Solving LASSO Problem for Sparse Encoding in Asynchronous Spiking Neural Networks

Alex Roseman
Department of Physics
Yale University
New Haven, CT
alex.roseman@yale.edu

Kenan Erol
Department of EECS
Yale University
New Haven, CT
kenan.erol@yale.edu

Siva Nalabothu
Department of Computer Science
Yale University
New Haven, CT
siva.nalabothu@yale.edu

Abrar Sheikh
Department of Electrical Engineering
Yale University
New Haven, CT
abrar.sheikh@yale.edu

Eytan Israel
Department of EECS
Yale University
New Haven, CT
eytan.israel@yale.edu

Luis Zuñiga Velásquez
Department of Chemical Engineering
Yale University
New Haven, CT
luis.zuniga@yale.edu

***Abstract*—In this paper, we present the design and partial implementation of an asynchronous spiking neural network (SNN) to solve the Constrained LASSO (C-LASSO) sparse encoding problem. The architecture would have been realized using a TSMC 180nm process and a central control unit that communicates with neurons through input and output channels. We implemented the spiking Locally Competitive Algorithm (S-LCA) to achieve sparse coding and validated the design at multiple levels.**

Testing demonstrated correct behavior at the CHP and decomposition levels, where spike outputs matched theoretical expectations. However, when mapping the design to production rules, discrepancies arose after initial outputs, prompting further debugging. To address these issues, we explored modifications, including an offset-based CHP implementation to prevent negative values and a dataflow-based re-implementation. Despite ongoing challenges, the project establishes a robust foundation for solving sparse optimization problems efficiently with asynchronous SNNs.

***Index Terms*—Spiking Neural Networks, Locally Competitive Algorithm, l_1-minimizing sparse coding problem**

I. Introduction

Sparse coding is a fundamental optimization problem where a signal $x \in \mathbb{R}^n$ is represented as a linear combination of sparse dictionary elements from an overcomplete basis $\Phi \in \mathbb{R}^{n \times m}$, with $m > n$. This is formulated mathematically as the Least Absolute Shrinkage and Selection Operator (LASSO) problem:

$$\underset{\alpha}{\text{minimize}} \quad \frac{1}{2}\|x - \Phi\alpha\|_2^2 + \lambda\|\alpha\|_1, \tag{1}$$

where $\alpha \in \mathbb{R}^m$ is the coefficient vector, and $\lambda > 0$ is a regularization parameter balancing reconstruction accuracy and sparsity. In applications such as feature selection and signal processing, constraints can be added to enforce non-negativity on the coefficients α. This leads to the Constrained LASSO (C-LASSO) problem [4]:

$$\underset{\alpha \geq 0}{\text{minimize}} \quad \frac{1}{2}\|x - \Phi\alpha\|_2^2 + \lambda\|\alpha\|_1. \tag{2}$$

A. The Locally Competitive Algorithm (LCA)

The LASSO problem can be mapped to the Locally Competitive Algorithm (LCA), which computes sparse representations using a network of competing neurons. This algorithm has been extended for implementation on neuromorphic hardware, where local competition enforces sparsity efficiently [2], [4]. The LCA dynamics are described as follows:

$$\frac{du_i}{dt} = b_i - u_i - \sum_{j \neq i} w_{ij} a_j, \quad a_i = T_\lambda(u_i), \tag{3}$$

where:

- u_i is the membrane potential of neuron i,
- $b_i = \Phi_i^T x$ represents the feedforward input,
- $w_{ij} = \Phi_i^T \Phi_j$ defines the inhibitory weights between neurons,
- a_i is the output activity governed by the soft-thresholding operator T_λ.

For the C-LASSO problem, the soft-thresholding operator T_λ is further restricted to enforce non-negativity by setting $T_\lambda(u) = \max(0, u - \lambda)$ [4].

B. Spiking Neural Network Implementation

The LCA can be implemented using Spiking Neural Networks (SNNs), which leverage discrete spike events instead of continuous values to communicate information. This approach gives rise to the Spiking Locally Competitive Algorithm (S-LCA), which efficiently solves the C-LASSO problem in an energy-efficient manner [1]. In SNNs, neurons operate using the leaky integrate-and-fire (LIF) model [3], with dynamics governed by:

$$\frac{d\mu_i}{dt} = b_i - \mu_i - \sum_{j \neq i} w_{ij} a_j(t), \tag{4}$$

$$v_i(t) = \int_0^t (\mu_i(s) - \lambda) ds, \quad v_i(t) \geq V_{\text{th}} \Rightarrow a_i(t) = 1 \text{ (spike)}, \tag{5}$$

$$a_i(t) = 1 \Rightarrow v_i(t) \leftarrow 0; \mu_j(t) \leftarrow \mu_j(t) - w_{ij}\alpha(t - t_{\text{spike}}), \tag{6}$$

$$\mu_i(t = 0) = b_i, \tag{7}$$

$$v_i(t = 0) = 0, \tag{8}$$

where:

- μ_i is the soma current of neuron i,
- v_i is the membrane potential, which integrates input current over time,
- λ is the bias current,
- w_{ij} defines the inhibitory weights,
- $a_j(t)$ represents the spike trains of neuron j,
- $\alpha(t) = e^{-t}$ for $t \geq 0$ and 0 otherwise, controlling the spike decay,
- V_{th} is the firing threshold.

To digitalize the problem, we make several changes to the continuous equations above. First, we discretize time in units of time constant τ [5]. v and μ no longer represent physical voltages and currents, but are state-holding variables that update their internal variables every timestep. When a spike happens at step t, the neurons have just finished their calculations for $v(t)$. They must incorporate the spike into their calculation for $\mu(t)$ because there is no held state across timesteps except for v and μ. It should also be noted that μ is computed after checking for a spike. So $\mu(t)$ and $v(t)$ evolve according to the following equations:

$$v_i(t+1) = v_i(t) + \tau(\mu_i - \lambda), \tag{9}$$

$$v_i(t) \geq V_{\text{th}} \Rightarrow a_i(t) = 1, \tag{10}$$

$$\mu_i(t+1) = \mu_i(t) + \tau(b_i - \mu_i(t)) - \sum_{j \neq i} w_{ij} a_j(t+1) \tag{11}$$

When a neuron spikes, its potential resets, and it sends inhibitory signals to competing neurons, enforcing sparsity. Over time, the spike rates of the neurons converge to the solution of the C-LASSO problem.

C. Our Implementation

In this project, we worked towards implementing a solution to the C-LASSO problem using the Spiking Locally Competitive Algorithm (S-LCA) on an asynchronous chip. Building upon the S-LCA framework, we designed the structure in CHP for an architecture of B bits and N neurons, and implement a prototype for a 32-bit system with 3 neurons, corresponding to one of the examples provided by Tang et al. [4]. The CHP description of the design was converted into production rules to enable its implementation using asynchronous circuits with the ACT language tools. After verifying these production rules using IRSIM and Xyce, the next step would be synthesizing the design into standard cells using the TSMC 180nm process.

II. Architecture

To implement the required equations in a circuit-friendly way with minimal multiply and divide operations, we modified our variables in the following way:

$$v_i^{''}(t+1) = \tau^{-2} v_i, \tag{12}$$

$$V_{th}^{''} = \tau^{-2} V_{th}, \tag{13}$$

$$\mu' = \tau^{-1}\mu, \tag{14}$$

$$\lambda' = \tau^{-1}\lambda, \tag{15}$$

$$w'_{ij} = \tau^{-1} w_{ij}, \tag{16}$$

$$\tau^* = -\log_2(\tau), \tag{17}$$

$$\mu'_i(0) = \tau^{-1} b_i \tag{18}$$

Using this change of variables, the equations (9)-(11) can be implemented without multiplication or division, thus considerably simplifying the design. The rearranged equations are shown below:

$$v''_i(t+1) = v''_i(t) + (\mu'_i(t) - \lambda'), \tag{19}$$

$$v''_i(t+1) \geq V''_{th} \Rightarrow v''_i(t+1) \leftarrow 0; a_i(t+1) = 1, \tag{20}$$

$$v''_i(t+1) < V''_{th} \Rightarrow a_i(t+1) = 0, \tag{21}$$

$$\mu'_i(t+1) = \mu'_i(t) + b_i - (\mu_i(t) >> \tau^*) - \Sigma_{j \neq i} w'_{ij} a_j(t+1), \tag{22}$$

$$a_j(t) = 1 \Rightarrow \mu_i(t+1) = \mu_i(t) - w_{ij}, \tag{23}$$

$$a_j(t) = 1 \wedge \mu_i(t) < \frac{V_{th}}{2} \Rightarrow \mu_i(t+1) = \frac{V_{th}}{2}, \tag{24}$$

$$\mu'_i(0) = \tau^{-1} b_i \tag{25}$$

With equations 19-25 being implemented on each neuron, we created a central unit that distributed global constants and spike information to all of the neurons. This reduced the number of channels required from $O(n^2)$ in an architecture where every neuron is connected to $O(n)$ channels.

The global constants loaded onto the chip go through an Arduino interface and upload $\lambda', \tau^*, V''_{Th}$ respectively. Once these global constants are sent from the control unit to every neuron, then b_i and w_{ij} are read and sent to the neurons. All of these values were generated using `/scripts/EENG_426_S_LCA_I_O.ipynb`.

We created three separate implementations of the asynchronous spiking neural network to solve the LASSO problem: CHP, CHP with additional positive offset, and dataflow.

A. CHP and CHP with Offset

In the regular CHP implementation, we have a control unit with N neurons. The control unit has an input channel connected to off-chip to receive the weights, and an output channel connected to off-chip to tell first which neuron spiked, and at what time step that neuron spiked.

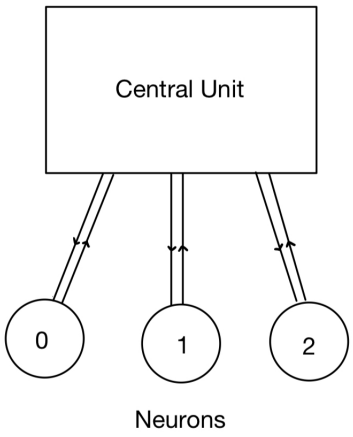

Fig. 1. Illustration of the central unit and neurons for the 32 bits, 3 neurons implementation. As shown in the image, each neuron connects to the central unit via two channels. Channels between neurons not shown on the image.

The control unit then has one 1-bit input (neuron to control) and one 32-bit output (control to neuron) channel per neuron. The global constants and global spike information are sent through the control-to-neuron channel. The neuron-to-control channels are 1-bit to tell the control unit whether the neuron has spiked or not.

In each neuron, spikes are calculated according to equations 19-25. In the CHP with additional offset, equations 19, 20, 22, 24, 25 respectively become

$$v_i''(t+1) = v_i''(t) - 2^{31} + (\mu_i'(t) - \lambda'), \tag{26}$$

$$v_i''(t+1) \geq V_{th}'' \Rightarrow v_i''(t+1) \leftarrow 2^{31}; a_i(t+1) = 1, \tag{27}$$

$$\mu_i'(t+1) = \mu_i'(t+1) + (2^{31} >> \tau^*) \tag{28}$$

$$a_j(t) = 1 \wedge \mu_i(t) < \frac{V_{th}}{2} \Rightarrow \mu_i = (1 + 3 * 2^{32} - 2V_{th}), \tag{29}$$

$$\mu_i'(0) = \tau^{-1} b_i + 2^{31} \tag{30}$$

B. Dataflow

We used the `dataflow` sublanguage to create various dataflow components, which were in turn combined to build the LCA. There are 14 of these components in total.

Component Summaries:

const

The `const` process generates a constant output stream on channel `Y` by initially routing the input channel `A` through internal control channels, then sends 1 forever. A control signal ensures proper handshaking and propagation.

min, max, smax

These processes compute the minimum and maximum value between two inputs `A` and `B` using splits.

count components

The `count,count_to,count_to_3` process generates a sequence of incrementing integers $(0, 1, 2, ...)$ and outputs them continuously on channel `Y`. `count_to` counts until a pint Q, and `count_to_3` counts to 3. The counter is implemented using internal addition, feedback loops, splits, and merges.

count_loop

The `count_loop` process generates a repeating sequence of integers $(0, 1, ..., Q)$. Once the counter reaches `Q`, it resets back to 0 and continues.

count_loop_1

The `count_loop_1` process is an optimized version of `count_loop` for a single-bit counter using less dataflow components.

count_repeat

The `count_repeat` process generates a sequence where each integer $x \in \{0, 1\}$ is repeated `R` times before incrementing.

count_repeat_to

The `count_repeat_to` process outputs a sequence where each integer is repeated `R` times. Once the counter reaches a specified maximum value `Q`, the value `Q` is output repeatedly.

neuron_connector

The `neuron_connector` process routes a predefined series of inputs (`lambda`, `tau`, `Vth`, and `const`) to the output channel `out`. After this sequence, the input `j` is repeatedly sent to the output indefinitely. Merges and control signals determine the order of routing.

n_copy

The `n_copy` process takes a single input and sequentially copies it to `N` output channels using a loop counter.

splitter

The `splitter` process serializes an `N`-bit input into `M`-bit chunks and outputs them one at a time, starting with the least significant chunk. This process assumes that `N` is a multiple of `M`. A loop counter and right shifts manage the serialization. This component is needed because the Arduino micro-controller (which we are using to send values to the chip) has 16 bit outputs, while our architecture runs on 32 bit.

merger

The `merger` process reconstructs a larger `M`-bit output from multiple `N`-bit inputs, where `M` is a multiple of `N`. It performs concatenation and accumulation on the smaller chunks. This component is needed because the Arduino micro-controller (which we are using to send values to the chip) has 16 bit outputs, while our architecture runs on 32 bit.

III. Off Chip Implementation

Arduino Code Summary

The chip is designed to interface with an Arduino, which manages the four-phase handshake and communication process. On the user side, a Python program allows the user to input data, run computations, and generate bit strings that the Arduino transmits to the chip. The Arduino then synchronizes the chip's output and sends it bit-by-bit to the Python program, which subsequently extracts the timestamp and neuron that spiked.

The user interacts with the chip through a Python interface, where they enter $\overrightarrow{y}, \Phi, V_t h, \lambda_P$, and $\tau^* = -\log_2 \tau$. Note that all of the inputs are integers. Also note that offset is internally set to 2^{31} due to it being exactly halfway- this allows "negative" and "positive" numbers to be represented within the circuit.

The Python code then computes $\overrightarrow{b} = \overrightarrow{y} \bullet \Phi$(assuming $\overrightarrow{y}$ is a horizontal matrix) and $W = \Phi \bullet \Phi^T$. It then converts all quantities to 32-bit binary strings (MSB first) and sends these to the Arduino. At this point, the Python waits for further communication from the Arduino, and becomes passive.

The Arduino code is structured into three main stages: reading data from Python (the stage just discussed), sending data to the chip, and reading data from the chip.

In the second stage, the Arduino then sends these values to the chip 16 bits at a time, internally breaking up the 32-bit quantities, and subsequently enters the third (and final phase). Once the chip enters the run phase, whenever a neuron spikes, the chip outputs a time-stamp and then the neuron id (with the chip breaking up the 32-bit quantities into 16-bit chunks). The Arduino reads these via the parallel-to-serial shift registers and serially sends these bits to the Python code for processing. On the user end, the Python stops updating after a user-defined number of spikes is recorded.

Pin Configuration

The following pins are initialized for communication:

- `dataOutPin` (Pin 13): Serial data output to the chip.
- `latchPin` (Pin 12): Register latch pin.
- `clockPin` (Pin 11): Shift register clock pin.
- `SH` (Pin 8): Shift control pin.
- `outCLK` (Pin 7): Output clock pin.
- `datafromChip` (Pin 6): Data input from the chip.
- `toChipReq` (Pin 10): Request line to the chip.
- `toChipAck` (Pin 3): Acknowledgment pin from the chip (for sending input to the chip- sent from chip).
- `fromChipReq` (Pin 2): Request line from the chip.
- `fromChipAck` (Pin 4): Acknowledgment pin to the chip (for receiving output data from chip- sent from Arduino).

Variables and States

Several variables are defined to store input data, process states, and chip communication status. The variables include arrays for different neuron parameters, such as `tauStar`, `lambdaP`, `vth`, `b`, and `w`. The code uses flags to track the current process state and handle data reading or writing operations.

Setup

The `setup()` function initializes serial communication, configures pin modes, and attaches interrupts for handling acknowledgment signals (for input into the chip) and data send requests (for output) from the chip.

Data Communication Functions

- **sendDataToChip:** This function sends 16 bits of data to the chip, either in MSB-first or LSB-first order, depending on the `msb_first` flag. It uses the `shiftOut` function to transfer the data bit by bit to the chip.
- **handleAck:** This interrupt service routine handles acknowledgment signals from the chip. It updates the `ackReceived` flag and manages the request lines for data transfer.
- **load_data_from_chip:** This interrupt service routine is called when the chip requests to send data. It samples bits from the chip 16 at a time and shifts them out to the Arduino.

Loop Function

The `loop()` function manages the different process states:

- **State 0 (Read from Python):** The Arduino reads data from the serial input, including variable names and values, and stores them in the corresponding arrays. Upon receiving the "Done" signal, it transitions to state 1.
- **State 1 (Send Data to Chip):** The Arduino sends data to the chip in 16-bit chunks. It sends values such as τ^*, λ', and neuron parameters. The data is sent in the specified bit order (MSB or LSB first).
- **State 2 (Read from Chip):** When the Arduino receives data from the chip, it triggers the `load_data_from_chip` function, which reads and shifts out the bits from the chip.

Interrupt Handling

Interrupts are used to manage the acknowledgment signals from the chip (`toChipAck`) and the request signals from the chip (`fromChipReq`). These interrupts help coordinate data transfer between the (synchronous) Arduino and the (asynchronous) chip.

Circuitry setup

Note that the circuit uses 3.3V logic in order to be compatible with the chip, Arduino, and the shift registers, without the need for logic converters. Both Arduino and the shift registers are relatively flexible, but the chip (being TSMC 180nm technology) cannot operate at a more typical 5V.

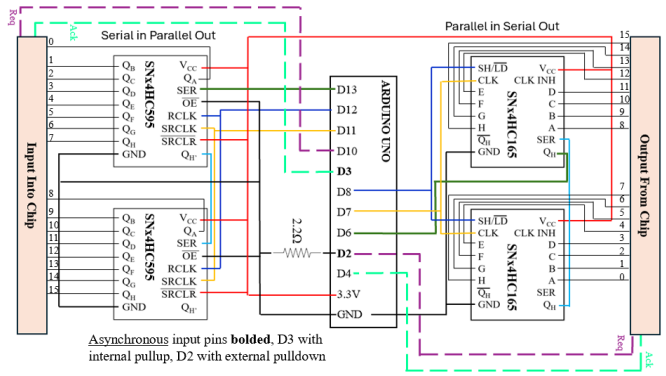

Fig. 2. Circuitry implementation for the chip (tan boxes). The specific serial-in parallel-out and parallel-in serial-out shift registers listed are shown as examples of commonly used parts that would be compatible with the chip- any other shift registers with similar internal logic and compatibility at 3.3V logic would suffice. Pins D3 and D2 are used for asynchronous input as these are the only pins on the Arduino Uno able to implement interrupts.

IV. Verification

We carried out testing at various stages of development to ensure that the SNN implementation matched the expected results from the referenced paper [4] (shown in Figure 2). This involved simulations and tests for the three different implementations—Python, CHP and CHP with offset, and Dataflow. For the ACT specific implemetations, we also conducted testing for their respective production rules outputs.

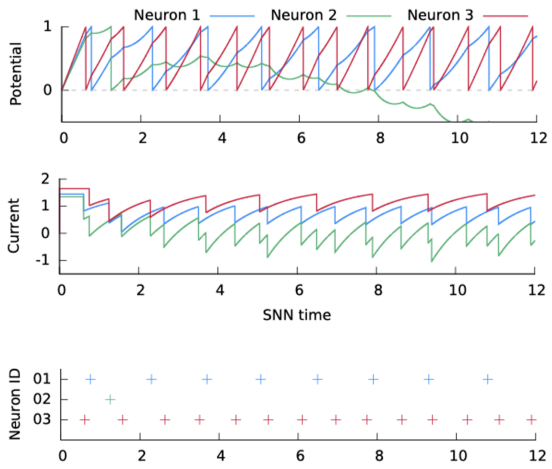

Fig. 3. Results from the 3 neuron, 32 bit implementation of the SNN in the paper by Tang et al., [4]

A. Jupyter Notebook Initial Validation

The initial implementation was done in Python as a baseline. This version of the spiking locally competitive algorithm (S-LCA) enabled us to simulate the behavior of the neurons precisely, observing all aspects of the dynamics:

- Spike timing $a_i(t)$,
- Membrane potentials $v_i''(t)$,
- Current dynamics $\mu_i'(t)$.

Python's implementation served as the gold standard because of its transparency and ability to simulate continuous behavior. As shown below, the results matched the ones published by Tang et al., [4]

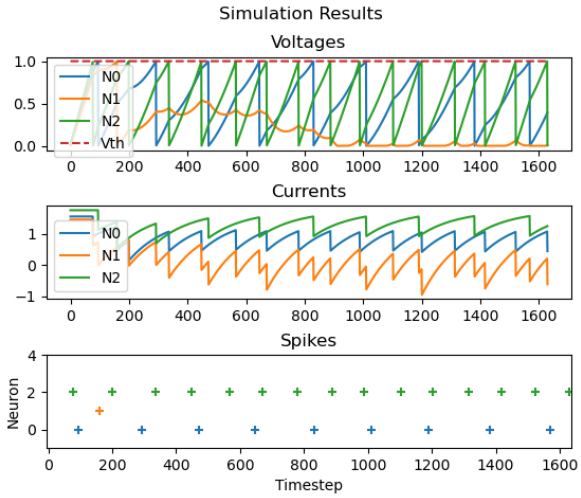

Fig. 4. Implementation of the three-neuron spiking neural network in Python. This simulation allowed us to observe all relevant aspects of the neural dynamics.

B. CHP Verification

The CHP implementation was tested using the ACT toolkit with `actsim`. A structured testing methodology was followed:

- Inputs: Precomputed values for $\lambda', \tau^*, V_{th}''$, and w_{ij}', generated using Python scripts.
- Outputs: Spike signals and neuron states were logged and compared with the Python results and reference outputs from Figure 2.

We designed the CHP (and chip in general) so that it only sends out time step and spikes so the results only show the spikes based on the time step for each neuron. As shown below, the results obtained for the CHP level testing matched those of the Python implementation and [4].

We decomposed the CHP with the `decomp` tool from `synth2` and obtained the same results as the ones we show above. We then used `ring` to get a file with the production rules but when testing the file, the results did not match that of the CHP or decomp levels. An interesting observation from the testing of the production rules file was that the first four values did match those of CHP and decomp but after that the values diverged. From this point, we parallelized our efforts to get to a working production rules file we could use for the next steps for the tape out. We attempted to re-implement the project with dataflow (results discussed in the next section) and re-write the CHP with an offset to avoid negative values as those might have been causing the problem. The overall results are summarized in the table in the Results and Comparison section.

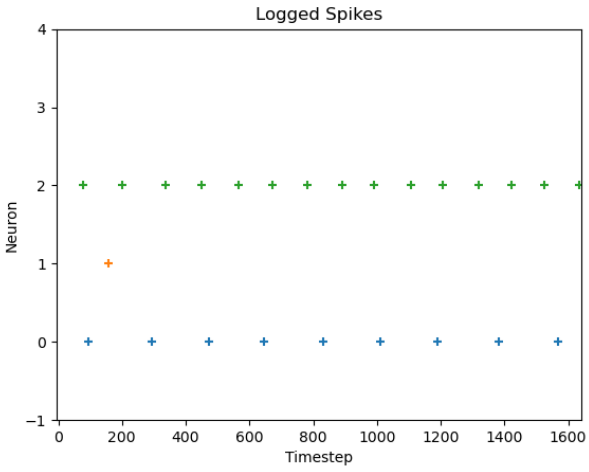

Fig. 5. Number of neuron that spiked at specific time points in the CHP test. The results match that of the paper by Tang et. al. in Fig. 3 and 4

C. Dataflow Verification

The dataflow implementation was tested using a structured methodology similar to the CHP verification. Testing was carried out at multiple levels:

- `test_neuron`: Simulated individual neuron behavior using predefined spike trains and weight configurations.
- `test_control`: Handled the routing of global constants and parameters to neurons.
- `test_chip`: Integrated full system testing, including I/O interfaces and communication channels.

Inputs were initialized through dedicated data channels while outputs—spike signals and timestep information were routed and verified. The results obtained demonstrated correct communication and behavior of the neurons in the dataflow architecture.

D. Results and Comparison

Below we have summarized the main results from our multiple efforts to implement the SNN with asynchronous circuits (and the Python standard).

TABLE I
SUMMARY OF TESTING RESULTS

Implementation	Testing Level	Result
CHP	CHP	Correct
	Decomp	Correct
	Prs	Incorrect values after 4th value
CHP with Offset	CHP	Correct
	Decomp	Correct
	Prs	Incorrect values
Dataflow	Dataflow	Correct
	Prs	No outputs
Python	N/A	Correct

[a]Correct: All outputs match expected results.
[b]Incorrect values after 4th value: Deviation starts after the 4th output.
[c]Incorrect values: Results do not match expected outputs.

For some reason we are still figuring out why our implementations keep failing at the production rules level. This has been very puzzling as we have tried significantly different approaches related to implementing the SNN and all of the approaches work at the higher level of abstraction. We know that the problem lies in the translation of such abstraction into low level rules but do not know whether the bug or bugs lie primarily on our design or the way such design is translated (or both). As a result, we have tried to iterate on our designs as well as to reach out to Karthi and Professor Manohar so they can look at the ACT-related tools on their end. Our hypothesis at the moment of submission is that reading in multiple values from a single channel leads to problems with the ring synthesis. For the CHP implementation specifically, we believe there is some sort of overflow happening on neuron 1 as it keeps spiking every time step after time step 361, so fixing this issue will fix the circuit.

A general summary of the tools used to translate our design into production rules is shown below:

TABLE II
SUMMARY OF TOOLS USED TO COMPILE DESIGNS

Implementation	ACT Tool
CHP	`decomp`
	`ring`
CHP with Offset	`decomp`
	`ring`
Dataflow	`dflowmap`

V. SYNTHESIS AND NEXT STEPS

We used ring synthesis to generate production rules with CHP [6]. For dataflow, we used dflowmap and netlist generation. Unfortunately at the time of submission, the prs for CHP, CHP with offset, and dataflow did not work as their implementations at a higher level of abstraction. We will figure out why this is the case before the tapeout deadline. Once there is a set of prs that work with either dataflow or CHP, we will implement the rect files with routing friendly cell layout.

BUGS IDENTIFIED IN ACT TOOLS

During our testing and analysis of the `maelstrom`, `ring`, and `dflowmap` functions, we identified several issues that required attention. These bugs, along with their descriptions and status of their resolutions, are outlined below:

1) **Non-Exclusive Guards in the `decomp` Function:** We identified an error in the `decomp` function of Maelstrom where multiple guards were generated with the same conditions but produced separate outputs. This behavior was incorrect, as the guards are intended to be mutually exclusive. This led to failures when trying to run `ring` on the `decomp` output. Karthi was able to resolve this issue and fix it.
2) **Handling of Internal Loops:** The Maelstrom tool encountered issues when parsing internal loops. Specifically, it failed to process them correctly, leading to

improper behavior. Sometimes this caused problems in running `ring` on the `decomp` output and other times while `ring` was able to execute, the output of running the code was different. Karthi identified the root cause and implemented a fix to address this problem.

3) **Handling of Negative Numbers:** As CHP also does not handle negative integers, we decided to implement a system where the bits would get added and subtracted normally, and we would use our own comparator (instead of $<$ or $>$) to compare the values using two's compliment. While this worked on the CHP and `decomp` level, it did not work with `ring`. The `ring` had problems related to bitwidth and other issues which made us need to implement the offset.
4) **Variable Initialization in Maelstrom:** When trying to convert our CHP to PRS, the `decomp` would output a file that needed to be edited before it could be used for `ring` to ensure that every variable is initialized before it is used in a function, and that every initialization is done with a constant instead of making it equal to another variable. While this is an easy fix, it disrupts the flow of the process and takes a significant amount of time when it can be added to the automation. Furthermore, we assumed that every variable that was not already initialized was supposed to be set to 0, yet this might also be a reason for our incorrect outputs. Karthi let us know that he is aware of the issue and is working on a fix.
5) **Segmentation Fault on Left Shift Operation:** We (accidentally) discovered that performing a left shift operation (<<<) by 2^{32} bits caused a segmentation fault. This issue was reported to Karthi for further investigation and resolution.
6) **Incorrect Behavior from `ring` and `dflowmap` PRS Outputs:** As noted earlier, we were able to obtain the intended outputs when testing the CHP-level code, as well as the code processed through Maelstrom's `decomp` function. However, when running Maelstrom's `ring` function, the PRS generated by it gave incorrect outputs when we ran actsim. Similarly, while dataflow tested successfully and produced correct results, using `dflowmap` to convert dataflow to PRS resulted incorrect outputs. At this stage, it is unclear whether the issue lies in our code or in the Maelstrom and `dflowmap` commands. Further investigation is required to pinpoint and resolve the problem.

ACKNOWLEDGMENT

We sincerely thank Congyang Li for guiding us at the outset by recommending key papers to read. We are also deeply grateful to Karthi Srinivasan for invaluable assistance with debugging circuit decomposition, PRS production, and CHP compilation. Our gratitude extends to Professor Rajit Manohar for teaching us the theory in class and for his guidance on dataflow and other issues during office hours. Lastly, we thank Matt Dobre for sharing his final versions of the routed cells.

REFERENCES

[1] M. Davies et. al., "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," in IEEE Micro, vol. 38, no. 1, January 2018.
[2] S. Shapero, C. Rozell, P. Hasler, "Configurable hardware integrate and fire neurons for sparse approximation", in Neural Networks, vol. 45, pp. 134-143, September 2013.
[3] M. Davies, et al., "Advancing neuromorphic computing with loihi: A survey of results and outlook." in Proceedings of the IEEE, vol. 109, no. 5, pp. 911-934. 2021.
[4] P. T. P. Tang, T. H. Lin, M. Davies, "Sparse coding by spiking neural networks: Convergence theory and computational results." arXiv preprint arXiv:1705.05475 (2017).
[5] K. L. Fair, D. R. Mendat, A. G. Andreou, C. J. Rozell, J. Romberg, D. V. Anderson,"Sparse Coding Using the Locally Competitive Algorithm on the TrueNorth Neurosynaptic System," in Frontiers in neuroscience, vol. 13, 754. https://doi.org/10.3389/fnins.2019.00754
[6] K. Srinivasan and R. Manohar, "Maelstrom: A Logic Synthesis Technique for Asynchronous Circuits," unpublished manuscript, submitted for publication, 2024.