Simulation and DSE of Neuromorphic Architectures for RSNNs

Willigers, Stefan L.C.

Award date:
2022

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Parallel Architecture Research Eindhoven (PARsE)

Simulation and DSE of Neuromorphic Architectures for RSNNs

Master's Thesis

Stefan Willigers

Supervision:
Sherif Eissa
Henk Corporaal

Assessment committee:
Sherif Eissa
Henk Corporaal
Federico Corradi
Charlotte Frenkel

August 12, 2022

Contents

Chapter 1

Introduction

Machine learning is a branch of AI that uses algorithms and data to perform cognitive tasks. Recently, machine learning has become a hot topic again. This increase in popularity is due to the increase in computation power and data abundance allowing the accuracy of especially neural networks to improve by leaps and bounds. As a result, neural networks have proven effective in a wide range of applications. Effective applications include fields like computer vision which deals with the analysis of image and video data by computers, or the field of natural language processing which tries to interpret natural languages like English, French, German or Dutch. Deep learning has achieved super-human performance in many of these cognitive tasks.

Currently, most of the work in deep learning is in the field of ANNs (artificial neural networks). ANNs can achieve super-human performance on cognitive tasks, but they suffer from being power-hungry and time-consuming to create and run. When looking at the biological counterpart for AI, the brain, we see that the brain consumes less power and learns information easier than current ANN-based systems. Two avenues of research are studying the brain in the context of AI. Cognitive neuroscience tries to decode how the brain works. And neuromorphic AI, which uses neuroscience concepts to improve the efficiency of computers on cognitive tasks. This work's main area of interest will be SNNs (spiking neural networks), part of neuromorphic AI research. SNNs try to improve on ANNs using several principles inspired by the brain. SNNs are also often called the third generation of neural networks [1].

There is already dedicated hardware for running ANNs. Due to ANNs being so popular, this avenue of research is also intensively researched. This popularity results in the more mature hardware specialized in running applications required by the ANN field. There even are frameworks specializing in DSE for ANNs like ZigZag [2]. However, that maturity does not hold for SNN hardware. The design and implementation of efficient hardware for SNNs are compared to ANNs still in their infancy. Hardware specialized in

simulating SNNs is called neuromorphic hardware.

1.1 Problem statement

In this work, we want to research neuromorphic hardware and the design choices involved. More specifically, the neuromorphic hardware we want to research is multi-core and therefore has a NoC (Network on Chip). Additionally, we want to do DSE (design space exploration) of different SNN applications and mappings. Using a hardware model, we want to learn what trade-offs are and how significantly they affect the hardware design. More specifically, the questions that we want to answer are as follows:

- What is the effect of core size on the throughput and energy of the chip? Smaller cores cause more traffic, but the memories consume less energy.
- How can core computation unit design affect the area, energy, and throughput? Having more functional units allow more throughput. However, more functional units will increase area and power leakage.
- What is the influence of increasing a router's buffer sizes on the throughput and area usage of the hardware? Increasing buffer sizes of the routers may increase the router's throughput but will increase the area usage of the router.
- Does limiting the number of wires between routers greatly influence the throughput of the hardware? Limiting the number of wires can limit the transfer speed between routers. This limited speed will decrease overall throughput.

1.2 Contributions & overview

To enable DSE, we plan to create a simulator that models the behavior of neuromorphic hardware. This simulator will include a cost model that can quantify the area and energy cost of the core and a discrete event-based simulator that can functionally model the hardware, including the congestion of the NoC. Additionally, the simulator will be execution-driven and, therefore, able to predict accuracies. The hardware will be multi-core and use a 2D-mesh as communication fabric, and the core design will be digital and time-multiplexed. Additionally, the core design will use the axonal approach. To do our experiments, we will train our own networks based on the SRNN (spiking recurrent neuronal network) networks of [3]. Then we will use these networks as a benchmark to evaluate the performance of different hardware configurations. The hardware configurations are made in such a way that they can answer the research questions.

The remainder of the work will be as follows: First, we will give a bit of background into SNNs, neuromorphic hardware, and simulators in chapter 2. Then we will investigate the current state of the art and see where we can improve upon it in chapter 3. After researching the state-of-the-art, we will explain the simulator that was built to solve the problem. The explanation of this simulator is given in chapter 4. Finally, we can do our experiments in chapter 5, after which we can discuss the results and the simulator in chapter 6, followed by the conclusion in chapter 7.

Chapter 2

Background

Before looking at related work, we first need to know some background information to understand the related work. Section 2.1 will look into how neuromorphic computing works. After that, we will look into some choices we can make while designing neuromorphic hardware in section 2.2. Finally, we will look at the design choice of implementing an execution-driven versus a trace-driven simulator in section 2.3, which is an important design decision for simulators.

2.1 Neuromorphic computing

To build efficient hardware for SNNs, we first need to know how SNNs actually work. Therefore, we will first look into how biological neurons work and what the interesting properties of biological neurons are in section 2.1.1. We will then see how ANNs work and how they implement some of the ideas of biological neurons in section 2.1.2. Finally, in section 2.1.3 we will look at how SNNs work.

2.1.1 Biological neurons

The neurons in SNNs are based on their biological counterparts. There is a large variety in biological neurons [4]. However, all kinds of neurons work according to the following general concept, see figure 2.1. A biological neuron can be divided into three functionally distinct parts [4]: dendrites, axons, and somas. A biological neuron has dendrites which are the input of neurons. These dendrites receive action potentials. Action potentials are rapid changes in the membrane potential of a neuron. The axons have the opposite function. They transmit action potential and are the output of a neuron. This current is transferred to other neurons' dendrites. A synapse connects the dendrites and axons. The sending neuron is called the presynaptic, and the receiving neuron is postsynaptic. Many dendrites feed into a soma. The soma can, in turn, drive multiple axons. The soma is a processing unit and adds a lot of non-linearity to the neuron [4]. The soma holds

potential. This potential will rise according to the current flowing from the postsynaptic part of the synapse into the soma. The potential rise in the soma is proportional to how "well-connected" the synapse is. This "connectedness" of the synapse can also change based on effects like, for example, LTP (long-term potentiation) [5]. If the potential in the soma reaches some threshold, the neuron will spike. This spike is an action potential traveling down the axon to another postsynaptic neuron. This spike can then cause an increase of potential in postsynaptic neurons, which can subsequently make these postsynaptic neurons spike. Finally, the exact mechanism that generates the spike will also reset the potential of the soma back to some resting potential.

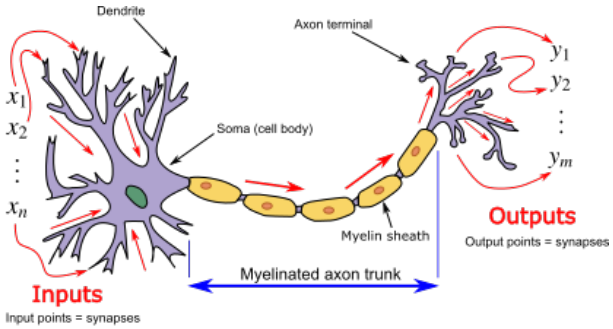


Figure 2.1: An image depicting a biological neuron from [6]. It shows a simplified overview of a neuron with the dendrites, soma and axons. Additionally, it shows how the action potential from the dendrites is fed into the soma and how a spike generated in the soma travels down to the axons.

Biological neurons have several remarkable features that make them interesting [7]:

1. Spikes: In the brain, biological neurons communicate by using spikes. These spikes typically have a duration of 1-2 ms and amplitude of 100 mV [4]. Even if the spikes vary slightly, it does not matter for the information-carrying ability of the spike. The information-carrying ability will stay the same as long the timing stays consistent. This resiliency is because the timing of neurons largely carries the information in the brain and which neurons fire rather than the value of the action potential [4, 7].

2. Sparsity: Most of the time, the neurons in the brain are silent and do not spike [8]. This sparsity is good for efficiency because silent neurons consume less energy by not spending the energy on generating spikes. As the generation and transmission of spikes by cerebral neurons consume a significant part of the energy budget in the cerebral cortex [9], this can result in substantial energy savings.

3. Static suppression: The sensory system is more reactive to changes than static information. An example of this reactivity is the visual system. The visual system is, for example, quick to adapt to changes in color brightness [10].

2.1.2 ANNs

ANNs work differently from their biological counterparts [11]. Figure 2.2 shows a visual reference. Additionally, equation 2.1 shows a simplified equation describing an ANN neuron that can also be used as a reference. Generally, ANNs are functional and do not have a time dimension, unlike their biological counterpart. Exceptions to this functional model are recurrent architectures like RNNs [12], LSTMs [13] and GRUs [14].

$$y = f(\sum_{i=1}^{N} x_i w_i + b) \tag{2.1}$$

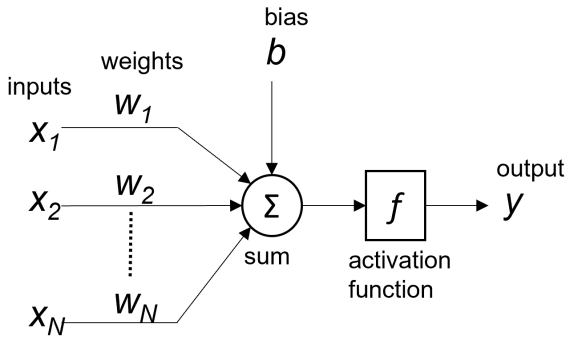


Figure 2.2: Graphical depiction of the ANN neuron model.

A neuron will receive a set of inputs $x_i \in \mathbb{R}$ and produce a single output $y \in \mathbb{R}$. This is done by multiplying each of the inputs x_i with the corresponding weight $w_i \in \mathbb{R}$. The weights w_i control how important each of the inputs is. During training, the weights are changed to minimize the difference between the predicted y and the desired y. In addition to the weights, a bias $b \in \mathbb{R}$ is added, which biases the neuron. Finally, an activation function f maps the summed potential to the output activation. The goal f is to add non-linearity. If this is not done, the neuron will only be able to model linear relations, meaning that the output is only a linear combination of the input. Even if more neurons are added in series, i.e., more layers are added, the output will still be a linear combination of the input.

Their biological counterparts primarily inspire ANNs. The weights model the synapse "connectedness", the activation function models the soma, the inputs x_i model the dendrites, and finally, the output y models the axon.

However, this likeness of ANN neuron model to their biological counterpart does not result in any of the remarkable features of biological neurons mentioned earlier exhibiting themselves. SNNs try to remediate the lack of these features.

2.1.3 SNNs

SNNs are a new third generation of deep learning models that tries to improve on the biological inaccuracies of ANNs, the second generation of neural networks [1]. SNNs are similar to ANNs in that they can model identical topologies. This similarity means that SNNs can model fully connected, convolutional, and recurrent layers similarly to ANNs. However, the difference between the two is in the neuron model. Whereas the neuron model of an ANN is purely functional[1], the neuron models of SNNs always have a state. In the same way that there are many possible activation functions in ANNs, SNNs have many different neuron models. The most ubiquitous and straightforward neuron model is LIF (Leaky Integrate-and-Fire).

A LIF neuron tries to model a biological neuron. The explanation of the LIF neuron model is, therefore, also comparable to the one given in section 2.1.1. An example of how the neuron model works is shown in figure 2.3. A LIF neuron works as follows [7]: Unlike ANN models, the LIF is not purely combinatorial and has a state variable: the potential. This neuron potential resembles the potential of the soma in a biological neuron. If the potential exceeds some threshold, then the neuron will spike, and the potential will be reset to 0[2]. LIF neurons also have an exponential decay, meaning that the potential will decrease exponentially over time.

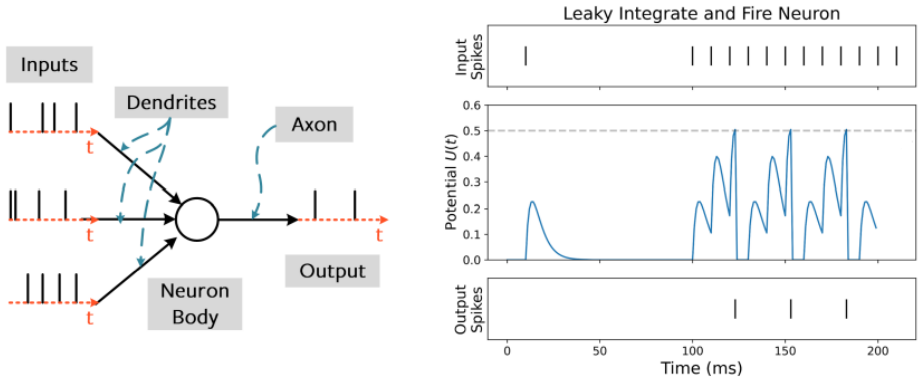


Figure 2.3: Figure from [7] showing the dynamics of a LIF neuron.

Equation 2.2 shows the equation for a LIF neuron model. LIF neurons a persisting potential $u(t)$. The potential of the neuron changes according

[1]Once again, except for RNNs, LSTMs, and GRUs
[2]Actually resting potential, but the resting potential is assumed to be 0

to the input current $I(t)$, resistance R, and decay rate τ_m. The leakage will exponentially decay the potential with a rate of τ_m. The left term on the right side will, henceforth, be called the decay term, and the right term will be, henceforth, the input term. If the potential exceeds some threshold θ, the neuron will fire, which means the potential is reset to 0. Equation 2.3 shows that the amount of current received is proportional to the spikes and weights. Whenever a spike $s_i(t)$ arrives on synapse i at time t, then the potential is increased proportional to the weight w_i.

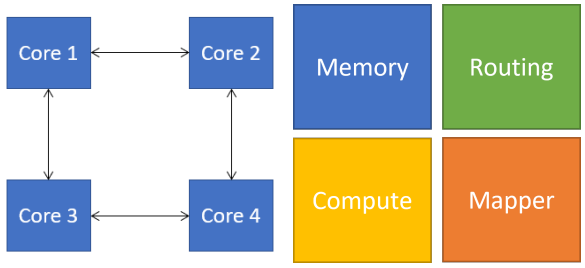
$$\tau_m \frac{du}{dt} = -u(t) + I(t) \cdot R \tag{2.2}$$

$$I(t) = \sum_{i=1}^{N} s_i(t) \cdot w_i \tag{2.3}$$

2.2 Neuromorphic hardware

Unlike CPUs (central processing unit) and GPUs (graphic processing units), neuromorphic hardware does not adhere fully to standard Von Neumann architecture [15, 16, 17, 18]. Instead of having a large shared memory that each PE (processing element) shares, each PE accesses the local memory next to it. This locality of memory reduces the latency and energy overhead caused by expensive memory access and limits the effects of the Von Neumann bottleneck.

Figure 2.4 shows an example model of a neuromorphic multi-core chip. A neuromorphic processor consists of several cores connected by a communication fabric; see figure 2.4a. Figure 2.4b shows a typical digital neuromorphic core. A neuromorphic core consists of multiple functional components: a router or an I/O interface that handles the sending or receiving spikes. A mapper unit that is responsible for mapping outgoing spikes to the target cores' addresses. It also consists of memory to store synapses and neuron states and an ALU unit that can emulate the required neuron operations that need to be supported. Most designs do not exactly follow this scheme of separation into these exact functional components. Some, for example, have multiple separate pieces of memory [17, 19]. However, the memory, compute, routing and mapping are always in there in some form and therefore are the most unifying aspects of a digital neuromorphic core.



(a) Multiple cores connected by a 2D-mesh NoC

(b) Main constituent parts of a core.

Figure 2.4: Simplified overview of multi-core neuromorphic mesh.

2.2.1 Hardware design choices

There are multiple techniques to simulate SNNs; see figure 2.5. An SNN can be simulated either using digital hardware or using analog hardware. One approach is analog hardware where a physical model is constructed that models the dynamics of the SNN neuron model. The other is a digital hardware where the formulas governing the dynamics are simulated. The digital hardware can simulate the decay in two ways: time-driven and event-driven. We will explain more about the difference between these two ways in the section about digital hardware. This choice between time-driven and event-driven is essential as it will significantly affect the design of a neuromorphic chip.

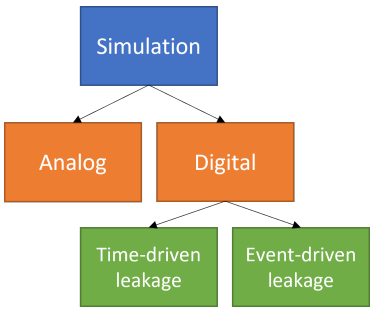


Figure 2.5: A tree showing the difference choices that can be made when simulating an SNN.

Analog hardware

There are two ways to model neurons in the analog domain: analog CMOS [20] and memristor-based. Memristor-based setups are commonly used in a crossbar arrangement to model all-to-all synaptic connections between inputs and outputs. We will explain more about these memristor-based crossbars as they have gained popularity recently.

These crossbars can perform parallel matrix-vector multiplication with very low energy requirements [21]. Figure 2.6 shows such a crossbar. The

general concept is as follows [22]: There is an array of horizontal word lines and vertical bit lines. A memristor then connects these. A memristor is a circuit component that can modulate its conductance [21]. Synapse weights can then be stored in these modulated conductances. So, when a spike arrives on a synapse, the top word line can be set to high. Then an amount of current will flow, proportional to the memristor state, from the word line to the bit line. After which, an ADC (analog-digital converter) can read the output value on the bit line. A CMOS (Complementary metal-oxide-semiconductor) circuit can then use this output value to simulate LIF behavior. This circuit is digital. One bit line for the negative values and one bit line for positive values is used because memristors can not store negative values. Although this analog design may look like the perfect solution, it still has some problems. NVM (non-volatile memory) technologies like memristors can suffer from limited read endurance [23], meaning that the weight value would have to be rewritten after a certain amount of reads. In addition they can suffer from several defects and variabilities [22, 21].

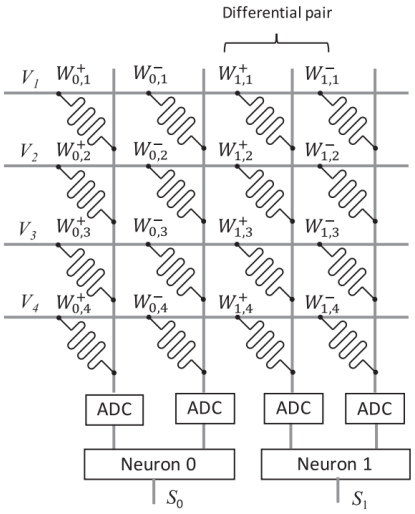


Figure 2.6: A simplified example from [22] of an RRAM-based crossbar.

Digital hardware

Digital simulation emulates the differential equations governing the neurons. For linear models, the input term of section 2.1.3 only changes the potential when a spike arrives and therefore is easy to implement. One can change the potential of the neuron whenever a spike comes in. However, the decay term is a problem. The decay term changes the potential over time. It is, therefore, more difficult to calculate what the potential should be at some later point in time. Luckily the decay is exponential, and the solution for exponential decay is already known. Equation 2.4 shows the solution to the

decay term of equation 2.2. Here, t' is the time the potential needs to be known, and t is the time of the last known potential.

There are two approaches to using this solution: event-driven [19] and time-driven [18]. The time-driven takes constant steps, meaning that $t' - t$ is constant. Equation 2.5 shows the solution to the decay for time-driven simulation. The equation assumes a discretization such that $t' = t + 1$. An advantage of these constant steps is that the term $\exp(-\frac{t'-t}{\tau_m})$ becomes constant and therefore possible to precompute. As a result, the computations are significantly simplified because no exponential calculations and storing the last known potential are needed. The signal to apply the decay can be issued in many ways, in ODIN [17], the signal is manually triggered by an event sent to the core, while for TrueNorth [18], the signal comes from a clock-based 1000 Hz signal.

On the other hand, an event-driven simulation has no such assumption; in event-driven designs, t' can take any value as long as $t' \geqslant t$. Equation 2.6 shows that only the time of equation 2.4 has to be discretized. A consequence of variable time steps is that in addition to $u(t)$, the t also needs to be stored because the step sizes are not constant. However, the amount of computation in an event-driven simulation is proportional to the number of spikes, not the number of timesteps. This proportionality of computation to the number of spikes can be advantageous if the number of spikes is sufficiently sparse spatially and in time.

$$u(t') = u(t) \cdot \exp(-\frac{t' - t}{\tau_m}) \tag{2.4}$$

$$u[t + 1] = u[t] \cdot \exp(-\frac{1}{\tau_m}) \tag{2.5}$$

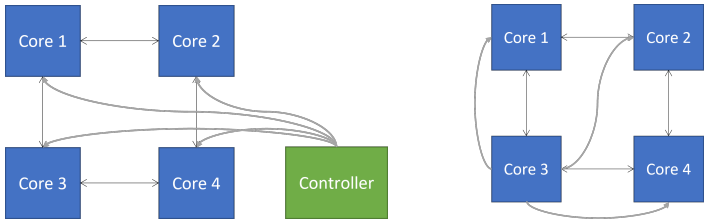
$$u[t'] = u[t] \cdot \exp(-\frac{t' - t}{\tau_m}) \tag{2.6}$$

Synchronization

If neurons spread over multiple cores have to generate spikes and apply decay simultaneously, then each should share the same sense of time. However, this synchronization is not always required. For example, ODIN [17] only has a way to trigger a leakage event manually, and the ODIN chip immediately spikes when the threshold is reached.

Nonetheless if there is synchronization, then there are many approaches to doing it. We will distinguish between two different directions: A central synchronization approach [18] where the control to advance to the next timestep is centered on one component. Fig 2.7a shows an example of central synchronization where a central controller sends a synchronization event over the communication fabric every 1ms to synchronize.

On the other hand, when using distributed sync [16] shown in figure 2.7b, all cores decide together to advance to the next timestep. Several different implementations of distributed synchronization are possible. One implementation requires sending a done signal to a central controller, which, when receiving all done signals, will send a synchronization event to all cores. In this case, the synchronization is not spatially distributed but distributed in terms of control. Another implementation requires each core to send synchronization events to the neighboring cores to signify that the current core is done. An example of this implementation is Loihi [16]. In this case, the synchronization is distributed spatially and in terms of control.



(a) Central sync **(b)** Distributed sync

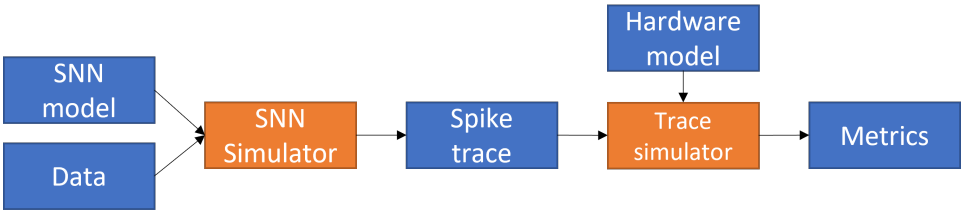
Figure 2.7: Synchronization methods

2.3 Trace-driven vs execution-driven simulators

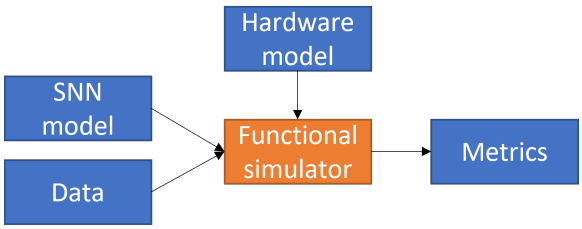
When designing hardware simulators, one can choose between an execution-driven [24, 22] and a trace-driven simulator [25, 23]. Figure 2.8 shows the difference between an execution-driven and trace-driven simulator using a visual diagram. The main essence of the figure is that an execution-driven simulator has a "built-in" SNN simulator. The execution-driven simulator does both the SNN and hardware simulation. In contrast, a trace-driven simulator outsources this to a separate external SNN simulator. Each approach has different pros and cons. Trace-driven simulators have no model of the SNN. Thus spike drops or other events happening out of order can not influence the neuron dynamics and, therefore, the simulator can not predict the model's accuracy. Consequently, trace-driven simulators are less flexible in the hardware details they can model accurately.

Ultimately, this comes down to a trade-off between more abstraction or less abstraction. A more abstract model will have fewer assumptions and details and be applicable in more situations. However, a more detailed model can simulate specific scenarios more accurately. A more abstract model will also be faster to simulate as the simulator must simulate fewer details. Trace-driven simulators are, in this case, the more abstract model. They apply to more simulations, e.g., changing neuron model, and they are faster run, but they will lose some accuracy in simulations, e.g., can not predict model accuracy. Finally, using a non-self-written simulator can save

work, which is another advantage for trace-driven simulators.



(a) Trace-driven simulation



(b) Execution-driven simulation

Figure 2.8: Trace-driven vs execution-driven simulator. Blue boxes show inputs/outputs. Orange boxes show programs.

Chapter 3

Related work

After knowing some of the background information about neuromorphic hardware and neuromorphic hardware simulator we can take a look at what the other works do. Research on neuromorphic hardware is needed such that a hardware model that is relevant to the current SOTA (State Of The Art) can be implemented. Section 3.1 will look into some of the neuromorphic hardware chips. Section 3.2 will look into some of the simulators that are used to simulate neuromorphic hardware. This knowledge is need to to know whether we can find a way to improve or take a different approach. Finally, in section 3.3 we can put together what we learned about simulators and neuromorphic hardware and determine the type of hardware and the kind of simulator we would like to implement.

3.1 Neuromorphic hardware

One of the earlier big neuromorphic processors developed is TrueNorth [18]. It is a neuromorphic processor developed by IBM. It is fully digital and does time-driven SNN simulation. TrueNorth's NoC is an asynchronous 2D-mesh. The core supports multiple neurons by time multiplexing. The synchronization is done centrally by a clock running at 1000 Hz. Therefore, the TrueNorth processor is a prime example of the central synchronization paradigm. Interestingly, even though the processor is digital, it still implements a digital crossbar. The main goals of the TrueNorth processor were to minimize energy consumption, maximize throughput, have a real-time operation, have scalability, be defect tolerant, and have one-to-one equivalence between hardware and software.

ODIN [17] is different in that it only has one core instead of many cores. ODIN can simulate both a LIF neuron and phenomenological models for the Izhikevich neuron model. ODIN does not have synchronization. The closest resemblance to synchronization is a manually triggered leakage event. ODIN neurons immediately trigger if they reach threshold. ODIN also implements a phenomenological model that mimics but not is Izhikevich model. A phe-

nomenological model rather than a differential equation is simulated and, therefore, does not fit either event-driven or time-driven simulation. Unfortunately, the small size of ODIN limits the applicability to anything else than edge computing applications.

Loihi [16] is a neuromorphic processor developed by Intel. It has 128 cores that are each able to simulate 1024 neurons. Like TrueNorth, it is connected by an asynchronous 2D mesh, but unlike TrueNorth, the synchronization is not global but distributed using barrier sync messages. The Loihi paper does not give enough information about the core design to know whether it uses event-driven or time-driven simulation. Unlike TrueNorth and ODIN, Loihi focuses more on having a very flexible neuromorphic processor. Loihi, for example, has a reconfigurable on-chip learning engine. Loihi even keeps track of spike traces, which is essential for some learning algorithms.

NeuroEngine [19], takes a clear event-driven simulation approach. The main contributions are in the proposal of several techniques to increase the simulation efficiency of two-stage neuron models. Two-stage neuron models are a special class of neuron models where instead of the potential increasing instantly on an input spike arriving, the potential increases slowly over time. The authors theorize that this non-instant increase could allow the network to model more complex dynamics and reach higher accuracy. The synchronization scheme is once again not clearly defined in the paper.

The SpiNNaker [26, 27] and SpiNNaker2 [28] are computer clusters. Unlike other neuromorphic processors, this architecture uses a specialized ARM core. The choice of a general-purpose ARM core increases the system's flexibility significantly. The system can simulate a vast range of SNN topologies and neuron models. The SpiNNaker system can run a time-driven [29] simulation. No literature could be found that claimed that it could also do event-driven simulation.

3.2 Neuromorphic hardware simulators

There is already some research in using hardware simulators to evaluate neuromorphic hardware. However, interestingly, there is a significant variety in the methods of these simulators and the modeled hardware.

PyCARL [25] simulates hardware in a trace-driven way primarily focused on crossbar-based architecture. The central concept of this trace-driven approach is to get a trace of spikes from an external SNN simulator like CARLsim [30]. This trace is then simulated on a hardware model to extract metrics. This trace consists of the time and neuron of each spike in the network. PyCARL does acknowledge that hardware details can affect the accuracy of the SNN simulation. To monitor the accuracy impact of the simulated hardware, they added the spike disorder and the inter-spike

interval distortion metrics. Nonetheless, these metrics can only provide estimates of the model's accuracy. Running a dataset on the simulator and reading the accuracy will not work as it is trace-driven. On the other hand, the simulator can extract information like latency, throughput, and energy of the hardware design. These metrics can help estimate congestion of the NoC, which most of the experiments also test.

NeuroXplorer [23] shares some of the same authors as PyCARL. It is, therefore, no surprise that this work improves on its predecessor PyCARL. The most relevant difference is that, compared to PyCARL, which is only compatible with CARLsim, this framework is compatible with many other neuromorphic simulators like Brian [31], and Nest [32]. The model accuracy metrics are the same as PyCARL. Yet, in terms of hardware metrics, this work has more metrics. In addition to the metrics of PyCARL, the model also focuses on metrics monitoring the crossbar like circuit aging, resource utilization, and endurance.

The work [22], henceforth, called the system level simulator, also focuses on crossbar-based architectures. However, in addition to focusing on different aspects of RRAM, the paper also has a different approach to simulating. This work uses an execution-driven simulator, simulating the neuronal dynamics and evaluating accuracy.

NEUTRAMS [24] is another execution-driven simulator. However, this simulator has a decoupling between the functional and timing models. The user can then give the time that a certain operation should take, independent of the functionality. This decoupling means that the user can, for example, configure that a synaptic operation should take a certain number of cycles. Simulators like the earlier mentioned system-level simulator, NeuroXPlorer, and PyCARL do not explicitly show any flexibility in changing the timing parameters of the core independently from functionality. This flexibility in the timing model allows the user to model more scenarios. In addition, the NEUTRAMS simulator also claims to be amiable to changing the execution substrates, e.g., ANNs vs SNNs. All of these options allow the NEUTRAMS simulator to be very flexible.

Although not neuromorphic hardware simulators, Compass [33], NengoLoihi [34], and the Intel lava framework [35] also simulate SNNs. But these simulators mostly emulate execution semantics. Simulating execution semantics can be helpful in some scenarios, like investigating scalability. Still, these simulators do not model any buffers, NoC, or other hardware mechanics and are, therefore, not considered neuromorphic hardware simulators.

3.3 Unexplored approach

A large part of the related work focuses on simulating crossbar-based architectures. Simulators like PyCARL, NeuroXPlorer, the system level simulator, and NEUTRAMS all focus on crossbar-based architectures. However, section 2.2 shows that not all hardware follows this crossbar-based architecture. Moreover, this focus on crossbar-based architectures results in few simulators supporting architectures that use a digital time-multiplexed core like Loihi.

What can also be seen is that most simulators either choose to do a trace-driven simulation or an execution-driven simulation of a crossbar. The trace-driven designs are acceptable. However, they do lose all knowledge of the neuron model being simulated. This means that after the trace is extracted, no design exploration in, for example, quantization of the neuron model can be done. Also, fewer different hardware models can be explored in a detailed way because some hardware models may let spikes drop in sacrifice for more efficient hardware. If a trace-driven simulator is used then the accuracy effects of dropping spikes can not be determined. Thus, an execution-driven simulator will allow the design of both hardware that does and does not drop spikes.

Most of the execution-driven simulators are like the system-level simulator which focuses on the execution-driven simulation of a crossbar with an emphasis on simulating low-level electrical circuits. As a result of focusing on these low-level mechanics, these execution-driven designs are less flexible. For example, it makes it more challenging to run different neuron models. An exception to this rule is, for example, turning LIF into an IF (integrate-and-fire) neuron model by setting leakage to 0. But then, the processor simulates a more complicated neuron model while the processor could use a less complex one. NEUTRAMS does not take this approach and can create a high-level crossbar simulator by adding abstractions in the right places. Yet, NEUTRAMS currently is still quite focused on crossbar-based architectures. Also, NEUTRAMS is a clock-driven simulator, which can be non-ideal in terms of performance.

The direction of a digital architecture-focused simulator that can do an execution-driven simulation focusing on high-level aspects and modelling non-crossbar-based hardware would be interesting to research. Unfortunately, not many simulators can be found that take this direction. If this simulator is made event-driven, it is even possible to have a configurable core timing model like NEUTRAMS. This event-driven simulation will result in a large amount of flexibility.

Chapter 4

Simulator

In the previous section, we learned what the current SOTA is doing regarding hardware and simulators. Now that we know the current SOTA and how we want to improve upon it, we can discuss the simulator that was built. Firstly, in section 4.1 we will look at a quick overview of the constituent programs of the simulator. In section 4.2 we will look at our SNN model. In section 4.3 we will look at the hardware model we want to evaluate and use for the simulator. In section 4.4 we will look at how we can quantify the energy and area cost for each of our hardware components. Finally, in section 4.5, we will review the simulator's software architecture shortly.

4.1 Simulator overview

Firstly, we will make the simulator source code available as open source. The source code can be found at `https://github.com/tetanw/SpikingDSE`. The simulator consists of several separate programs that together form the complete simulator. The implementation of these programs is in either Python or C#. The more compute-intensive programs use C#. The rest uses Python. The simulator produces energy, latency, and area numbers given a hardware model, SNN model, cost model, and input traces. The simulator is execution-driven, but it is also possible to do trace-driven simulation in the future by extending the simulator (see section 4.5). The hardware focus of the simulator will be on digital mesh-based neuromorphic hardware that is multi-core. The SNN focus will be on simulating the SRNN networks of [3]. This work was selected for its good performance on a wide range of benchmarks. Even better, the source code was available for use. The networks are recurrent though so the hardware should be able to simulate recurrent SNN networks.

Figure 4.1 shows an overview of the constituent programs of the simulator. The main program is the DES (discrete-event simulator). It is a discrete-event simulator. Discrete-event simulations model a system as a series of events happening at increasing discrete times. More specifically, it

is a process-based (or process-oriented) discrete-event simulation [36]. The process-based part means that the simulated system is viewed as many communicating parallel processes. This process-based DES design of the simulator is based on SimPy [37], and POOSL [38].

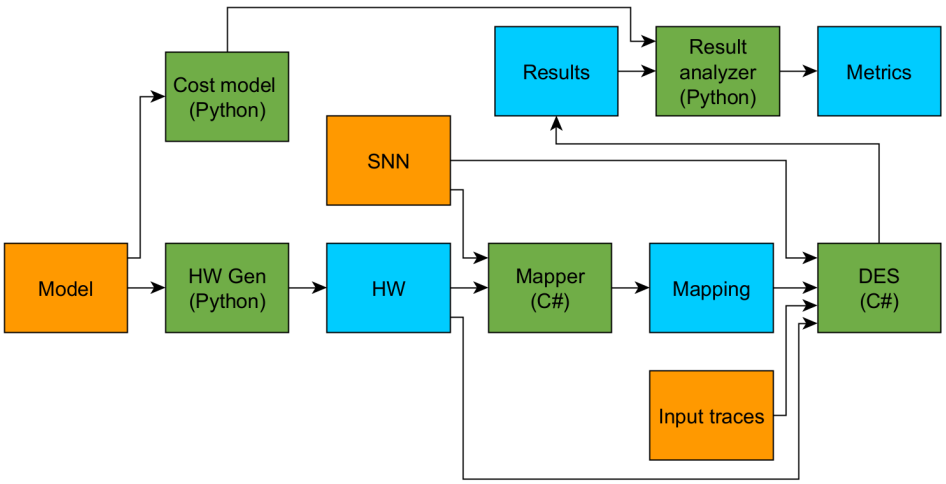


Figure 4.1: Simulator overview. The orange boxes show inputs, the green boxes show the programs, and finally, the blue boxes show intermediate results.

The main function of the DES is to produce results like how many packets a router handled, how many memory accesses each core did, how much time an inference took, etc. Creating these results needs a specification of the hardware model. This HW specification can, for example, contain how many cores the hardware consists of, parameters of each core, locations of cores on the mesh, buffer sizes of routers, etc. But most importantly, it also contains parameters regarding how much time each operation takes. Because the simulator is discrete-event based, the latencies are also parameters. The advantage of having latencies as parameters is that the simulator is more flexible. The disadvantage is that it requires an effort from the user to find realistic values for these parameters. Moreover, these parameter values may not always be readily available.

The other program either produces input files needed for the DES or analyzes the results of the DES. The SNN model describes the SNN architecture that the hardware simulates. Additionally, the simulator needs input traces. These input traces contain which neuron spikes at which timestep in the input layer. The model file is an enriched version of the hardware file. In addition to all the information about the HW, it contains memory size parameters. The HW generator generates the HW specification file based on the model file. The model file makes more assumptions about the hardware. For example, the model files assume that all cores are homogeneous in size, whereas, for the simulator, this does not necessarily have to be the

case. By keeping the HW and model file separate, the DES does not have these assumptions like homogeneity and can, therefore, be more flexible. The mapper creates a mapping of the layers of the SNN to the cores. The mapper is part of the experimental setup and will be explained in section 5.1.2. The cost model calculates the energy cost or area cost of certain operations. For example, given synapse size in bits and the maximum number of synapses that a core may store, it can calculate the synapse memory size. The cost model can then use this memory size to calculate the leakage, dynamic read energy, area, etc. The result analyzer can then use this cost model and the results from the DES to calculate the metrics.

4.2 SNN model

The simulator needs a description of the SNN model. The simulator assumes that an SNN network consists of a series of layers. More specifically, the simulator expects an input layer followed by any number of hidden layers followed by an output layer. Figure 4.2 shows an example of such an SNN. Input layer "i" is followed by a series of hidden layers "h1" and "h2", ending with an output layer "o". Henceforth, a connection from one layer to another will be called a forward connection. A connection between a layer and itself will be called a recurrent connection.

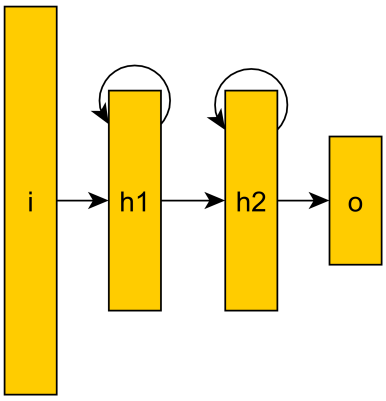


Figure 4.2: Example of what an SNN model should look like. It consists of an input layer "i" followed by recurrent hidden layers "h1" and "h2", and finally, output layer "o".

Each layer should also have a size in terms of the number of neurons in that layer. Additionally, different sets of parameters are required depending on the layer type. There are two layer types supported by this simulator ALIF and LI, both of which are from [3]. An ALIF layer requires passing the weights, decay constants, biases, etc. These parameters need to come from a trained PyTorch model. Section 5.1.1 explains more about this parameter extraction and the origin of these trained models. The output layer of [3] requires a separate layer implementation as it is functionally different

compared to the ALIF layers. Henceforth, the name of this output layer will be the LI layer. Running the LI requires fewer parameters as it is a simpler version of the ALIF layer. It only requires weights, decay constants, and threshold voltage. In figure 4.2, "h1" and "h2" would be ALIF layers, and "o" would be the output layer of [3] as these are the only hidden and output layer types available.

The simulator can also split layers into multiple parts if a core can not fully fit a layer due to constraints. The word layer, in this simulator, actually means a subset of a layer's neurons which can be the full layer but does not have to be. So, a layer part. The layer splitting is a complicating factor in the design of the hardware. If the layer is recurrent and is split into multiple parts located on different cores, then spikes need to be sent between cores to communicate the recurrent spikes. Likewise, the layer before the splitted layer must send the spikes to multiple destinations.

4.3 Hardware model

Figure 4.3 shows a general model of the hardware. As mentioned earlier, the hardware will focus on simulating mesh-based chips. The number of slots to fit cores or controllers in the width and height of the mesh is configurable. In the case of figure 4.3 the hardware model is configured with a width of 2 slots and a height of 2 slots. In addition, the hardware model will always have the controller on the bottom left slot of the mesh, with cores filling the remaining slots.

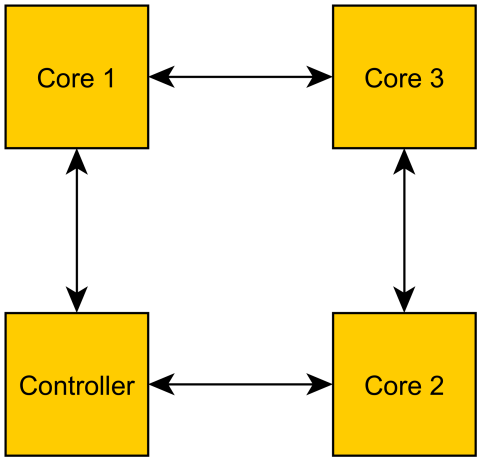


Figure 4.3: An example of a possible configuration of the hardware model. The figure shows a mesh of 2 by 2 with a controller on the bottom-left and cores all around connected by links.

The remainder of this section will discuss how the synchronization (section 4.3.4) works and discuss the three big component parts of the hardware

model: the core (section 4.3.1), the controller (section 4.3.2), and the mesh (section 4.3.3).

4.3.1 Core

Unlike TrueNorth and Loihi, the core design for this simulator uses the concept of layers. Although, especially Loihi and DYNAPs [15] try to limit the memory size on the chip needed to store the connectivity. We also chose to limit the memory size. We chose a layer-based system because it is closest to how the SRNN model of [3] is trained. The implementation of this work does not allow connection between all neurons. Like ANNs, it limits connectivity by grouping neurons in layers and limiting layers to be connected in an acyclic way.

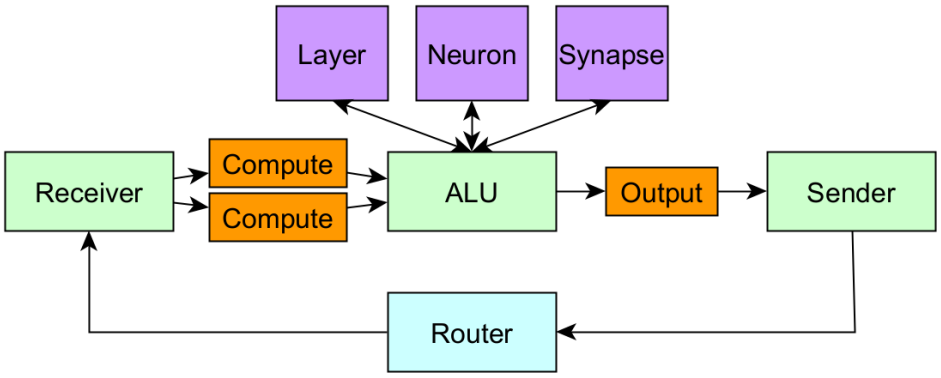


Figure 4.4: Simplified overview of core. Green shows the three main components. Orange shows the buffers. Purple the memories. And light-blue the router.

Figure 4.4 shows an overview of the core's hardware model. The core has three main components: the receiver, ALU, sender, and two buffers: the compute and output. The receiver is responsible for receiving packets from the mesh and handling them correctly. There are two compute buffers. One buffer for the current timestep's spikes and one buffer for the next timestep's spikes. The ALU reads the current timestep's buffer, and the next timestep's buffer is filled with spikes generated by other cores' spikes for the next timestep. Therefore two buffers are needed. If the receiver receives a spike packet, it should be queued in the compute buffer of the next timestep. If the receiver receives a sync packet, it should signal the ALU to start synchronizing the compute buffer of the next timestep. The sender's job is to wait for packets in the output buffer and send them to the router.

The ALU is the most complicated part of the core. When a synchronization signal comes in from the receiver, the ALU will read spike events until the buffer is empty. Each spike packet will result in the core updat-

ing the potentials of the relevant layer. After all, spikes are processed, the ALU must synchronize all neurons. This synchronization includes applying leakage and checking whether the threshold has been reached, among other things. Connected to the ALU are the layer, neuron, and synapse memories. These memories must be written to and read for the ALU to do its job. The layer memory stores information about where the neurons and synapses are stored, network parameters, connectivity data, etc. The neuron memory stores the potential and other neuron parameters. The synapse memory stores the synapse weights. Section 4.4.3 describes what the memory looks like in-depth.

The core has to update the potentials of all neurons when a spike is received. The TrueNorth paper [18] documents two approaches: the dendritic and axonal approaches. Figure 4.5 shows these two approaches. The dendritic approach gets the neuron state from memory once and then integrates all input synapses. The axonal approach sees every input spike as an event that updates neuron potentials.

The advantages and disadvantages of each approach are as follows. Firstly, the dendritic approach requires that all spikes need to be stored before synchronizing, whereas the axonal does not. On the other hand, this means that the axonal approach can update as the spikes come in, which can be an advantage for the axonal approach in certain circumstances.

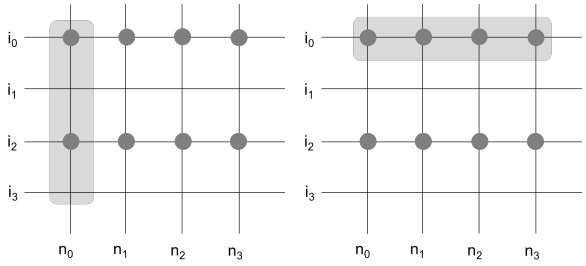
The dendritic approach only requires getting the neuron state as often as there are neurons in a layer, whereas the axonal approach requires more neuron state accesses. Namely, the number of neuron state accesses depends on the number of neurons in a layer and then more depending on the number of spikes. However, this is eased by not having to fetch the whole neuron state while integrating but only the neuron potential. On the other hand, the dendritic approach must fetch the spiking state of the same input neuron multiple times. So, the axonal approach results in more neuron state accesses and the dendritic approach in more spiking state accesses. Finally, the axonal approach will comparatively perform better in sparse scenarios as the number of fetches scales with the number of spikes.

The axonal approach is advantageous in terms of throughput. The dendritic approach has constant working time[1]. In contrast, the amount of time for the axonal approach is more proportional to the number of spikes. So once again, the axonal approach will likely work better in case of high sparsity. Therefore, this work uses the axonal approach because the benchmarks can have relatively high sparsity.

ALU

The ALU will integrate the spikes followed by synchronizing the neurons. Parallelism is a configurable parameter that denotes the number of neurons

[1] this does not necessarily have to be the case



(a) Dendritic approach **(b)** Axonal approach

Figure 4.5: Integration approaches. The vertical axis show the input neurons. The horizontal axis the neurons of the layer to be updated. The dots on the intersection show waiting spikes. The image shows that for the dendritic approach you iterate over all input connection on each neuron whereas for the axonal approach you update all neurons on a layer on receiving a spike.

during synchronization and the number of neurons during integration that is updated simultaneously. For example, when the parallelism is 4, then the ALU will perform four synaptic operations at the same time and synchronize four neurons at the same time.

One of the most important design decisions for the ALU is how many functional units of each kind are needed. The more functional unit can increase the throughput but take up more area and contribute to leakage. The two processes that the ALU needs to perform are integration and synchronization. The most complicated of these is synchronization. Figure 4.6 shows a dataflow graph for the synchronization of the neurons.

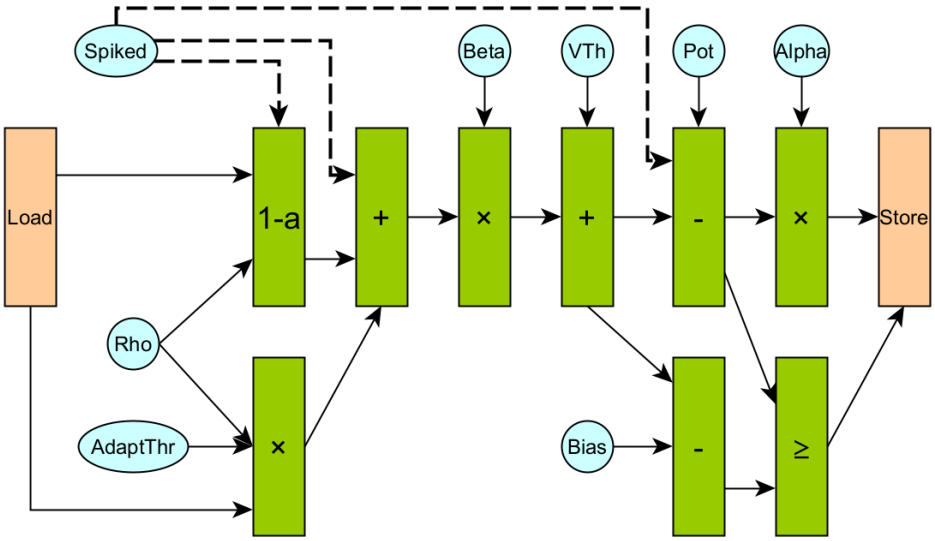


Figure 4.6: Dataflow for synchronization of a neuron. The green boxes show the operations. The blue circles show the variables used in the operations. The dotted line denotes that the operation will only happen when `Spiked` is true.

The latency of synchronizing a neuron and integrating a synapse are

configurable in the simulator. The user can change two parameters: the latency and the initiation interval. The latency describes the time it takes for a value to go from load to store. The initiation interval is the amount of time between operations, i.e., inverse of the throughput. Three designs are possible. A fully serial design where each operation happens in series. A parallel design that still is serial but that will do as much in parallel as possible. The latency will be the same for both the serial and parallel design. Finally, there is a pipeline design. This design schedules the operation in a pipeline. This design will have about the same latency as fully parallel but have a lower initiation interval.

Comparatively, the integration of a spike is easy. Figure 4.7 shows the data flow for that process. Here not much design is possible. It is assumed that the integration uses the same functional units as the synchronization process.

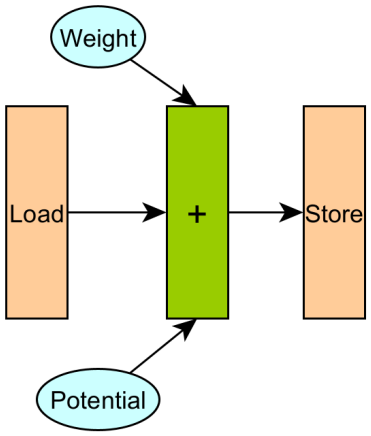


Figure 4.7: Dataflow for integrating a spike for an ALIF neuron. The green boxes show the operations. The blue circles show the variables used in the operations.

4.3.2 Controller

The controller has multiple responsibilities. Firstly, it handles the input layer. Meaning that from the controller the input spikes are sent to the first hidden layer. Secondly, the controller handles the output layer. Meaning it also receives the spikes from the final hidden layer. Layers mapped to the core are assumed to take no time and energy. This also implicates that the input and output layer consume no energy. Finally, the controller is responsible for orchestrating the synchronization. This explained in more detail in section 4.3.4. The controller has no configurable parameters regarding latencies for receiving spikes or regarding synchronization. These parameters can be added in future works to make the simulator more accurate.

4.3.3 Mesh

The mesh consists of many connected routers. The simulator uses a packet-switched router with input and output buffers. Most routers are virtual-channel based. However, this combination of input and output buffers allows for the evaluation of the difference between input buffering and output buffering. Additionally, the model chosen by the simulator is simpler to implement.

Figure 4.8 shows an overview of the router design used in this simulator. The green show boxes show processes, and the orange boxes show buffers. The inputs are responsible for receiving the transmitting packet. The input buffers will store the packets after being received. The outputs are responsible for sending the packets onto the next router. More precisely, they read a packet from the output buffer and send it to an input buffer of another router. The switch is responsible for moving a packet from an input buffer to the correct output buffer.

The routing policy is XY-routing. XY-routing is required to prevent deadlocks due to cyclic dependencies when routing packets. The router also uses round-robin arbitration to guarantee that one direction does not starve all other directions. The round-robin arbitration works as follows: it stores the direction in which the switch switched the last packet. Then, when a new packet is available for routing, the direction after the last switched direction is checked first.

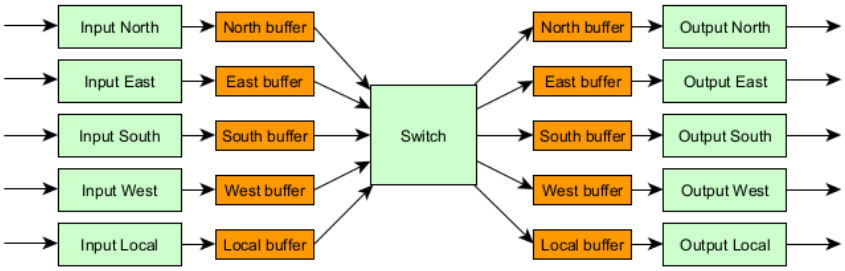


Figure 4.8: Model of an XY-Router as used in the simulator. Orange boxes show buffers. Green boxes show processes.

Figure 4.9 shows a diagram depicting the latencies that a packet encounters when travelling through a switch. First, it will encounter latency on the input. This latency is equal to the transfer latency parameter if it is non-local input and the input transfer latency if not. The reasoning behind this differentiation is that local connections may be faster because the local connection is closer to the core. The same latencies also hold for the output link. If the router switch is already busy or the output is busy, the packet may have to wait in the buffer. This waiting is the congestion. Then comes the switching delay. The simulator assumes that the delay in routing, arbi-

tration, and traversal can be summarisable into one number, the switching delay.

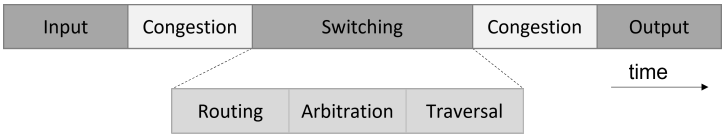


Figure 4.9: Diagram of the latencies that a packet can encounter when traversing a router.

4.3.4 Synchronization

This hardware model uses a distributed synchronization approach. However, it is quite an unusual variant as it uses the controller to decide when to continue instead of all cores deciding together.

Figure 4.10 shows how synchronizing two cores in practice. First, the controller sends the spike events from the input layer to the first hidden layer in each step. Then the controller sends a synchronization event that prompts the cores to start consuming the spikes in the compute buffer. After integrating the spikes in the compute buffer (see section 4.3.1), the core synchronize each neuron. After completion of the synchronization of all neurons, each core will report that it is ready to the controller. A new synchronization event will be issued if the controller has a message from each relevant core that it has completed synchronization. A relevant core is a core that has mapped layers. If there are no layers on a core, then it is not involved in synchronization.

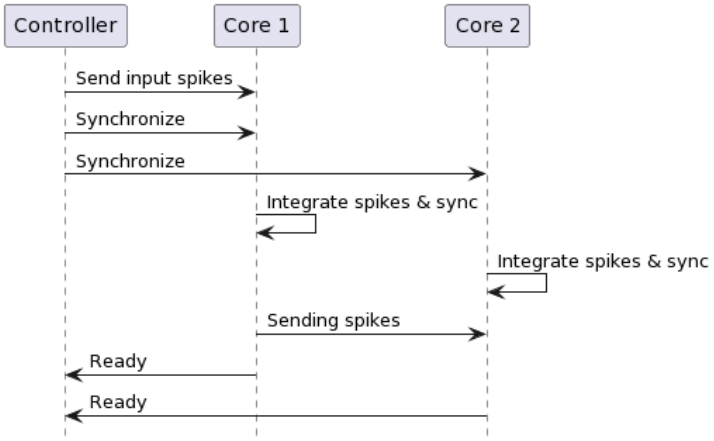


Figure 4.10: The sequence of action that the hardware goes through when doing one timestep. This is an example of a hypothetical hardware with one controller and two cores.

This scheme, however, has a problem. The core should only report that it has completed synchronizing after the spiking events generated during the synchronization have arrived at their destination cores. However, because

of the packet-switched mesh, a core can not know when a spike has arrived. As a result, it is possible that a core already got a synchronization signal before the spike from that timestep could arrive at the compute buffer. Thus there is a race condition. This race condition influences the accuracy of the simulation. This race condition should be relatively rare, so this defect is allowed.

4.4 Cost model

The cost model assumes that the hardware will run at 100 MHz with a core voltage of 1.1 V, a temperature of 25 °C, and that a 40 nm fabrication process is used. The cost model also assumes some level of power and clock gating. As a result, if a core does not have any layers, then it is assumed that the core does not contribute to energy consumption. Additionally, the idle energy consumption of functional units is assumed to be zero.

In section 4.4.1 and section 4.4.2 we will give the memory and buffers an area and energy cost. Section 4.4.3 will show the memory layout. After this section we should know the depth and width of each memory and how each parameter is laid out in the memories. In section 4.4.4 and section 4.4.5 we will look at energies and area for the core and the router. Finally, we will look how the earlier individual energy and area consumption sum up to a model for the whole chip in section 4.4.6.

4.4.1 Memories

Numbers of a commercial 40 nm SRAM memory library are used to find good memory models for the simulator. The version of memory used has a mux factor of 4 and uses the dense memory type. The simulator requires a model for the area, static power, and dynamic energy. Although the value of these characteristics can depend on many variables, only two will be considered the size of the memory in bits S and the width of the memory in bits W.

The area usage of the memory logically depends mostly on the number of the memory capacity in bits. If the memory is small, then the area usage will be mostly dominated by the peripheral circuitry. Equation 4.1 shows the formula used to determine a memory's area. This equation model has an R-squared[2] value of 0.9986 Thus, it is quite a good fit.

$$A_{mem} = (0.4586 \cdot S + 12652) \cdot 10^{-6} \quad [\text{mm}^2] \tag{4.1}$$

[2]R-squared is a value denoting which proportion of the variance of the data is explained by the model. So, it measures how well a model fits the data. A value of 1 is a perfect explanation of variance.

Similarly, the leakage also linearly depends on the capacity of the memory. Equation 4.2 shows the amount of leakage for a certain memory size S. The fit for this equation has an R-squared value of 0.9968, so it is also an accurate fit.

$$P_l = (8 \cdot 10^{-5} \cdot S + 1.822) \cdot 1.1 \cdot 10^{-6} \quad [\text{W}] \tag{4.2}$$

Finally, the dynamic read and write energy are more complicated as they also depend on the width of the memory. Equation 4.3 and equation 4.4 show the write and read energy respectively. The W parameter signifies the width of the memory in bits. Do keep in mind that for the neuron and synapse memory the width is not the size of a neuron and a synapse itself as described in section 4.4.3, but instead need to still be multiplied by the parallelism P. So, the hardware model assumes that an increase in parallelism on the ALU will also increase the width of memory access. The fitting for this equation was done using a Moore-Penrose inverse, giving a least-squares solution. Overall, the error between data and model is less than 20% for both the read and write energy. The model even becomes more accurate with an increase in memory size.

$$E_w = (3.32 \cdot 10^{-5} S + 0.20 \cdot W + 3.71) \cdot 10^{-12} \quad [\text{J}] \tag{4.3}$$

$$E_r = (4.68 \cdot 10^{-5} S + 0.31 \cdot W + 3.23) \cdot 10^{-12} \quad [\text{J}] \tag{4.4}$$

Equation 4.5 shows the total energy consumption of the memory for one inference, where N_r is the number of rows read from memory, N_w is the number of rows written to memory, and t amount of time that the inference took.

$$E_{mem} = E_w \cdot N_w + E_r \cdot N_r + P_l \cdot t \quad [\text{J}] \tag{4.5}$$

4.4.2 Buffers

The hardware model assumes that both the compute and output buffer are FIFO buffers. Each of these buffers will block if it is full. The cost model will also assume that the FIFO buffers are implemented using SRAM. In addition, the cost model assumes that dequeuing a value from the buffer only requires a memory read, and enqueueing a value only requires a memory write. Counters are typically required to store the next available space to read and write, especially in the case of ring buffers. The cost model assumes that these are implemented as registers. Thus, they do not cause extra rows to be read and written to the SRAM memory. The cost model also assumes that these registers' read and write energies are also zero.

4.4.3 Memory layout

Section 4.3.1 explained how the core works functionally but did not yet describe how the memory in the buffers and the memories are laid out. An explanation of different parameters of the cost model is needed first. Table 4.1 describes these parameters.

Parameter	Description
N_{neuron}	Number of neurons on a core
N_{syn}	Number of synapses on a core
N_{layer}	Number of layers on a core
N_{split}	Number of times a layer can be split
$N_{w,x}$	Number of slots of the mesh in the width
$N_{w,y}$	Number of slots of the mesh in the height
N_{fanin}	Maximum fan-in synapses
N_{types}	Number of different packet types
P	Numbers of neurons/synapses executed concurrently

Table 4.1: The parameters used in memory layout model

Mesh packets

To dimension the router's buffers, the size of the different type of packets need to be known. The system uses three kinds of packets: the spike packet, the sync packet, and the ready packet. Figure 4.11 shows the memory layout for each packet type. The packet should always denote the destination core of the packet. The packet also holds the packet type. This way, the core or controller will know which kind of packet was received. The largest packet kind should dimension the input and output buffer width for the router. For this system, this will always be the spike packet. The size of the largest packet, i.e., the spike packet, will be called S_{packet}. The depth of a router's buffers is a configurable parameter.

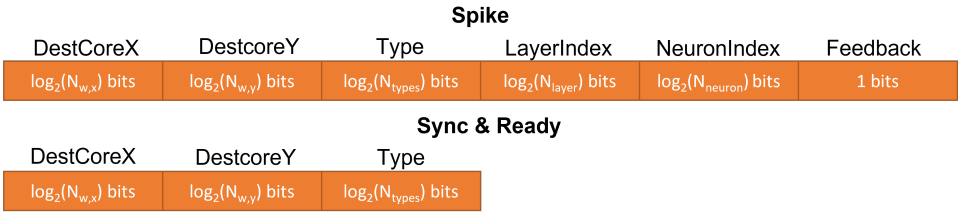


Figure 4.11: Memory layout of all mesh packet types.

Core buffers

Figure 4.12 shows the memory layout of the core's buffers. The compute buffer will hold the spikes until a synchronization event arrives. It must therefore be able to store the information regarding a spike. For that, it needs the layer that received the spike, the neuron that spikes in the previous layer, and whether it is a feedback spike. The compute buffer should be able to hold two times the maximum number of spikes according to the fan-in of the core, because there are actually two buffers. It should contain the spikes of the current timestep, and it should hold the spikes of the next timestep already being received. So, the the compute buffer should be able to hold $2 \cdot N_{fanin}$ items. The output buffer will store the packets ready to be sent on the mesh and, therefore, should have a width equal to the packet size. The depth is configurable.

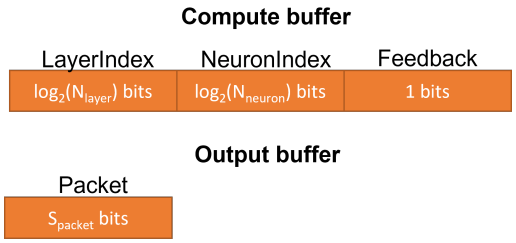


Figure 4.12: Memory layout of the core's buffers. S_{packet} is the size of a packet.

Core memory

Figure 4.13 shows the memory layout of the neurons, layers, and synapses. The figure shows the names of the parameters corresponding to a certain piece of memory and the number of bits. The neuron, synapse, and layer should contain all parameters required to simulate a neuron, which will be stored in the `NeuronState`, `SynapseState`, `LayerState` respectively. Section 5.1.5 will contain more details on what these states contain for the experiments. The `NeuronOffset` is important when a layer is split. For example, if a layer has a size of 32 neurons and is split into two parts of 16, then the second part will have a layer offset of 16 neurons. Of course, the `size` corresponds to the size of the layer mapped onto the core. In our earlier example, both parts would have a size of 16 neurons. The `NeuronStart`, `ForwardStart`, and `RecurrentStart` denote where in the memories the start address of the neuron and the start of the forward and recurrent synapse for that layer are. Finally, the layer also needs to store the fanout for the layers. This fanout requires storing the coordinates of the cores and the layer address on the destination core. The core requires two rows of fanouts, one for the forward connections and one for recurrent connections.

The number of rows in the synapse, neuron, and layer memory will be N_{syn}, N_{neuron}, and N_{layer} respectively. The compute's parallelism will affect the width and number of rows of the synapse memory and the neuron memory. For example, if the parallelism is two, the neuron memory will be twice as wide and half as deep. However, parallelism does not influence the layer memory.

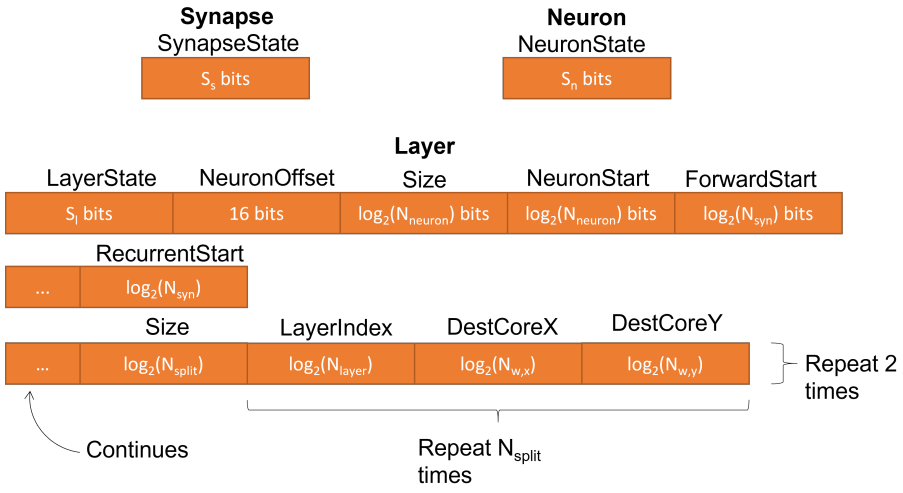


Figure 4.13: Memory layout of the neuron, synapse, and layer memory. S_s, S_n, and S_l describe the synapse, neuron and layer state sizes. These are described in section 5.1.5.

4.4.4 Core

The energy and area numbers for the ALU on the core are based on the work of [39]. The work documents leakage power, area, and dynamic energy for several different types of functional units. Specifically, the 10 ns number should be used as the hardware model assumes 100 MHz.

Equation 4.6 shows the are usage of the the ALU. $\mathbb{F}$ denotes the set of all functional units. $N(f)$ denotes the number of functional units of type f on the core. $N(f)$ is a model parameter as this depends on how the ALU is designed (see 4.3.1). $A(f)$ denotes the area consumption of functional unit f. This area consumption comes from [39].

$$A_{alu} = \sum_{f \in \mathbb{F}} A(f) \cdot N(f) \quad [\text{mm}^2] \tag{4.6}$$

Equation 4.7 shows the energy consumption of the ALU. The first term is the dynamic energy consumption of the ALU and the second the leakage of the ALU. $E(f)$ is the dynamic energy consumption of the functional unit according to [39]. And the $N_O(f)$ is the number of operation of that type performed on that functional unit. This parameter is a result of the DES.

t is the amount of time the inference took. Finally, $P_l(f)$ is the leakage corresponding to the functional unit, also from [39].

$$E_{alu} = \sum_{f \in \mathbb{F}} E(f) \cdot N_O(f) + t \cdot \sum_{f \in \mathbb{F}} N(f) \cdot P_l(f) \quad [\text{J}] \tag{4.7}$$

Finally, equation 4.8 and 4.9 show the the area and energy consumption of a whole core respectively.

$$A_{core} = A_{alu} + A_{neuron} + A_{syn} + A_{layer} + A_{compute} + A_{output} \quad [\text{mm}^2] \tag{4.8}$$

$$E_{core} = E_{alu} + E_{neuron} + E_{syn} + E_{layer} + E_{compute} + E_{output} \quad [\text{J}] \tag{4.9}$$

4.4.5 Routers

Each router will also have an energy and area consumption. The area that the switch takes up is difficult to quantify and will therefore not be considered. Therefore only the area of all memories is used. Equation 4.10 shows this equation. $A_{r,input}$ and $A_{r,output}$ are the areas of the input and output buffers, respectively.

$$A_{router} = 5 \cdot A_{r,input} + 5 \cdot A_{r,output} \quad [\text{mm}^2] \tag{4.10}$$

The energy consumption of the router is based on [40]. The router design assumed for the work assumes a virtual-channel router which is different from this work that assumes a router that assumes a router with an input and output buffer. However, the energies should not be too different. Additionally, the numbers of this work are for a 130 nm process while the model of the simulator assumes a 40 nm. Therefore a correction needs to be applied. This correction will only correct for the difference in core voltages. The work does not mention the core voltage used for the chip. So it is assumed that a core voltage of 1.2 V was used. This means that all numbers needs to be corrected by a factor: $C_f = \frac{1.1^2}{1.2^2}$.

Equation 4.11 shows a modified version of equation 2 of [40]. The equation of [40] assumes that the amount of hops per packet are counted. However, for the simulator this is not feasible. Therefore, the simulator counts the the number of hops per router N_h and the number of packets switches N_s. This is enough to make an equivalent equation. The length of the links L are modelled by equation 4.12. This equation assumes that the router, the ALU, and the memory, among others fit perfectly together. Realistically, this is not the case, but this will still give a good indication.

$$E_{router} = ((1.37 + 0.12L)N_h + 0.98N_s) \cdot C_f \cdot 10^{-12} \quad [\text{J}] \tag{4.11}$$

$$L = \sqrt{A_{router} + A_{core}} \quad [\text{mm}] \tag{4.12}$$

4.4.6 Chip total

Finally, the total area and energy consumption of chip are described by equation 4.13 and 4.14 respectively. $\mathbb{R}$ is the set of all routers, and $\mathbb{C}$ is the set of all active cores. An active core is a core with at least one layer mapped to it. The controller's slot will not contribute to the area of the chip. Additionally, the energy consumption of the controller is not considered.

$$A_{chip} = (N_{w,x} \cdot N_{w,y} - 1) \cdot (A_{core} + A_{router}) \quad [\text{mm}^2] \tag{4.13}$$

$$E_{chip} = \sum_{c \in \mathbb{C}} E_{core,c} + \sum_{r \in \mathbb{R}} E_{router,r} \quad [\text{J}] \tag{4.14}$$

4.5 Software architecture

Since the simulator is a research tool, it is written in such a way that it is extensible. Figure 4.14 shows the decomposition that enables this flexibility[3]. For example, a user can implement custom hidden layers. This new custom implementation could be a neuron model with different mechanics. Or interestingly, a custom implementation of a hidden layer could use a trace file instead of computing the numbers on the fly. This way, the simulator can become trace-driven. The figure shows that ALIF and LI are implementations of this hidden layer class. A hidden layer implementation requires a description of how it syncs, how it integrates, and whether it is recurrent. In addition, the layer requires the size of the layer in neurons and the input size. The input size is the size of the previous layer.

In addition, a user can change the communication fabric as well. A new class that extends `Comm` needs to be created to make a custom implementation. For now, only a mesh fabric is implemented. But this could also be extended to cover a bus or fat-tree architecture. Another interesting approach could be implementing a communication model where the simulators model a mesh by just delaying the packets traveling over the mesh based on the number of hops. This model would not model congestion but would be simpler to model and, therefore, faster simulation.

The reason that the program is written as a system of programs and not just one program is also to facilitate extensibility. An advantage of this is that, for example, if the mapper is not well-liked, a user can write a new mapper in a different programming language to generate the mapping files.

[3]Not all methods and fields are shown for the sake of brevity

Finally, the simulator has a reporting system. Figure 4.1 showed that the DES simulator writes its results to an output file. The results file is a CSV file. The results file contains a concatenation of the strings where each string represents a column. These strings come from the `Report` method on all `Core` instances and the `Comm` instance. For the `Mesh` instance, the reporting consists of the result of the `Report` method of all routers. Adding new columns is as simple as adding an extra column in the report method of the core or router.

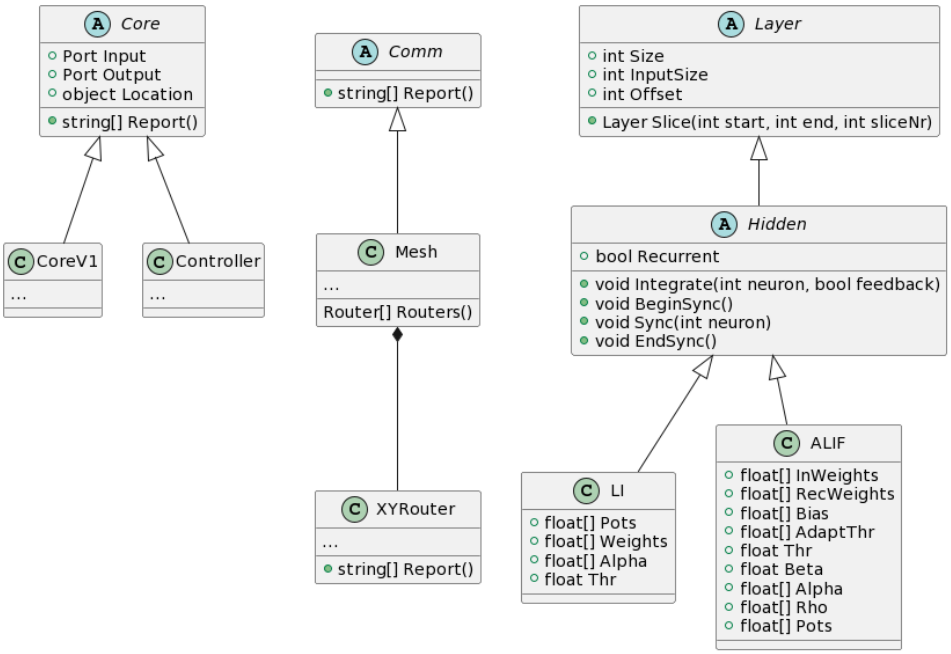


Figure 4.14: Simplified description of the DES's `Core`, `Comm` and `Layer` abstractions. It also shows how the current core, controller, mesh and layer classes use the abstraction.

Chapter 5

Experiments

Now that we know how the simulator works, we can do some experiments. But before we start the experiments, we first need to talk about the experimental setup in section 5.1. After that, we need to determine which metrics will be used to evaluate the experiments in section 5.2. Finally, in section 5.3 we can start doing the experiments to answer our research question.

5.1 Experimental setup

There is still some explanation left before we do our results. This section will focus on explaining more about the experimental setup. Section 5.1.1 will talk about the networks that were trained to evaluate our hardware and with which we will do our experiments. Section 5.1.2 discusses the mapper made to map the networks on the hardware for our experiments. Finally, we need to show what values were chosen for most of the parameters of our hardware model. Section 5.1.3, section 5.1.4, and section 5.1.5 will show the configurations for the ALU, routing and memory, respectively.

5.1.1 Networks

The experiments will use four datasets during its DSE: SDH, SSC, S-MNIST, and PS-MNIST. These are a subset of the datasets used by [3]. Each dataset will have two networks with different configurations. One network per dataset will have the same layer sizes as the original work. The other network will have larger layer sizes (SHD, S-MNIST, PS-MNIST) or smaller layer sizes (SSC). The networks were chosen so that they are fully connected and follow the paradigm of a series of inputs followed by an output value. The ECG dataset was therefore not selected as it requires output prediction simultaneously with inputting. The convolutional neural networks were also not selected as they need more complicated mapping. There are also a few differences in SNN execution.

Compared to [3] there are a few changes, the most major being that the transfer of spikes from one layer to another takes one timestep and is

not immediate[1]. Thus we cannot use exactly the same networks as the original paper. New ones have to be trained. Table 5.1 shows these newly trained models. The accuracies achieved for the self-trained models are comparable to that of the original work. So, they are sufficient to perform the experiments.

Network	Configuration	Our acc.	Acc. from [3]
SHD1$^+$	700x128x128x20	84.41%	84.4%
SHD4	700x512x256x20	85.42%	-
SMNIST3$^+$	40x256x128x10	96.19%	97.82%
SMNIST4	40x512x256x10	96.95%	-
PSMNIST1$^+$	40x256x128x10	91.17%	91.0%
PSMNIST2	40x512x256x10	93.05%	-
SSC2$^+$*	700x400x400x35	73.13%	74.2%
SSC3	700x256x128x35	66.40%	-

Table 5.1: Self-trained networks using the neuron dynamics of [3]. $^+$: networks with the same layer sizes as the paper. *: was not presented in paper but was included in source-code.

Datasets SHD and SSC are from the same work [41]. The SHD and SSC dataset must be discretized as part of the data preparation. As a result, the dataset's number of timesteps depends on the precision chosen for this discretization process. The SHD and SSC have 100 and 250 timesteps for this work, respectively. The original work uses the same parameters. Choosing different discretization precisions is beneficial as it provides more variety in the benchmarks.

PS-MNIST and S-MNIST are both based on the MNIST dataset. S-MNIST is supposed to be a sequential version of MNIST. S-MNIST is simply a version of MNIST for which each image is flattened and then convoluted. In this work, this convolution will have a stride of 2 and a kernel size of 8. PS-MNIST is similar to S-MNIST, but PS-MNIST permutes the pixels beforehand. This permutation is the same for all samples. Like the original work, the PS-MNIST and S-MNIST are population-encoded, meaning that there is one extra ALIF layer and that extra ALIF layer is fed analog numbers instead of spikes. This extra population encoding layer has 40 neurons for both S-MNIST and PS-MNIST. The simulator does not allow something other than spikes. As a result, the simulator can not use the input from the dataset itself. But instead, the spike trace of the population encoding layer is extracted. That spike trace is then used for the simulator.

5.1.2 Mapping

The simulator also requires a mapping of the SNN layers onto the hardware cores. This is complicated by the fact that if a layer does not fit wholly

[1]The modified SRNN implementation is published with the source code

on a core then it can be split into multiple parts and be fitted to multiple cores. In addition, there are several constraints when fitting a layer to a core. Namely, the number of neurons may not exceed the number of neurons that can fit in neuron memory. The number of synapses can not exceed synapse memory. The layer can also not be split into too many parts as the core needs to store the location of each part of the split layer in a row on the layer memory. In addition, only a limited number of layers can be stored on each core constrained by the number of rows in the layer memory. Finally, the amount of fan-in connections is limited as well because of the limited size of the compute buffer. As the compute buffer must be able to hold all of the spikes of the incoming synapses until the synchronization event comes in.

Since there are too many constraints it is not feasible to make a manual mapping from layers to cores. Therefore a simple mapping algorithm was implemented to do the mapping automatically. Future work does not have to use this mapper they can write their own mapper. The algorithm is based on a first-fit bin packing algorithm. Algorithm 1 shows the main concept. The algorithm first tries to fit a layer to a core in such a way that the layer does not have to split if it has to split then it will fit the layer to the first cores. This is what the `FindWholeFit` function does.

To determine whether the layer fits a core wholly, algorithm 2 is used. This algorithm will be called the `MaximumCut` algorithm, as it tries to fit the largest subset of layer's neurons that fits on a core. This algorithm tries to find the maximum amount of neurons that fit on from layer $l \in \mathbb{L}$ with size n onto a core $c \in \mathbb{C}$, where $\mathbb{L}$ is the set of all layers in the SNN model and $\mathbb{C}$ the set of all cores. When the maximum cut is the size of the layer then we know that the whole layer will fit on the core.

Also, the cores are sorted in order according to priority. The highest priority core on the list will be check first. Many orderings are possible. Figure 5.1 shows two of those. The experiments will solely use the column-major sorting as it is easiest to use, but diagonal sorting is also implemented.

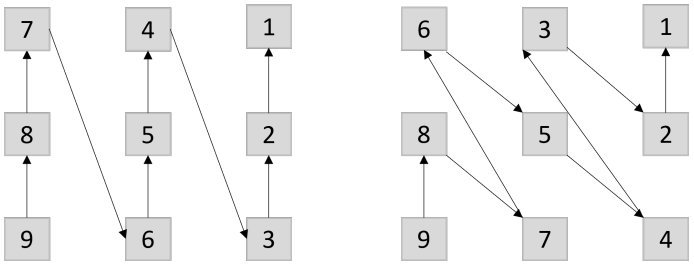
Some layers may not be split. For example, the input and output layer may be problematic if split over multiple cores. If no core can be found that fits the layer as a whole and the layer is not splittable then the algorithm will return with an error. If the layer is splittable, then the layer can be split over multiple cores. The `FindSplitSet` finds this set of cores. It checks all cores in the sorted order and tries to fit as many neurons on them on each core possible using the `MaximumCut` algorithm. If the full layer can not be mapped then an empty set will be returned. In this case, the algorithm will also return a failure. If not, then the layer will be split and actually assigned to the cores. If all layers mapped without any error then the mapping is valid.

Algorithm 1 FirstFit

for $l \in$ L **do** ▷ Iterate over all layers in network
  $c \leftarrow$ FindWholeFit ▷ Finds a core that can fit the layer as a whole
  if l is not splittable **then**
    return Fail
  end if
  $s \leftarrow$ FindSplitSet ▷ Find a set of cores that the layer can split over
  if s is empty **then**
    return Fail
  else
    AssignSplitSet(s) ▷ Applies the splitting
  end if
  return Mapping
end for

Algorithm 2 MaximumCut

function MaximumCut($l \in \mathbb{L}, c \in \mathbb{C}, n$)
  if c.nrLayers = c.maxLayers **or** c.fanin + layer.inputSize + (layer.reccurent ? layer.size : 0) $\geqslant$ c.maxFanIn **or** l **not in** c.acceptedLayers **then**
    return 0
  end if
  $neuronLim \leftarrow c.maxNeurons - c.nrNeurons$
  if l.recurrent **then**
    $synapseLim \leftarrow c.freeSynapses/(l.inputSize + l.size)$
  else
    $synapseLim \leftarrow c.freeSynapses/l.inputSize$
  end if
  return Min($neuronLim$, $synapseLim$, n)
end function



(a) Column-major **(b)** Diagonal

Figure 5.1: Different ordering of core priorities.

5.1.3 ALU config

Section 4.3.1 mentioned that multiple ALU configurations were possible depending on the number of functional units used. The configuration matters most for the synchronization of neurons, not the integration, as the synchronization is more complex. The experiments will explore three configurations: serial, parallel, and pipeline. In the serial configuration, everything operation required for syncing happens serially. The parallel configuration is similar to serial but instead tries to parallelize operations maximally. Finally, there is the pipeline configuration that assumes a pipeline. In the pipeline configuration, each operation has its functional units. Consequently, the pipeline configuration runs at the fastest speed possible. The default ALU configuration will be the serial configuration for all experiments.

Table. 5.2 shows, among others, the latencies for these configurations. This work assumes that a load and store happen in a separate cycle. There might be possibilities to optimize this, but a more conservative estimate is preferred. As a result, the pipeline will only be able to reach an initiation interval of 20 ns instead of 10 ns. In addition, the hardware model assumes that if an operation is skipped due to the neuron not spiking, it will still take one cycle[2]. In addition to latencies, table. 5.2 also shows the number of functional units used in a core as a function of the number of parallel synapses or neurons P. The simulator assumes that the same functional unit does an equality check and subtraction as an addition, meaning that the area and leakage are shared. The simulator must make this assumption because the work of [39] does not have separate functional units documented for these operations.

No latencies are associated with receiving a spike or sync event on the receiver, creating a spike to put into the output buffer. Additionally, writing or reading a buffer also take no time. Only the ALU operations take time. A user can add more latencies in future work to make the simulation more accurate.

	Sync. II [ns]	Sync. Lat. [ns]	Int. II [ns]	Int. Lat. [ns]	#Addf32	#Multf32
Serial	110	110	30	30	$1P$	$1P$
Parallel	80	80	30	30	$2P$	$1P$
Pipeline	20	80	20	30	$6P$	$3P$

Table 5.2: Parameter values as a result of which ALU configuration is chosen. The ALU configuration influences both the timing and the number of functional units.

[2]One cycle is 10 ns. The HW is assumed to run at 100 MHz (see section 4.4)

5.1.4 Routing config

In addition, to core latencies, there are also router latencies. Table. 5.3 shows the router latencies. As mentioned in section 4.3.3, the delays for transferring from the core to the router can differ from one router to another. The delay for an inter-router transfer is the transfer delay. The input and output delays give the delay for transferring from a local core to the router. The table shows the delays are the same for all kinds of transfers. Finally, the switch delay represents multiple operations that need to be done by the switch, like arbitration, route computation, and switch travel. The latency model assumes that arbitration takes two cycles, and route computation and switch traversal take one.

Param. Name	Value [ns]
InputDelay	10
OutputDelay	10
SwitchDelay	40
TransferDelay	10

Table 5.3: Baseline router latency parameters used to do the experiments.

Three different buffering parameters relate to the NoC. The core output buffer depth and the router's input buffer and output buffer depth. The core output buffer will not be changed and will be kept at a depth of 1. The NoC experiments will change the input and output buffer depth, but for the other experiments, we will keep it at a default of 1.

5.1.5 Memory config

This section will define S_n, S_l, S_s, and N_{types} of section 4.4.3. The other parameters of the section will be defined in section 5.3.1. We want to simulate the model of [3]. Figure 5.2 shows how the neuron state, synapse state, and layer state should look like to achieve this. All of the parameters are 32-bit floating-point values as no quantization is assumed. The neuron state has 5 parameters and one boolean to denote whether it spiked last timestep. As a result, the neuron state, S_n, is quite large with 161 bits. A synapse only requires the storage of the weight. The synapse state size, S_s, therefore, is 32 bits large. The layer state stores the threshold, `VTH`, and beta as these parameters only need to vary based on layers. The size of the layer state, S_l, is 64 bits.

Our current model only has only three kind of packet types. This is not realistic as more than three kinds of packets would be needed to configure the memories and for debugging purposes. Therefore this parameter was made configurable the number of packet types N_{types} was made configurable.

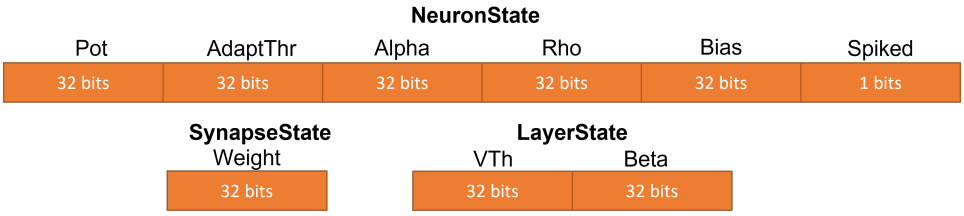


Figure 5.2: Memory layout of the neuron states.

The experiments will assume that a hardware implementation will require 8 different kind of packet types. Therefore N_{types} is 8.

5.2 Evaluation

The experiments have three main metrics to measure the performance of the hardware: the synaptic energy, the synaptic area, and the throughput efficiency. The synaptic energy measures the energy efficiency of the chip. The total energy is not used because the total energy usage is very dependent on the number of synaptic operations. As a result, it would be difficult to distinguish efficiency by looking at total energy. The synaptic energy is calculated by the following formula: $E_{sop} = E_{chip}/N_{sop}$, where E_{chip} is the energy consumption of the whole chip from section 4.4.6 and N_{sop} is the number of synaptic operations.

Likewise, the synaptic area is used instead of the total area. The synaptic area measures the area required to store one synapse. The synaptic area is calculated by the formula: $A_{syn} = A_{chip}/N_s$, where N_s is the number of synapses that the chip can hold, A_{chip} is the the total area of chip from section 4.4.6.

The throughput efficiency measures the number of synaptic operations per second that can be achieved per square millimeter of chip area. The following formula calculates it: $TE = N_{sops}/T/A_{chip}$ where T is the amount of time taken to run through the whole dataset. Henceforth, the throughput efficiency will be called just "throughput".

Most design space exploration is about making trade-offs. In a trade-off, one metric will become better, and one will become worse. Consequently, it is difficult to distinguish between an efficient and an inefficient trade-off. Therefore, in this work, we will seek to maximize the EAT (Energy-Area-Throughput) measure. Equation 5.1 shows the equation to calculate the EAT. This EAT calculation will be done on a per network basis.

$$EAT = TE/E_{sop} \tag{5.1}$$

Representing the metrics of all networks for all experiments is infeasible. Instead, to summarise the results of all networks, the arithmetic mean is used to summarise all numbers. Each experiment name will also be annotated with its experiment ID between brackets. These IDs are important for section 5.3.4 as they can be used to cross-reference the experiments. Additionally, these IDs can also be used to lookup up the metrics on a per network basis instead of the summarised numbers. These unsummarised numbers are provided in appendix A.

5.3 Experimental results

Finally, now that we have set up everything, we can do some experiments and try to answer our research questions. Section 5.3.1 seeks answer the question related to the effect of core size. Section 5.3.2 seeks to answer what

effect the design of the ALU will have. Section 5.3.3 will answer two of our research question. It will answer both the influence of buffering and changing the number of wires. Finally, in section 5.3.4, we will look at what accuracy the networks run in our simulator predict versus what the PyTorch trained models predicted. We are looking at the accuracy to evaluate whether the synchronization problem does not have too large of an effect on the results and whether there were no mistakes in the simulator code.

5.3.1 Core size

This experiment tries to vary the core size and examine the effect of changing core sizes on the metrics. Table. 5.4 shows the core sizes that will be tried. Not only should the maximum number of neurons and synapses scale, but we should also scale the maximum splits and number of cores. If the layer does not fit one core, then the layer needs to be split. When a layer splits, more destination cores have to receive the spike. These destination cores are stored in the layer memory. The layer memory can not have an infinite width. This is what limits the maximum amount of splits. If core sizes are small, the mapper must split the layers into more parts. Consequently, more cores and more allowed splits are needed.

Param. name	Small	Medium	Large	Huge
MaxNeurons (N_{neuron})	32	64	256	512
MaxSynapses (N_{syn})	32K	64K	256K	512K
MaxFanIn (N_{fanin})	2048	2048	2048	2048
MaxLayers (N_{layer})	4	4	4	4
MaxSplits (N_{split})	64	16	8	4
Parallel (P)	4	4	4	4
Mesh ($N_{w,x}$x$N_{w,y}$)	8x8	4x4	3x3	2x2
ALU	Serial	Serial	Serial	Serial

Table 5.4: The different core sizes that will be explored for this experiment. MaxSplits refers to the maximum of parts a layer can be split into. Between brackets are the corresponding parameters from table 4.1.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot 10^{21}$]
Small (exp4)	134	14.6	21.6	109
Medium (exp1)	130	35.0	18.2	269
Large (exp3)	186	7.0	15.6	37
Huge (exp2)	263	6.0	15.2	23

Table 5.5: Results of varying the core sizes. Configurations from table 5.4 are used.

Table 5.5 shows the results of the experiments using the core sizes from table 5.4. The results show that the medium size is the best in terms of EAT.

The trend is that larger core sizes seem to have a higher energy consumption than smaller core sizes. However, this trend is broken for the small core size, where it is smaller but consumes slightly more energy.

An energy breakdown may give a better idea of what is going on. Table. 5.6 shows the energy breakdown. The energy breakdown shows which component consumes what amount of energy. The data clearly shows that most energy goes to reading and writing memories. When the core size decreases, the synapse memory energy significantly decreases. This decrease is the result of the smaller memories on smaller core sizes. The medium size performs better than the small size because the increase in energy by the layer memory, router, and static on the small core counteracts the decrease in synapse memory energy. As the core size decreases, more layer splits occur, and the core must send far more spike packets. This splitting, therefore, increases the consumption due to routing. Besides, the core must deal with integration events that update fewer neurons. Consequently, the overhead of getting the layer information for each spike integration becomes more significant, which increases the layer memory energy consumption. Additionally, the layer memory is wider due to the increased number of splits.

One other interesting observation: the router does not consume much energy relatively. In hindsight, this makes sense. Generally, the energy to switch and transfer a packet is on the order of 20 pJ. The cost of updating a neuron for the hardware model is 80 pJ. In addition, one packet transfer will cause multiple neuron updates. So, the packet transfer cost is minimal compared to the neuron update cost for a whole layer. Splitting the layers more will increase the energy due to routing relatively, but it will still be a relatively minor energy consumer.

Component	Energy Small [mJ]	Energy Medium [mJ]	Energy Large [mJ]	Energy Huge [mJ]
Static	23	20	25	44
Layer Mem.	23	3	1	1
Neuron Mem.	259	260	261	263
Synapse Mem.	49	76	235	447
Compute buffer	2	0	0	0
Output buffer	0	0	0	0
ALU	31	31	31	31
Router	7	2	1	1
Total	393	391	554	788

Table 5.6: Energy breakdown: energy consumption per component. This breakdown is for the SHD1 network. Static energy for layer memory, etc are included separately in the static category.

There is a general trend toward the cores increasing throughput as the core size shrinks. The throughput increases as cores become smaller because

there is more computation power for fewer synapses. As a result, the integration of spikes takes less time. However, if cores become smaller, the mapper must split layers into more parts. Consequently, there is congestion due to more traffic on the mesh. Congestion seems prevalent for the small core size as the throughput is lowered. The transfer latencies in table 5.7 show that there indeed is a lot of congestion. A packet transfer is far slower on the small core size than the large ones. So, it is plausible that the congestion partially causes the throughput to slow down. In both the large and huge configurations, there is almost no congestion, but there is still no doubling of throughput. This bad throughput scaling may be due to the general latencies involved in synchronization. For example, it takes time for the ready event to travel to the controller. And it takes time for the controller's input spikes to arrive at the cores containing the first hidden layer.

Network	Lat. Small [ns]	Lat. Medium [ns]	Lat. Large [ns]	Lat. Huge [ns]
SHD1	58.3	52.3	50.0	50.0
SHD4	91.8	67.6	51.5	50.5
SMNIST3	77.6	61.4	50.1	50.0
SMNIST4	83.3	66.5	51.1	50.0
PSMNIST1	89.9	63.2	50.1	50.0
PSMNIST2	87.0	67.8	51.3	50.0
SSC2	76.8	59.6	50.7	50.3
SSC3	90.5	67.9	50.0	50.0
Mean	81.9	63.3	50.6	50.1

Table 5.7: Transfer latencies: the time from when a packet has been received until it after it has been sent to a next router. The results are showing the mean latency per network. Shortest latency possible is 50ns.

Finally, there is a trend towards larger cores requiring less area to store a synapse for the synaptic area. We may get an explanation by looking at each component's area consumption. Table 5.8 shows this area breakdown. This table shows that as the core size increases, the overhead will decrease, as components other than the neuron memory and synapse memory will remain relatively constant in size. Accordingly, the synaptic area will be lower, as the area dedicated to, for example, a router does not allow more synapses to run on the hardware.

Component	Area Small [mm^2]	Area Medium [mm^2]	Area Large [mm^2]	Area Huge [mm^2]
Layer mem.	0.01 (1.4%)	0.01 (0.8%)	0.01 (0.2%)	0.01 (0.1%)
Neuron mem.	0.02 (2.8%)	0.02 (1.7%)	0.03 (0.7%)	0.05 (0.6%)
Syn. mem.	0.49 (69.0%)	0.97 (82.9%)	3.86 (94.1%)	7.71 (96.6%)
Compute buf.	0.03 (4.2%)	0.03 (2.5%)	0.03 (0.7%)	0.04 (0.5%)
Output buf.	0.01 (1.4%)	0.01 (0.8%)	0.01 (0.2%)	0.01 (0.1%)
ALU	0.02 (2.8%)	0.02 (1.7%)	0.02 (0.5%)	0.02 (0.3%)
Router	0.13 (18.3%)	0.13 (10.9%)	0.13 (3.2%)	0.13 (1.6%)
Total	0.71	1.19	4.10	7.96

Table 5.8: Area breakdown: area usage per component in a core

5.3.2 ALU design

The goal of this experiment is to try and find out what the influence is of adding more functional units to a core. Adding more functional units should increase leakage but improve throughput. Changing the ALU configuration can be done in two ways: by changing the ALU configuration between serial, parallel, and pipeline or increasing the amount of parallelism. The goal is also to find the trade-off between these two choices.

Firstly, we will explore the ALU configurations. The different configurations of table 5.2 with a medium core size are tried. Table 5.9 shows the results for the different ALU configurations. The serial and parallel configurations are not that different, which makes sense considering that they are quite similar in latency and number of functional units. However, the pipeline configuration is quite a bit different. We expected that these configurations would trade area and energy for throughput. However, in this case, the increase in throughput seems to be worth the relatively small increase in area and energy. The energy increase is due to the increased leakage because of more functional units. However, it has to be kept in mind that the simulator does not measure idle energy. Thus energy consumption could realistically be worse.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [$SOP^2/(m^2Js) \cdot 10^{21}$]
Serial (exp1)	130	35.0	18.2	269
Parallel (exp5)	131	35.9	18.3	274
Pipeline (exp6)	136	43.0	19.0	315

Table 5.9: The results of a medium core size design with different ALU configurations.

Next, we will explore an increase in the parallelism of the medium core size design. Table. 5.10 shows the results for this experiment. Increasing the parallelism is very effective in increasing the throughput of the cores. The throughput of the ALU must have been quite compute-bound. More

interestingly, the energy goes down as well. This decrease in energy is because when the parallelism increases, each memory access retrieves more synapses. As a result, the cost of accessing a large memory can be divided over multiple synapses. Hence, this saving overhead even offsets the increase in leakage. Hence, energy savings. This savings is significant for the synapse memory as it is big but not wide. Increasing the parallelism is far more effective than just changing the ALU configuration. The hardware design has a parallelism of 16 is better than the pipeline ALU configuration from earlier on every metric.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot 10^{21}$]
Parallel 4 (exp1)	130	35.0	18.2	269
Parallel 8 (exp7)	120	51.7	18.4	429
Parallel 16 (exp8)	117	67.0	18.9	572
Parallel 32 (exp38)	119	76.6	19.9	644

Table 5.10: The result of increasing parallelism for the medium core size.

Knowing that parallelism can significantly decrease synapse memory energy consumption, it may be interesting to see what happens when the large core size increases in parallelism. The larger cores have a high energy consumption mainly due to the high synapse memory energy. So, particularly the larger core sizes can save a lot of energy by using more parallelism. Table 5.11 shows the results when parallelism is increased for the huge core size. The results show a significant improvement in terms of energy consumption. So much so that the huge core size with high parallelism can at least be competitive with the medium core size. The cost model shows that the read energy of the synapse memory was originally 146 pJ per synapse. With increased parallelism of 64, it is only 15 pJ to read one synapse. Now, the neuron memory causes most of the energy consumption. It takes 82 pJ to update the neuron potential and 15 pJ to read the synapse.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot 10^{21}$]
Parallel 4 (exp2)	263	6.0	15.2	23
Parallel 16 (exp10)	144	22.5	15.3	156
Parallel 32 (exp9)	125	40.2	15.4	322
Parallel 64 (exp11)	115	65.9	15.7	570

Table 5.11: The result of increasing parallelism for the huge core size.

It would also be interesting to see what would happen when the parallelism is increased on the smaller core sizes. An increase in parallelism should theoretically increase traffic on the mesh. The small core size already suffers from congestion. Thus an increase in parallelism on the small

core size should not increase the throughput much. Table 5.12 shows the results when core sizes are small. The results confirm these suspicions. An increase in parallelism still leads to an increase in throughput but not much. More interestingly, even the energy starts increasing from parallelism of 16 onwards. The increase in static power is likely the main cause for this. For parallelism of 4, it is still 23 mJ, while for parallelism of 16, it is already increased to 44 mJ, when the SHD1 network is used. The increase in static power is especially problematic for the parallelism of 32 design, where even EAT decreases.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot 10^{21}$]
Parallel 4 (exp4)	134	14.6	21.6	109
Parallel 8 (exp15)	132	17.3	22.1	132
Parallel 16 (exp16)	138	18.7	23.1	137
Parallel 32 (exp17)	155	18.3	25.1	122

Table 5.12: The result of increasing parallelism for the small core size.

5.3.3 NoC design

The goal of the following experiment is to research the effect the NoC has on the chip. Specifically, we will study the impact of increasing buffering depth and transfer time on the metrics. We will first increase the amount of buffering in the routers. An increase in buffering should help smooth out peak traffic but comes at the cost of more area usage. In the earlier experiments, we kept the buffer depth at 1. In the following experiments, both the input buffer and output buffer are increased separately to see the influence of both independently. Table 5.13 shows the results for increasing the buffer sizes on the NoC's routers. The results show that increasing the input buffer size is more effective than increasing the output buffer size. On the other hand, increasing the output buffer size only has a minimal effect. This minimal effect may either be due to the hardware not being sensitive to increasing buffer depths or relatively low congestion on the medium core hardware.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot 10^{21}$]
In & Out 1 (exp1)	130	35.0	18.2	269
In 5 (exp12)	130	35.9	18.2	275
Out 5 (exp14)	130	35.2	18.2	270
In & Out 5 (exp13)	130	37.4	18.2	288

Table 5.13: The result of increasing buffer depth for the medium core size.

Before concluding any trend about buffer depth, it is good to get more data. The medium has some congestion, but the smaller core design has

more congestion. It would be interesting to look at buffering results on the smaller core size. Table 5.14 shows these results. The small core size shows the same trend. According to the results, increasing the input buffer size is more effective than increasing the output buffer size. This better throughput for increasing input buffer depth contradicts the literature. Normally, the literature assumes that output buffering provides more throughput than input buffering [42].

The most likely explanation is that the ALU is stalling because there is no free space in the output buffer of the core. An increase in the input buffer depth of the router effectively increases the output buffer space on the ALU as only a 10ns link separates these buffers. A custom experiment is performed where the the buffer sizes where the input buffer and output buffer depths were kept at 5 for all directions except for local which was kept at 1. "In 5" and "Out 5" from table 5.13 then only have a throughput of 35.2 and 35.2, respectively. Likewise, "In 5" and "Out 5" from table 5.14 then have a throughput of 15.0 and 15.0, respectively. These results show that increasing in anything other than the local direction does not have any effect for the medium design. However, For the small core size experiment's buffering this does have an effect. The input and output buffering have about the same effect. These results show that there is not a large difference between input and output buffering on a router.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot 10^{21}$]
In & Out 1 (exp4)	134	14.6	21.6	109
In 5 (exp18)	133	15.2	21.6	114
Out 5 (exp19)	134	15.0	21.6	113
In & Out 5 (exp20)	133	16.5	21.6	125

Table 5.14: Results of increasing buffer sizes on the router on the small core size.

Furthermore, we can try increasing the buffer sizes more and see where the throughput gains stop. Table 5.15 shows the result of this experiment. After increasing the input buffer size to 5, the throughput does not increase anymore.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot 10^{21}$]
In 1 (exp1)	130	35.0	18.2	269
In 5 (exp12)	130	35.9	18.2	275
In 10 (exp21)	130	36.0	18.2	277
In 20 (exp22)	130	36.1	18.2	277
In 40 (exp23)	130	36.0	18.2	276

Table 5.15: The result of increasingly larger input buffer depths for the medium core size.

Maybe this diminishing throughput is due to using the medium core size that has less congestion. Or, due to it only happening on the input buffer sizes. We can check this by doing a set experiments on the small core size with increasing output buffer depths. Table 5.16 shows the results. It shows that even when using the small core size about the same effect occurs. The throughput quickly diminishes after a buffer depth of 5.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot 10^{21}$]
Out 1 (exp4)	134	14.6	21.6	109
Out 5 (exp19)	134	15.0	21.6	113
Out 10 (exp33)	133	15.1	21.6	113
Out 20 (exp34)	133	15.2	21.6	114
Out 40 (exp35)	133	15.2	21.6	114

Table 5.16: The result of increasingly larger output buffer depths for the small core size.

Next, the amount of wires between two routers also determines the time it takes to transfer a packet between two routers. Minimizing the number of wires between routers can reduce area consumption. Sadly, the simulator cannot model the area consumption of wires. However, the transfer time can still be increased, which can model the throughput effect of decreasing the number of wires. For example, if a flit is usually 10 bits and there are 10 wires, it results in a 10 ns transfer time. Then by increasing the transfer time to 20 ns, we can model a situation where the number of wires would 5. We ran all of the previous experiments with a transfer time of 10 ns. Table 5.17 shows the results of increasing the transfer time to 20 ns, 40 ns, and 80 ns on the medium core size. The data shows that the throughput decreases quite sharply. The synaptic area and EAT are useless as wires have no area model. In reality, the synaptic area would have to reduce. Theoretically, increasing the buffer sizes should increase throughput and even out this effect. Table 5.18 shows the results of increasing the input and output buffer depths when the transfer time is 40 ns. These results even show an improvement in throughput. So, in this case, the throughput increase of larger buffer depth compensates for the larger transfer time. When the buffer sizes are 5, the throughput is only 0.2 lower than it would be with a transfer time of just 10 ns. In real hardware, this would be an interesting trade-off to make.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot 10^{21}$]
Trans. 10 ns (exp1)	130	35.0	18.2	269
Trans. 20 ns (exp24)	130	33.9	18.2	259
Trans. 40 ns (exp25)	131	31.8	18.2	242
Trans. 80 ns (exp37)	132	27.9	18.2	212

Table 5.17: The result of increasing the transfer latency for the medium core size.

Exp. Name	Energy [pJ/SOP]	Throughput [MSOP/s/mm^2]	Area [μm^2/syn]	EAT [SOP^2/(m^{2}Js) $\cdot$ 10^{21}]
In & Out 1 (exp25)	131	31.8	18.2	242
In & Out 5 (exp26)	130	37.2	18.2	286
In & Out 10 (exp27)	130	37.3	18.2	287

Table 5.18: The result of increasing buffer depth for the for the design with a transfer latency of 40 ns from table 5.17.

5.3.4 Accuracy deviations

As mentioned earlier, because our simulator is execution-driven, it can also give accuracy predictions. We can compare this prediction to our PyTorch model's accuracy and get an idea if there are any mishaps in execution. Figure 5.3 shows this difference. The figure shows that the simulator does not entirely predict the same outputs as the PyTorch model. If we look closely at the figure, there is a component that constantly causes most networks to mispredict no matter what experiment. And there is another component that deviates on a per experiment basis.

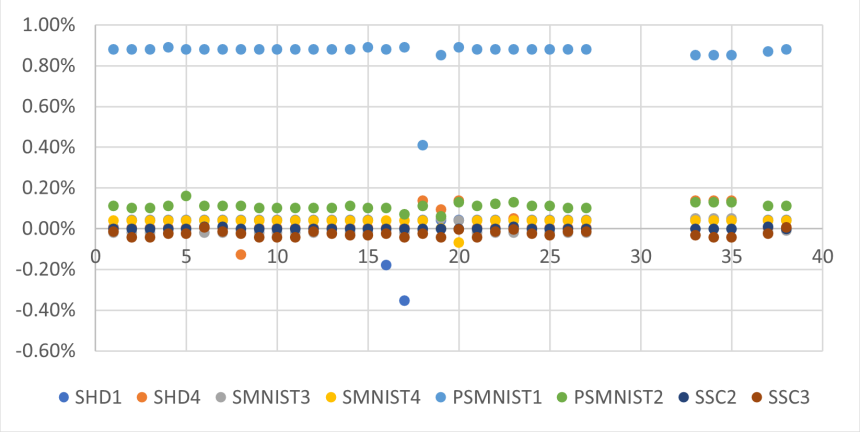


Figure 5.3: This figure shows on the y-axis the difference in accuracy between when the network runs on the PyTorch model versus when run on the simulator. The figure shows the accuracy per experiment per network. The numbers on the horizontal axis refer to the experiment IDs.

A bug likely causes the first constant component in the simulator. Again, this is an execution-driven simulator, so any difference in calculations can result in wrong execution and, therefore, lower or higher accuracies. The cause of this constant component is unknown. It can be that the parameters extracted from the PyTorch model are not extracted with enough accuracy. Or that there is a bug in the code. The research into the problem was inconclusive.

However, the second component, the difference on a per experiment basis, can be explained. As mentioned in section 4.3.4 the synchronization

model used to do the experiments is not perfect. This is due to a race condition in the synchronization algorithm. The synchronization problem can cause spikes to arrive a timestep too late. This is the likely cause of the second component.

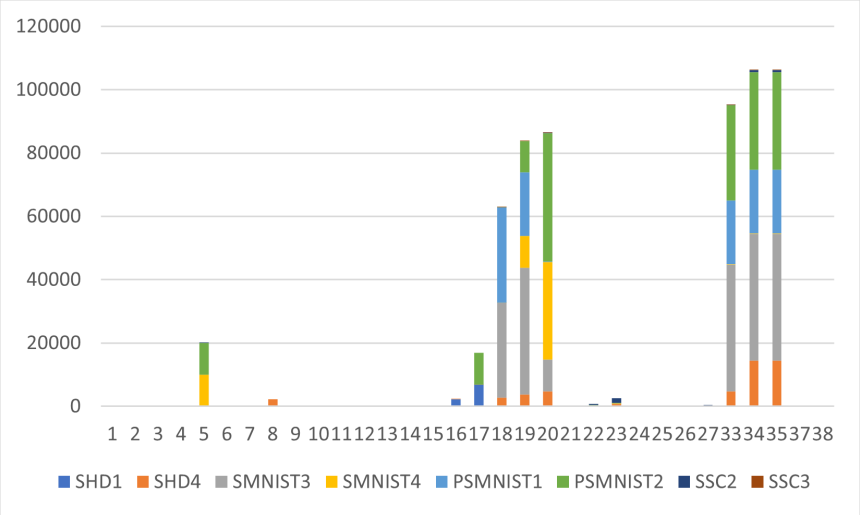


Figure 5.4: Stacked bar graph of the number of spikes arriving at the wrong timestep in our hardware model on the simulator. The bars show the number of mistimed spikes on a per network per experiment basis. The numbers on the x-axis refers to the experiment ID.

To confirm this suspicion, we can let the simulator measure the number of spikes that arrive either too early or too late. Henceforth, such a spike will be called a faulty spike. Figure 5.4 shows the number of faulty spikes. We can see that the differences in figure 5.3 overlap with the number of faulty spikes from figure 5.4.

We can also take the circumstances that these faulty spikes occur. They often occur on hardware designs where there is a combination of much buffering on the NoC and a lot of congestion. This makes sense, as this is caused by the synchronization problem mentioned in section 4.3.4. The synchronization problem is a race condition. If there is more congestion and more buffering, then two different paths can have more opportunity for different latencies. Therefore, there are more chances for race conditions to happen.

Chapter 6

Discussion

Now that experiments are done we can do an analysis and try to draw some conclusions from each experiment and the hardware in general. After that we can reflect and see what can be done better in the future. Section 6.1 will provide an analysis of the general hardware and will compare the simulated hardware to the literature. Section 6.2 will discuss the results of the core size, ALU design, and NoC design experiments. In section 6.3 we will reflect on the design and approach of the simulator. Finally, in 6.4 we will look at possible avenues of improvement for the modelling and hardware design.

6.1 Hardware analysis

The current designs are comparatively still very inefficient compared to the SOTA. TrueNorth has a synaptic area of 1.60 μm^2. Loihi has a synaptic area of about 0.46 μm^2. ODIN has a synaptic area of 0.68 μm^2. Exp 1 has a synaptic area of about 18.2 μm^2, which is an order of magnitude worse, even though the simulator most likely underestimates the area. The synapse memory takes up most of the area, which is a significant problem. Partially, the bit width of the synaptic weights can be blamed. Other literature uses less precise synaptic weights. For example, ODIN uses 4 bits per synapse. TrueNorth and Loihi can even go down to 1 bit per synapse. When taking the number of bits per synapse down to 4 and 1 bits, decreasing the neuron size to 21 bits and dividing the area of functional units by 8 on exp1 will lead to a synaptic area of 5.0 μm^2 and 3.7 μm^2, respectively. Still larger than most of the literature, but at least somewhere in the same ballpark. With these optimizations the router now consumes half of the area and therefore is the main area consumer. Also, keep in mind that Loihi on and ODIN use a more advanced fabrication process and therefore have denser memories. This simulator assumes a 40 nm process. Since 90% of the more area efficient designs is used by the synapse memory, more area savings can be found by increasing memory density, adding memory compression, or adding pruning.

The same holds for synaptic energy. The current designs are too inef-

ficient. ODIN takes 12.7 pJ of energy per synaptic operation. Loihi takes 23.6 pJ per synaptic operation. TrueNorth takes 26 pJ per synaptic operation. Comparatively, a SOP energy of 130 pJ for exp1 even though it is not the best is still quite alarming. Although this again is primarily due to the lack of quantization. Suppose we do an experiment where we reduce the bit width to 4 bits. In this case, we reduce neuron size to 21 bits, so 4 bits for most parameters, and all the stats of the ALU divided by 8 on exp1, the energy per synaptic operation is reduced to 21.1 pJ.

A bad hardware decision largely causes the remaining inefficiency. Choosing the axonal approach itself is not that big of a problem, but the disadvantage of the axonal approach is that it needs to fetch the neuron potentials more. This fetching would not be a problem if the neuron state is small, but the current implementation has quite a large neuron state, and it does not have a separate neuron potential memory, meaning it has to fetch the whole neuron state. So, either the neuron potential needs to be put in a separate memory or the dendritic approach needs to be used. We can look at what separating the neuron potential would save. Originally, updating the whole neuron state results in energy consumption of about 12.4 pJ. By only fetching the neuron potential, this can likely be reduced to 3.0 pJ with enough of parallelism. This is a saving of about 9.4 pJ. This would reduce the energy per SOP further down to 11.7 pJ.

So, if quantization and the hardware are optimized, then the simulated design can be quite competitive. What has to be kept in mind is that the simulator assumes that its hardware runs at a clock speed of 100 MHz on a 40 nm process at 1.1 V. The earlier calculations also assumed floating-point calculations instead of the integer ones if quantization is used. So, the synaptic energy can be even better.

6.2 Experimental analysis

Even though the designs are not perfect, we can still draw some conclusions from the experiments: Firstly, assuming that cores do not do external memory banking and all have the same parallelism and ALU configuration, smaller cores will have better energy consumption. But they do not have to be so small that the layers have to be split so often that routing and other overheads become a problem. Secondly, larger cores scale better because they fundamentally produce less traffic on the mesh. After all, large cores do not split the layers as much. Finally, routing energy, in this work, does not significantly contribute to energy consumption. Although, this may be due to the neuron and synapse memory taking up so much energy, thereby making the energy consumption of the routing relatively insignificant.

The ALU configuration can increase the design's throughput at the cost of an increase in energy and area. Changing the ALU configuration leads

to a bit of improvement in EAT. However, increasing the parallelism of the core will be more efficient because this also leads to wider memory access. Above significantly improving the throughput, it also lowers the energy consumption due to reducing the cost of accessing the large synapse memory. Increasing parallelism is particularly effective for larger core sizes. However, it has to be said that this is partially due to the assumption that a wider ALU also comes with wider memory accesses. The hardware can also be designed so that one synapse can be calculated in parallel, but instead of only fetching one synapse, multiple synapses will be fetched in a batch. Also, this assumes that no special techniques are applied to the memory to reduce the dynamic energy access like banking.

Increasing the buffer depths by a small amount can have a small effect on the throughput while not increasing the synaptic area by much. However, for medium core design and small core design, most of the increase in throughput is due to a virtual increase in the core's output buffer. Nevertheless, without this effect increasing all input or output buffers still results in a small increase in throughput. Increasing the input buffer depths is preferable in this scenario as output buffering is difficult in terms of hardware implementation [42]. Increasing the input and output buffer depths together leads to the most increase in throughput but would require the most area and lead to the hardware difficulty mentioned earlier. Anything above 5 packets in depth does increases the throughput only sparingly and therefore should not be done. Increasing the transfer times diminishes the throughput of the hardware in quite a substantial way. However, most of this decrease in throughput can be recovered by increasing the buffer depths. It may be interesting to see where the trade-off lies in decreasing the number of wires and increasing buffer depth in terms of area usage. However, this question requires that the simulator also gets an area model for the wires.

Two sources cause the accuracy to deviate. One unknown that is likely caused by the simulator causes all experiments to deviate. And one that is caused by the synchronization problem. The unknown only causes a real problem for the PS-MNIST1 network. For the other ones, the accuracy deviation is acceptable. The accuracy loss due to the imperfect synchronization is also not disastrous. The accuracy loss only happens when there is more buffering and congestion. Worst case, the accuracy only deviated by 0.5% due to this problem. The reason for this small deviation was that faulty spikes only occurred rarely and not often in the same dataset sample. Luckily, the networks are resilient to at least some spikes dropping. However, it is the case that if too many spikes drop, then accuracy does start to suffer a bit. Depending on the application the accuracy deviations may be an issue or not. However, for most applications this will likely not be an issue.

6.3 Simulator reflection

Additionally, there are some fundamental problems with the current simulator. For one, the hardware being modeled is relatively unique. Not many chips in the literature use this specific hardware. This makes it difficult to compare the hardware to the SOTA. Also, the model is only as accurate as the accuracy of the model and parameters. If the simulator does not model a real-world chip in a behaviorally accurate way, then the simulator will lack accuracy. Certain effects may exist in the simulator but not in the real world. Additionally, if parameters are chosen wrongly, the same will hold. One problem with the parameters is that the latencies are also parameters, which introduces more room for inaccuracies. Finding the right area and energy models, especially for the router, came with great difficulty, as a result, these models could be more accurate.

Luckily the simulator is very open to changes. The simulator is implemented so that the DES simulator itself only runs the SNN on the hardware and produces latencies and other metrics and the number of operations performed. Whatever program feeds the DES simulator or analyzes the results does not matter. So, if someone wants to build another cost model program, they can. They can even implement their own core, controller, NoC, or SNN models in the DES simulator. They can also write their own mapper because the DES simulator only requires a mapping file that any program can generate.

6.4 Future work

In hindsight, many improvements could have made the hardware far more efficient. Due to only running fully connected layers and having 32-bit synapses, the synapse memory was immense compared to all other components. There are many solutions to this overly large synapse memory, but the main one is quantizing and reducing the bit width of the synapses. Almost all other neuromorphic hardware chips like TrueNorth, Odin, and Loihi do this. The main problem for this work was that the original model of [3] did not have any results based on quantization and quantizing the model is complex and can easily cause the accuracy to suffer. So, it should really be researched in a separate work. Another interesting solution could be pruning. However, this likely requires more irregular memory access, which makes increasing the width of memory access more difficult.

We can improve the synchronization process instead of waiting for all cores to be ready and then sending the input spikes. The controller can already start sending the input spikes for the next timestep immediately after the synchronized signal. This improvement would increase throughput by decreasing idle times on the core, and there would be no known downside.

The synchronization problem has a problem where some spikes arrive late. We can solve the problem in multiple ways. For one, each spike event being sent over the mesh can have an acknowledgment. However, this leads to much more traffic over the mesh, possibly limiting throughput and increasing energy consumption. In addition, if the core waits for each acknowledgment before sending the next spike, this will significantly increase latency. Another solution would be to send a kind of "end" event. After synchronizing a layer for the current timestep, the core will send an "end" event over all outgoing connections. This will tell all other cores that they will not get any more spikes. The cores will then only be able to communicate a ready signal when all "end" events are received for all incoming layers to the core. This would lead to a relatively small communication overhead and a minimal increase in latency.

Besides the hardware, we can still improve the model accuracy as well. There is no area model for the wires between routers and also the crossbar on the router. This is because finding good numbers for a crossbar and wires was difficult. So, currently, the router is modeled as a sum of all the buffers, but this is an underestimation and does not fully resemble the real world.

The current model assumes that there are no idle energies, especially for the functional units on the core. This lack of idle energy makes the model only accurate when for the actual hardware, everything is clock gated. However, this is likely not always the case. Therefore, in future work, it would be preferable to model the idle energies as well.

The latency model is also lacking. There is, for example, no latency for placing data into buffers. So, currently, the transfer of a spike from the ALU to the output buffer happens immediately, which is not preferable. Additionally, the buffers should be modeled so that if multiple processes try to access the same buffer, some processes must wait.

Currently on a router with an input buffer, output buffer, and a simple round-robin arbitration scheme is modeled. However, the current literature does not seem to use output buffer and input buffer but instead uses virtual channels. So, in the future, maybe a model for a virtual channel-based router would be interesting.

Chapter 7

Conclusion

Our goal was to research neuromorphic hardware and the design choices involved. We were able to show that building bigger cores is better for congestion on the mesh but that it comes at the cost of energy consumption due to large memories. Additionally, we could show that the synapse memory is the leading area consumer, and both the synapse and neuron memory are responsible for a significant part of the energy consumption. We also showed that when designing ALUs, much of the improvement in throughput comes at the cost of higher energy consumption due to more leakage. We also showed that increasing the parallelism of the hardware decreased the energy consumption while increasing the throughput significantly. This was mainly due to the wider memory access saving the energy overhead of larger memory sizes for the synapse memory. Additionally, we could show that increasing the input and output buffer could lead to a small increase in throughput at the cost of a small amount of area. When increasing the depths the throughput will soon encounter diminishing returns. Decreasing the amount of wires can decrease the throughput, but that can mostly be helped by increasing the buffer size. Finally, we were able to show that the accuracy is not perfect. Mainly due to two problems. For one, the simulator can not perfectly match PyTorch models. Secondly, the synchronization technique used by the hardware is not perfect. Nevertheless, this problem only manifests itself in scenarios with high congestion and much buffering.

To achieve our goal we developed a neuromorphic simulator that would allow us to do DSE. The simulator especially had to focus on non-crossbar-based time-multiplexed chips. Although the hardware model of the simulator is not competitive with current neuromorphic hardware due to current design choices, the simulator can still give good insights on how to design multi-core neuromorphic chips. Especially if, in future work, the simulator is changed such that quantization and the dendritic approach are both used. The simulator written in this work is a good start but can still be improved. Still, this simulator shows that using a simulator can be a promising approach to researching neuromorphic hardware.

Bibliography

[1] W. Maass, "Networks of spiking neurons: The third generation of neural network models," Neural Networks, vol. 10, no. 9, pp. 1659–1671, 1997.

[2] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators," IEEE Transactions on Computers, vol. 70, no. 8, pp. 1160–1174, 2021.

[3] B. Yin, F. Corradi, and S. M. Bohté, "Effective and efficient computation with multiple-timescale spiking recurrent neural networks," CoRR, vol. abs/2005.11633, 2020.

[4] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition. USA: Cambridge University Press, 2014.

[5] D. Purves and S. M. Williams, Neuroscience. 2nd edition. Sinauer Associates 2001, 2001.

[6] Wikipedia, "Biological neuron model — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Biological%20neuron%20model&oldid=1092801228`, 2022. [Online; accessed 17-June-2022].

[7] J. K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, and W. D. Lu, "Training spiking neural networks using lessons from deep learning," CoRR, vol. abs/2109.12894, 2021.

[8] S. Ahmad and J. Hawkins, "How do neurons operate on sparse distributed representations? a mathematical theory of sparsity, neurons and active dendrites."

[9] C. Howarth, P. Gleeson, and D. Attwell, "Updated energy budgets for neural computation in the neocortex and cerebellum," Journal of Cerebral Blood Flow & Metabolism, vol. 32, no. 7, pp. 1222–1232, 2012. PMID: 22434069.

[10] M. A. Webster, "Evolving concepts of sensory adaptation," F1000 biology reports, vol. 4, pp. 21–21, 2012. 23189092[pmid].

[11] A. Krogh, "What are artificial neural networks?," Nature Biotechnology, vol. 26, pp. 195–197, Feb 2008.

[12] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," Heliyon, vol. 4, pp. e00938–e00938, Nov 2018. 30519653[pmid].

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Comput., vol. 9, p. 1735–1780, nov 1997.

[14] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," CoRR, vol. abs/1409.1259, 2014.

[15] S. Moradi, Q. Ning, F. Stefanini, and G. Indiveri, "A scalable multi-core architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps)," CoRR, vol. abs/1708.04198, 2017.

[16] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," IEEE Micro, vol. 38, no. 1, pp. 82–99, 2018.

[17] C. Frenkel, J. Legat, and D. Bol, "A 0.086-mm^2 9.8-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28nm CMOS," CoRR, vol. abs/1804.07858, 2018.

[18] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 34, no. 10, pp. 1537–1557, 2015.

[19] H. Lee, C. Kim, Y. Chung, and J. Kim, "Neuroengine: A hardware-based event-driven simulation system for advanced brain-inspired computing," in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021, (New York, NY, USA), p. 975–989, Association for Computing Machinery, 2021.

[20] C. Bartolozzi and G. Indiveri, "Synaptic Dynamics in Analog VLSI," Neural Computation, vol. 19, pp. 2581–2603, 10 2007.

[21] Y. Li and K.-W. Ang, "Hardware implementation of neuromorphic computing using large-scale memristor crossbar arrays," Advanced Intelligent Systems, vol. 3, no. 1, p. 2000137, 2021.

[22] M. K. F. Lee, Y. Cui, T. Somu, T. Luo, J. Zhou, W. T. Tang, W.-F. Wong, and R. S. M. Goh, "A system-level simulator for rram-based neuromorphic computing chips," ACM Trans. Archit. Code Optim., vol. 15, jan 2019.

[23] A. Balaji, S. Song, T. Titirsha, A. Das, J. L. Krichmar, N. D. Dutt, J. A. Shackleford, N. Kandasamy, and F. Catthoor, "Neuroxplorer 1.0: An extensible framework for architectural exploration with spiking neural networks," CoRR, vol. abs/2105.01795, 2021.

[24] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints," in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1–13, 2016.

[25] A. Balaji, P. Adiraju, H. J. Kashyap, A. Das, J. L. Krichmar, N. D. Dutt, and F. Catthoor, "Pycarl: A pynn interface for hardware-software co-simulation of spiking neural network," CoRR, vol. abs/2003.09696, 2020.

[26] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber, "Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation," IEEE Journal of Solid-State Circuits, vol. 48, no. 8, pp. 1943–1953, 2013.

[27] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," Proceedings of the IEEE, vol. 102, no. 5, pp. 652–665, 2014.

[28] C. Mayr, S. Höppner, and S. B. Furber, "Spinnaker 2: A 10 million core processor system for brain simulation and machine learning," CoRR, vol. abs/1911.02385, 2019.

[29] S. J. van Albada, A. G. Rowley, J. Senk, M. Hopkins, M. Schmidt, A. B. Stokes, D. R. Lester, M. Diesmann, and S. B. Furber, "Performance comparison of the digital neuromorphic hardware spinnaker and the neural network simulation software nest for a full-scale cortical microcircuit model," Frontiers in Neuroscience, vol. 12, 2018.

[30] M. Beyeler, K. D. Carlson, T.-S. Chou, N. Dutt, and J. L. Krichmar, "Carlsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks," in 2015 International Joint Conference on Neural Networks (IJCNN), pp. 1–8, 2015.

[31] M. Stimberg, R. Brette, and D. F. Goodman, "Brian 2, an intuitive and efficient neural simulator," eLife, vol. 8, p. e47314, Aug. 2019.

[32] M.-O. Gewaltig and M. Diesmann, "Nest (neural simulation tool)," Scholarpedia, vol. 2, no. 4, p. 1430, 2007.

[33] R. Preissl, T. M. Wong, P. Datta, M. Flickner, R. Singh, S. K. Esser, W. P. Risk, H. D. Simon, and D. S. Modha, "Compass: A scalable simulator for an architecture for cognitive computing," in SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11, 2012.

[34] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. Stewart, D. Rasmussen, X. Choo, A. Voelker, and C. Eliasmith, "Nengo: a python tool for building large-scale functional brain models," Frontiers in Neuroinformatics, vol. 7, 2014.

[35] Intel, "Lava:a software framework for neuromorphic computing." `https://github.com/lava-nc/lava`.

[36] A. Cuomo, M. Rak, and U. Villano, "Process-oriented discrete-event simulation in java with continuations-quantitative performance evaluation.," in SIMULTECH, pp. 87–96, 2012.

[37] P. Grayson, O. Lünsdorf, and S. Scherfke, "Simpy." `https://gitlab.com/team-simpy/simpy`. [Online; accessed 17-June-2022].

[38] B. Theelen, O. Florescu, M. Geilen, J. Huang, P. van der Putten, and J. Voeten, "Software/hardware engineering with the parallel object-oriented specification language," in 2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007), pp. 139–148, 2007.

[39] Y. Wu, A. Azur, and T. Po-An, "Accelergy-aladdin-plug- in."

[40] P. Wolkotte, G. Smit, N. Kavaldjiev, J. Becker, and J. Becker, "Energy model of networks-on-chip and a bus," in 2005 International Symposium on System-on-Chip, pp. 82–85, 2005.

[41] B. Cramer, Y. Stradmann, J. Schemmel, and F. Zenke, "The heidelberg spiking data sets for the systematic evaluation of spiking neural networks," IEEE Transactions on Neural Networks and Learning Systems, pp. 1–14, 2020.

[42] R. S. Ramanujam, V. Soteriou, B. Lin, and L.-S. Peh, "Design of a high-throughput distributed shared-buffer noc router," in 2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip, pp. 69–78, 2010.

Appendix A

Experimental results

Table A.1, table A.2, and table A.3 show the unsummarised metric values on a per network per experiment basis. If, for example, the throughput, energy, or EAT for the SSC3 network on exp1 is needed, then these tables show the values.

	shd1	shd4	smnist3	smnist4	psmnist1	psmnist2	ssc2	ssc3
exp1	128	130	131	132	130	131	134	125
exp2	258	261	262	269	262	267	268	256
exp3	182	184	190	187	189	185	192	183
exp4	129	137	132	135	132	134	145	126
exp5	129	131	131	133	131	131	135	126
exp6	134	139	135	138	135	136	143	130
exp7	117	121	121	122	120	121	125	115
exp8	113	121	116	119	116	118	123	111
exp9	123	122	126	128	126	126	127	120
exp10	142	142	145	148	145	146	146	140
exp11	114	113	117	118	117	116	117	111
exp12	128	130	131	132	130	130	134	125
exp13	128	129	131	132	130	130	133	125
exp14	128	130	131	132	130	130	134	125
exp15	125	140	129	133	128	132	147	122
exp16	127	153	131	139	130	137	162	124
exp17	135	185	141	155	139	153	196	133
exp18	128	137	132	135	131	133	145	126
exp19	128	138	132	135	131	133	145	126
exp20	128	136	132	134	131	132	143	126
exp21	128	130	131	132	130	130	134	125
exp22	128	130	131	132	130	130	133	125
exp23	128	130	131	132	130	130	134	125
exp24	128	130	131	132	131	131	134	125
exp25	129	132	131	133	131	131	135	125
exp26	128	129	131	132	130	130	133	125
exp27	128	129	131	132	130	130	133	125
exp33	128	138	132	135	131	133	145	126
exp34	128	138	132	135	131	133	145	126
exp35	128	138	132	135	131	133	145	126
exp37	129	134	132	134	131	132	137	126
exp38	113	127	117	121	116	120	128	111

Table A.1: Per network synaptic energy results for all experiments in pJ/SOP

	shd1	shd4	smnist3	smnist4	psmnist1	psmnist2	ssc2	ssc3
exp1	15.2	39.9	27.8	50.1	29.9	50.9	36.7	29.8
exp2	5.0	7.6	4.8	6.9	4.9	7.1	6.8	5.3
exp3	3.7	9.7	4.8	9.6	5.1	9.9	7.9	5.1
exp4	8.1	13.8	13.1	20.2	14.4	20.6	12.1	14.3
exp5	15.4	40.1	29.0	51.8	31.1	52.2	37.5	29.8
exp6	19.8	45.5	35.8	61.4	38.5	61.8	43.6	37.3
exp7	25.6	52.7	44.1	72.9	47.5	73.7	50.0	47.1
exp8	38.1	59.7	60.8	92.3	65.3	92.6	61.5	65.5
exp9	29.1	47.8	32.4	49.2	32.1	50.9	46.2	34.1
exp10	17.5	27.7	18.0	26.4	18.0	27.3	25.7	19.2
exp11	43.3	77.4	52.6	84.5	51.3	87.4	75.7	55.4
exp12	15.3	40.6	28.9	51.4	31.0	52.2	37.2	30.4
exp13	15.8	44.0	29.3	53.5	31.4	54.4	39.8	30.9
exp14	15.2	40.1	27.9	50.5	30.0	51.4	36.8	29.9
exp15	10.7	14.8	16.3	23.5	17.9	23.9	13.5	18.0
exp16	12.4	15.1	18.0	24.8	19.9	25.2	13.8	20.1
exp17	12.7	14.4	17.8	24.1	19.8	24.4	13.2	20.2
exp18	8.2	14.0	13.7	21.5	15.2	21.8	12.6	14.8
exp19	8.2	13.9	13.5	21.3	14.9	21.5	12.5	14.5
exp20	8.9	16.2	14.5	23.3	15.9	23.8	14.2	15.5
exp21	15.3	40.7	29.0	51.8	31.0	52.5	37.4	30.4
exp22	15.3	40.8	28.9	51.9	30.9	52.5	38.0	30.4
exp23	15.3	40.7	28.8	51.8	30.8	52.5	37.4	30.3
exp24	14.8	37.4	27.4	48.4	29.5	49.3	34.9	29.4
exp25	14.0	33.2	26.5	45.4	28.5	46.1	31.8	28.5
exp26	15.7	43.9	29.1	53.0	31.2	54.0	39.6	30.8
exp27	15.7	44.0	29.3	53.4	31.3	54.4	39.8	30.8
exp33	8.2	14.0	13.5	21.5	15.0	21.8	12.5	14.6
exp34	8.2	14.1	13.5	21.5	15.0	21.8	12.5	14.7
exp35	8.2	14.1	13.5	21.5	15.0	21.8	12.5	14.7
exp37	12.7	26.9	24.4	39.6	26.4	40.2	26.8	26.4
exp38	48.8	63.4	72.2	103.0	77.4	102.4	66.4	79.0

Table A.2: Per network throughput results for all experiments in MSOP/s/mm^2

	shd1	shd4	smnist3	smnist4	psmnist1	psmnist2	ssc2	ssc3
exp1	119	308	213	379	229	390	274	238
exp2	19	29	18	26	19	27	25	21
exp3	20	52	25	52	27	54	41	28
exp4	63	100	99	150	110	154	84	114
exp5	119	306	221	391	238	398	279	237
exp6	149	327	265	446	286	454	306	288
exp7	218	436	366	597	396	610	401	410
exp8	338	495	522	776	563	787	500	592
exp9	237	391	257	386	255	404	365	283
exp10	123	195	124	179	124	187	176	138
exp11	379	684	451	716	439	750	645	498
exp12	120	314	221	390	238	400	278	244
exp13	123	341	225	406	242	418	299	247
exp14	119	310	213	383	230	394	275	239
exp15	86	106	126	176	140	182	92	147
exp16	98	99	137	179	153	184	85	162
exp17	94	78	126	156	143	160	67	152
exp18	64	102	104	160	116	163	87	117
exp19	64	101	102	158	114	161	86	115
exp20	69	120	110	174	122	180	100	123
exp21	120	314	222	393	238	403	280	243
exp22	120	314	221	394	237	403	285	243
exp23	120	314	221	393	237	402	280	243
exp24	115	287	209	366	226	377	260	235
exp25	109	253	202	342	218	352	235	227
exp26	123	340	223	403	240	415	297	247
exp27	123	341	224	406	241	418	299	247
exp33	64	102	102	160	114	163	87	116
exp34	64	103	102	160	115	164	87	117
exp35	64	103	102	160	115	164	87	117
exp37	98	201	185	297	202	304	195	210
exp38	433	498	618	852	667	856	517	714

Table A.3: Per network EAT results for all experiments in $\mathrm{SOP}^2/(\mathrm{m}^2\mathrm{Js}) \cdot 10^{21}$