Formalizing Neuromorphic Architecture

Nikita Sharma

1 Introduction

As we observe the nearing end of Moore's law and Dennard scaling, new forms of computation are being explored. Neuromorphic architecture [6] is one of these such proposals. Neuromorphic hardware takes inspiration from the brain for its structure. We define this structure in more detail in Section 2.

In this project we set out to formalize a representation of Neuromorhpic architecture in Coq. We demonstrate how some programs for general compute may be executed on this architecture and present several directions for futures work.

The paper is organized as follows. We give background and motivation in Section 2. We describe our formalization of the architecture Section 3. We give a description of several programs we simulate Section 4. Finally, we suggest future work in Section 5 and conclude in Section 6.

2 Background

2.1 Neuromorphic Hardware

Neuromorphic hardware takes inspiration from the brain for its structure. The architecture is composed of "neurons" and "synapses" that are all connected. These components communicate with one another using "spikes". This lends itself to an event driven structure where large components of the hardware are left idle, leaving us with very small power utilization.

In terms of hardware, several neuromorphic architecture implementations use memristors [3] because of their capabilities to remember and store value even without power. This allows us to move further away from CMOS and traditional von Neumann designs. This architecture aims to eliminate the division between compute and memory and instead combine them together with memoruy being intrinsicaly represented in this neuron/synapse architecture.

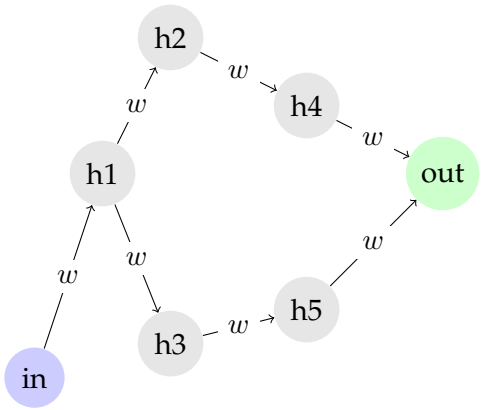

Fig. 1. A sample configuration that has one input neuron, in, and one output neuron, out. We also show five hidden neurons that are internally computing. Each edge is also corresponded to some weight.

Neuromorphic architectures are generally being used to implement spiking nerual networks (SNNs). However, it has been shown that this architecture can be used to compute general graph algorithms [8] [9] and it is even turing-complete [4].

To program this architecture, we provide a list of neurons and all of the synapses that connect them. Each of these neurons has an associated potential and a fixed threshold value. When potential exceeds threshold, the neuron fires all of its outgoing synapses and resets its own potential to 0. The connected neurons add the weight of the connecting synapse to their own potential. This allows values to propagate across the graph [10]. We show an example configuration in Figure 1.

2.2 Motivation

Neuromorphic architecture provides a framework for computation that is incredibly scale-able and and has inherently low power consumption. The hardware is idle until it is spiked by an incoming synapse or a firing neuron. This means that the large areas of the hardware may remain idle at any

given time. Further, we can combine several Neuromorphic chips as they naturally combine to work together. Combining chips allows us to configure larger graphs [2] [1].

3 Formalization

We provide a formalization of this architecture and demonstrate how we might run programs with it in Coq.[1] In order to do this, we set some limitations on the parameters discussed in Section 2:

1) Every neuron has a default potential of 0
2) Synapses can have integer weights and neurons can have integer potentials
3) Every synapse has a delay of exactly one timestep

Further, we explore the idea of leak in the neurons. Leak is a configurable parameter that says if, and how long it takes for, a neruon's potential to leak back to it's default value. We consider both implementations with leak and no leak, described respectively in Sections 3.1 and 3.2.

The first thing we define in this architecture is what a neuron is. This definition allows us to specify a neurons threshold. Taken with inspiration from our homeworks, we represent the current potential for each neuron by a list of pairs. We assume that a neuron not in this list has a default potential of 0.

```
Definition potentials :=
        list (Neuron * Z).
```

Finally, we represent our graph as a list of synapses between two neurons with an integer weight.

```
Definition graph :=
        list (Neuron * Neuron * Z).
```

The combination of the list of current neuron potentials and list of all synapses in the graph define the current state of our architecture. It gives us the current values stored in the hardware and all of the connections that define our graph.[2]

```
Inductive state : Type :=
    pair: potentials -> graph -> state.
```

We define how a program can step from one step to the next in the next two sections.

1. All code is available at https://github.com/nsharma1231/cs345h-final

2. It is important to note a neuron that is disjoint from the rest of the graph (i.e. a singleton graph within the larger graph), is pointless in this architecture. It's behavior will have no impact nor will any other part of the graph have an impact on it. For this reason, it is okay that are state only considers neurons that are connected to one another without requiring a list of all nodes.

3.1 Leak

When we simulate a hardware configuration in which a neuron leaks its potential every timestep, we have every neuron reset its potential to 0 across every call to step. To step the architecture one timestep, we recurse over our list of synapses and, if the neuron the synapse originates from has a potential greater than or equal to its threshold, it has "fired", and we add the weight of the synapse to the potential of the neuron the synapse ends at. We ensure that the neuron that just fired resets its potential to the baseline.

3.2 No Leak

With no leak, our step semantics look nearly identical to the ones listed in Section 3.1 with a few additions to account for potentials persisting across timesteps. When recursing over our list of synapses, we check if the neuron the synapse originates from should not fire, and make sure we add that value to our resulting potential list, ensuring we don't double count potentials from neurons with multiple outgoing synapses.

4 Simulation

We simulate several examples of using a neuromorphic programming scheme to illustrate how this architecture might be used for general compute. We adapt the algorithms from J. S. Plank et. al. [7] to work with the assumptions we laid out in Section 3. Our graph of neurons and synapses to compute the and of two numbers (with leak) is shown in Figure 2. Here is how we write this program in Coq:

```
Definition in1 := neuron "A" 1.
Definition in2 := neuron "B" 1.
Definition out := neuron "A&B" 2.
Definition synapses := [(in1, out, 1);
                        (in2, out, 1)].

Example computes_in_one_step:
    forall a b,
    eval 1 (initial_state a b) out =
                               andb a b.
Proof. destruct a, b; auto. Qed.
```

We specify each of our three neurons and define the connections between them. We then run the program for one cycle and read our result from the output neuron. We encode inputs and outputs as

$$(potential \geq threshold) \iff value = true$$

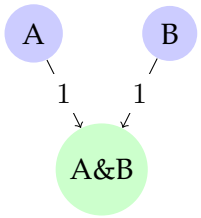

Fig. 2. A graph that computes the and of two numbers. In this circuit, the node "A&B" has a threshold of 2 and "A" and "B" have threshold 1.

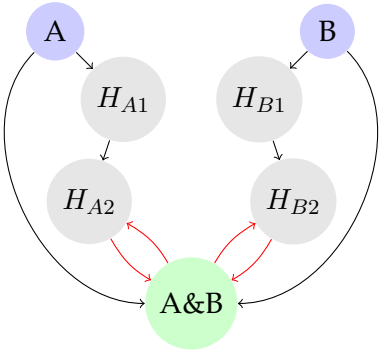

Fig. 3. Every neuron except "A&B" has threshold 1. "A&B" has threshold 2. The red edges indicate a synapse with weight -1 and the remaining black edges indicate a synapse with weight 1.

For this and circuit, we are able to observe the result in one timestep.

Unfortunately, if we use this same configuration in an environment where potential does not leak, we would run into problems with erroneous false positives with subsequent queries (i.e. $(1,0),(0,1)$). A graph that correctly addresses this problem with an environment that does not leak is shown in Figure 3.

For brevity, we omit the rest of the circuits that we implement as they provide no insight beyond the circuits we have already presented. We provide implementations in both a leak and no leak environment for and, or, and xor.

5 Future Work

5.1 Architectural Properties

One thing we attempted to do was prove properties about the architecture. However, this proved to be incredible difficult. For example, let's consider a lemma that is attempting to show the step function is indeed firing all neurons whose potential exceeds their threshold. We initially tried to show that, if a neuron is connected to an outgoing synapse from a neuron that will fire in the next step, then the resulting neuron should end up with a potential greater than or equal to the weight of the synapse. Quickly, we realized this was incorrect – as we have negative weights and potentials, this theorem might not be true.

We leave for future work identifying and proving intrinsic properties of this architecture that hold true for arbitrary graph configurations.

5.2 Compiler

With the work we have presented in this project, we provide the building blocks to create a compiler for a language that evaluates boolean expressions. The properties of this architecture we demonstrate in this project that allow for boolean arithmetic lends itself extremely well to an extension of conditional evaluation.

5.3 Performance Guarantees

Perhaps most relevant to the motivations behind this architecture, would be to prove how power consumption of neuromorphic circuitry compares to traditional von Neumann style circuitry. Prior work in this area has been discussed [1], but is severely limited and we have not found any published papers that have done this work.

Resources for verifying hardware in Coq [11] exist and have established support for traditional CMOS circuits. However, most neuromorphic hardware implementations are moving away from traditional CMOS circuitry and instead exploring the usage of components such as memristors and RRAM [2] [5] [3]. This means that getting any proof of correctness guarantees would involve having to find a way to represent the circuitry.

This is the area for the most work to be done. Being able to formalize power consumption would be incredibly helpful in understanding what workloads could benefit from running on Neuromorphic hardware.

6 Conclusion

This was an incredibly fun project to work on and to see what all could be represented in Coq. There are several exciting different paths that this project can take on. In this project we show that there is a way to represent this language in Coq in a way that makes it (relatively) easy to configure and use. Currently, Neuromophic architecture has very limited use cases confined to SNNs. We hope that with further work it can be shown that this architecture might be a viable option for more general compute problems as well.

REFERENCES

[1] James Bradley Aimone, Yang Ho, Ojas D. Parekh, Cynthia A. Phillips, Ali Pinar, William Mark Severa, and Yipu Wang. Brief announcement: Provable neuromorphic advantages for computing constrained shortest paths.

[2] Gangotree Chakma, Md Musabbir Adnan, Austin R. Wyer, Ryan Weiss, Catherine D. Schuman, and Garrett S. Rose. Memristive mixed-signal neuromorphic systems: Energy-efficient learning at the circuit-level. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(1):125–136, 2018.

[3] L. Chua. Memristor-the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, 1971.

[4] Prasanna Date, Thomas Potok, Catherine Schuman, and Bill Kay. Neuromorphic computing is turing-complete. In *Proceedings of the International Conference on Neuromorphic Systems 2022*, ICONS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Matthew Kay Fei Lee, Yingnan Cui, Thannirmalai Somu, Tao Luo, Jun Zhou, Wai Teng Tang, Weng-Fai Wong, and Rick Siow Mong Goh. A system-level simulator for rram-based neuromorphic computing chips. *ACM Trans. Archit. Code Optim.*, 15(4), jan 2019.

[6] Robert A. Nawrocki, Richard M. Voyles, and Sean E. Shaheen. A mini review of neuromorphic architectures and implementations. *IEEE Transactions on Electron Devices*, 63(10):3819–3829, 2016.

[7] James Plank, Chaohui Zheng, Catherine Schuman, and Christopher Dean. Spiking neuromorphic networks for binary tasks. In *International Conference on Neuromorphic Systems 2021*, ICONS 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Catherine D. Schuman, Kathleen Hamilton, Tiffany Mintz, Md Musabbir Adnan, Bon Woong Ku, Sung-Kyu Lim, and Garrett S. Rose. Shortest path and neighborhood subgraph extraction on a spiking memristive neuromorphic implementation. In *Proceedings of the 7th Annual Neuro-Inspired Computational Elements Workshop*, NICE '19, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Catherine D. Schuman, Shruti R. Kulkarni, Maryam Parsa, J. Parker Mitchell, Prasanna Date, and Bill Kay. Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science*, 2(1):10–19, Jan 2022.

[10] Catherine D. Schuman, James S. Plank, Garrett S. Rose, Gangotree Chakma, Austin Wyer, Grant Bruer, and Nouamane Laanait. A programming framework for neuromorphic systems with emerging technologies. In *Proceedings of the 4th ACM International Conference on Nanoscale Computing and Communication*, NanoCom '17, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Lixue Xia, Boxun Li, Tianqi Tang, Peng Gu, Xiling Yin, Wenqin Huangfu, Pai-Yu Chen, Shimeng Yu, Yu Cao, Yu Wang, Yuan Xie, and Huazhong Yang. Mnsim: Simulation platform for memristor-based neuromorphic computing system. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 469–474, 2016.