in a 3-level queue, only 1 of the 4 nodes of the last level can be occupied. Increasing the size of the queue to 4 levels requires the addition of another 3-level queue

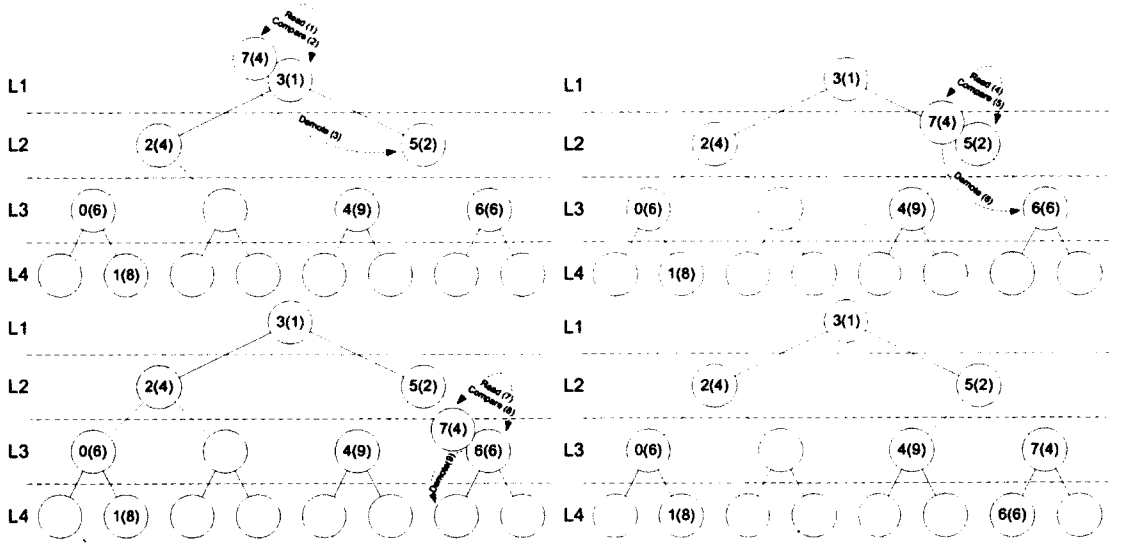

Figure 5.10 Insert operation in the structured heap queue. The pair of numbers in a node indicates the identification number and, inside parentheses, the priority of the element (in this example, the lower the number inside parentheses, the higher the priority). The operation consists in the insertion of the element 7 with priority 4. **Top left.** Initial state of the queue and first steps of the insert operation. Elements 7 and 3 are read and compared to determine that the former is to be demoted. **Top right.** Intermediate steps of the insert operation. Elements 7 and 5 are read and compared and the former is demoted. **Bottom left.** Last steps of the delete-insert operation. Elements 7 and 6 are read and compared and element 6 is demoted. **Bottom right.** Final state of the queue after the insert operation.

next to the first one plus a new root node on top of them. The queue can now sort up to 8 elements. Still, only one element can go all the way down each of the 3-level queues. This reasoning holds true whatever the size of the queue. The memory-optimized structured heap queue reduces the size of the last level of the memory tree by 75%. An example of a 4-level memory-optimized structured heap queue is shown in figure 5.12. The last level of a memory-optimized structured heap queue has to be handled differently than the other ones, since each of its nodes has 2 parent nodes. Also, each path down the tree is now shared by 2 elements.

5.4.5 Merger

The merger is required when several PEs are used in parallel. It operates on the output of every PE to determine the very next event to be processed.

Table 5.2 Pipelining delete and insert operations in the structured heap queue

	Clock cycle							
Operations	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1st Delete	r(L2)×2	c(L1)	w(L1)	r(L3)×2	c(L2)	w(L2)	r(L4)×2	c(L3)
2nd Delete	-	-	-	-	-	-	r(L2)×2	c(L1)
1st Insert	r(L1)	c(L1)	w(L1)	r(L2)	c(L2)	w(L2)	r(L3)	c(L3)
2nd Insert	-	-	-	r(L1)	c(L1)	w(L1)	r(L2)	c(L2)
1st Delete	r(L2)×2	c(L1)	w(L1)	r(L3)×2	c(L2)	w(L2)	r(L4)×2	c(L3)
1st Insert	-	r(L1)	c(L1)	w(L1)	r(L2)	c(L2)	w(L2)	r(L3)
2nd Delete	-	-	-	-	-	-	-	r(L2)×2

This table shows how to pipeline several operations in the structured heap queue. Each operation requires reading one or two elements on a given level of the tree, comparing them together or with a newly inserted element, writing back one of the elements on the current level and passing the operation to the next level down the tree. In the table, r() stands for a read, c() is a comparison and w() is a write. The "Lx" inside parentheses indicates the level of the tree where an operation takes place. The "×2" indicates that 2 simultaneous reads must be performed. A "-" sign means that the operation has not started yet.

5.5 Results

A network of 65 536 neurons is implemented on a Xilinx XC5VSX50T FPGA clocked at 100 MHz. The HSNN is setup for an image segmentation experiment, which means that each neuron has 8 synapses for a total of 513 184 implemented synapses. The HSNN can support many other regular network topologies and, in a fully-connected network of 65 536 neurons, there would be more than 4 Gsynapses implemented. Since synaptic weights are computed on-the-fly and are not stored in memory, this high number of synapses does not require an increased amount of memory.

5.5.1 Resource usage

Most of the computations are memory-based and few resources are required to implement a PE. Most of the memory is used to store the neurons' state and to implement the event queue. The membrane potential and firing time values are 13-bit wide so the membrane model and inverse membrane model LUTs can each be implemented using 3 BRAMs. The pixel values and synaptic weights are respectively 8-bit and 9-bit wide. The resources usage for the whole system is given in table 5.3 and table 5.4 details these numbers for each functional block of the design.

Table 5.3 Resource usage for the whole system

RESOURCE	USED	AVAILABLE	% USED
FFs	3 368	32 640	10%
LUTs	4 671	32 640	14%
Slices	2 855	8 160	35%
BRAMs	130	132	98%

Table 5.4 Resource usage for each functional block

FUNCTION	FFs	LUTs	BRAMs
Controller	503	635	0
Processing element	19	73	50
Topology solver	16	26	0
Neuron memory	1	1	44
Weight calculator	0	8	0
Membrane model	1	14	3
Synapse model	0	10	0
Inverse membrane model	1	14	3
Event queue	2 846	3 965	80
Merger	N/A	N/A	N/A

5.5.2 Performance

A PE can process one synapse every 7 clock cycles. The performance of the system in terms of events processed per clock cycle depends on the number of synapses per neuron. Equation 5.1 can be used to get an approximate of the system's performance. The processing pipeline has a latency of 7 and a depth of 9 (including the heap queue).

$$\text{Performance}_{\text{events/clock cycle}} = \frac{1}{9 + N_{\text{synapse/neuron}} \times 7} \tag{5.1}$$

5.5.3 Image segmentation

The results of an image segmentation task using the event-driven HSNN are presented in this section. For this experiment, the role of the computer is slightly different than with the time-driven HSNN. Here, the synaptic weights are computed on-the-fly on the FPGA when they are needed. At initialization, the computer simply transfers the pixel values of the image to segment and the random initial potential of the neurons into the HSNN. The HSNN is then run for a given time and the final membrane potential values are sent back to the computer for visualization. The parameters of equations 3.1 and 3.3, using the pixels' level of grey as features, are set to: $I_0 = 6.918$, $\tau = 0.1447$, $v^{threshold} = 1$, $w_{max} = 0.0325$,

$\alpha = 100$, $\delta = 6$. The network was run for 200 ms. The original and segmented images are presented in figure 5.13.

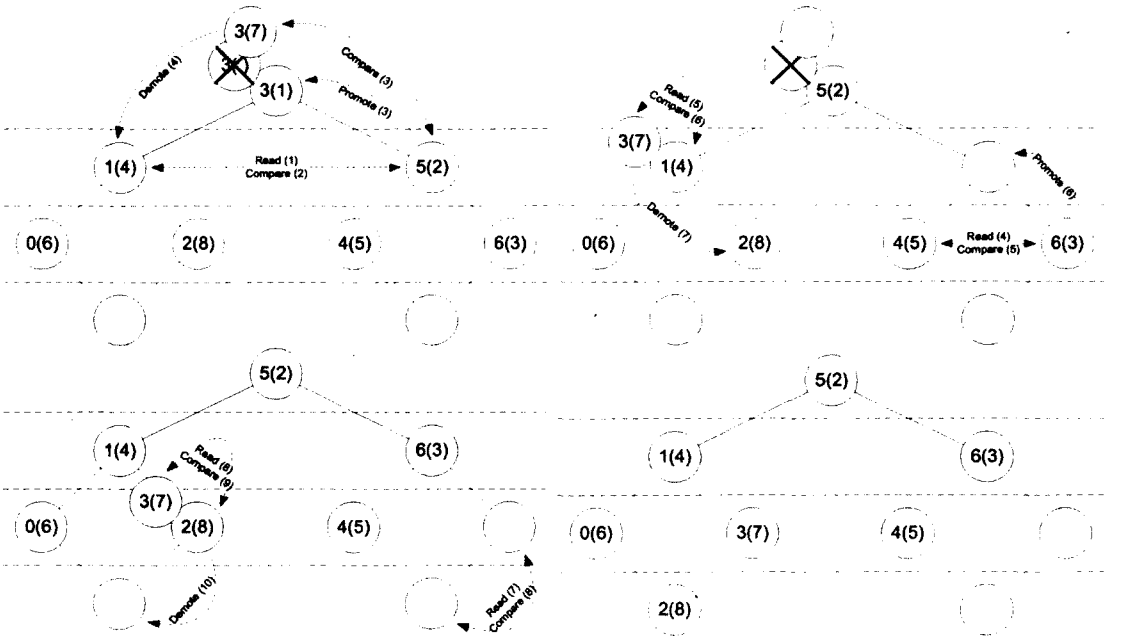

Figure 5.11 Delete-insert operation in a memory-optimized structured heap queue. The operation consists in the deletion of the element with identification number 3 directly followed with the insertion of the same element, changing its priority. The pair of numbers in a node indicates the identification number and, inside parentheses, the priority of the element. (in this case, the lower the number inside parentheses, the higher the priority). **Top left.** Initial state of the structured queue and first steps of the delete-insert operation. Elements 1 and 5 are read and compared to determine which one should be promoted to replace the deleted element. If the promoted element has a higher priority than the element inserted in the delete-insert operation, the latter is demoted. Otherwise, the inserted element would end up at the root node and the promoted element be returned to its former position. **Top right.** Intermediate steps of the delete-insert operation. Elements 4 and 6 are read and compared and the latter is promoted. Elements 3 and 1 are read and compared and the former is demoted. A new delete-insert operation can be issued during the 7^{th} clock cycle. **Bottom left.** Last steps of the delete-insert operation. If there were elements left in the path of the promoted element, they would be compared and moved appropriately. Elements 3 and 2 are read and compared and the latter is demoted. **Bottom right.** Final state of the structured queue after the delete-insert operation.

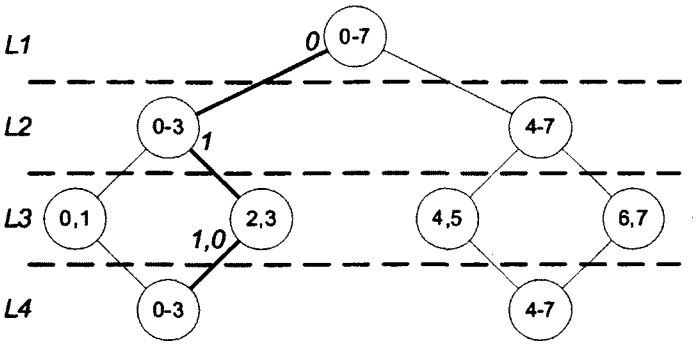

Figure 5.12 Memory-optimized structured heap queue. The numbers in a node indicate which elements can possibly occupy this node. The path associated to an element is found by branching either left or right depending on the binary representation of the element's identification number. In this version, the paths are shared by pairs of 2 elements. In the figure, the path shown in bold is shared by elements 2 and 3. Elements share paths, but they still can be distinguished by reading their information in the nodes. The last level of the tree uses 75% less memory than in the naive version.

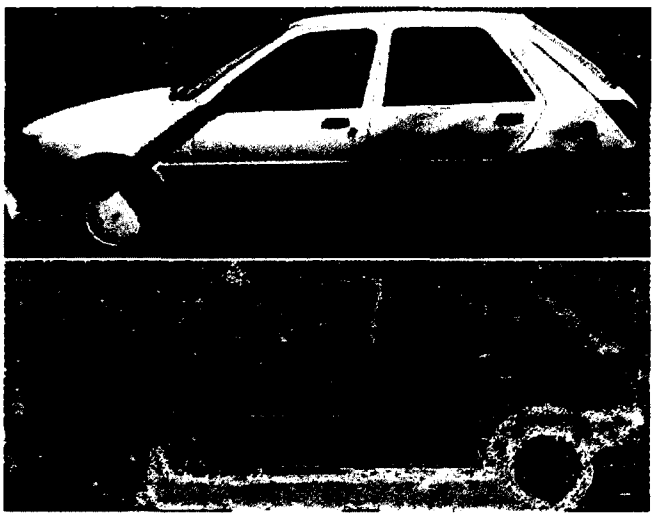

Figure 5.13 Image segmentation experiment. **Top**: The original 406×158 pixel image. **Bottom**: The segmented image, where different colors represent different segments. The segmentation is solely based on pixel values and the HSNN merges the lower part of the image with the tires of the car. Also, the foliage in the background is segmented with preservation of the texture.

CHAPTER 6

DISCUSSION

6.1 Evaluation of the bit-slice implementation

6.1.1 Performance

The use of a square matrix to store the synaptic weights is burdensome, but it has two major advantages. First, it allows the simulation of networks with any topology, making the HSNN very flexible in this regard. Also, the memory scheme is well suited for the implementation of plasticity, since there is a place allocated in the memory for every possible weight in the network. Plasticity rules are currently under evaluation and synaptic plasticity will be a part of future versions of the HSNN.

The time required to process the 448 time steps of an oscillation period is proportional to the number of isolated spiking groups. During one oscillation period, every neuron will fire exactly once. If all the neurons are synchronized, their spikes will be processed during the same spike propagation step. A spike propagation step always takes the same exact amount of time to execute, no matter how many spikes occurred in the time step. Since the use of the neural network for signal processing is based on synchrony, groups of synchronous neurons always form as the system runs. In the end, when the processing is done, large groups of synchronous neurons exist, spiking simultaneously, and the bit-slice architecture exhibits very high performance. When the processing starts, though, since the membrane potential of the neurons are initialized randomly, very few simultaneous spikes occur. The performance of the system thus varies greatly during the processing, hurting the overall figure.

The HSNN is constructed by duplicating and interconnecting slices. This repetitive design makes for easy placement on an ASIC. The routing is further simplified by the minimal amount of global signals. The slices mainly require local interconnection to neighboring units, hence the wire lengths are kept short. This data exchange scheme makes the system scalable, since wire lengths and the number of wire crossings will not increase with the amount of neurons.

Several factors limit the clock speed of the current design. The spike detection signal, a 648-input OR gate, is identified as the critical path of the design, with a delay typically around 9 ns. Also, almost all of the slices and BRAMs on the FPGA are used. This complexifies the routing of the circuit, leading to increased delays. More specifically, the weight memory is a huge assembly of BRAMs sharing common control signals. Those signals must be routed all over the FPGA, also involving delays around 9 ns. To improve the timings without having to use a different FPGA, it would be necessary to manually place parts of the design on the device. Redesign of the spike detection signal alone would not allow the use of a faster clock. It is however hard to predict how much, if at all, manual placement would improve performance.

6.1.2 Future work

The current implementation has been designed with focus on fully connected networks. As a result, it handles sparsely connected networks less efficiently. However, it is possible to slightly modify the system to store membrane potentials and synaptic weights in a shared memory with a variable boundary. Different network topologies would result in different memory utilization ratios. Using time multiplexing, the system could efficiently handle small networks with dense connectivity and larger ones with sparse connectivity. This would be advantageous for regular network topologies, such as 8-neighbor connectivity. For non-regular connectivity, the scheme for fully connected networks would be used.

ASICs would also be an interesting next step, as the bit slice architecture would be readily amenable to that technology.

6.2 Evaluation of the event-driven implementation

6.2.1 Performance

The performance of the PE could be improved by modifying the implementation of the heap queue as suggested in [Ioannou and Katevenis, 2007]. By using memory banks with one more read and write interfaces, the heap queue could service the dequeue-enqueue operation twice as fast. The throughput of a PE would double as well.

The FPGA implementation of the HSNN presented here yields a maximum clock frequency of 100 MHz. This value is currently limited by the sub-optimal placement of the assemblies of BRAMs used to implement the heap queue. According to [Ioannou and Katevenis, 2007],

an ASIC implementation of a pipelined heap queue can achieve a 350 MHz clock frequency. This would result in a 3.5× improvement on performance for the HSNN.

The number of implemented neurons is currently limited by the amount of on-chip memory. By moving the neuron state memory off-chip and using a bigger device, like the Xilinx XC5VSX240T FPGA, it would be possible to increase the size of the network to approximately 250 kneurons and segment images of 500×500 pixels. The network size could be further increased by adding additional levels to the heap queue using off-chip memories. Each level of the heap queue requires its own read and write interfaces, and a straightforward solution uses a separate memory chip for each one. Every additional heap level requires a memory of size twice that of the previous one. Roughly, to implement a network of 500 kneurons, 1 MB of off-chip memory is necessary to store the extra level of the queue. Approximately 1 Mneurons could be implemented by further adding 2 MB of off-chip memory, and so on.

6.2.2 Future work

To improve the performance of the HSNN, the use of multiple PEs is considered. For example, 9 PEs could be used in parallel to accomplish the segmentation experiment. Each synapse of the neurons would be assigned to a different PE. That way, the performance of the HSNN would be improved by a 9× factor and it would process one event every 7 clock cycles. The number of PEs is arbitrary and does not depend on the actual network topology. However, it is useless to implement more PEs than the number of synapses per neuron. The most important functional blocks required to use several PEs in parallel, the controller and the merger, are already implemented in the current design.

In the HSNN presented here, spikes are processed in the order of their occurrence. But as they interact, neurons quickly synchronize and tend to fire simultaneously. These synchronous events could be processed in parallel using several PEs. The increase in performance is hard to predict, as it depends on the activity of the network. In the segmentation experiment, groups of fewer than 5 neurons are rare and this means that a 5 × improvement of the performance would be realistic with the use of 5 PEs in parallel.

6.3 Comparison of both systems

The performance of both systems depends on different parameters. For the bit-slice system, the performance can be anything between 6.7 Mspikes/s and 11 kspikes/s, depending on

the average number of synchronous neuron groups. With the event-driven system, the performance for a given network topology is fixed. In the image segmentation experiement, with 8 synapses per neurons, the system processes approximately 300 kspikes/s. For the bit-slice system to achieve an average performance of 300 kspikes/s, there would have to be only 22 synchronous groups of neurons, that is, the input would have to have a very simple structure. This dependence on the level of synchrony in the network, in combinaison with the random initialization the the membrane potentials, is what hurts the performence of the bit-slice implementation. In a real signal processing context using complex inputs, the event-driven system will almost always yield a better performance than the bit-slice one.

In the event-driven system, the events are processed at the exact time of their occurrence, with a precision depending on the variables' widths. In the bit-slice, time-driven approach, the spike times are approximated due to the use of a fixed time step. Since the neuron model used is so simple, the added precision in the event-driven approach comes at almost no expense. In fact, it considerably reduces the computational burden of the SNN, since the state of the neurons does not need to be regularly updated. For a network as simple as the one used in the ODLM, an event-driven implementation is a very good solution.

Since the ODLM mostly relies on regular network topologies and uses fixed weight values, the on-the-fly calculation of the weights is a very efficient strategy. A huge amount of memory can be saved compared to the use of a weight matrix as in the bit-slice system. In the signal processing experiments, the calculation is further simplified by the fact that weights are a function of a single variable, the difference of the pixels' level of grey.

The precision in the calculation of the neuron model has a huge impact on processing speed. Experiments have shown that, for the same network setup, it takes 10 times the number of spikes to reach convergence when using the 4-segment piecewise linear approximation than it takes with the more precise LUT-based calculation of the event-driven implementation. The small number of PEs used in the event-driven system allows the use of the much more precise approximation of the model because it does not have to be duplicated a large number of times. With the bit-slice architecture, hardware resources are dedicated to the neuron model for each implemented neuron. It is thus very costly to use a fine approximation of the model. Since the event-driven system uses a very small number of PEs, it is advantageous to spend resources to approximate the neuron model more precisely and benefit from the processing speed boost. If a large number of PEs is used, than a compromise between resource usage and performance must be made.

6.4 Final thoughts

The design choices made for the event-driven implementation fit very well the ODLM algorithm. Between the two systems presented, the second one clearly is a more efficient implementation of the ODLM. Moreover, the event-driven system can be enhanced by using additional PEs to process simultaneous events in parallel. There is however an intrinsic limit to the parallelism achievable while implementing the ODLM algorithm. The algorithm is serial in nature, the events having to be processed in the order of their occurrence. As a result, an increase in the number of PEs will not necessarily bring an increase in performance. The ODLM algorithm is thus not suited for a massively parallel implementation, as demonstrates the poor performance of the bit-slice architecture. To address this, the core algorithm should be modified to make the calculations independent. It should be possible to accomplish this while maintaining the other properties of the ODLM algorithm.

We suggest one way to modify the ODLM algorithm to make it more hardware friendly. Neurons would still be characterized by a phase, but instead of generating spikes, they would iteratively pull other neurons closer in the phase space. During an iteration, the state of a neuron is modified by computing the equivalent of a center of mass, created by the neurons it is connected to. That way, the state of all the neurons can be updated in parallel, based on the state of the network at the previous iteration. An example of image segmentation using this algorithm is shown in figure 6.1. In this experiment, the neurons' state was updated using the following equation

$$s_i \leftarrow s_i + \frac{\sum_j ((s_j - s_i) \times w_{ij})}{\sum_j w_{ij}} \tag{6.1}$$

where s_i is the current state of neuron i, the sums are taken over all neurons j to which neuron i is connected and w_{ij} is the connection weight between neuron i and neuron j.

The algorithm was actually run on a computer, so the state of the neurons are computed serially, but they are only updated at the end of an iteration. The algorithm only uses the information available at the beginning of an iteration, meaning it could be implemented in parallel. As can be seen in figure 6.1, the resulting segmentation is not perfect. In the proposed algorithm, each neuron tries to pull its neighbors closer in the phase space. There is a form of competition between the neurons inside a segment, since there is not mechanism to elect one neuron toward which all the others would move. There should be plenty of ways to address this issue and get a satisfying segmentation. One straightforward, but

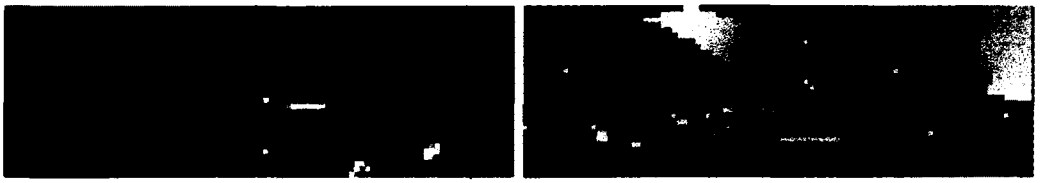

Figure 6.1 Image segmentation experiment using the hardware friendly algorithm. **Top.** Original 89×30 pixel image of an airplane. **Bottom.** Segmented image, after 200 iterations of the described algorithm. The resulting segmentation is not perfect, as a gradient appears in most of the segments. This is due to a form of competition between the neurons inside the same segment.

not theoretically sound, way to remove the unwanted competition is to assign a randomly generated strength value to the neurons. A condition stating that a neuron does not get pulled by another one having a lower strength value could then be used. This would result in the election of one single neuron in each segment that could pull all the other ones.

CHAPTER 7

CONCLUSION (version française)

Dans ce mémoire, nous avons étudié l'implémentation matérielle de réseaux de neurones à décharges. Deux implémentations différentes de l'ODLM ont été présentées: une "bit-slice" et l'autre évènementielle. Il existe plusieurs autres implémentations possibles de réseaux de neurones à décharges sur FPGA, mais celles présentées dans ce travail se démarquent par leur polyvalence. Les systèmes présentés utilisent le "binding" pour accomplir différentes sortes de traitement de signal.

L'implémentation "bit-slice" utilise un grand nombre de petits éléments de calcul. Une grande partie de la mémoire interne du FPGA est utilisée pour stocker les valeurs de poids synaptiques, assurant que le système peut implémenter un réseau avec n'importe quelle topologie. Quand des neurones sont synchronisés, ils déchargent de façon simultanée et leurs décharges sont propagées en parallèle par le système. Le circuit peut ainsi atteindre une vitesse de calculs maximale de 6 millions de décharges par seconde. Un réseau de 648 neurones et 419 904 synapses est implémenté sur un FGPA Virtex-5 XC5VSX50T de Xilinx cadencé à 100 MHz. Le système est en mesure de segmenter des images de 23 par 23 pixels en 2 secondes et fait la comparaison de deux images préalablement segmentées de 90 par 30 pixels en 550 ms.

L'implémentation évènementielle utilise un seul élément de calcul qui consiste en un pipeline de 5 étages. Le système résout la topologie du réseau et calcule les valeurs de poids synaptiques à la volée. Il peut implémenter des réseaux ayant une topologie régulière comme la connectivité en 8 et la connectivité totale. Le système utilise une "structured heap queue", un algorithme de tri développé spécialement pour l'application, pour identifier le prochain neurone à émettre une décharge. Avec cette architecture, on peut implémenter sur le FPGA Virtex-5 un réseau de 65 538 neurones et un maximum de 4 millions de synapses. Le système traite les décharges au rythme de 1 synapse par 7 coups d'horloge et est en mesure de segmenter une image de 406 par 158 pixels en 200 ms. Le FPGA est cadencé à 100 MHz.

Le système évènementiel implémente le réseau de neurones d'une manière plus efficace que le système "bit-slice". Moins de ressources logiques sont utilisées et les performances obtenues dans les tâches de traitement d'images sont meilleures. La résolution de la topolo-

gie et le calcul des poids à la volée sont très bien adaptés à l'implémentation de l'ODLM, qui utilise des poids de valeurs fixes. La quantité de mémoire nécessaire pour implémenter un neurone est ainsi grandement réduite. Par contre, la nature séquentielle de l'algorithme, due au fait que les décharges doivent être traitées dans l'ordre de leur occurrence, n'est pas bien adaptée à une implémentation matérielle massivement parallèle. Pour remédier à cette situation, il faudrait modifier le paradigme utilisé. À la place d'utiliser un réseau de neurones à décharges, il serait possible de représenter les entrées du système par des masses réparties dans l'espace qui s'attirent et se repoussent. Les interactions entre les masses pourraient être entièrement calculées d'une manière parallèle ce qui entraînerait un gain en performance.

CHAPTER 8

CONCLUSION (english version)

In this master thesis, we studied the hardware implementation of spiking neural networks for signal processing. Two different FPGA implementations of the ODLM algorithm were presented: one bit-slice and the other event-driven. Many other hardware spiking neural networks exist, but these implementations stand out by their practical versatility. Since computation is based on binding, a variety of signal processing tasks can be readily accomplished by the HSNNs presented here.

The bit-slice implementation uses a large number of very small processing elements. It uses most of the on-chip memory to store a huge weight matrix to be able to implement SNNs of any topology. When several spikes are emitted simultaneously, the system processes them all in parallel, achieving a maximum processing speed of 6 Mspikes/s. A network of 648 neurons and 419 904 synapses can be implemented on a Xilinx XC5VSX50T FPGA. The bit-slice implementation can segment a small 23×23 pixels image in 2 seconds. It can perform a matching task on two pre-segmented 90×30 images in 550 ms.

The event-driven system uses a single processing element which consists in a 5-stage pipeline. In resolves the topology of the network on-the-fly and calculates the synaptic weights on demand. The system supports networks with regular topology such as 8-neighbor and fully-connected. It uses a structured heap queue, a pipelined sorting algorithm introduced in this work. Using this architecture, a maximum of 65 536 neurons and more than 4 Gsynapses can be implemented the FPGA. The system processes spikes at a rate of 1 spike per 7 clock cycles. It can segment a 406×158 pixel image in 200 ms. Both systems are implemented on a Xilinx XC5VSX50T FPGA clocked at 100 MHz.

The event-driven system implements the neural network in a more efficient way, requiring less logic resources to achieve the same processing power as the bit-slice one. Additionally, the on-the-fly resolution of the topology and weight calculation is more adapted to the ODLM algorithm and reduces the memory used by each implemented neuron. However, the ODLM algorithm is not very well suited for a massively parallel implementation. In the algorithm, the spikes generated by the neurons must be processed in the order of their occurrence, which means that more processing units on the hardware do not necessarily provide more processing speed. To address this issue the algorithm should be modified in

such a way that the computations become independent. One way to do so is to consider the neurons as behaving like masses in the phase space, iteratively pulling other neurons closer to them. In such an algorithm, the state of all the neurons can be updated in parallel based on their state at the previous iteration.

LIST OF REFERENCES

Agís, R., Ros, E., Díaz, J., Carrillo, R. and Ortigosa, E. M. (2007) Hardware event-driven simulation engine for spiking neural networks. *International Journal of Electronics*, volume 94, no. 5, pp. 469–480.

Bergeron, J. (2008) *Reconnaissance accélérée de formes par un réseau optimisé avec neurones à champs récepteurs synchrones*. Masters thesis, Université de Sherbrooke.

Bhagwan, R. and Lin, B. (2000) Fast and scalable priority queue architecture for high-speed network switches. In *19th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pp. 538–547.

Boahen, K. (2000) Point-to-point connectivity between neuromorphic chips using address events. *IEEE Transactions on Circuits and Systems II: Analog*, volume 47, no. 5, pp. 416–434.

Brown, R. (1988) Calendar queues: A fast O (1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, volume 31, no. 10, pp. 1220–1227.

Caron, L.-C., Mailhot, F., Rouat, J. (2011) FPGA implementation of a spiking neural network for pattern matching. In *2011 IEEE International Symposium on Circuits and Systems*, pp. 649–652.

Cheung, K., Schultz, S. R. and Leong, P. H. (2009) A parallel spiking neural network simulator. In *2009 International Conference on Field-Programmable Technology*. pp. 247–254.

Delorme, A. and Thorpe, S. (2003) SpikeNET: an event-driven simulation package for modelling large networks of spiking neurons. *Network: Computation in Neural Systems*, volume 14, no. 4, pp. 613–627.

D'Haene, M., Schrauwen, B., Van Campenhout, J. and Stroobandt, D. (2009) Accelerating event-driven simulation of spiking neurons with multiple synaptic time constants. *Neural computation*, volume 21, no. 4, pp. 1068–1099.

Eckhorn, R., Reitboeck, H., Arndt, M. and Dicke, P. (1989) Feature linking via stimulus-evoked oscillations: experimental results from cat visual cortex and functional implications from a network model. In *1989 International Joint Conference on Neural Networks*, pp. 723–730.

Furber, S. B. and Temple, S. (2007) Neural systems engineering. *Journal of the Royal Society*, volume 4, no. 13, pp. 193–206.

Furber, S. B., Temple, S. and Brown, A. (2006) High-performance computing for systems of spiking neurons. In *AISB'06 workshop on GC5: Architecture of Brain and Mind*, volume 2. pp. 29–36.

Girau, B. and Torres-Huitzil, C. (2007) Massively distributed digital implementation of an integrate-and-fire LEGION network for visual scene segmentation. *Neurocomputing*, volume 70, no. 7-9, pp. 1186–1197.

Hodgkin, A. and Huxley, A. (1952) A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, volume 117, no. 4, p. 500–544.

Hopfield, J. J. and Herz, A. V. M. (1995) Rapid local synchronization of action potentials: Toward computation with coupled integrate-and-fire neurons. *National Academy of Science USA*, volume 92, pp. 6655–6662.

Ioannou, A. and Katevenis, M. G. H. (2007) Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks. *IEEE ACM Transactions on Networking*, volume 15, no. 2, pp. 450–461.

Izhikevich, E. (2004) Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, volume 15, no. 5, pp. 1063–1070.

Jahnke, A., Schönauer, T., Roth, U., Mohraz, K. and Klar, H. (1997) Simulation of spiking neural networks on different hardware platforms. *Lecture Notes in Computer Science*, volume 1327, pp. 1187–1192.

Maass, W. (1997) Networks of spiking neurons: the third generation of neural network models. *Neural Networks*, volume 10, no. 9, pp. 1659–1671.

Mihalas, S. and Niebur, E. (2009) A generalized linear integrate-and-fire neural model produces diverse spiking behaviors. *Neural Computation*, volume 21, no. 3, pp. 704–718.

Milner, P. (1974) A model for visual shape recognition. *Psychological Review*, volume 81, no. 6, pp. 521–535.

Mirollo, R. E. and Strogatz, S. H. (1990) Synchronization of Pulse-Coupled Biological Oscillators. *SIAM Journal on Applied Mathematics*, volume 50, no. 6, p. 1645–1662.

Moon, S., Rexford, J. and Shin, K. (2000) Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on Computers*, volume 49, no. 11, pp. 1215–1227.

Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A. and Veidenbaum, A. V. (2009) A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural networks*, volume 22, no. 5-6, pp. 791–800.

Nischwitz, A. and Glunder, H. (1995) Local lateral inhibition: a key to spike synchronization? *Biological Cybernetics*, volume 73, no. 5, pp. 389–400.

Pearson, M. J., Pipe, A. G., Mitchinson, B., Gurney, K., Melhuish, C., Gilhespy, I. and Nibouche, M. (2007) Implementing Spiking Neural Networks for Real-Time Signal-

Processing and Control Applications: A Model-Validated FPGA Approach. *IEEE Transactions on Neural Networks*, volume 18, no. 5, pp. 1472–1487.

Peskin, C. (1975) *Mathematical aspects of heart physiology - Principles Of Electrophysiology With Special Reference To The Heart*. Courant Institute of Mathematical Sciences, New York University.

Pichevar, R. and Rouat, J. (2007) Monophonic sound source separation with an unsupervised network of spiking neurones. *Neurocomputing*, volume 71, no. 1-3, pp. 109–120.

Pichevar, R., Rouat, J. and Tai, L. (2006) The oscillatory dynamic link matcher for spiking-neuron-based pattern recognition. *Neurocomputing*, volume 69, no. 16-18, pp. 1837–1849.

Rast, A., Yang, S., Khan, M. and Furber, S. B. (2008) Virtual synaptic interconnect using an asynchronous network-on-chip. In *2008 International Joint Conference on Neural Networks*, pp. 2727–2734.

Rönngren, R. and Ayani, R. (1997) A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, volume 7, no. 2, pp. 157–209.

Ros, E., Ortigosa, E., Agis, R., Carrillo, R. and Arnold, M. (2006) Real-time computing platform for spiking neurons (RT-spike). *IEEE transactions on Neural Networks*, volume 17, no. 4, p. 1050–1063.

Roth, U., Jahnke, A. and Klar, H. (1995) Hardware Requirements for spike-processing neural networks. *Lecture Notes in Computer Science*, pp. 720–720.

Schaefer, M., Schoenauer, T., Wolff, C., Hartmann, G., Klar, H. and Rückert, U. (2002) Simulation of spiking neural networks-architectures and implementations. *Neurocomputing*, volume 48, no. May 2001, pp. 647–679.

Schrauwen, B., D'Haene, M., Verstraeten, D. and Campenhout, J. V. (2008) Compact hardware liquid state machines on FPGA for real-time speech recognition. *Neural networks*, volume 21, no. 2-3, pp. 511–523.

Seiffert, U. (2004) Artificial neural networks on massively parallel computer hardware. *Neurocomputing*, volume 57, pp. 135–150.

Serrano-Gotarredona, R., Oster, M., Lichtsteiner, P., Linares-Barranco, A., Paz-Vicente, R., Gomez-Rodriguez, F., Camunas-Mesa, L., Berner, R., Rivas-Perez, M., Delbruck, T., Liu, S.-C., Douglas, R., Hafliger, P., Jimenez-Moreno, G., Civit Ballcels, A., Serrano-Gotarredona, T., Acosta-Jimenez, A. J. and Linares-Barranco, B. (2009) CAVIAR: a 45k neuron, 5M synapse, 12G connects/s AER hardware sensory-processing- learning-actuating system for high-speed visual object recognition and tracking. *IEEE Transactions on Neural Networks*, volume 20, no. 9, pp. 1417–1438.

Strukov, D. B., Snider, G. S., Stewart, D. R. and Williams, R. S. (2008) The missing memristor found. *Nature*, volume 453, no. 7191, pp. 80–83.

Tang, W., Goh, R. and Thng, I. (2005) Ladder queue: An O (1) priority queue structure for large-scale discrete event simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, volume 15, no. 3, pp. 175–204.

Thomas, D. and Luk, W. (2009) FPGA accelerated simulation of biologically plausible spiking neural networks. In *2009 IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, pp. 45–52.

Thompson, C. (1983) The VLSI complexity of sorting. *IEEE Transactions on Computers*, volume 100, no. 12, pp. 1171–1184.

Upegui, A., Penia-Reyes, C. and Sanchez, E. (2003) A functional spiking neuron hardware oriented model. *Lecture notes in computer science*, pp. 136–143.

van Emde Boas, P., Kaas, R. and Zijlstra, E. (1976) Design and implementation of an efficient priority queue. *Theory of Computing Systems*, volume 10, no. 1, pp. 99–127.

Van Vreeswijk, C., Abbott, L. F. and Ermentrout, G. B. (1994) When inhibition not excitation synchronizes neural firing. *Journal of computational neuroscience*, volume 1, no. 4, pp. 313–321.

Vega-Pineda, J., Chacon-Murguia, M. and Camarillo-Cisneros, R. (2006) Synthesis of Pulsed-Coupled Neural Networks in FPGAs for Real-Time Image Segmentation. *2006 IEEE International Joint Conference on Neural Network*, pp. 4051–4055.

Von Der Malsburg, C. (1981) *The correlation theory of brain function*, July 1981. pp. 95–119.

Wang, D. and Terman, D. (1997) Image segmentation based on oscillatory correlation. *Neural Computation*, volume 9, no. 4, pp. 805–836.

Wang, H. and Lin, B. (2007) Pipelined van Emde Boas Tree: Algorithms, Analysis, and Applications. *26th IEEE International Conference on Computer Communications*, pp. 2471–2475.

Watts, L. (1994) Event-driven simulation of networks of spiking neurons. *Advances in neural information processing systems*, pp. 927–927.