Programming Scalable Neuromorphic Algorithms with Fugu

Brad Aimone

Center for Computing Research
Sandia National Laboratories

jbaimon@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

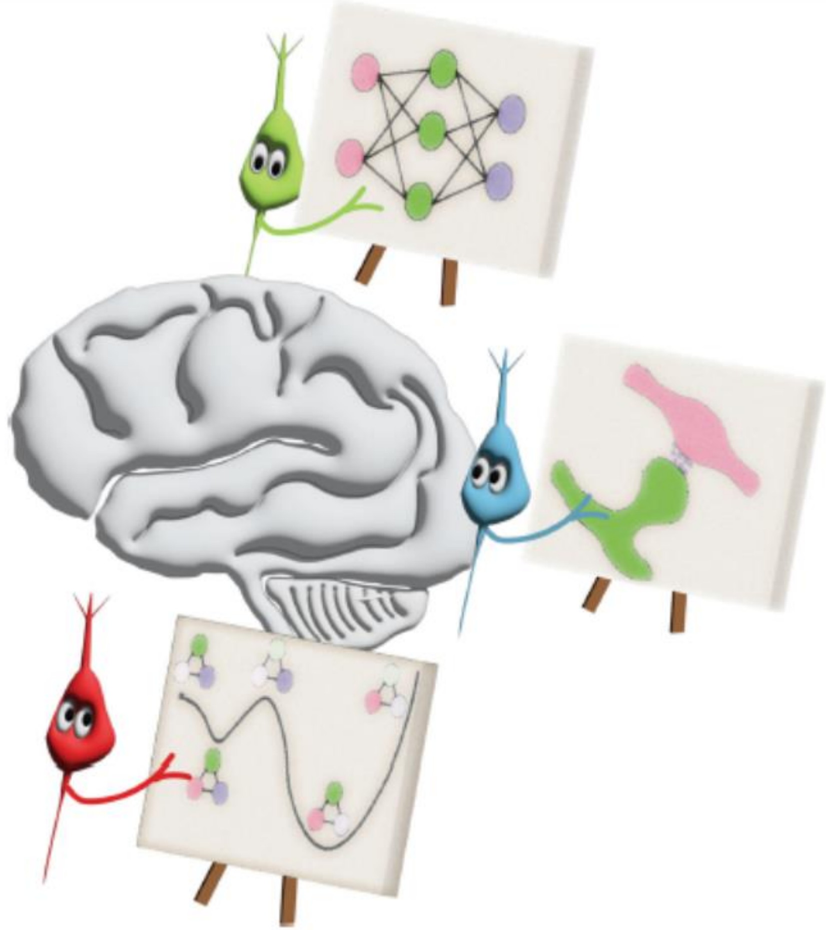
SAND2023-14932PE

Thank You!

Fugu
- William Severa, Craig Vineyard, Srideep Musuvathy, Yang Ho, Leah Reeder, Michael Krygier, Fred Rothganger, Suma Cardwell, Ingrid Lane, Aaron Hill, Zubin Kane, Sarah Luca, …

STACS, N2A, and Neural Simulations
- Felix Wang, Fred Rothganger, Brad Theilman, …

Broader Sandia Neuromorphic Algorithms Team
- Darby Smith, Ojas Parekh, Rich Lehoucq, Frances Chance, Corinne Teeter, Mark Plagge, Ryan Dellana, Shashank Misra, Conrad James, Chris Allemang, Brady Taylor, Yipu Wang, William Chapman, Efrain Gonzalez, James Boyle, Cale Crowder, Clarissa Reyes, Cindy Phillips, Ali Pinar, …

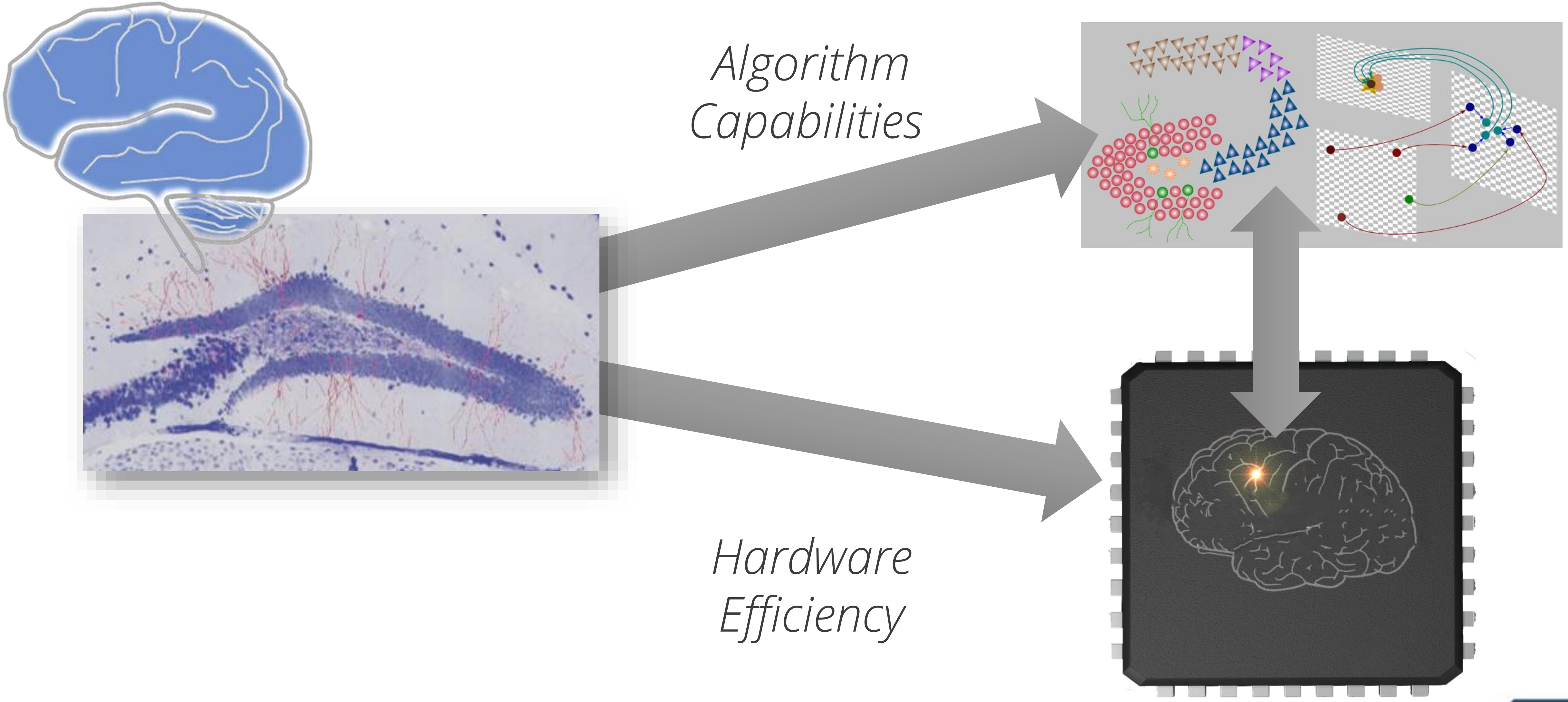


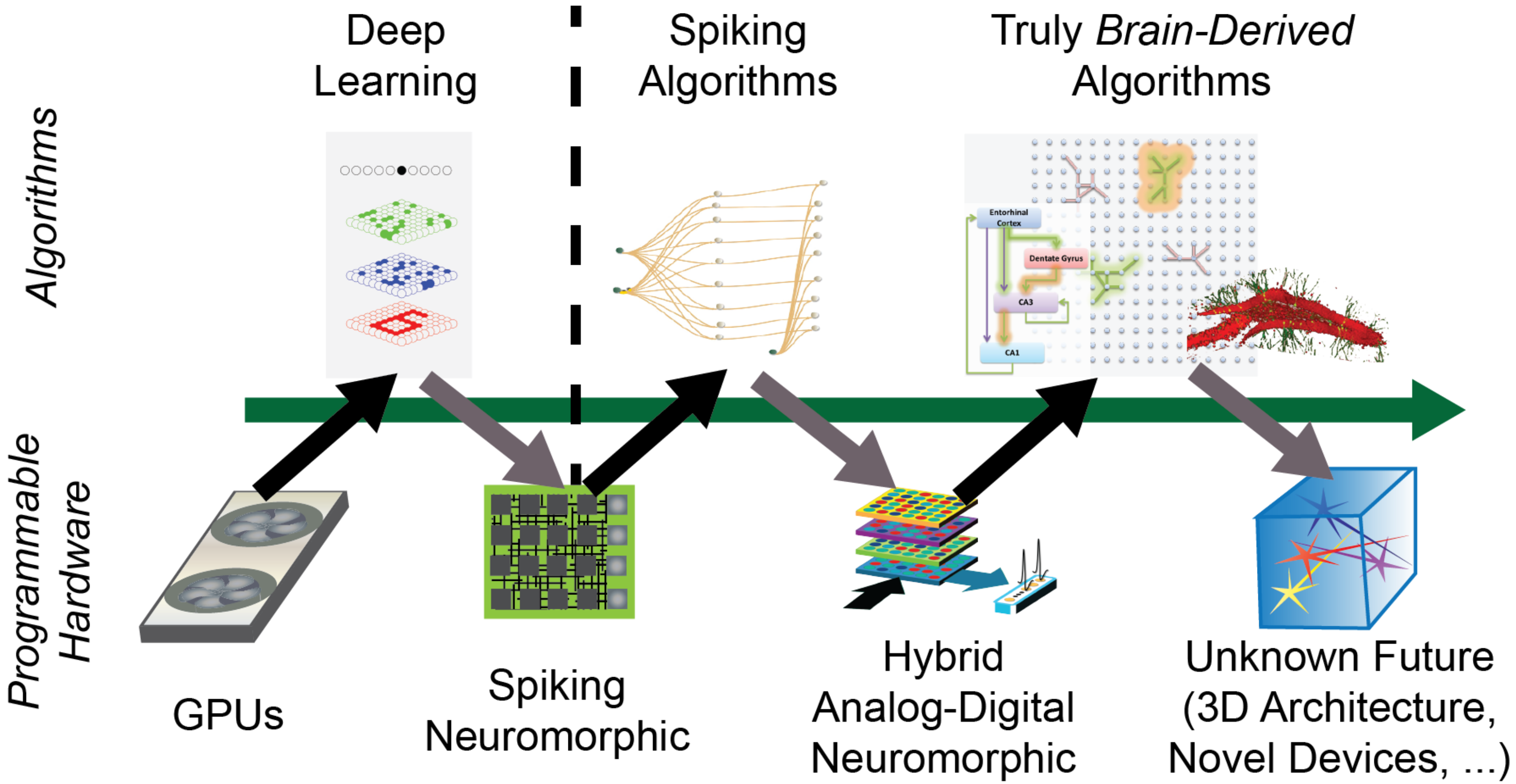
U.S. DEPARTMENT OF ENERGY | Office of Science

jbaimon@sandia.gov

Sandia National Laboratories

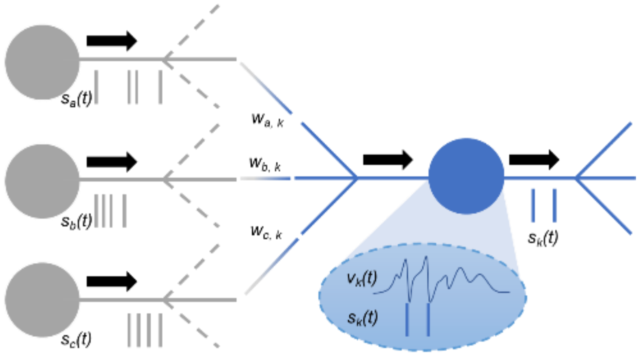
ASC

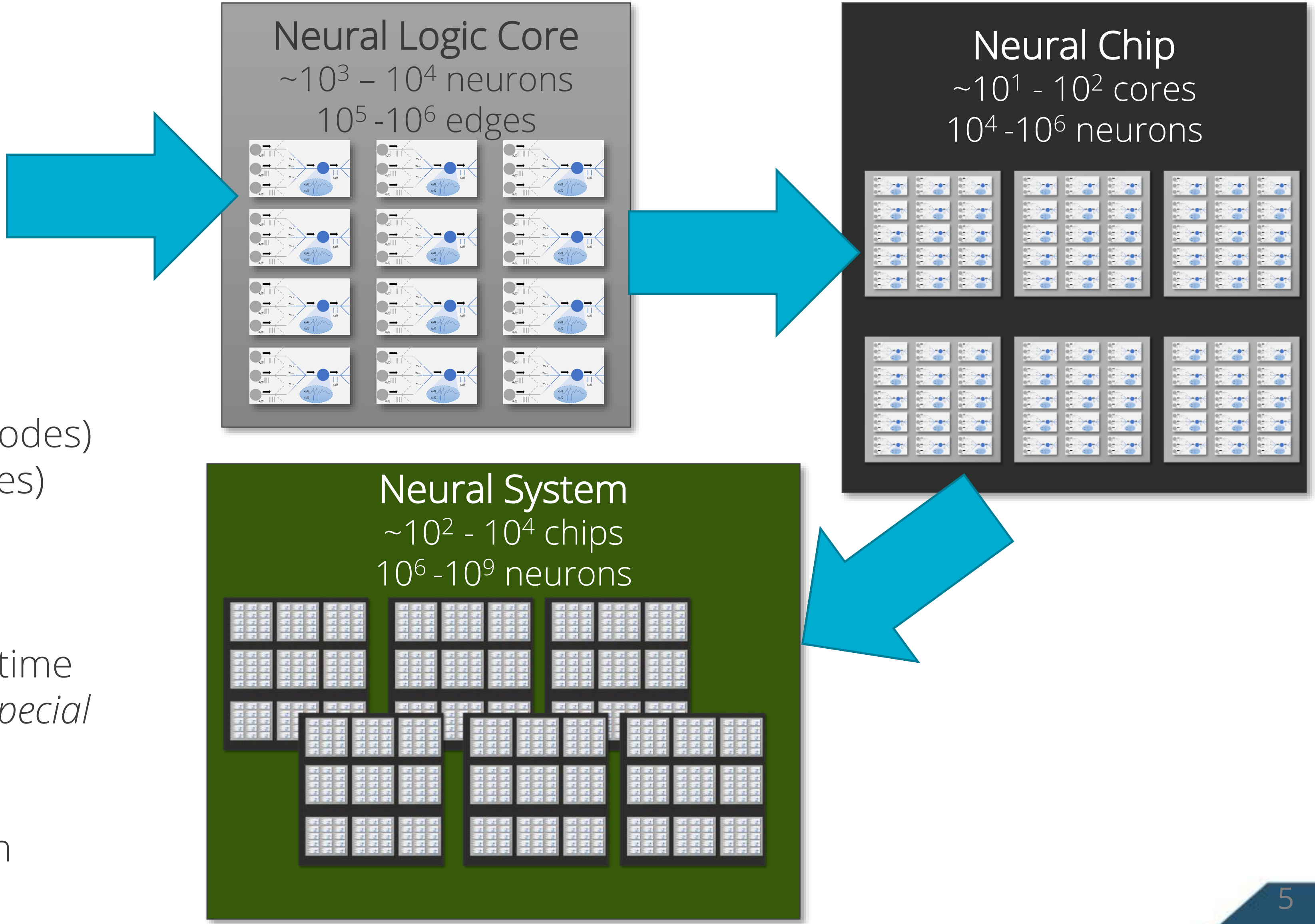

Algorithm Capabilities
Hardware Efficiency



Aimone JB, Advanced Intelligent Systems, 2023

Neuromorphic computing today





Computational Primitives:

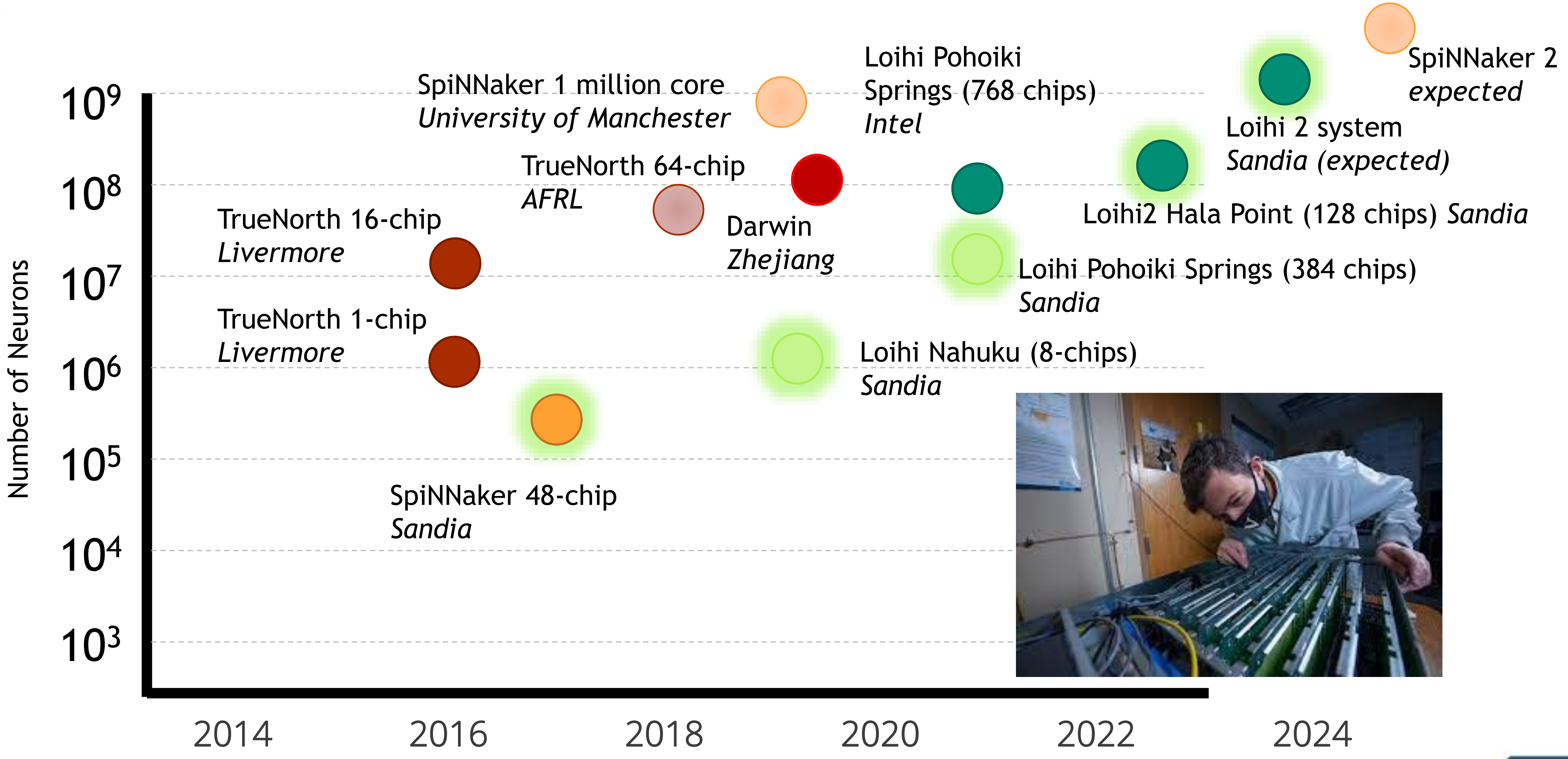
Spiking Neurons (vertices / nodes)

Synapses (connections / edges)

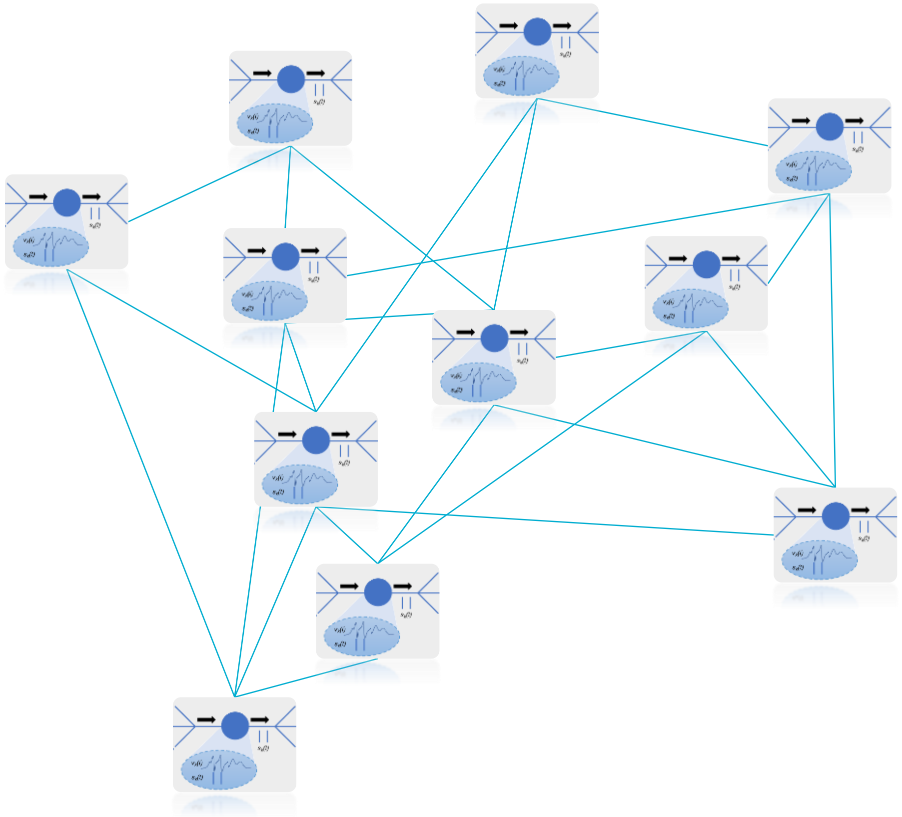
Programmable as arbitrary graphs

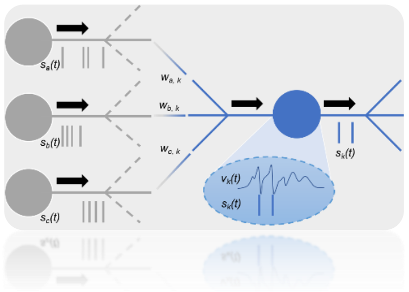
- Edges: Directed and weighted
- Nodes: Threshold gate logic + time
- *Artificial neural networks are a special case*
- Programmability, theoretical, analysis and software are open research questions

Sandia has some of the largest spiking neuromorphic systems



Neuromorphic hardware jumped ahead of the rest of the stack



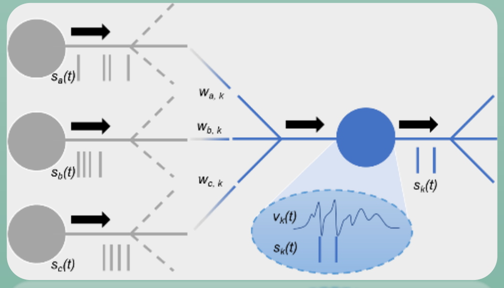


We need

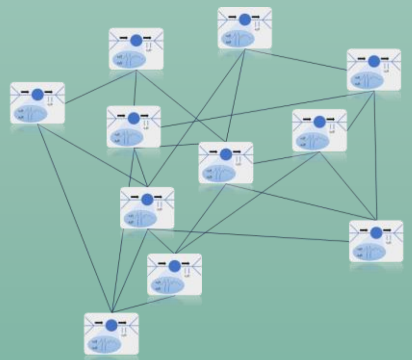
- Driving Applications
- Systems Interface
- Software and Programming Paradigm
- Theoretical Framework

A quick aside: most neuromorphic hardware is *not* designed for artificial neural networks

Neuromorphic Hardware



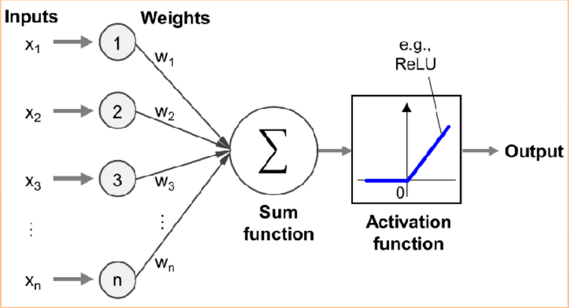
Spiking neurons



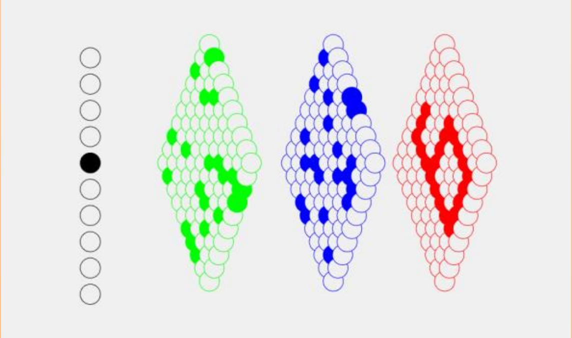
Arbitrary connectivity

- Continual learning integrated into operation
- Inherently temporal
- Dynamical tasks?

Artificial Neural Networks



Continuous neurons



Linear algebra-like networks

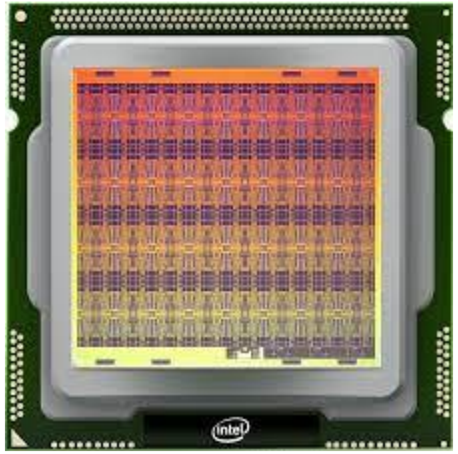
- Distinct training and inference modes
- Time is largely avoided
- Computer vision and natural language processing

Modern Artificial Intelligence

https://openai.com/research/overview

The Brain

Neuromorphic Hardware

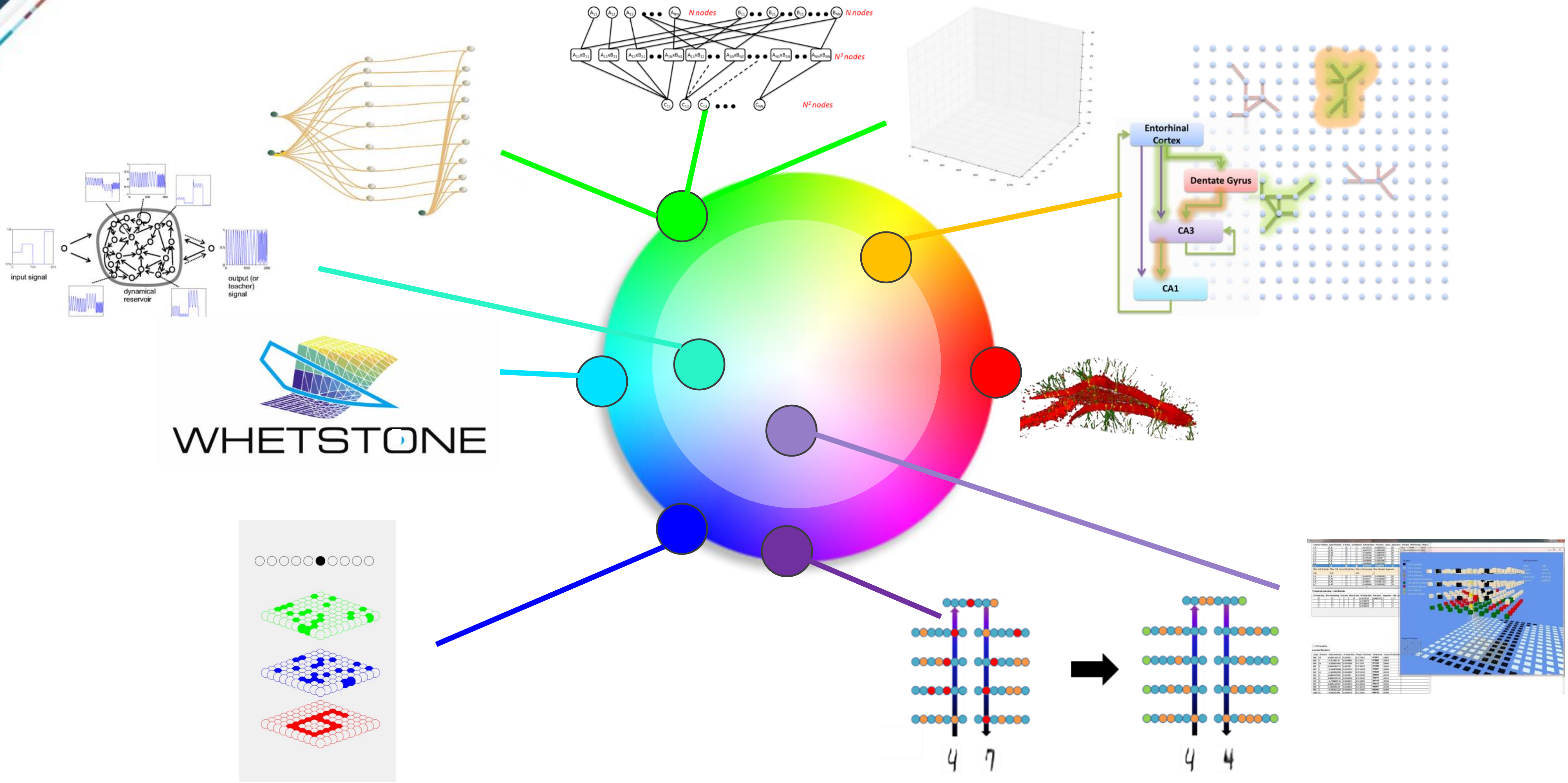


Intel Loihi Chip

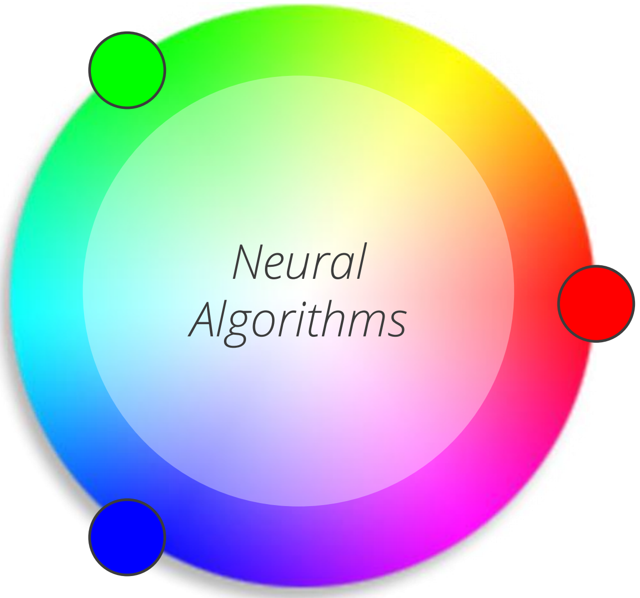
All of these have a lot of neurons...

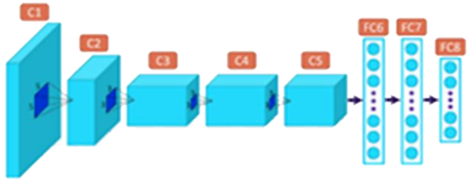
... clearly these are not all equivalent in what they do

As a community, we all mean different things when we talk about neural algorithms…



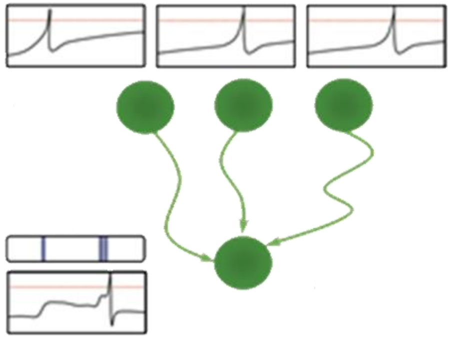
Different classes of neural algorithms have received varied attention

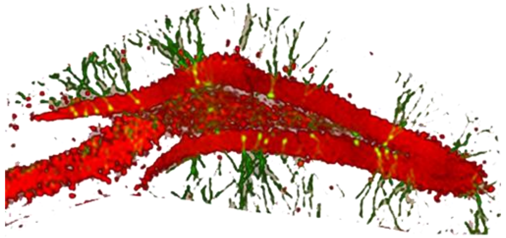




Artificial neural networks
- Generic layers of non-linear nodes
- SGD optimization of weights
- Powerful machine learning capabilities through learning sequential non-linear mappings and function approximation

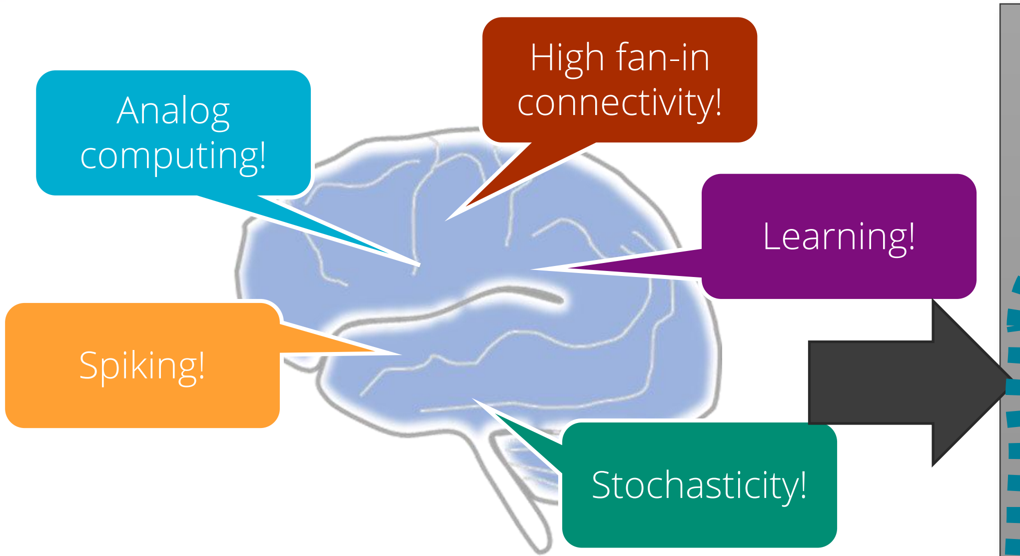
Spiking neural algorithms
- Hand-crafted circuits of spiking neurons
- Model of parallel computation
- Energy efficiency through event-driven communication and high fan-in logic





Neuroscience-constrained algorithms
- Circuit architecture based on local and regional neural connectivity
- Computation incorporates broad range of neural plasticity and dynamics
- *Generally still unexplored from algorithms perspective*

… meanwhile, hardware is rapidly evolving and scaling

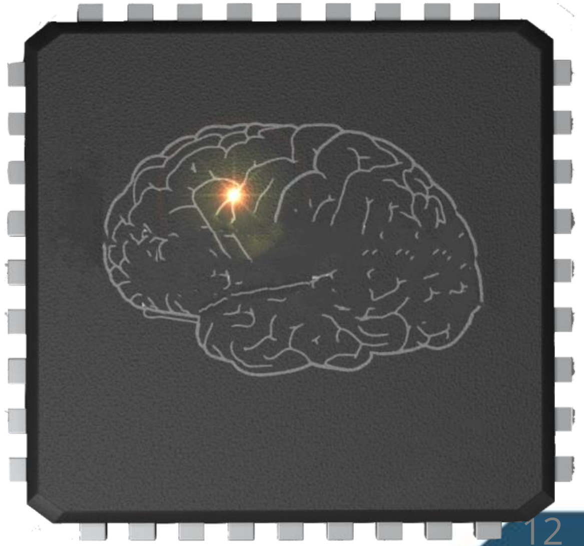


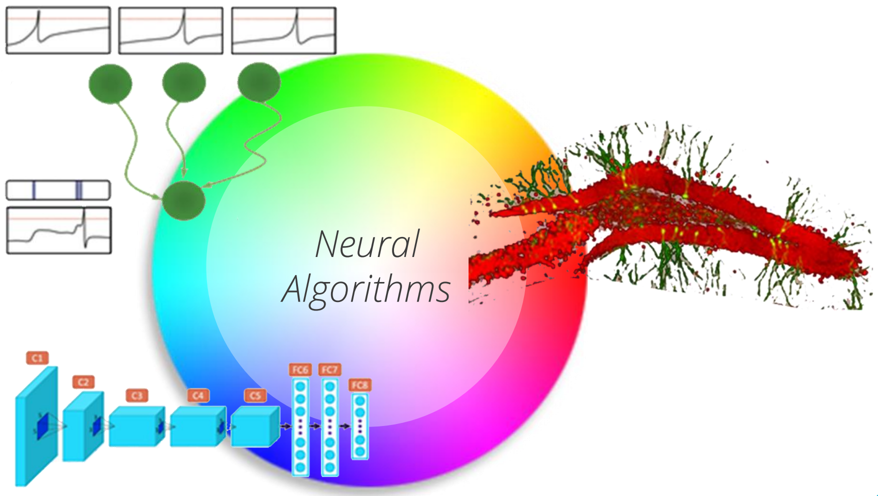
Realized Features of Brain Inspiration in Neuromorphic Hardware

- Event-driven communication
- Graph based connectivity
- Processing in Memory
- In situ learning
- Analog computation
- Post-Moore's Law Devices
- Ubiquitous stochasticity

Realized Today for 10x-100x energy savings

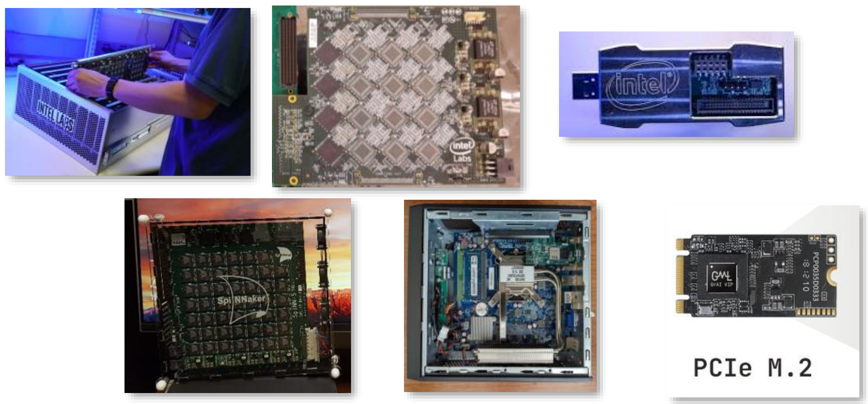
Active Research for additional power savings



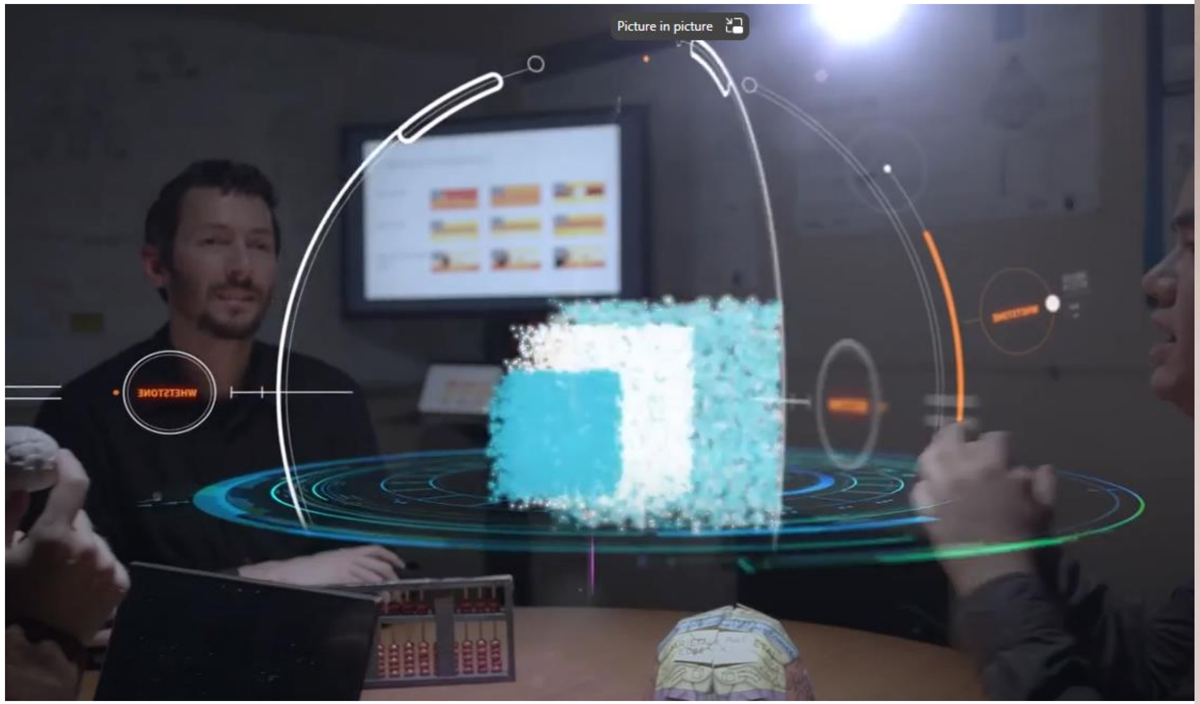


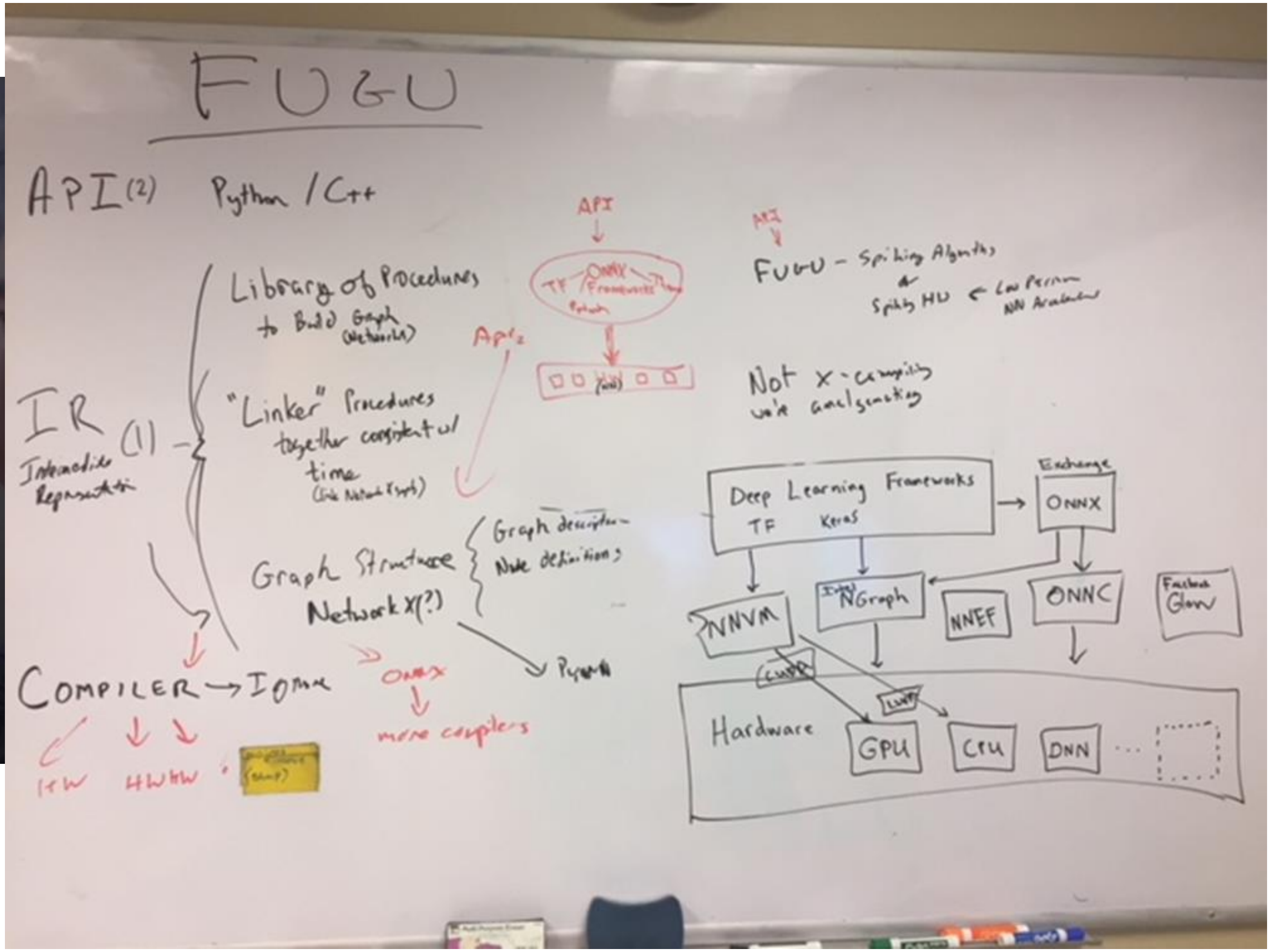
Rapidly evolving and diverse set of algorithms

How do we work across both of these?

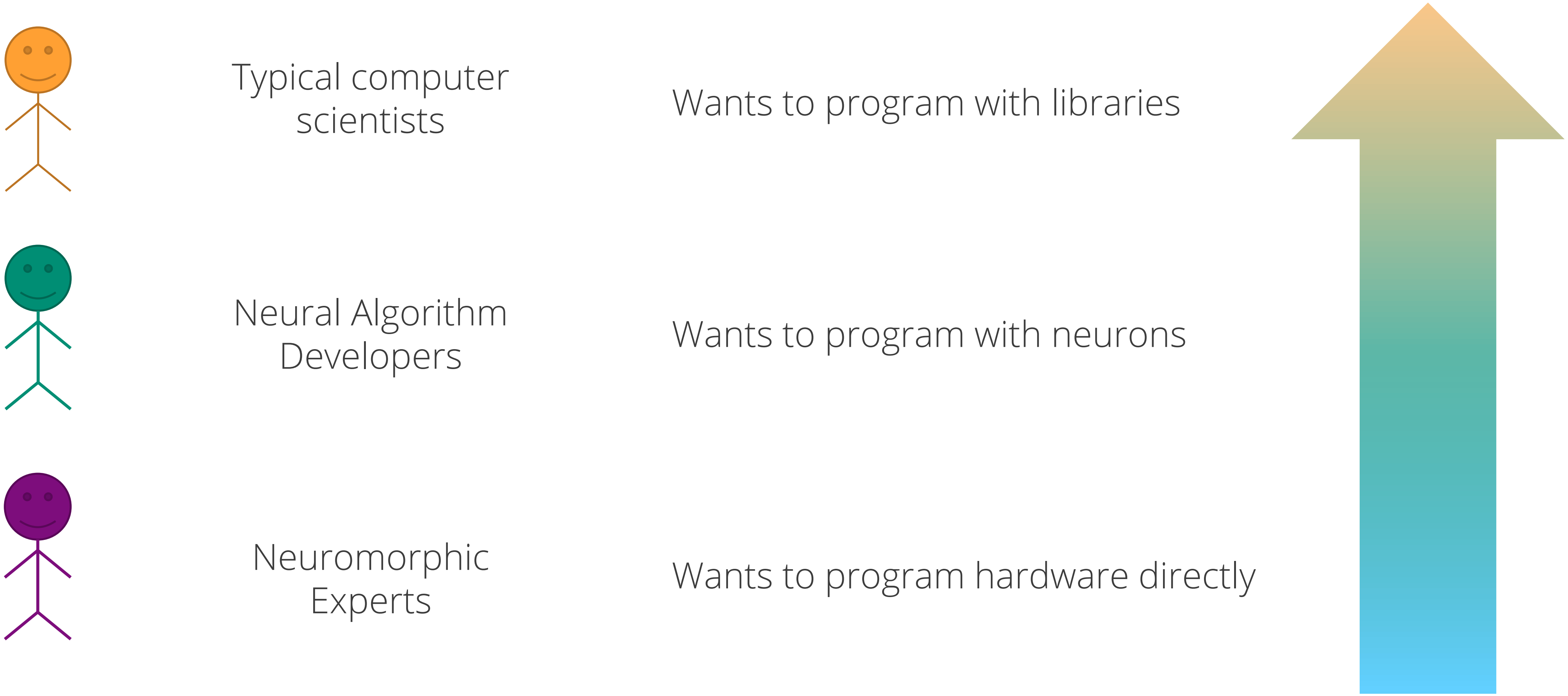


Rapidly evolving and diverse set of hardware


Picture in picture
WHETSTONE


FUGU
API (2)
Python / C++
Library of Procedures
to Build Graph
(Network)
IR (1)
Intermediate Representation
"Linker" Procedures
together consistent w/
time
Graph Structure
Network X (?)
Graph description
Node definitions
COMPILER
API
ONNX
more compilers
PyNN
FUGU - Spiking Algorithms
Spiking HW
Not x-compiling
Deep Learning Frameworks
TF
Keras
Exchange
ONNX
NNVM
NGraph
NNEF
ONNC
Glow
Hardware
GPU
CPU
DNN

Fugu aims to bring neuromorphic solutions to general computing world



How to Interact with Fugu

We are always looking for collaborators!

If you use Fugu for research, please cite our ICONS paper: Aimone, Severa, Vineyard. *Composing neural algorithms with Fugu*, 2019.

- End User - Works with bricks, scaffolds and backends
  - Should **clone** Fugu repository and import fugu.
  - New code should go in its own project-specific repository
- Brick/Backend Builder - Creates new Bricks/Backends for End User
  - If the code is generally applicable, create a feature **branch** from Fugu, write code, **merge request.** Recommended to e-mail wg-fugu@sandia.gov to coordinate and collaborate first.
  - If the code is project-specific or sensitive, create your own repository and inherit from Brick / Backend
- Core Fugu - Modifies Core parts of Fugu
  - Create a feature **branch** from Fugu, create code, **merge request.** Recommended to e-mail wg-fugu@sandia.gov to coordinate and collaborate first.
  - Large suggestions/collaborations will require discussions with wg-fugu@sandia.gov

What is Fugu? And Why?

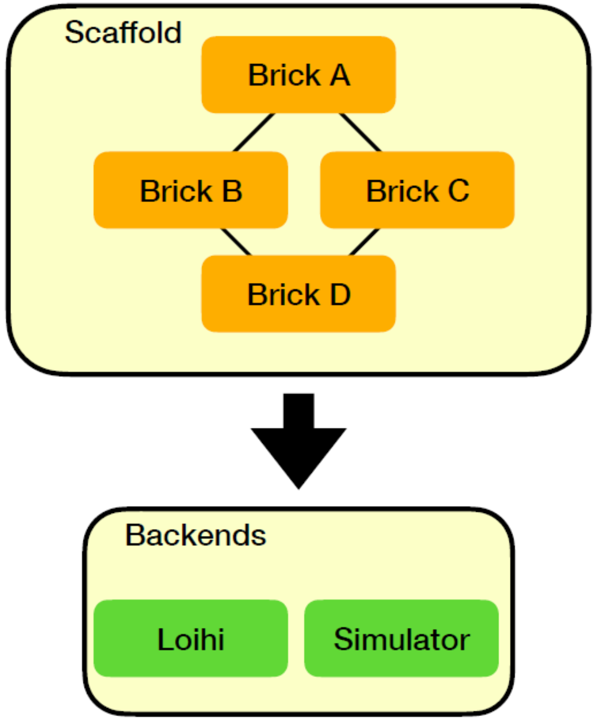
Neuromorphic Challenges

- Neuromorphic platforms remain a challenge to program
- Lack of interoperability between research outputs

Fugu

- Open-source library for spiking neural networks
- A unified, (mostly) hardware agnostic, framework to enable neuromorphic algorithm development
  - Bricks: roughly represents a function
  - Scaffolds: represents an application
- Design goals: easy-to-use, lower barrier of entry, improved code efficiency and re-use
- In active development

—F U G U—



The name Fugu is inspired by the Japanese word for pufferfish; which, of course, have spikes. Furthermore, Fugu is considered a culinary delicacy due to the presence of low levels of the neurotoxin tetrodotoxin, or TTX, which has significant value in studying the electrophysiology mechanisms underlying biological action potentials.

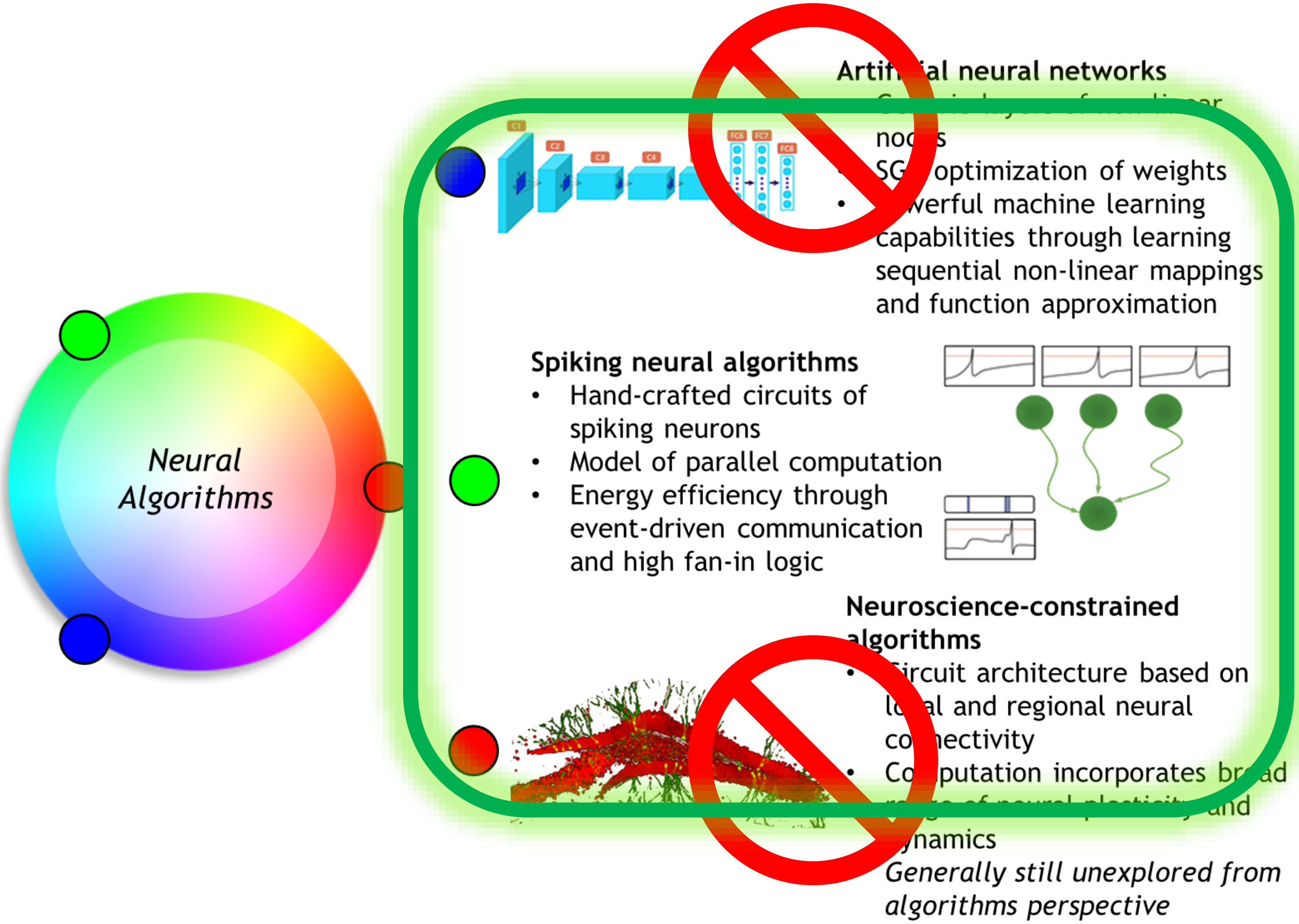
What Fugu is *not*

- Fugu is *not* a deep learning or spiking neural network training tool
  - Fugu *can* leverage outputs of SNN tools as bricks in a computation

- Fugu is *not* a neurobiology modeling tool
  - Fugu *can* leverage outputs of SNN tools as bricks in a computation

- Fugu is *not* a replacement for hardware-specific software infrastructure (e.g., Lava)

- Fugu *IS* an intermediate representation that allows development of explicit scalable, parallelizable algorithms for neuromorphic systems



How Fugu Works

Scaffold

Sequence Classifier

Bricks are similar to functions or dataflow blocks

Bricks

- Input
- Pooling
- Mean
- Convolution
- Dot
- Linear
- Liquid State Machine
- Sparse Coding

Four main steps for Fugu

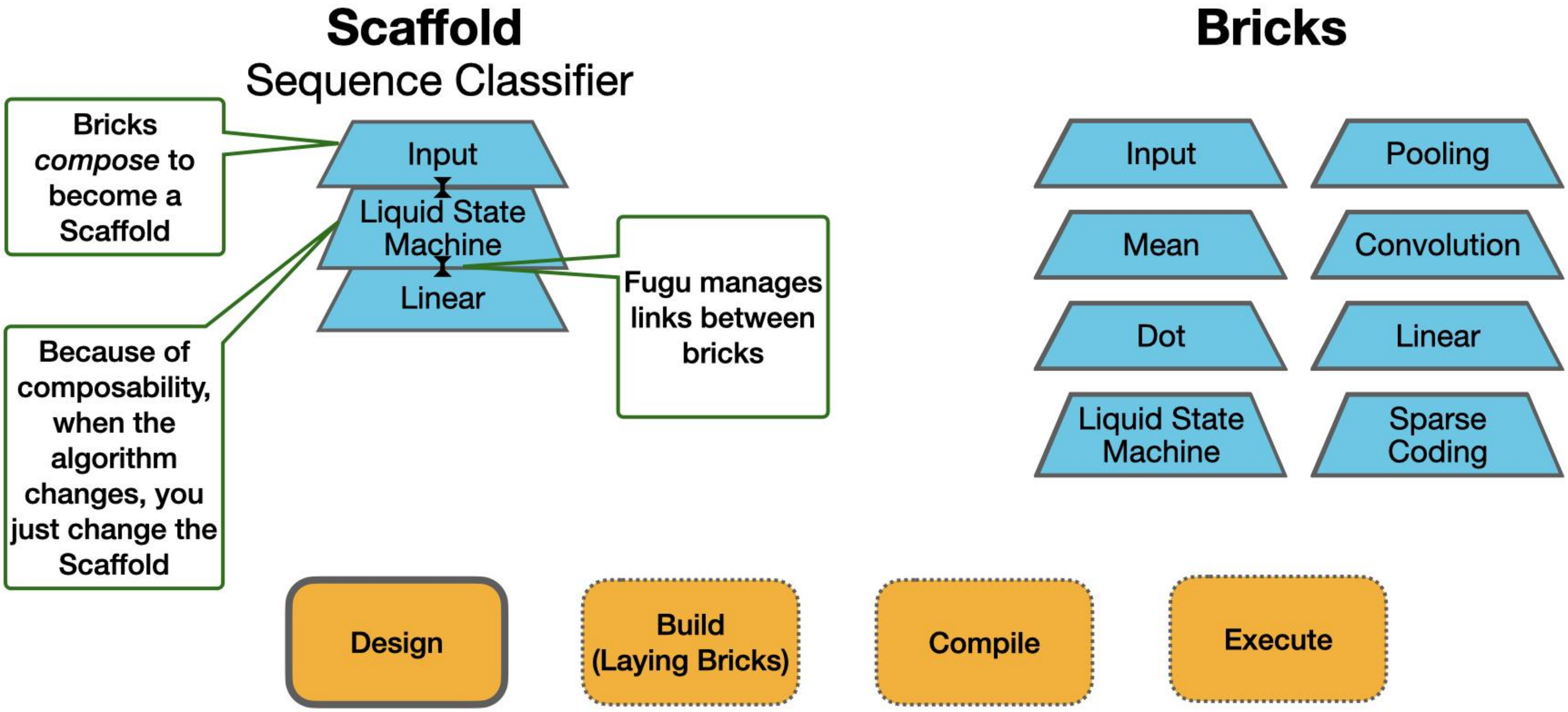
Design

Build (Laying Bricks)

Compile

Execute

How Fugu Works



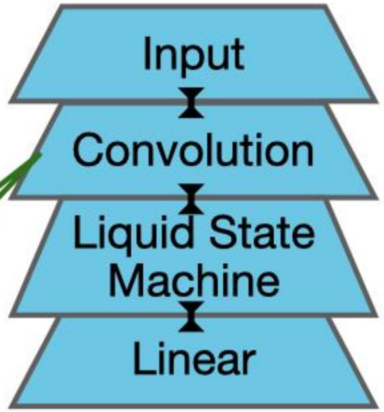
How Fugu Works

Scaffolds are directed, acyclic and can represent complex algorithms

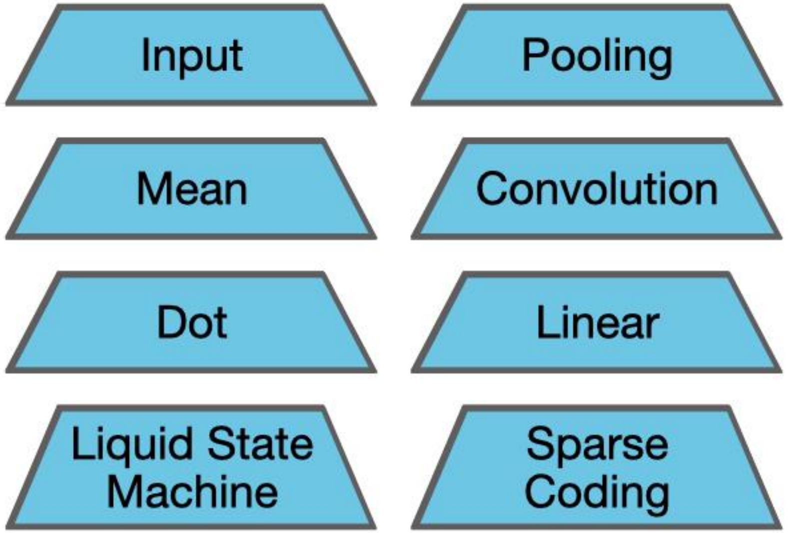
Because of composability, when the algorithm changes, you just change the Scaffold

Scaffold

Sequence Classifier



Bricks



Design

Build (Laying Bricks)

Compile

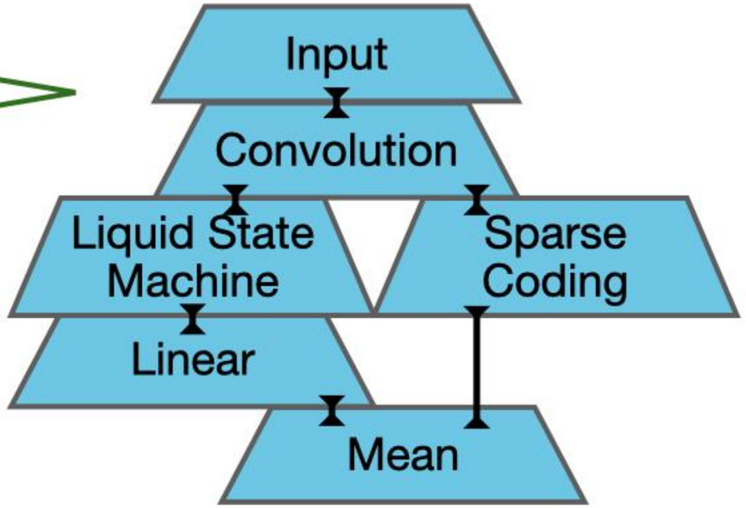
Execute

How Fugu Works

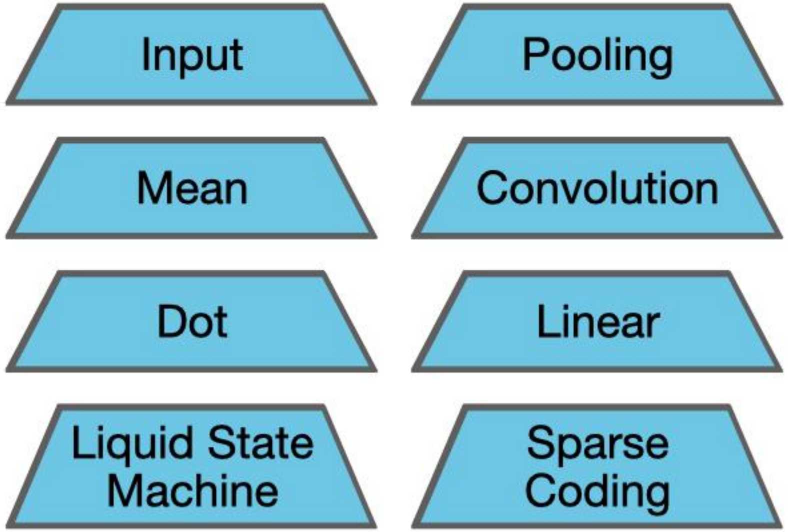
Scaffolds are directed, acyclic and can represent complex algorithms

Scaffold

Sequence Classifier



Bricks



Design

Build (Laying Bricks)

Compile

Execute

How Fugu Works

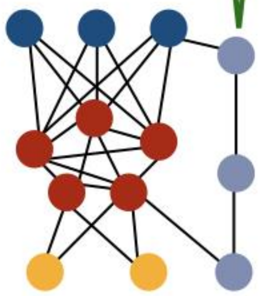
Scaffold

Sequence Classifier

Input

Liquid State Machine

Linear



Fugu also adds in some special neurons to help control information flow

Bricks

Input

Pooling

Mean

Convolution

Dot

Linear

Liquid State Machine

Sparse Coding

At Build, the Bricks provide instructions on how to build a spiking neural network

Design → **Build (Laying Bricks)** → Compile → Execute

How Fugu Works

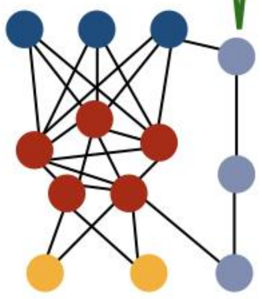
Scaffold

Sequence Classifier

Input

Liquid State Machine

Linear



Fugu also adds in some special neurons to help control information flow

This Intermediate Representation (IR) is platform-agnostic

Bricks

Input

Pooling

Mean

Convolution

Dot

Linear

Liquid State Machine

Sparse Coding

At Build, the Bricks provide instructions on how to build a spiking neural network

Design

Build (Laying Bricks)

Compile

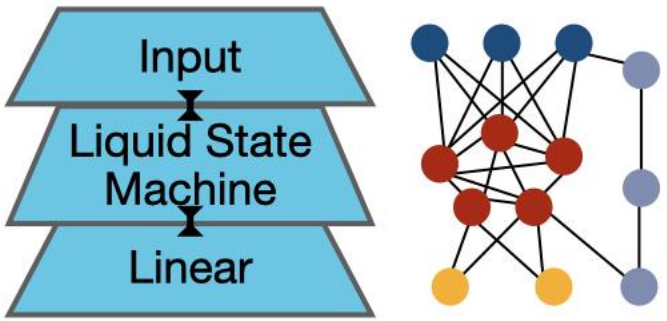
Execute

How Fugu Works

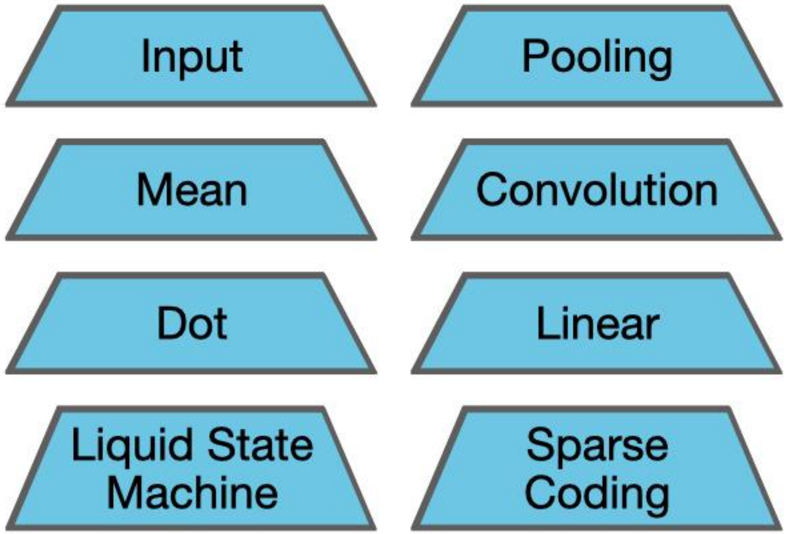
Scaffold

Sequence Classifier

Bricks are scalable, so if you dimension changes, so will your IR network



Bricks



Design

Build (Laying Bricks)

Compile

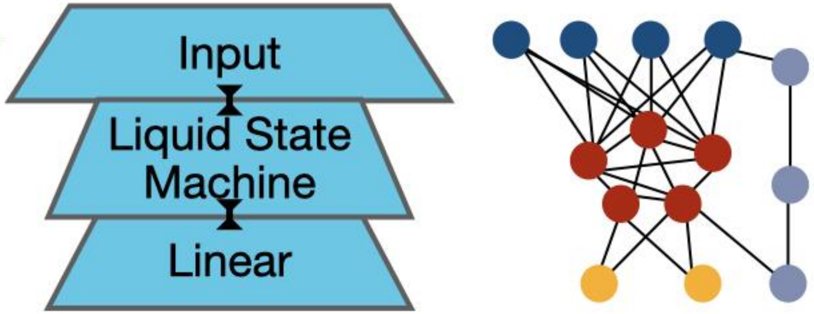
Execute

How Fugu Works

Scaffold

Sequence Classifier

Bricks are scalable, so if you dimension changes, so will your IR network

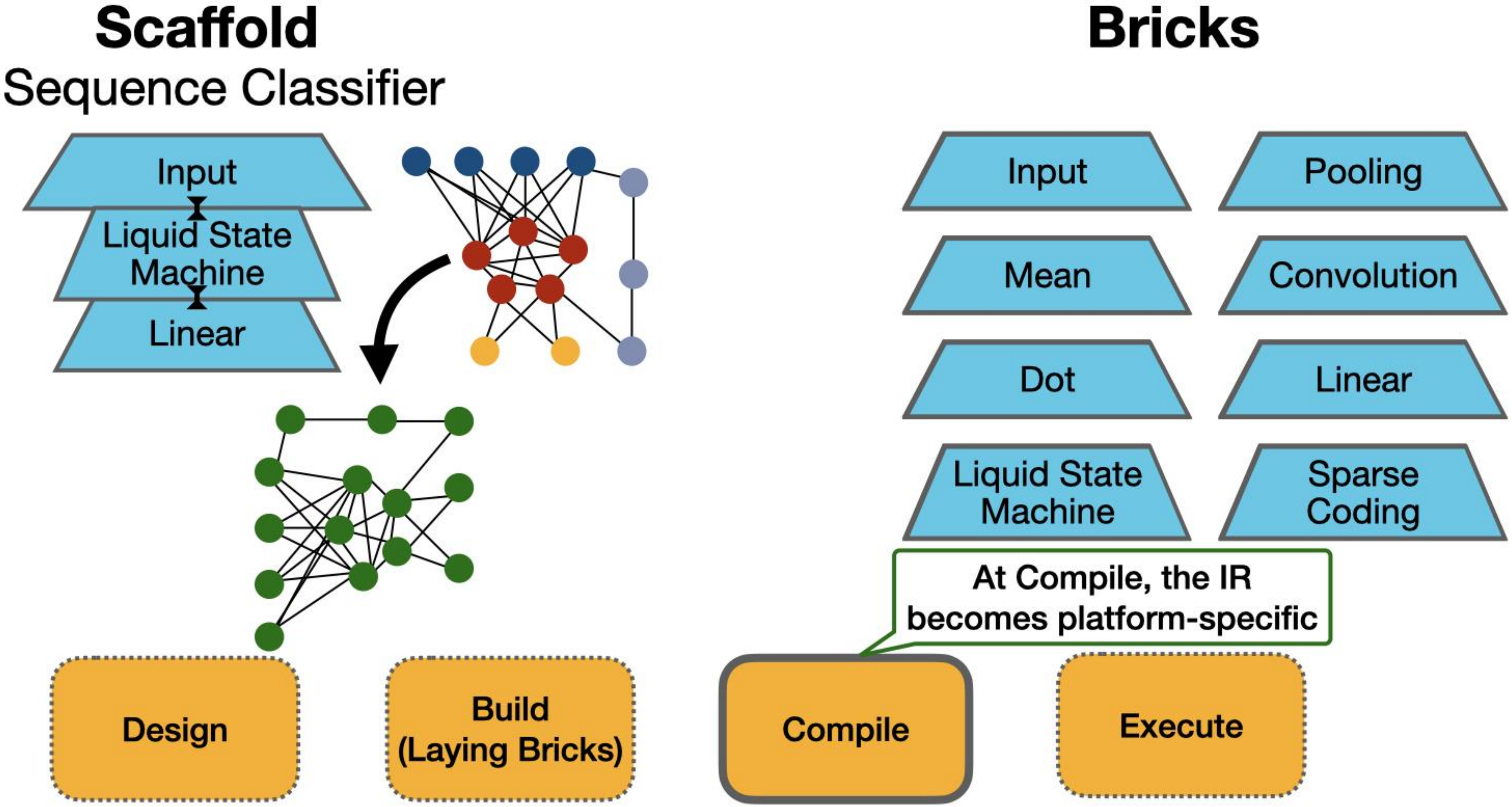


Bricks

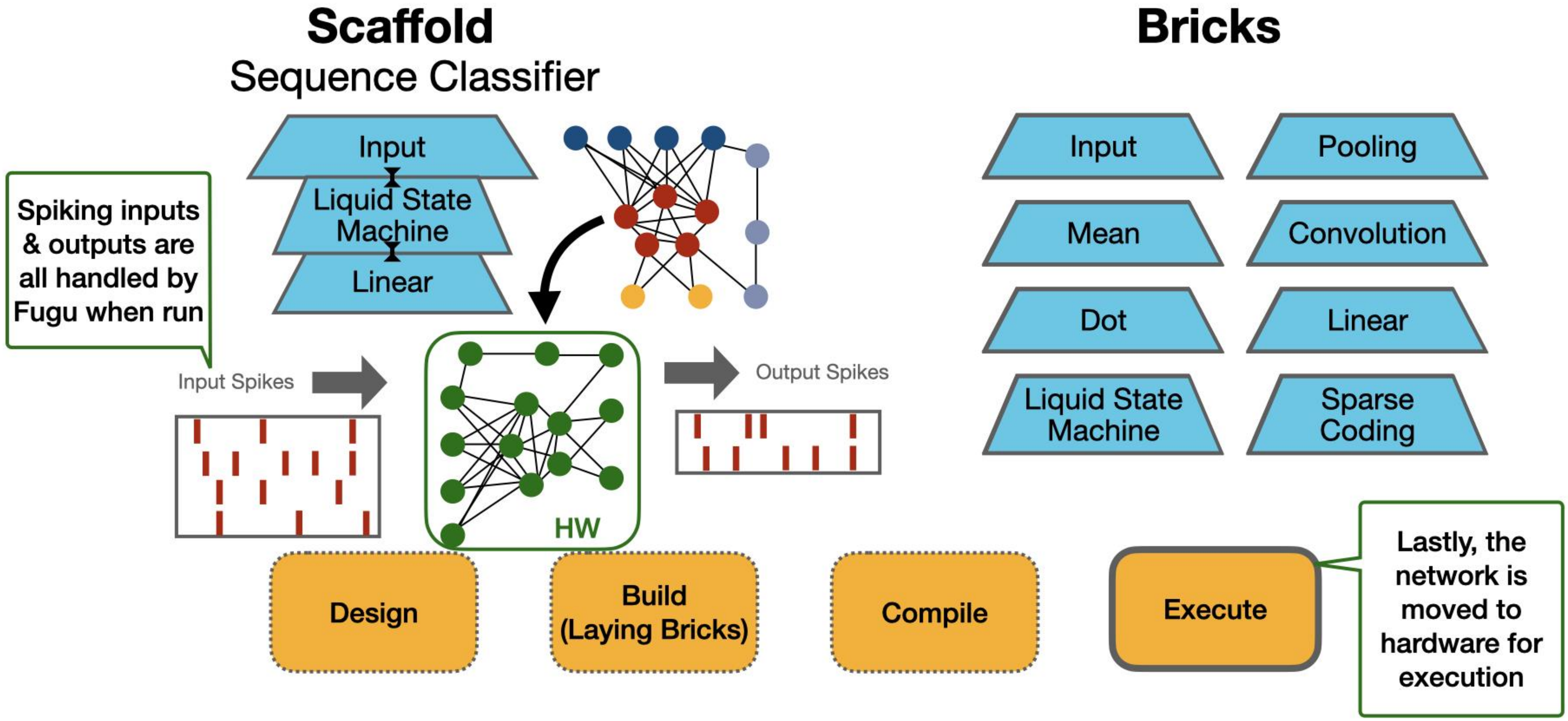
- Input
- Pooling
- Mean
- Convolution
- Dot
- Linear
- Liquid State Machine
- Sparse Coding

Design → **Build (Laying Bricks)** → Compile → Execute

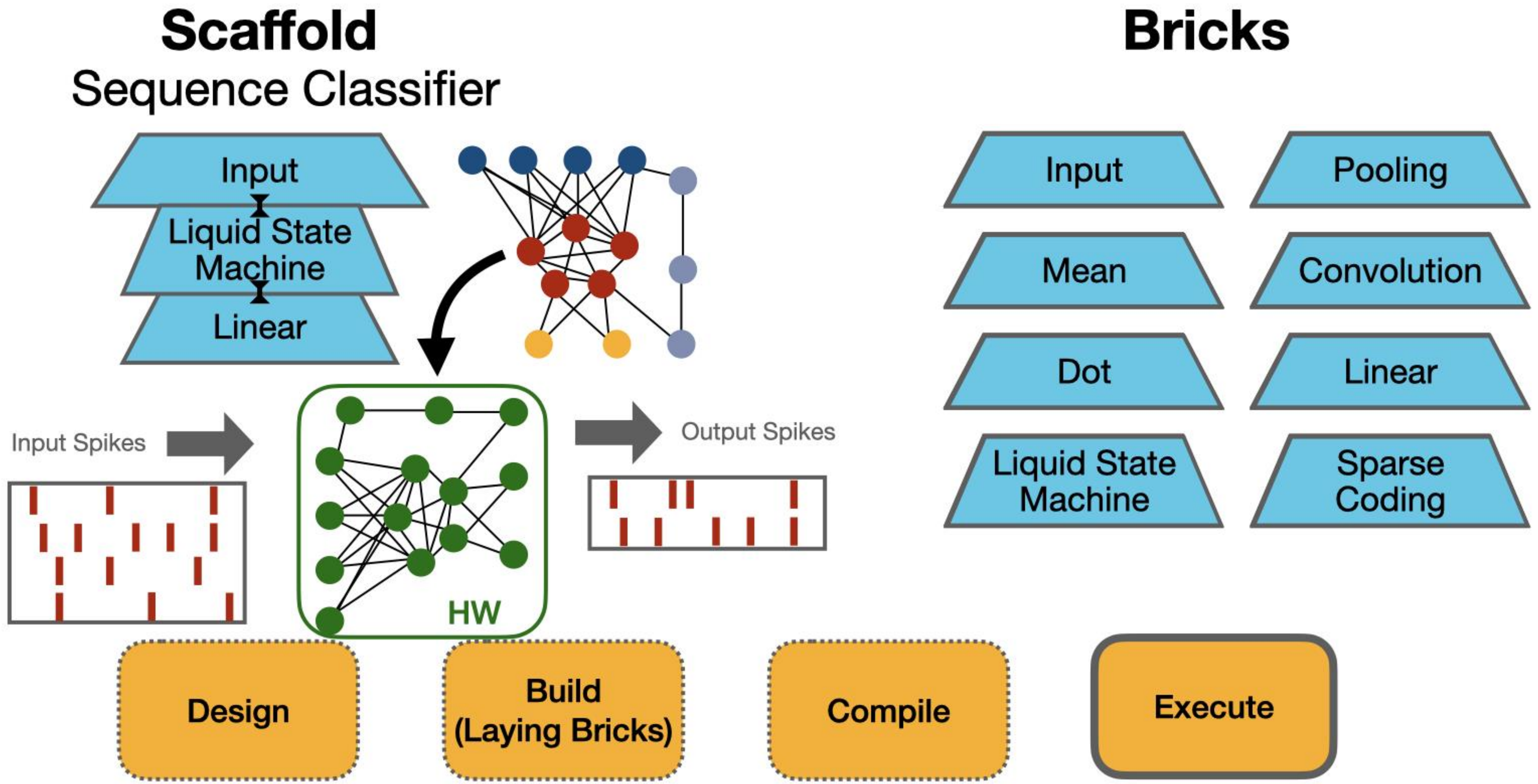
How Fugu Works



How Fugu Works



How Fugu Works



Okay... so what exactly is a brick?

...some intuition using a simple arithmetic example

Spiking Neural Streaming Binary Arithmetic

James B. Aimone, Aaron J. Hill, William M. Severa, & Craig M. Vineyard
Sandia National Laboratories
Albuquerque, New Mexico
Email: jbaimon@sandia.gov

Abstract—**Boolean functions and binary arithmetic operations are central to standard computing paradigms. Accordingly, many advances in computing have focused upon how to make these operations more efficient as well as exploring what they can compute. To best leverage the advantages of novel computing paradigms it is important to consider what unique computing approaches they offer. However, for any special-purpose co-processor, Boolean functions and binary arithmetic operations are useful for, among other things, avoiding unnecessary I/O on-and-off the co-processor by pre- and post-processing data on-device. This is especially true for spiking neuromorphic architectures where these basic operations are not fundamental low-level operations. Instead, these functions require specific implementation. Here we discuss the implications of an advantageous streaming binary encoding method as well as a handful of circuits designed to exactly compute elementary Boolean and binary operations.**

I. Introduction & Background

Fundamental to many paradigms of computing are Boolean functions and arithmetic operations. These core concepts can then be composed to build arbitrarily complex computations and set a foundation for comparing implementations and understanding computability. In pursuing an understanding of what computations neural circuits can perform, prior work has explored universal function approximation as well as Turing completeness [1], [2]. Accordingly, with that foundation in hand it is straightforward that spiking neurons can be used to compute arithmetic functions. However, here we not only provide several fundamental arithmetic computations as spiking neural streaming circuits, but use them as a means of understanding and enabling neuromorphic computing (NMC). Accordingly, here we present a set of streaming neural binary circuits implemented in Fugu, a neural algorithm composition framework, showing how more complex functions can be built upon operations such as addition and subtraction leading to multiplication.

Classic computational paradigms are incredibly efficient at performing these base building blocks of numerical computation, having been optimized for decades to minimize the computational kernel and maximize scalability [3], [4]. Exacting these computations in neurons both shows potential for how future, device breakthroughs in the development of neuromorphic hardware can enable classic numerical computations. And this work has been inspired partly by previous approaches for implementing logic and arithmetic in spiking networks, such as [5], [6], [7], [8]. But furthermore, this exploration also examines how the computational flexibility in spiking neural networks can be leveraged to enable compositionality for more complex computations. Alternatively, if the highly optimized canonical approaches were used to compute fundamental arithmetic operations which integrate neural subfunctions, there is a cost to convert in and out of neural circuity analogous to paying for analog to digital conversions.

II. Fugu

As a means of showing compositionality and scalability of spiking algorithms, we use the Fugu framework to represent the neural circuits presented here [9]. While implementation details vary based on NMC hardware, Fugu is a high-level framework specifically designed for developing spiking circuits in terms of computation graphs. Accordingly, with a base leaky-integrate-and fire (LIF) neuron model at its core, neural circuits are built as 'bricks'. These foundational computations are then combined and composed as 'scaffolds' to construct larger computations. This allows us to describe the streaming binary arithmetic circuits in terms of neural features common to most NMC architectures rather than platform specific designs.

In addition to architectural abstraction, the compositionality concept of Fugu not only facilitates a hierarchical approach to functionality development but also enables adding pre and post processing operations to overarching neural circuits. Such properties position Fugu to help explore under what parameterization or scale a neural approach may offer an advantage. For example, prior work has analyzed neural algorithms for computational kernels like sorting, optimization, and graph analytics identifying different regions in which a neural advantage exists accounting for neural circuit setup, timing, or other factors [10], [11], [12], [13].

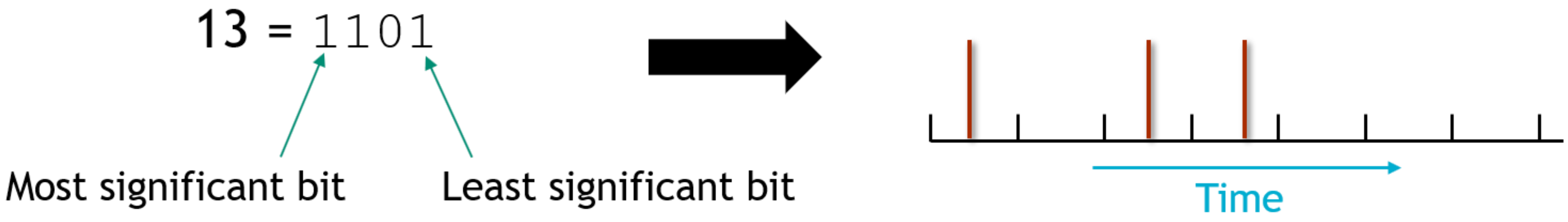
III. Spiking Binary Arithmetic

An open research question in neuroscience and neuromorphic computing is how to encode information [14]. The transmission of spikes can conveys information in their timing, enabling complex spatial temporal representations. However, here we do not exploit any novel spike encoding representations but rather use a binary representation of numbers which starts streaming the least significant bit first to the most significant bit last. Neurons can be used to represent many different coding schemes, but the main advantages of this "little-endian" temporal binary representation are that:

- One neuron is required per variable represented, with k timesteps required for a k-bit number.

ICRC, 2021

Little Endian in Time coding scheme



Single neuron encodes binary bits *in time* starting with least significant bit and moving up

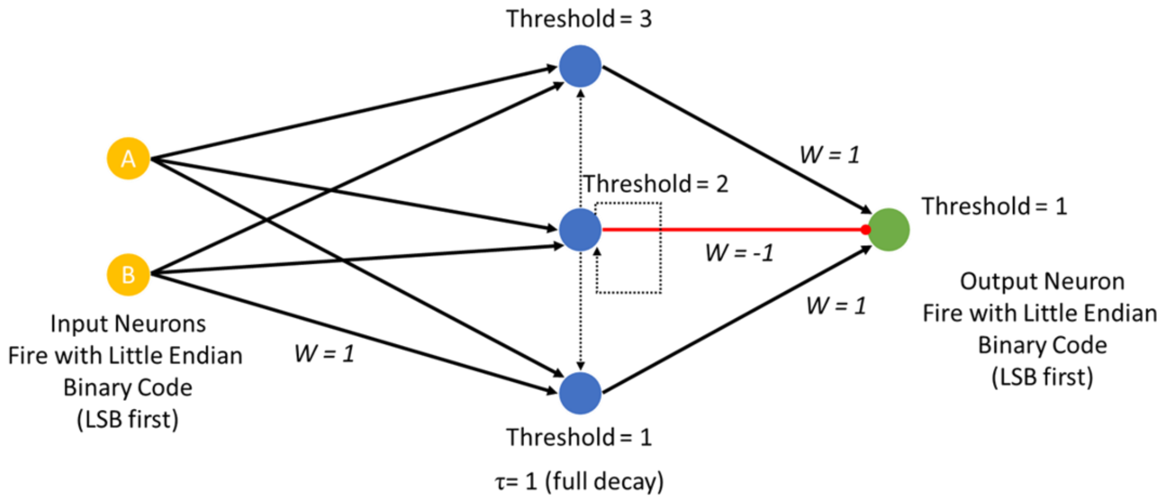
Benefits

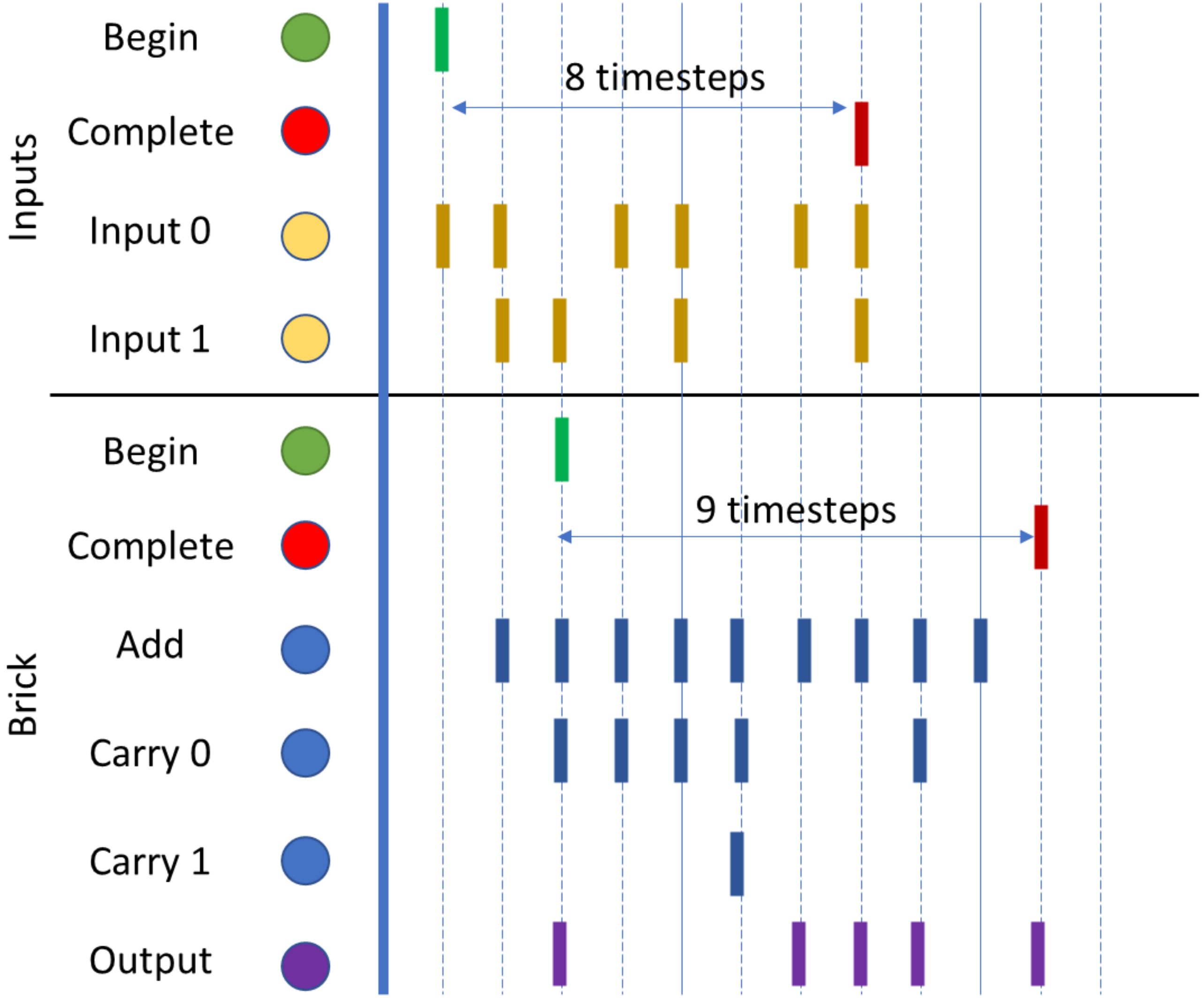
- Natively handles variable length numbers
- Well-suited for binary arithmetic
- Very efficient for neuron count and overall spike counts

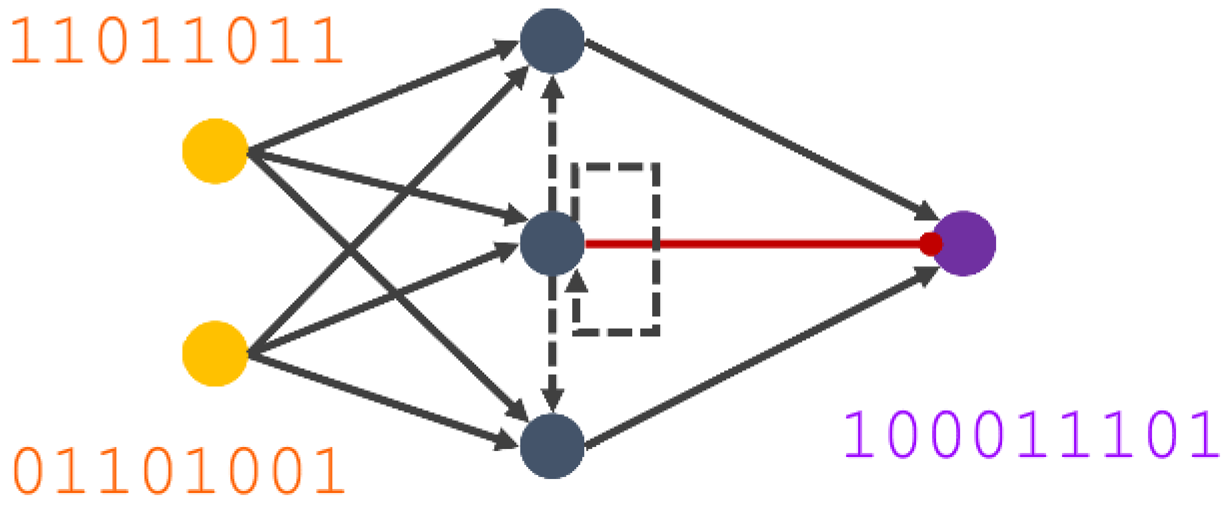
Drawbacks

- Requires cautious “bookkeeping” and/or external halt signal
- Computing in time adds a latency to calculations

Circuit for binary streaming adder

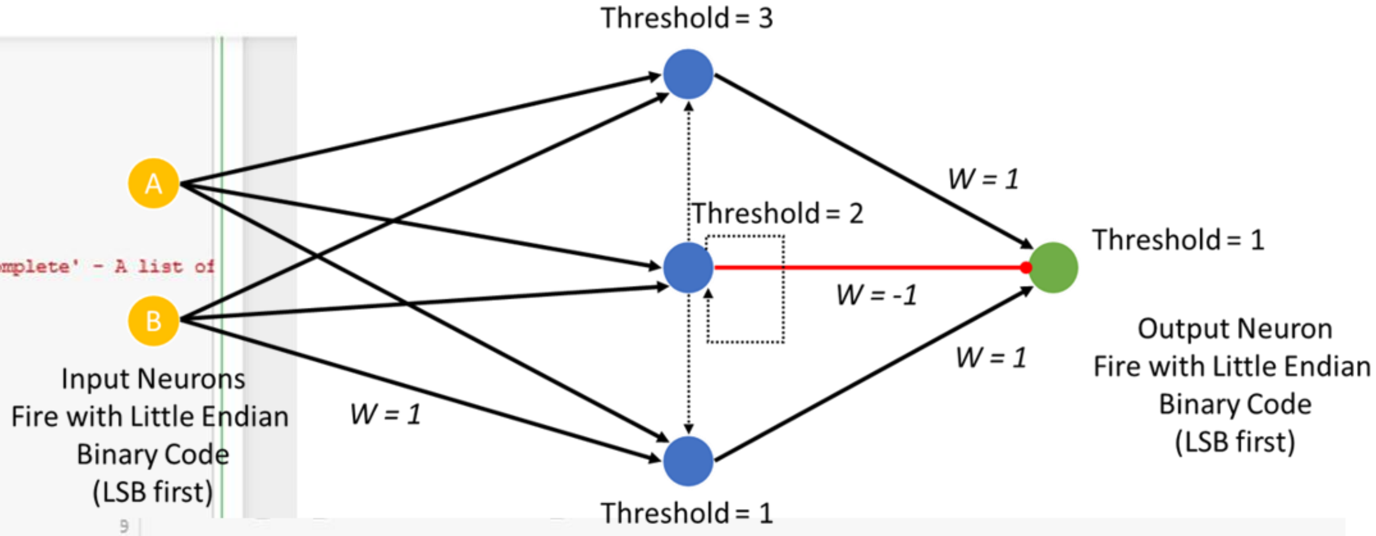





11011011
01101001
100011101

Example Fugu Code for binary streaming adder

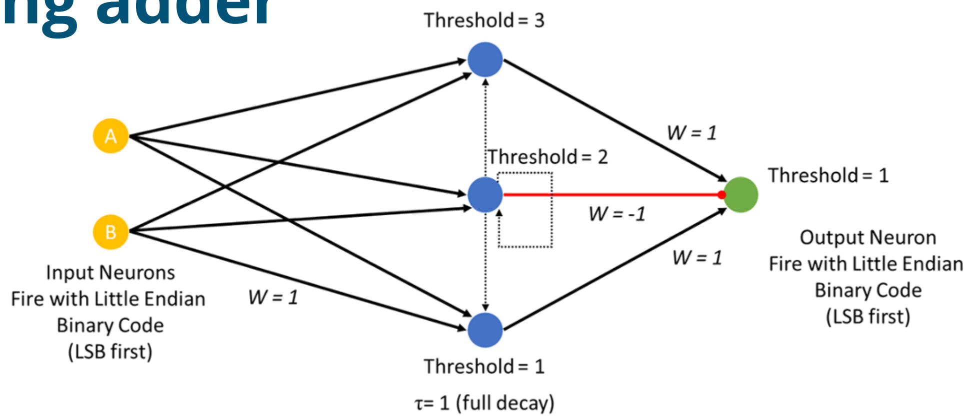
```python
        self.name = name
        self.supported_codings = ['binary-L']

    def build(self, graph, dimensionality, control_nodes, input_lists, input_codings):
        """
        Build streaming adder brick.

        Arguments:
            + graph - networkx graph to define connections of the computational graph
            + dimensionality - dictionary to define the shapes and parameters of the brick
            + control_nodes - dictionary of lists of auxillary networkx nodes.  Excpected keys: 'complete' - A list of
            + input_lists - list of nodes that will contain input
            + input_coding - list of input coding formats

        Returns:
            + graph of a computational elements and connections
            + dictionary of output parameters (shape, coding, layers, depth, etc)
            + dictionary of control nodes ('complete')
            + list of output
            + list of coding formats of output
        """

        if len(input_codings) != 2:
            raise ValueError("adder takes in 2 input on size n")

        output_codings = [input_codings[0]]

        new_complete_node_name = self.name + '_complete'
        new_begin_node_name = self.name + '_begin'

        graph.add_node(new_complete_node_name,
                       index = -1,
                       threshold = 0.0,
                       decay =0.0,
                       p=1.0,
                       potential=0.0)
        #graph.add_edge(control_nodes[0]['complete'], new_complete_node_name,weight=1.0,delay=1)

        graph.add_edge(control_nodes[0]['complete'], new_complete_node_name, weight=1.0, delay=2)
        graph.add_edge(control_nodes[0]['begin'], new_begin_node_name, weight=1.0, delay=2)

        complete_node = new_complete_node_name
        begin_node = new_begin_node_name

        #xor_node_name = self.name + '_0'

        l = len(input_lists[0])
```



τ= 1 (full decay)

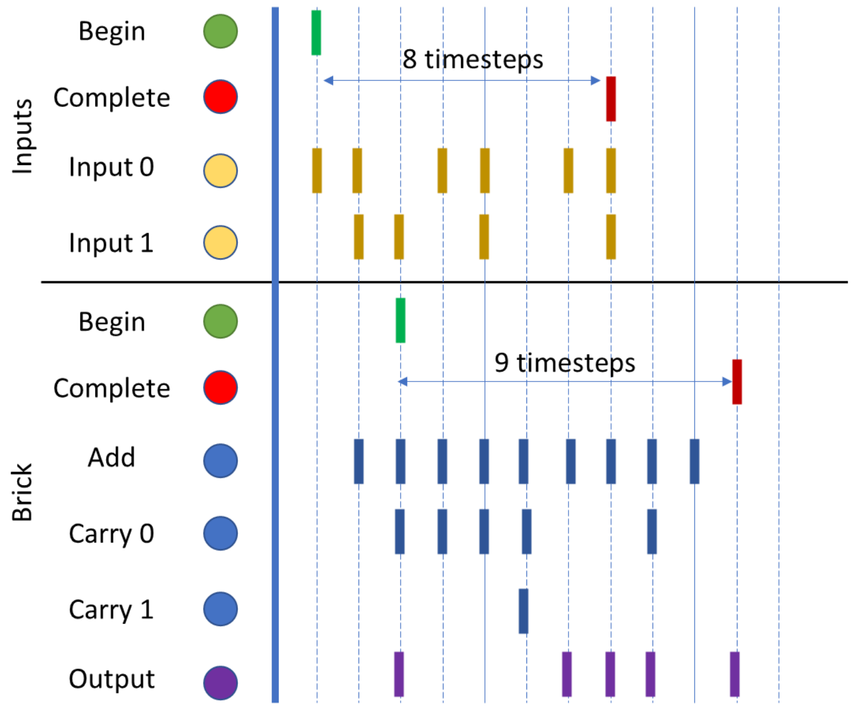
```python
        l = len(input_lists[0])

        #nodes
        graph.add_node(self.name + 'add', threshold=.9, decay=1.0, p=1.0, potential=0.0)
        graph.add_node(self.name + 'carry0', threshold=1.9, decay=1.0, p=1.0, potential=0.0)
        graph.add_node(self.name + 'carry1', threshold=2.9, decay=1.0, p=1.0, potential=0.0)
        graph.add_node(self.name + 'out', threshold=.9, decay=1.0, p=1.0, potential=0.0)
        #edges
        graph.add_edge(input_lists[0][0], self.name + 'add', weight=1.0, delay=1)
        graph.add_edge(input_lists[1][0], self.name + 'add', weight=1.0, delay=1)
        graph.add_edge(input_lists[0][0], self.name + 'carry0', weight=1.0, delay=1)
        graph.add_edge(input_lists[1][0], self.name + 'carry0', weight=1.0, delay=1)
        graph.add_edge(input_lists[0][0], self.name + 'carry1', weight=1.0, delay=1)
        graph.add_edge(input_lists[1][0], self.name + 'carry1', weight=1.0, delay=1)

        graph.add_edge(self.name + 'carry0', self.name + 'add', weight=1.0, delay=1)
        graph.add_edge(self.name + 'carry0', self.name + 'carry0', weight=1.0, delay=1)
        graph.add_edge(self.name + 'carry0', self.name + 'carry1', weight=1.0, delay=1)

        graph.add_edge(self.name + 'add', self.name + 'out', weight=1.0, delay=1)
        graph.add_edge(self.name + 'carry0', self.name + 'out', weight=-1.0, delay=1)
        graph.add_edge(self.name + 'carry1', self.name + 'out', weight=1.0, delay=1)


        self.is_built=True

        output_lists = [[self.name + 'out']]

        return (graph, self.dimensionality, [{'complete': complete_node, 'begin': begin_node}], output_lists
```

Example Fugu Code for binary streaming adder



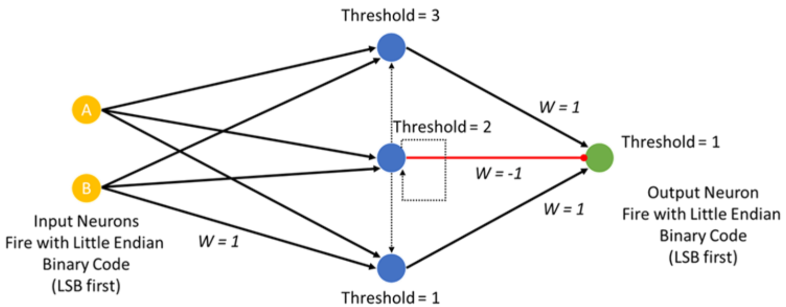

```python
# Simple addition example

scaffold = Scaffold()

# For addition, what we want to do is create a Scaffold taking two inputs and the adder brick

scaffold.add_brick(Vector_Input(np.array([[1, 1, 0, 1, 1, 0, 1, 1]]), coding='binary-L', name='Input0', time_dimens
scaffold.add_brick(Vector_Input(np.array([[0, 1, 1, 0, 1, 0, 0, 1]]), coding='binary-L', name='Input1', time_dimens
scaffold.add_brick(streaming_adder(name='adder1_'), [(0,0), (1, 0)], output=True)

scaffold.lay_bricks()
scaffold.summary(verbose=1)

backend = snn_Backend()
backend_args = {}
backend_args['record'] = 'all'
backend.compile(scaffold, backend_args)
result = backend.run(30)
print(result)
```

18	5.0	8.0
19	5.0	9.0
20	5.0	10.0
21	6.0	2.0
22	6.0	11.0
23	6.0	8.0
24	7.0	2.0
25	7.0	5.0
26	7.0	11.0
27	7.0	1.0
28	7.0	4.0
29	7.0	8.0
30	8.0	11.0
31	8.0	8.0
32	8.0	9.0
33	9.0	6.0
34	9.0	8.0
35	10.0	11.0

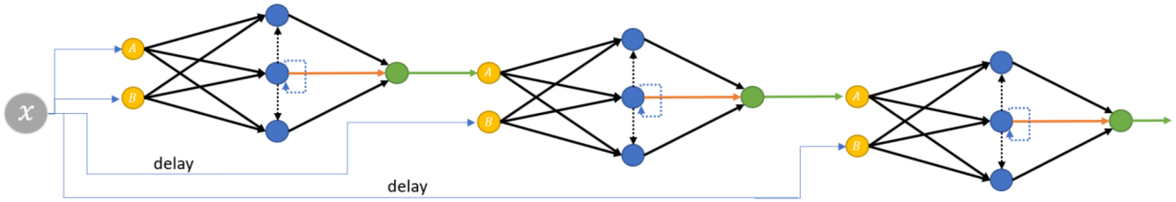


Growing suite of neuromorphic arithmetic logic

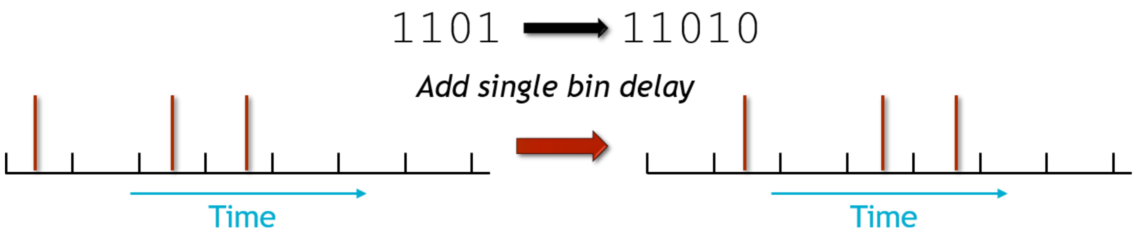
Adder: $y = A + B$



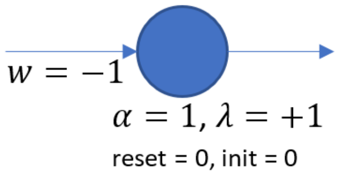
Multiply: $y = a*x$



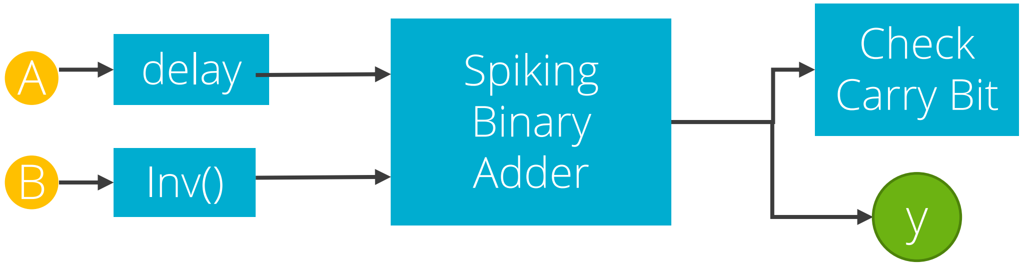
Shift multiplier: $y = 2$^d * A



Inverter: $y = !x$



Subtractor: $y = A - B$



Fibonacci Example

sandialabs / Fugu Public

Code · Issues · Pull requests · Actions · Projects · Security · Insights

Files

main

Go to file

- examples
  - notebooks
    - EightTwentyTutorial.ipynb
    - Example_Brick_Building.ipynb
    - FibonacciTutorial.ipynb
    - Fibonacci_Example.ipynb
    - adder.ipynb
    - arithmetic_example.ipynb
  - adder_example.py
  - fugu_test.py
  - lis_example.py
  - max_example.py
- fugu
- LICENSE
- README.md
- requirements.txt
- setup.py

Fugu / examples / notebooks / Fibonacci_Example.ipynb

SNL-NERL Initial release of Sandia Labs Fugu spiking neural algorithm tool

Preview · Code · Blame · 554 lines (554 loc) · 76.4 KB

Fugu Example Notebook - Fibonacci Sequence

Brad Aimone, 3/14/2022. Happy Pi Day.

This notebook shows how Fugu can be used to generate more complex arithmetic circuits from basic streaming arithmetic functions such as addition. The goal of this notebook is to show how more complex arithemetic fucntinos can be simply composed from Fugu bricks.

Neuron Coding Scheme

The examples in this notebook describe circuits that use inputs which are encoded using a little-endian-in-time (LEIT) coding scheme. LEIT coding is simple - think of a binary description of a number (19 = 10011), flip it around so the least significant bit is first (19 => 11001), and then have the input neuron spike at the first, second, and fifth timesteps.

Step 0: Setup

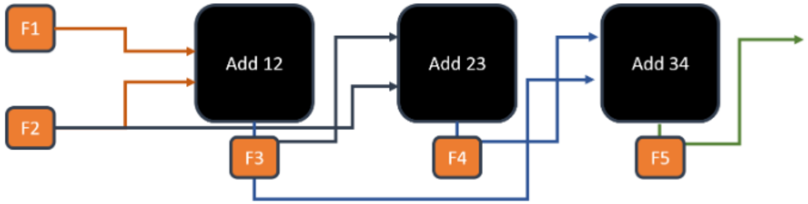
First, we need to import Fugu and other relevant libraries. Here, we will include the adder bricks

In [ ]:

```
import networkx as nx
import numpy as np
import fugu
from fugu import Scaffold, Brick
from fugu.bricks import Vector_Input
from fugu.backends import snn_Backend
from fugu.bricks import streaming_adder, temporal_shift
%load_ext autoreload
%autoreload 2
```

Example 1: Brute force circuit

The goal of this circuit is to implement the Fibonacci sequence by brute force; if we want to go 10 layers, we will have 10 adders.



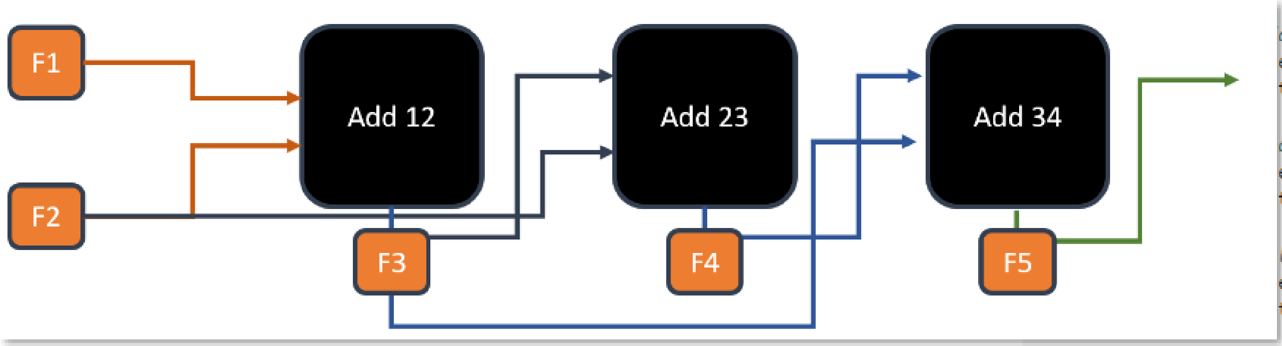
In [ ]:

```
scaffold = Scaffold()

F_1=[0,0]
F_2=[1,0]
shift_length_total = 2

scaffold.add_brick(Vector_Input(np.array([F_1]), coding='binary-L', name='F1', time_dimension=True), 'input') #0
```

ttps://github.com/sandialabs

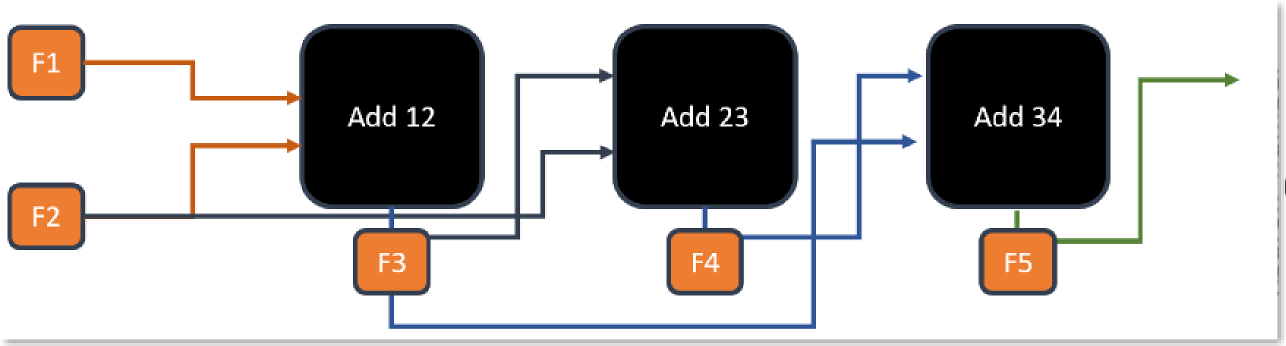
Fibonacci Example

```
F_1=[0,0]
F_2=[1,0]
shift_length_total = 2

scaffold.add_brick(Vector_Input(np.array([F_1]), coding='binary-L', name='F1', time_dimension=True), 'input') #0
scaffold.add_brick(Vector_Input(np.array([F_2]), coding='binary-L', name='F2', time_dimension=True), 'input') #1

# The first adder will add of F1 and F2.  This output is F3
scaffold.add_brick(streaming_adder(name='add_12_'), [(0,0), (1,0)], output=True) #2

# The second adder adds a time-delayed version (2 timesteps) of F2 and F3.  This output is F4
scaffold.add_brick(temporal_shift(name='shift_2_', shift_length=shift_length_total), [(1,0)], output=True) #3
scaffold.add_brick(streaming_adder(name='add_23_'), [(2,0), (3,0)], output=True)

ds a time-delayed version of F3 and F4.  This output is F5
emporal_shift(name='shift_3_', shift_length=shift_length_total), [(2,0)], output=True) #5
treaming_adder(name='add_34_'), [(4,0), (5,0)], output=True)

dds a time-delayed version of F4 and F5.  This output is F6
emporal_shift(name='shift_4_', shift_length=shift_length_total), [(4,0)], output=True) #7
treaming_adder(name='add_45_'), [(6,0), (7,0)], output=True)

ll do this for 10 elements
emporal_shift(name='shift_5_', shift_length=shift_length_total), [(6,0)], output=True) #9
treaming_adder(name='add_56_'), [(8,0), (9,0)], output=True)

scaffold.add_brick(temporal_shift(name='shift_6_', shift_length=shift_length_total), [(8,0)], output=True) #11
scaffold.add_brick(streaming_adder(name='add_67_'), [(10,0), (11,0)], output=True)

scaffold.add_brick(temporal_shift(name='shift_7_', shift_length=shift_length_total), [(10,0)], output=True) #13
scaffold.add_brick(streaming_adder(name='add_78_'), [(12,0), (13,0)], output=True)

scaffold.add_brick(temporal_shift(name='shift_8_', shift_length=shift_length_total), [(12,0)], output=True) #15
scaffold.add_brick(streaming_adder(name='add_89_'), [(14,0), (15,0)], output=True)

scaffold.add_brick(temporal_shift(name='shift_9_', shift_length=shift_length_total), [(14,0)], output=True) #17
scaffold.add_brick(streaming_adder(name='add_910_'), [(16,0), (17,0)], output=True)

scaffold.lay_bricks()
```

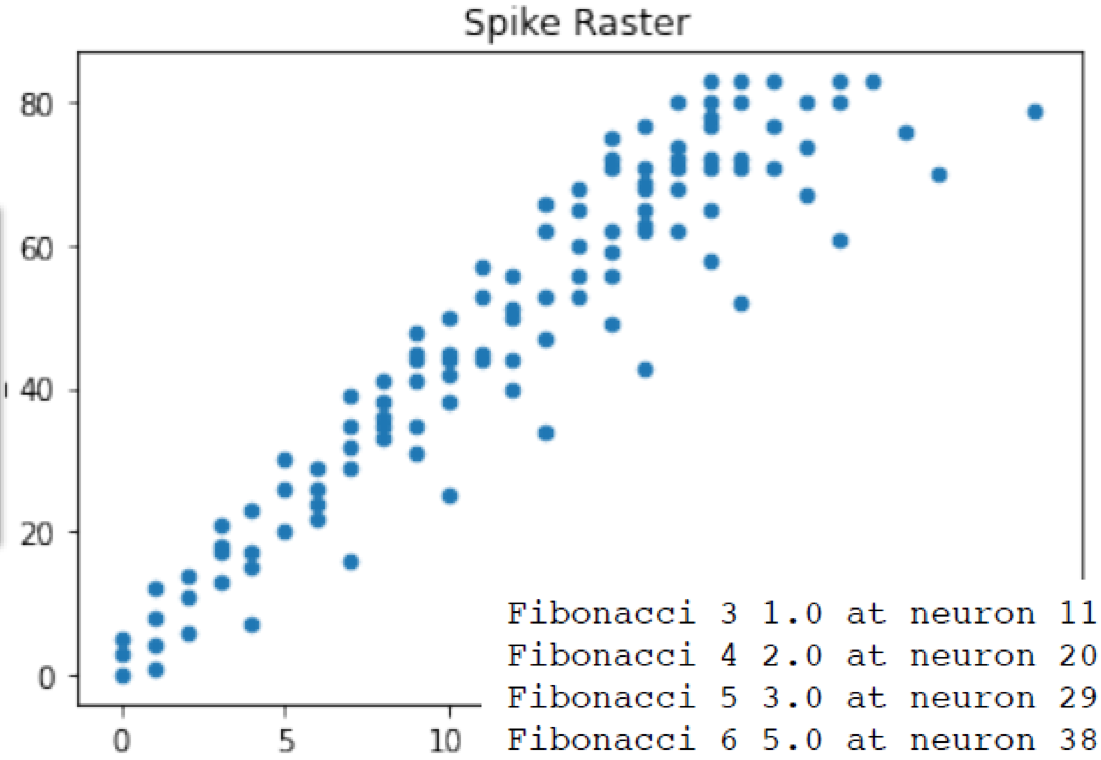


https://github.com/sandialabs/Fugu/blob/main/examples/notebooks/Fibonacci_Example.ipynb

Fibonacci Example

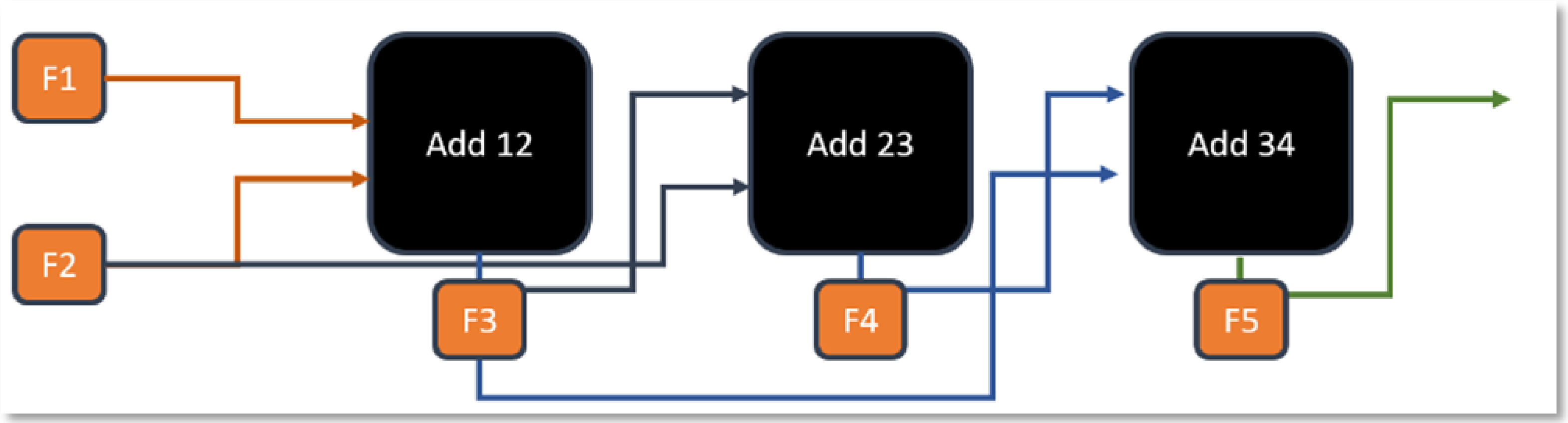


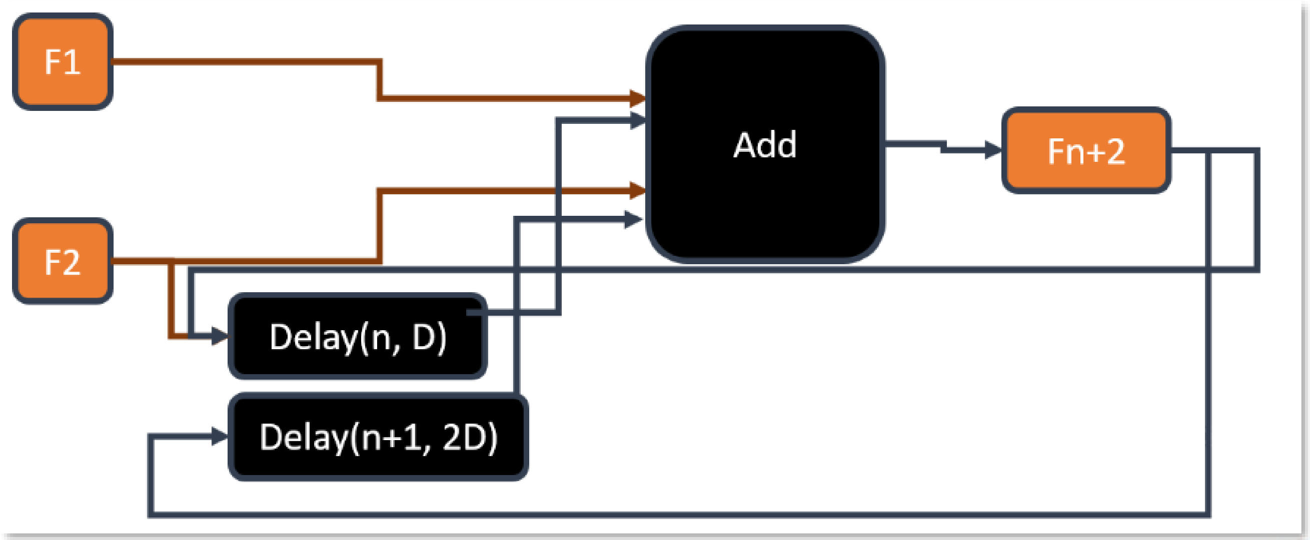
https://github.com/sandialabs/Fugu/blob/main/examples/notebooks/Fibonacci_Example.ipynb




```
Fibonacci 3 1.0 at neuron 11
Fibonacci 4 2.0 at neuron 20
Fibonacci 5 3.0 at neuron 29
Fibonacci 6 5.0 at neuron 38
Fibonacci 7 8.0 at neuron 47
Fibonacci 8 13.0 at neuron 56
Fibonacci 9 21.0 at neuron 65
Fibonacci 10 34.0 at neuron 74
```

Fibonacci Example





https://github.com/sandialabs/Fugu/blob/main/examples/notebooks/Fibonacci_Example.ipynb

Fugu Benefits and Limitations

Benefits

- Transparent hardware execution
- Compositional approach
- Easy scaling
- Lower barrier of entry
- 'Do what you're good at'

Limitations

- Designing bricks is challenging
- Not many bricks exist
- IR is static (in current version)
- Hardware execution is not 100% identical to simulator evaluation

Evolutionary Algorithms (EAs) can take advantage of the benefits and minimize the limitations

Evolutionary Algorithms

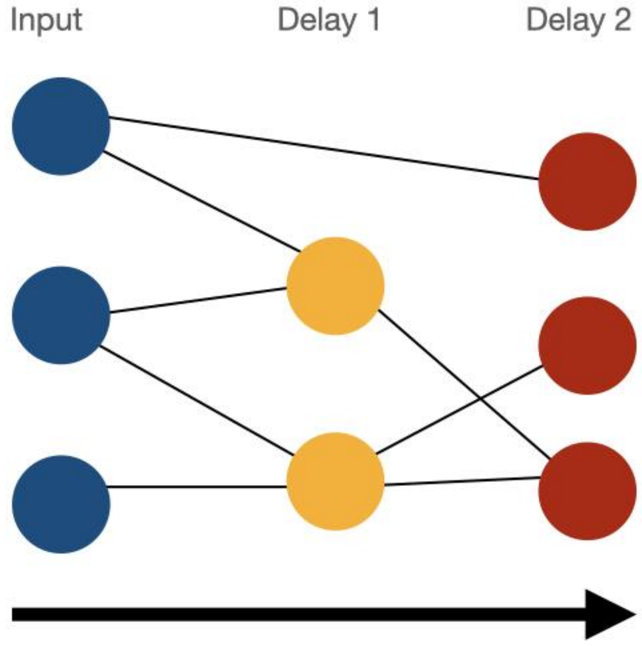
NEAT provides a ready-to-go EA approach

- Evolutionary Algorithms (EAs) are a family of optimization algorithms
  - Randomly alter and modify a population of candidate solutions
  - Evaluate candidate solutions against a fitness function
  - Keep best candidates and repeat
- "Evolutionary Optimization for Neuromorphic Systems" (EONS) is popular for spiking neural networks
- "NeuroEvolution of Augmenting Topologies" (NEAT) is a long-standing approach from traditional neural networks
  - Well-studied approach (4000+ citations) with multiple implementations
  - NEAT offers a strong, flexible framework for using EAs in Fugu

NEAT → Fugu Neuron Conversion

Fugu's LIF model adapts easily

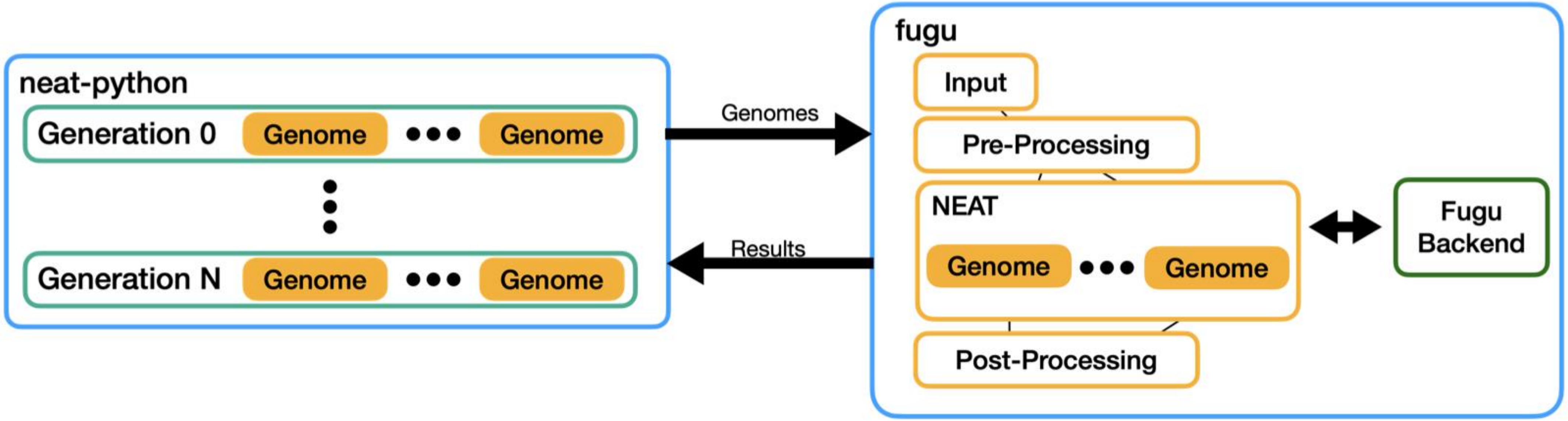
- NEAT is very flexible, we use the default feed-forward genome
  - Recurrent is a straightforward modification
- In NEAT's configuration, we force a 'threshold' activation function and a 'sum' aggregation.
- With this configuration, most the parameters have a clear correspondence (e.g. weight = weight, -bias = threshold)
- Delays are derived from the sequence activation of the layers (e.g. neurons in layer 5 connected to layer 3 will have delay 2)



Delays are determined by the layering of the NEAT network

Neat Fugu Approach

NEAT provides candidate networks are evaluated in-parallel within a Fugu scaffold

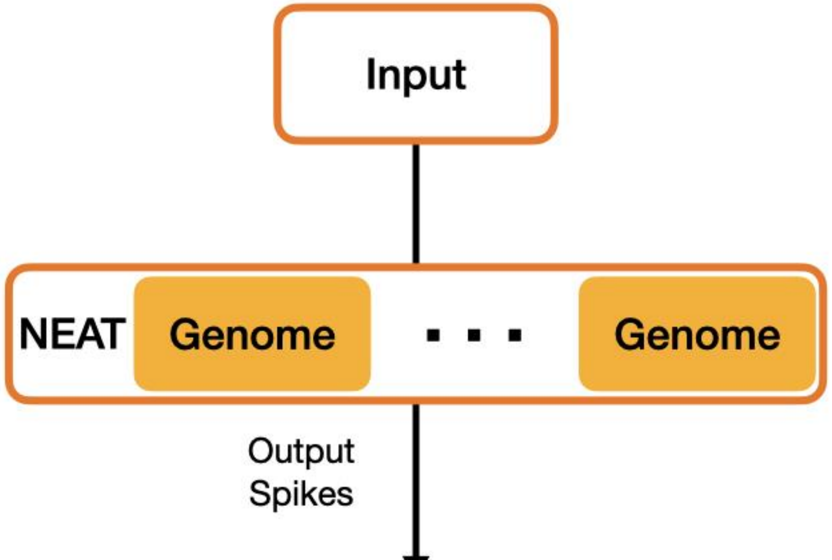


Example schematic of how neat-python interacts with Fugu.

Simple Scaffold Details

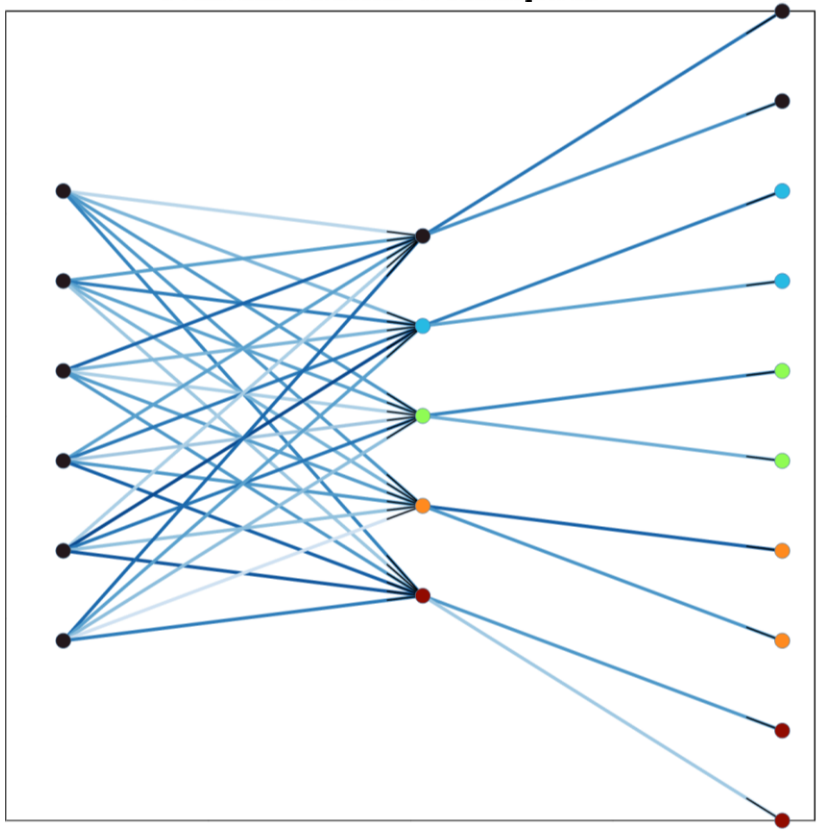
- In the basic case, we only need an input brick and a NEAT brick
- Input spikes are streamed in sequence
  - Fugu doesn't require a specific input coding
  - We use a Base P encoding for simplicity
  - A better encoding could improve performance
- Output spikes can be decoded to compute the fitness function
  - We usually use a Squared Error, normalized by the number of outputs (classes)
  - However, the fitness function can be whatever you define

A Basic NEAT Scaffold

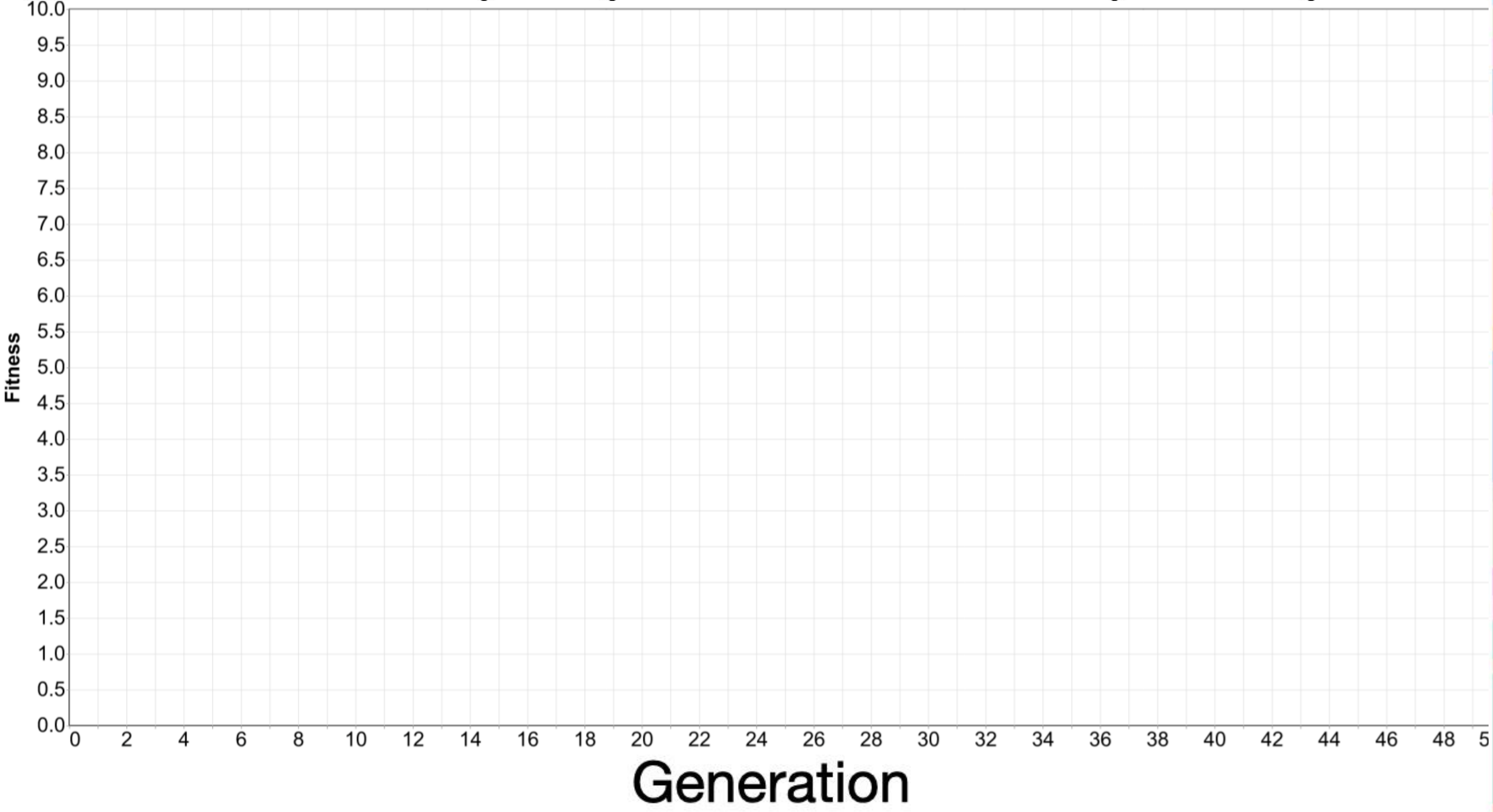


Example Population Update

Network Graph

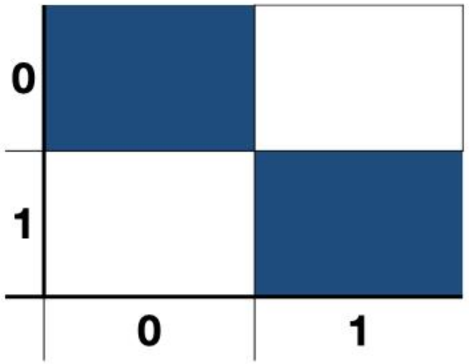


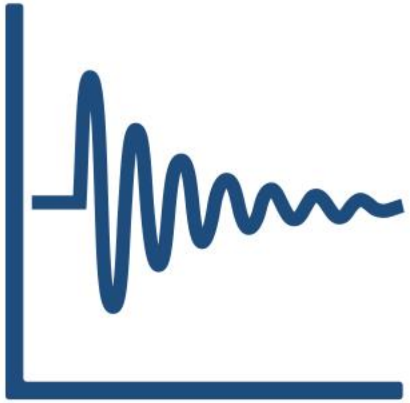
Max Fitness (solid) and Mean Fitness (dashed)



Testing Tasks with Basic Scaffold

NEAT + Fugu applies to a wide range of problems



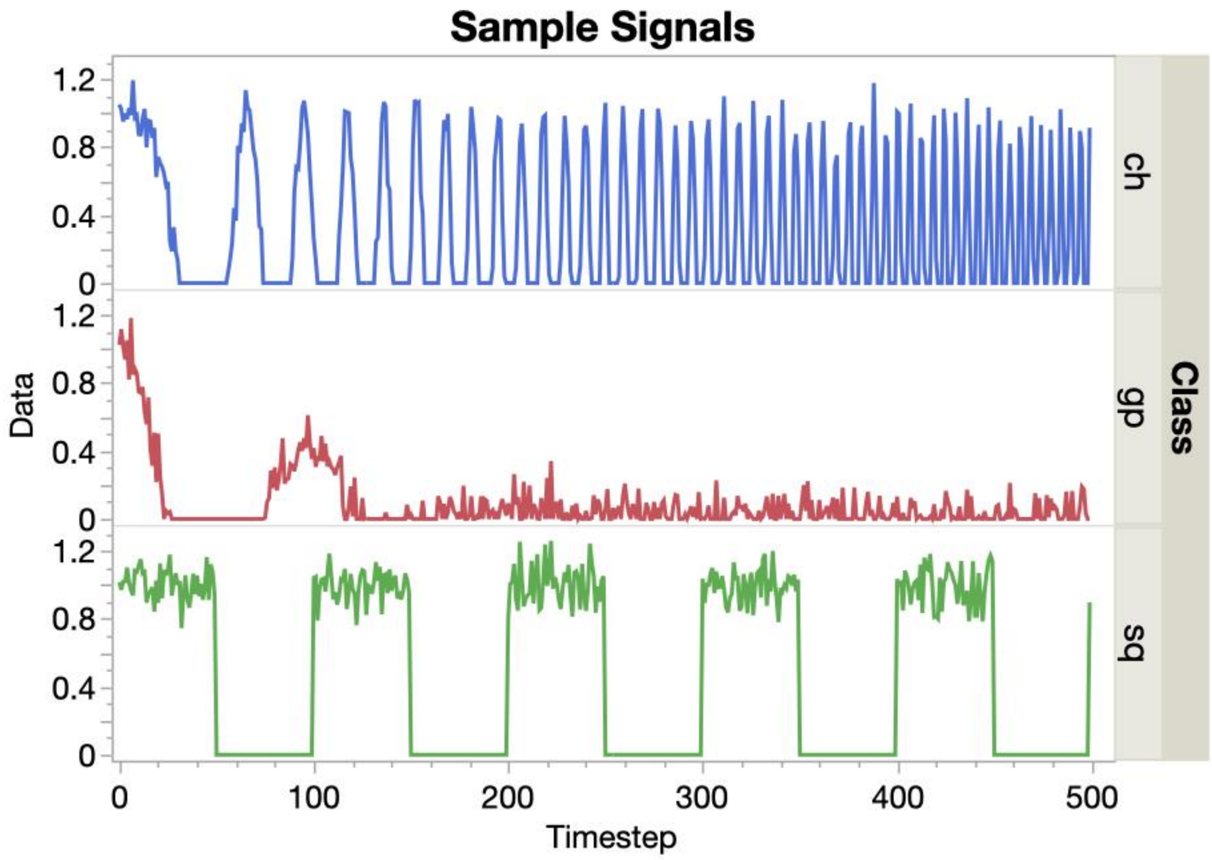


Task:	XOR	Iris	Agent	Signal
Fitness:	4/4	29.67/30	8/8	?

A Signal Classification Task

Dataset Generated by Adding Noise to Baseline Signals

Task: Classify the source signal of 500-timestep noisy time series signals

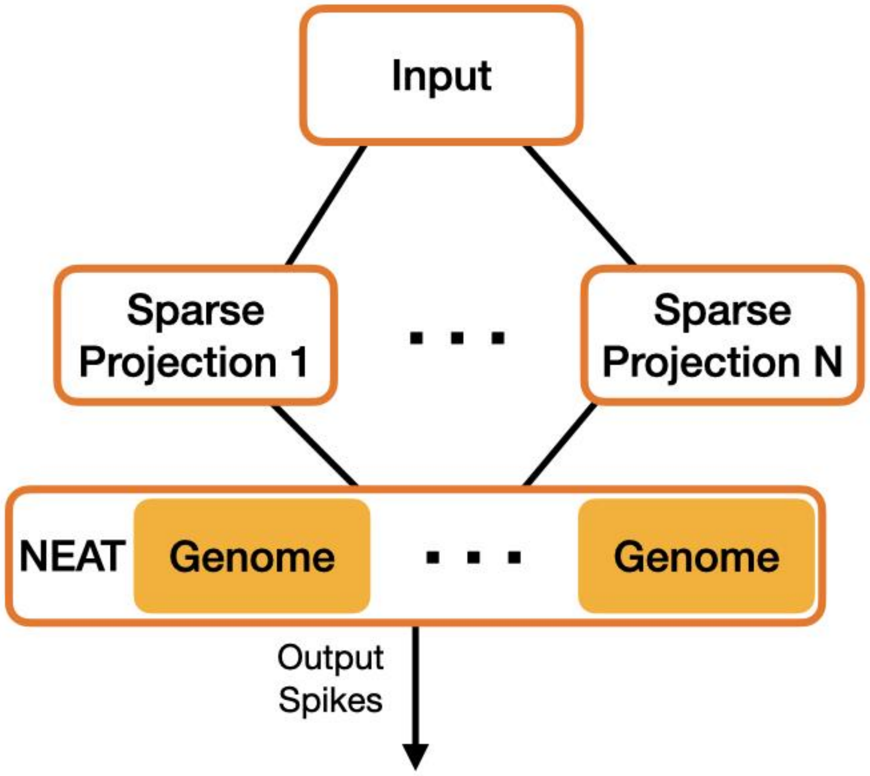


Using Composition

Effective Feature Extraction using Sparse Projection

- Random sparse projection reduces the timeseries into a 25-feature vector
- A learned NEAT classifier was trained from this representation, with a voting output
- With this scheme, it is easy to get high accuracy: 96%, much better!
- Show the flexibility and utility of Fugu composition
  - More non-EA options that just feature extraction: Post-processing, reference networks (distillation), validation, metrics, etc., etc....

Now with Sparse Projection



Fugu Enables Easy Hardware Execution

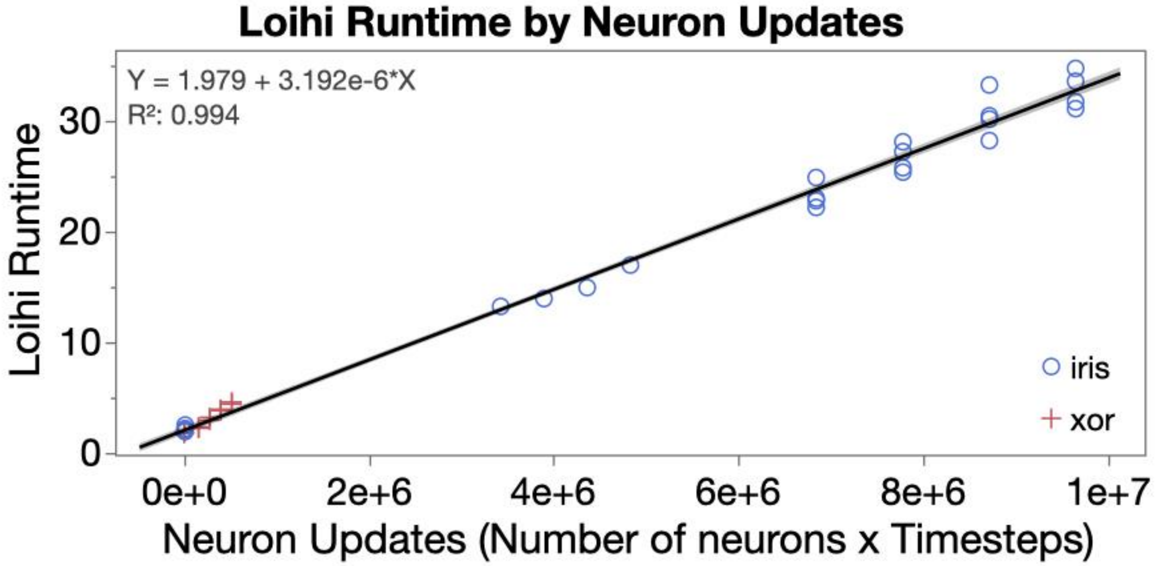
Scaling can be great. Compile times need improvement.

- Switching from simulator to hardware is easy in Fugu

```
backend = snn_Backend()
```

```
backend = loihi_Backend()
```

- Using hardware in-the-loop means you get the exact deployment behavior
- Also an easy, scalable hardware benchmark
- We performed a quick benchmark using Intel Loihi



- Evolutionary learning + Fugu
    - Easy to use, Modular approach
    - Platform-agnostic
    - Composition has benefits and deep implications
- On-hardware evaluation
    - Accurate deployment
    - Easy benchmarking

- Possible Extensions:
    - Real tasks and applications
    - Detailed benchmarking
    - Algorithmic modifications
    - Scaleable learned networks

Open-source Fugu is available on github

https://github.com/sandialabs/Fugu

Fugu-Neat is available in `fugu.experiemental`

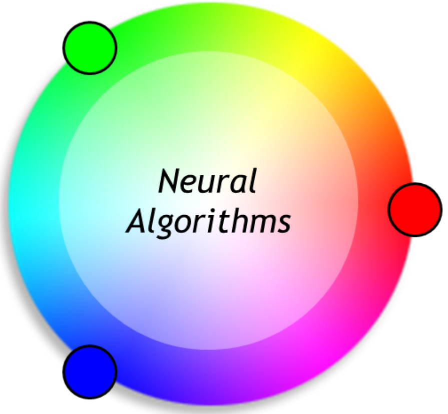
Full-Stack Neuromorphic Booklet

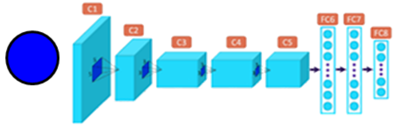
https://www.sandia.gov/app/uploads/sites/223/2022/12/Full-Stack-Neuromorphic-SAND2022-10373M.pdf

Sandia's Neural Exploration & Research Laboratory

https://neuroscience.sandia.gov

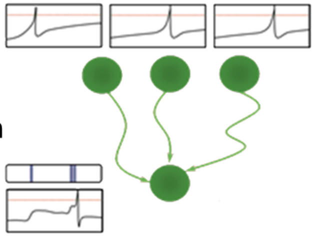
So what about these other cognitive and AI applications of neuromorphic?

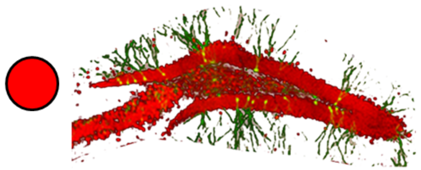




Artificial neural networks
- Generic layers of non-linear nodes
- SGD optimization of weights
- Powerful machine learning capabilities through learning sequential non-linear mappings and function approximation

Spiking neural algorithms
- Hand-crafted circuits of spiking neurons
- Model of parallel computation
- Energy efficiency through event-driven communication and high fan-in logic

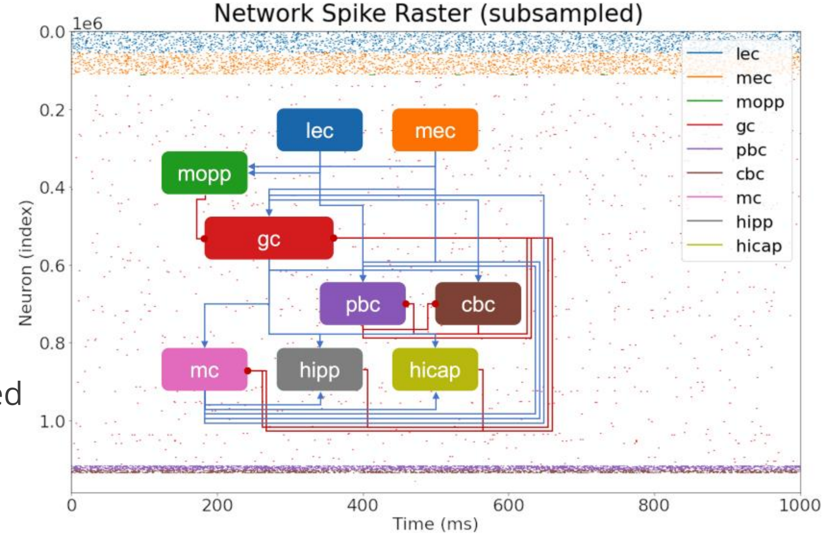




Neuroscience-constrained algorithms
- Circuit architecture based on local and regional neural connectivity
- Computation incorporates broad range of neural plasticity and dynamics
- *Generally still unexplored from algorithms perspective*

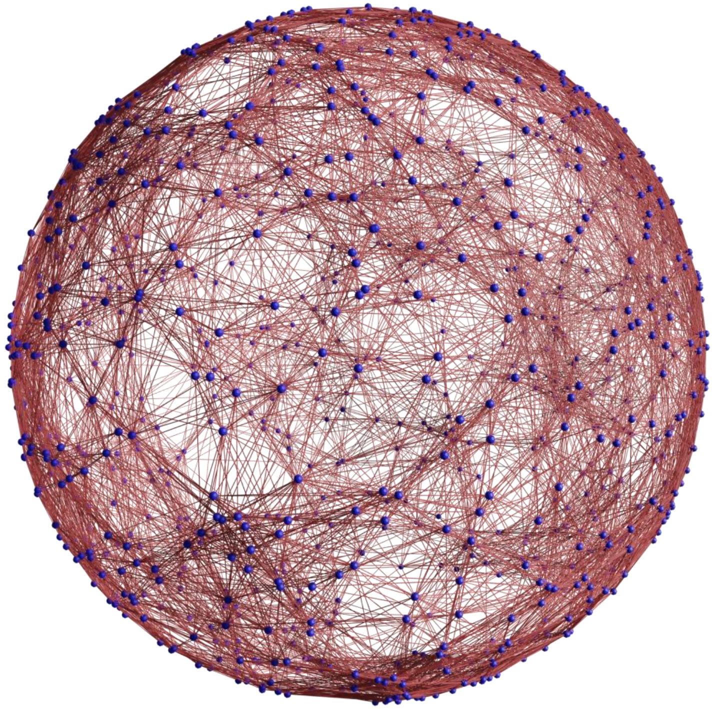
Simulation Tool for Asynchronous Cortical Streams (STACS)

- STACS is a large-scale spiking neural network simulator built on top of the Charm++ parallel programming framework (workload decomposed as parallel objects)
  - Network models are represented as a directed graph, and neuron and synapse model dynamics are formulated as stochastic differential equations:
    - $dx = f(x)dt + \sum_{i=1}^{n} g_i(x)dN_i$: time-driven computation, event-driven communication
  - Simulation is supported by partition-based data structures, and serialized through SNN extensions to the distributed compressed sparse row (SNN-dCSR) data format
    - These facilitate network generation, graph-aware partitioning, checkpoint/restart, scalable multicast communication, and tool interoperability
  - STACS may be used as a standalone simulator, as a backend to tools like N2A or Fugu, and also interface with external devices through YARP
- Available at: https://github.com/sandialabs/STACS



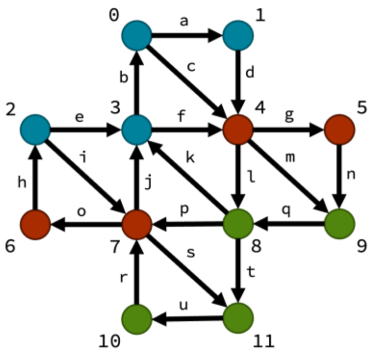
Simulation of a biologically inspired spiking neural network of about 12M neurons and 70B synapses

Selected features of STACS



STACS supports network generation over spatially-defined topologies. Here, a visualization of a 6000 neuron network over a sphere.

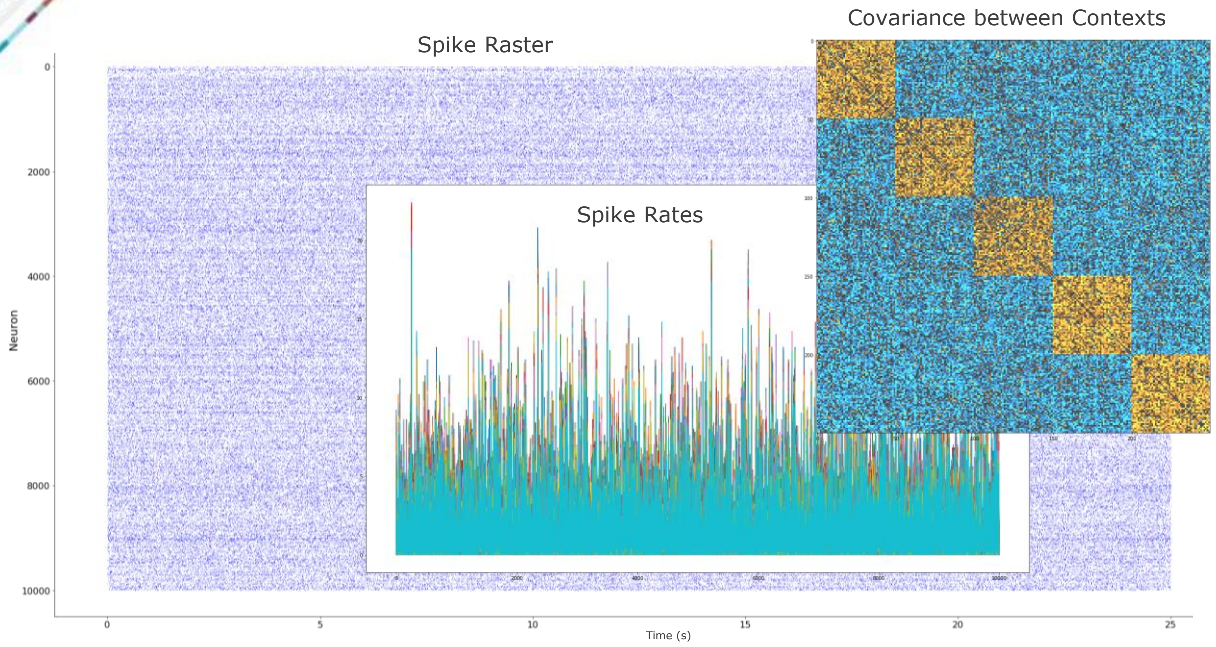
The SNN-dCSR format supports network (re)partitioning, making it suitable for computational parallelism, whether its target platform is between nodes on an HPC system or between chips on neuromorphic hardware.



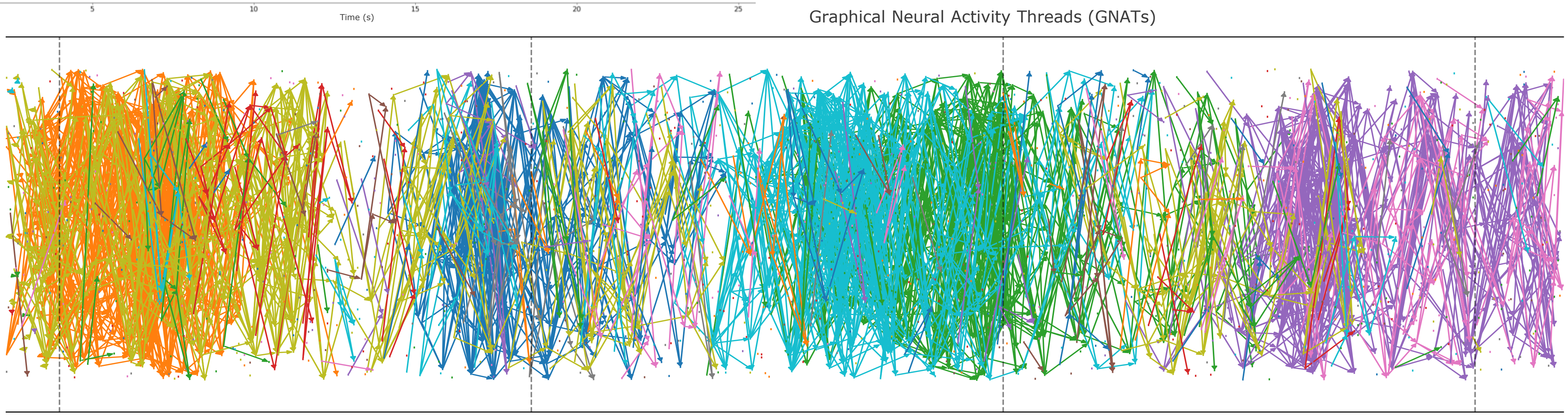

```
dist     adjcy.0       adjcy.1            adjcy.2
0 0      1 3 4         0 1 3 5 8 9        3 4 7 9 11
4 13     0 4           4 9                4 5 8
8 29     3 6 7         2 7                7 11
12 42    0 2 4 7 8     2 3 6 8 10 11      7 8 10

   state.0          state.1             state.2
   0 ∅ b ∅          4 c d f ∅ ∅ ∅       8 ∅ l ∅ q ∅
   1 a ∅            5 g ∅               9 m n ∅
   2 ∅ h ∅          6 ∅ o               10 ∅ u
   3 ∅ e ∅ j k      7 i ∅ ∅ p r ∅       11 s t ∅
```

Example 3-way partitioning of a simple network with 12 vertices and 21 directed edges using the SNN-dCSR format. Directed edges with associated state are bolded.

Selected features of STACS

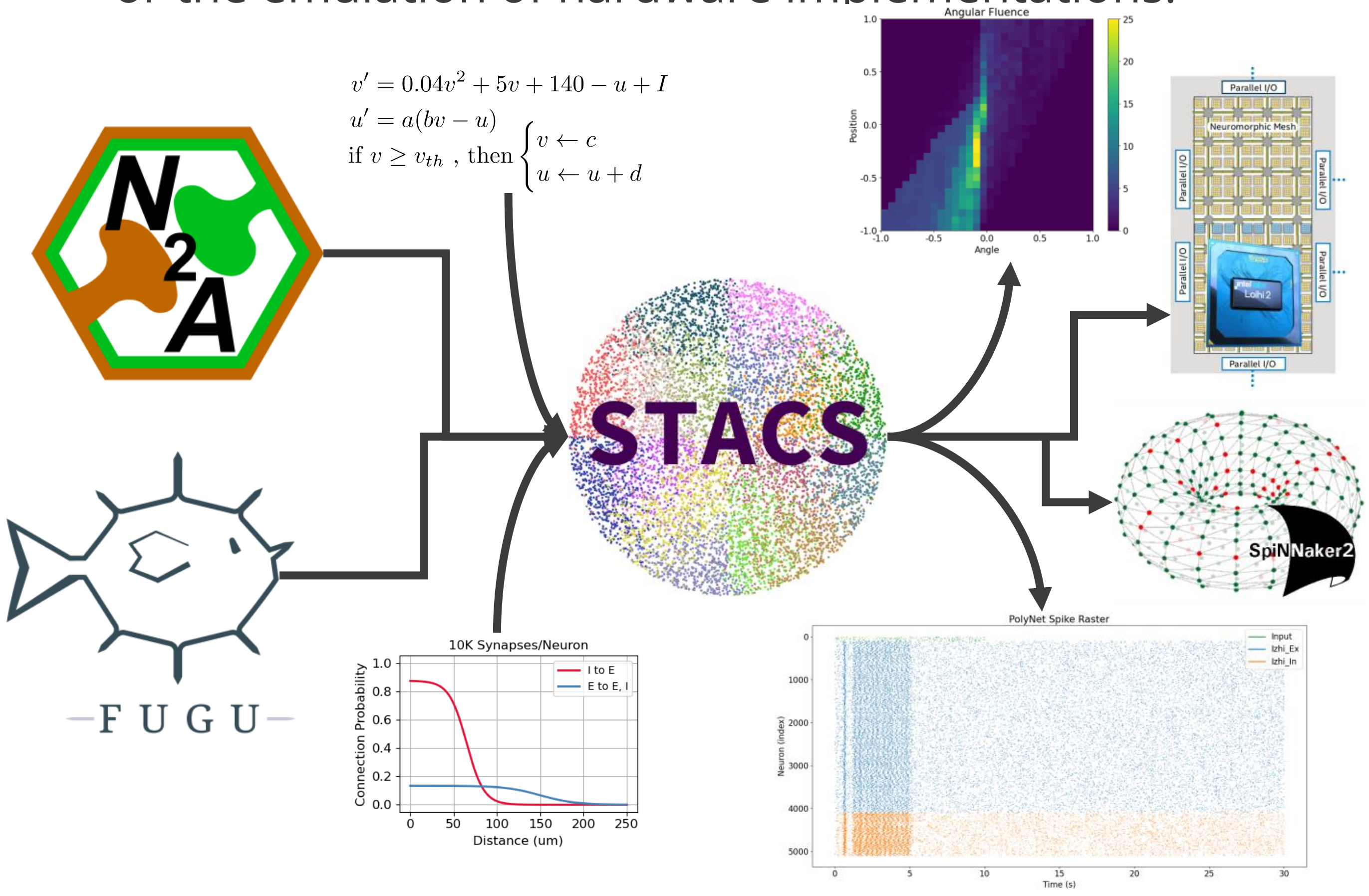


Network snapshots, state probes, and spike events can be recorded from large-scale and long-running simulations for sophisticated offline analysis. Computing the separability of spike rates between contexts (left). Identifying recurrent causal activity patterns through a combination of network structure and spiking activity (below).

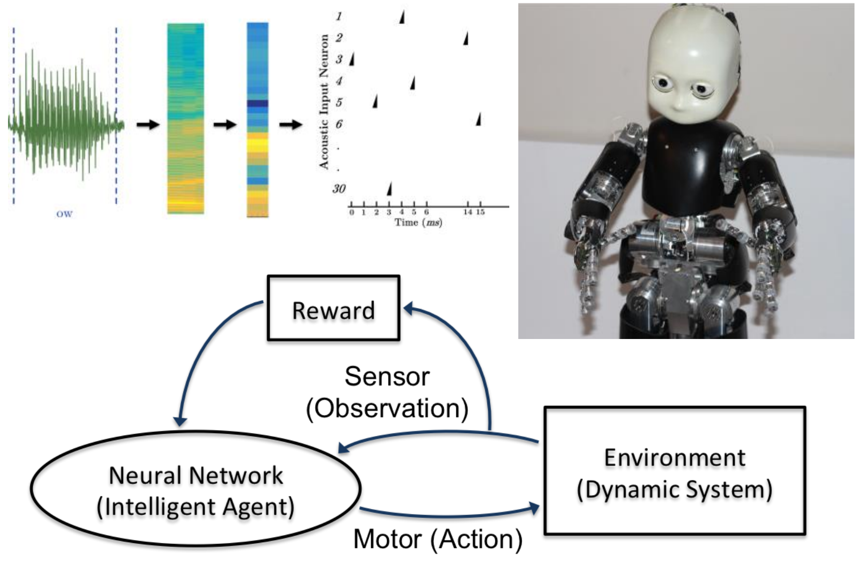


Selected features of STACS

STACS supports user-defined model dynamics (structured as SDEs through an abstract class), which enables algorithm and model exploration, or the emulation of hardware implementations.



STACS also supports interfacing with devices through YARP, which enables closed-loop system exploration and external communication and control.



Where STACS sits within the neuromorphic ecosystem

- Description Languages
  - PyNN, NeuroML, NineML, etc.
- Software Frameworks
  - N2A, Fugu, Lava, Nengo, etc.

STACS is able to interoperate with software frameworks such as N2A or Fugu through:
- Translating between higher-level network description languages (N2A → YAML)
- As a simulation backend for instantiated networks (Fugu → NetworkX → SNN-dCSR)

- Network Simulators
  - NEST, NEURON, Brian, GeNN, etc.

STACS is primarily a spiking neural network simulator

- Data Formats
  - NIR, SONATA, NetworkX, GEXF, etc.
- Hardware Platforms
  - Loihi, SpiNNaker, BrainScaleS, etc.

The partition-based SNN-dCSR data format supports external tool interoperability:
- Such as graph partitioners (e.g. ParMETIS), and network analysis (e.g. GNATFinder)
- It also provides a path toward mapping instantiated networks to different neuromorphic hardware platforms

N2a

Sandia's attempt to answer the above questions.

- "Neurons to Algorithms"
- In development since 2011
- Open source, available on Github

Main features

- Object-oriented, declarative language
- Integration with Git for team-based modeling
- Runs jobs on supercomputers
- Parameter sweeps and optimization

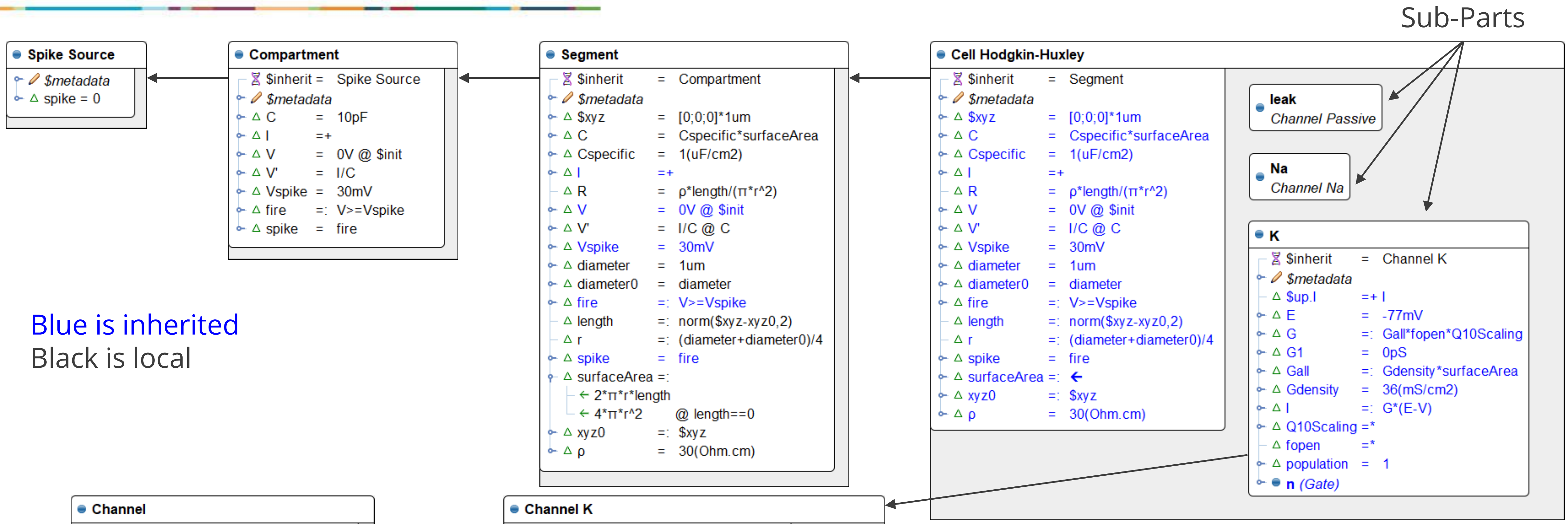
Needs

- Better supercomputer backend
- More support for model-exchange formats

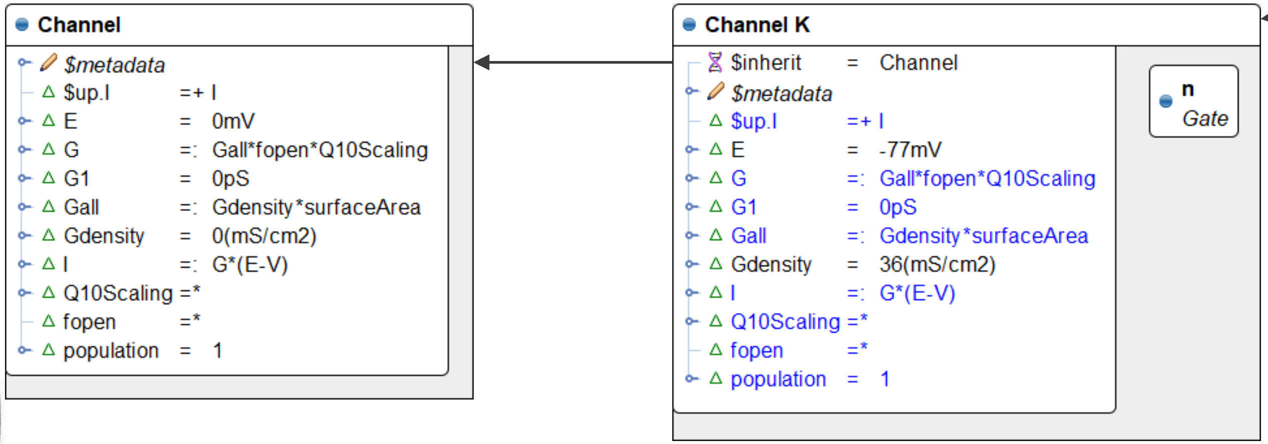
https://github.com/sandialabs/n2a

Please join us in building this tool!

Inheritance



Blue is inherited
Black is local



- The part set illustrated here is based on the NeuroML schema and its LEMS code.
- N2A can read and write NeuroML/LEMS models.
- You can define any part set you wish, even in a domain other than neuroscience.

Summary

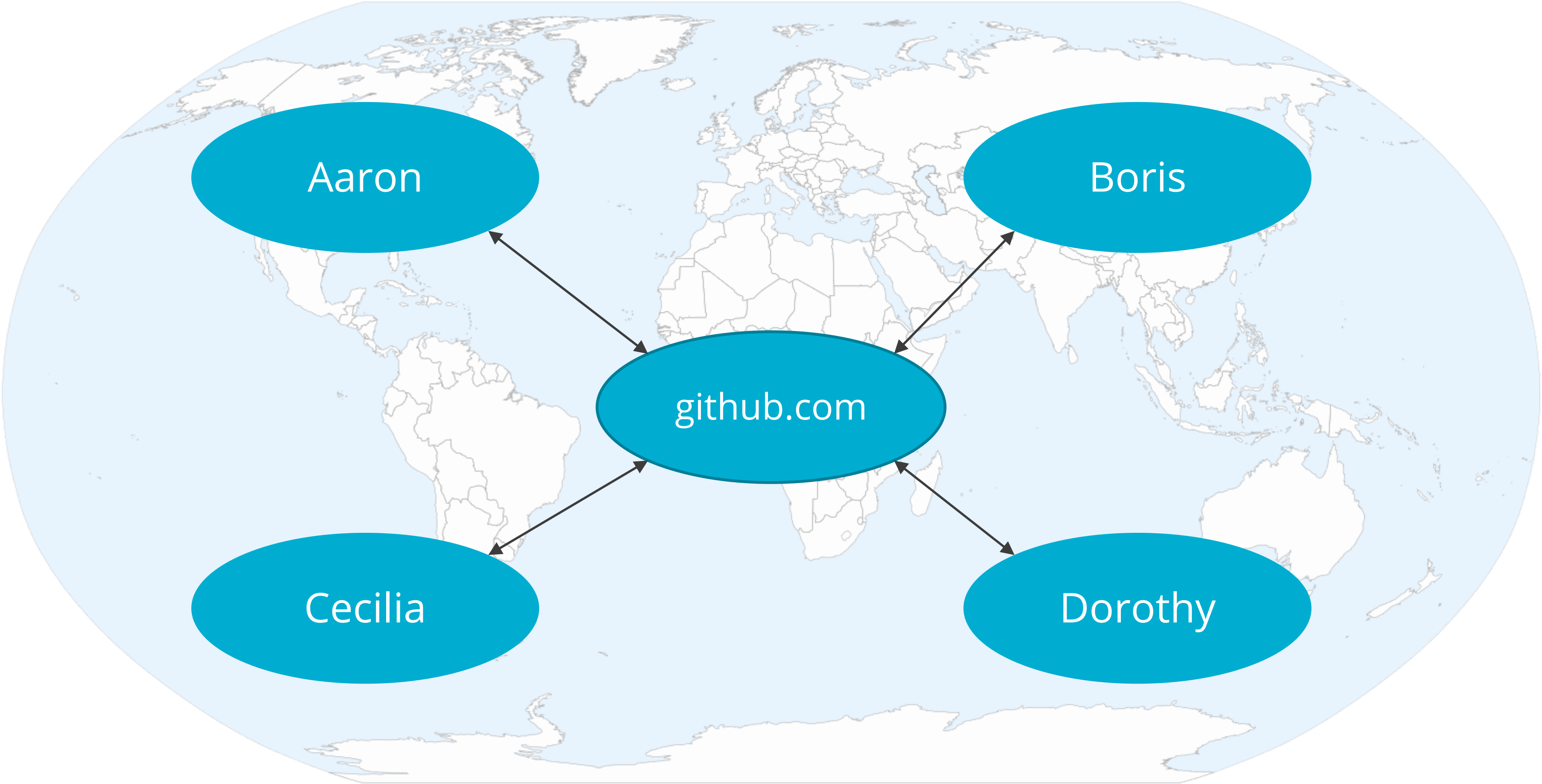
- Dynamical system modeling language and tool
- Simulator agnostic
- Enables teams of neuroscientists to collaborate on large-scale / complex models

- Parts defined with simple set of equations. No need to program.
- Build complex structures from simple structures by reusing parts.

- Publicly available: https://github.com/sandialabs/n2a
- Continuous development, supported by major national laboratory

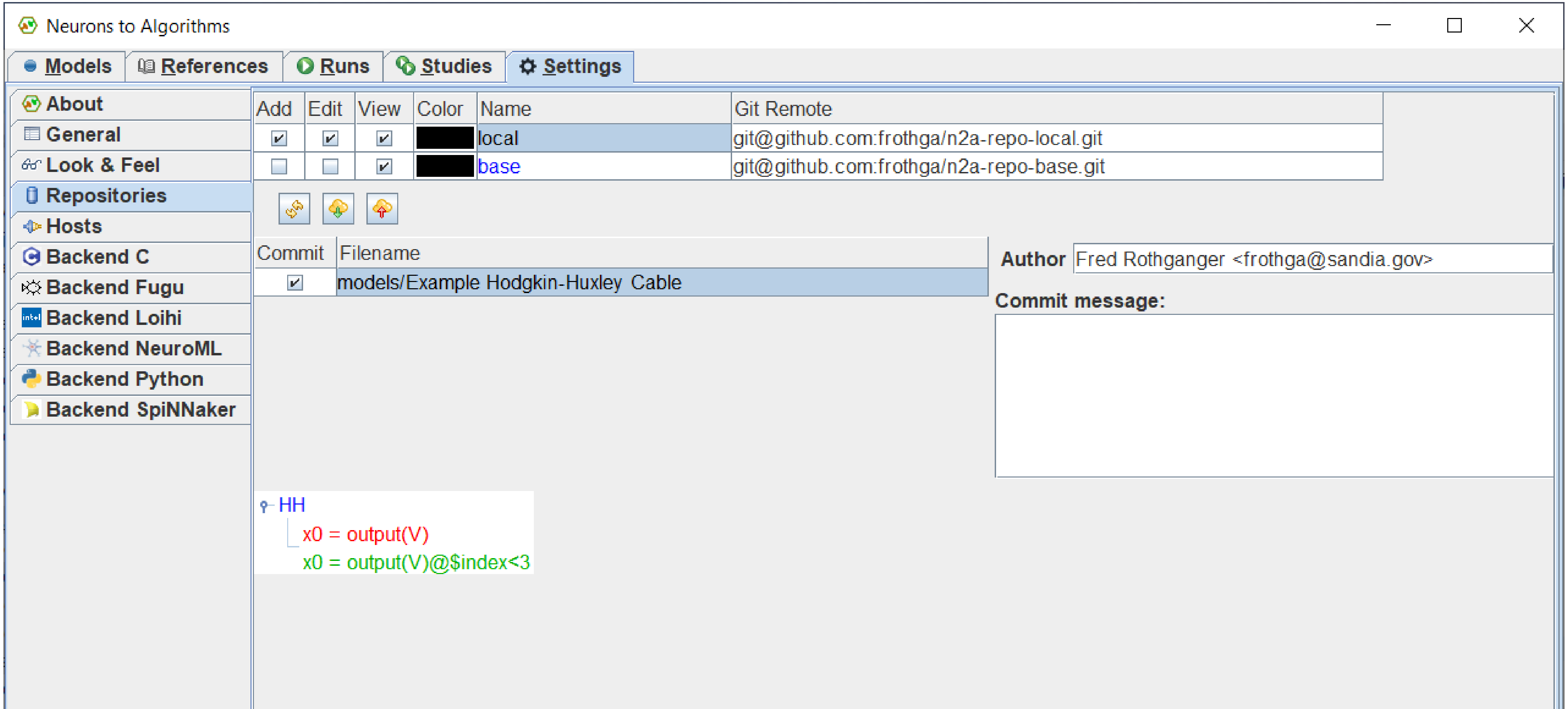
Collaboration

- Repository – collection of Parts, version-controlled with git
- Each user can have several repositories
- Each repo can be linked with an upstream git server

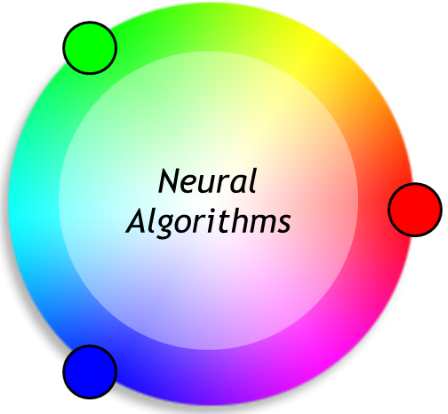


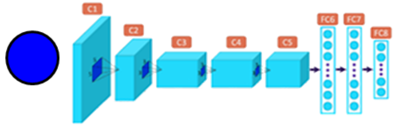
Collaboration

- UI for managing repositories
- Merges Git pulls correctly, preserving local work



So what about these other cognitive and AI applications of neuromorphic?



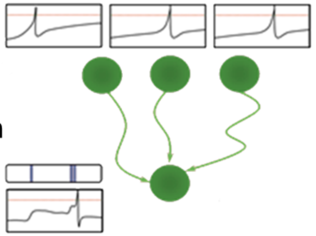


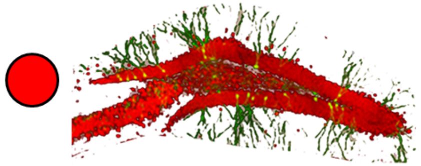
Artificial neural networks

- Generic layers of non-linear nodes
- SGD optimization of weights
- Powerful machine learning capabilities through learning sequential non-linear mappings and function approximation

Spiking neural algorithms

- Hand-crafted circuits of spiking neurons
- Model of parallel computation
- Energy efficiency through event-driven communication and high fan-in logic





Neuroscience-constrained algorithms

- Circuit architecture based on local and regional neural connectivity
- Computation incorporates broad range of neural plasticity and dynamics
- *Generally still unexplored from algorithms perspective*

snnTorch Norse SpykeTorch PySNN WHETSTONE

FUGU

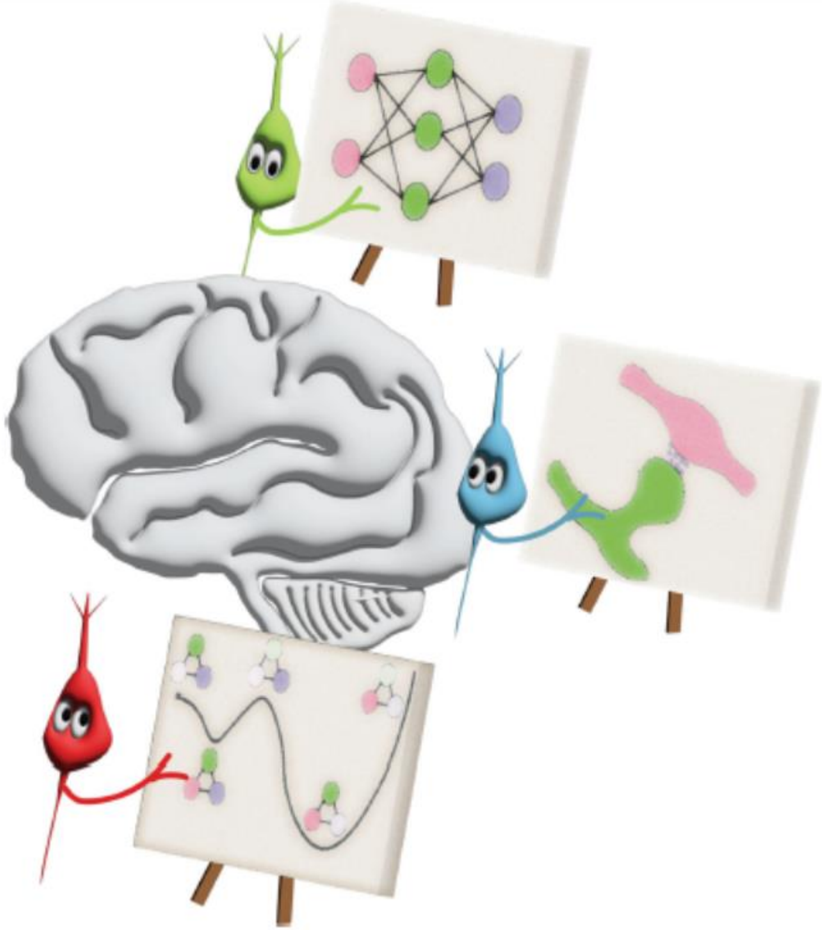
STACS N2A BRIAN nest::simulated()

Thank You!

Fugu
- William Severa, Craig Vineyard, Srideep Musuvathy, Yang Ho, Leah Reeder, Michael Krygier, Fred Rothganger, Suma Cardwell, Ingri Lane, Aaron Hill, Zubin Kane, Sarah Luca, …

STACS, N2A, and Neural Simulations
- Felix Wang, Fred Rothganger, Brad Theilman, ..

Broader Sandia Neuromorphic Algorithms Team
- Darby Smith, Ojas Parekh, Rich Lehoucq, Franc Chance, Corinne Teeter, Mark Plagge, Ryan Dellana, Shashank Misra, Conrad James, Chris Allemang, Brady Taylor, Yipu Wang, William Chapman, Efrain Gonzalez, James Boyle, Cale Crowder, Clarissa Reyes, Cindy Phillips, Ali Pina …



U.S. DEPARTMENT OF ENERGY | Office of Science

jbaimon@sandia.gov

Sandia National Laboratories

ASC