SANA-FE: Simulating Advanced Neuromorphic Architectures for Fast Exploration

James A. Boyle, Mark Plagge, Suma George Cardwell, Frances S. Chance, and Andreas Gerstlauer

***Abstract*—Neuromorphic computing is concerned with designing computer architectures inspired by the brain, with recent work focusing on platforms to efficiently execute large spiking neural networks (SNNs). Future designs are expected to improve their capabilities and performance by incorporating novel features such as emerging neuromorphic devices and analog computation. There is, however, a lack of high-level performance estimation tools to evaluate the impact of such features at the architectural level, to evaluate architectural trade-offs, and to aid with co-design and design-space exploration. Existing neuromorphic simulators either do not consider hardware performance, only model abstract SNN dynamics or are targeted to a single specific architecture.**

In this work, we propose SANA-FE, a novel simulator that can rapidly and accurately estimate performance and energy efficiency of different SNN-based designs. Our simulator uses a general and configurable architecture description format that can specify a wide range of neuromorphic designs. Using such an architecture description, SANA-FE simulates system activity when executing a given spiking application at an abstract time-step granularity, and it uses activity counts and per-activity performance metrics to estimate energy and latency for each time-step. We further show a calibration methodology and apply it to model performance of Intel's Loihi platform. Results demonstrate that our simulator can predict Loihi's energy and latency for three real-world applications, within 12% and 25%, respectively. We further model IBM's TrueNorth architecture, simulating a random network over 20× faster than existing discrete-event based TrueNorth simulators. Finally we demonstrate SANA-FE's design-space exploration capabilities by optimizing a Loihi baseline architecture for two applications, reducing run-time by 21% while increasing dynamic energy usage by only 2%.

***Index Terms*—neuromorphic computing, machine learning, analytical tools, codesign**

I. Introduction

NEUROMORPHIC computing uses neural-inspired elements to accelerate and efficiently execute a wide range of applications, such as mimicking biological circuits [1]–[3], solving NP-hard optimization problems [4], [5] and accelerating machine-learning at the edge [6]. In particular, neuromorphic architectures have been implemented to efficiently execute Spiking Neural Networks (SNNs). SNNs extend artificial neural networks (ANNs) by encoding information in time as either rates or delays between spiking events, shared between neurons via their weighted connections. SNN-based platforms are event-driven, resulting in naturally sparse and noise-tolerant computation.

J. Boyle and A. Gerstlauer are with The University of Texas at Austin, TX, USA. M. Plagge, S. G. Cardwell and F. S. Chance are are with Sandia National Laboratories, NM, USA.

Manuscript received June 5, 2024; revised October 28, 2024.

A range of different SNN-based architectures have been proposed and implemented [7]. However, there are a number of design choices when creating a spiking neuromorphic architecture – both high-level, for example the numbers of cores on the chip, and low-level, such as the features, hardware device types and algorithms supported in the computational neuron blocks. This design requires tools in early design stages to rapidly explore corresponding design spaces. Existing functional SNN simulators model the behavior of spiking neural networks, but do not capture hardware details of any underlying execution platform, such as performance or energy consumption [8], [9]. By contrast, performance modeling of SNN-based platforms using traditional hardware simulation techniques e.g., at the RTL or cycle-accurate level, is too slow to support rapid, early design-space exploration. Higher-level simulators of SNN platforms exist, but they are based on discrete-event models that simulate the precise timing of every event [10], which generally is still too slow.

In this paper, we propose SANA-FE, a novel high-level simulator of advanced neuromorphic architectures for fast exploration. SANA-FE is flexible and extensible allowing modeling of different architectures, and estimating the energy consumption and timings of a design executing a SNN application. We define generic and canonical file formats for the simulator to describe both SNN-based hardware platforms and SNNs mapped onto them. SANA-FE uses an abstract and high-level execution model that groups and simulates hardware events at a coarse time-step granularity. Activity collected in each step is used by the simulator to estimate the total performance and dynamic energy used per step.

We further introduce a method for calibrating SANA-FE to accurately match performance and energy of existing hardware. We propose a hierarchical approach to calibrate SANA-FE at the functional unit, core, tile and whole chip level in a bottom-up fashion. Using our methodology, we calibrate SANA-FE against Intel's Loihi SNN-based platform, and compare energy and performance estimates against measured values on three benchmark applications [11]–[13].

In prior work, we proposed an initial version of SANA-FE [14]. However, our earlier simulator was only demonstrated for small benchmark applications, and it did not accurately track the latency impact of cross-core interactions and network-on-chip (NoC) contentions. In this work we extend our simulator with a scheduling algorithm to model cross-core interactions, calibrate our simulator against real-world hardware, showing that it can accurately track latency and energy for real-world architectures, and furthermore demonstrate SANA-FE's rapid design-space exploration capabilities.

Our contributions are as follows:

- We introduce SANA-FE, a fast and accurate simulator for estimating the performance and energy efficiency of SNN-based hardware architectures. SANA-FE uses a novel approach to simulate spiking hardware at an abstract, coarse-grained time-step granularity while accurately modeling chip activity including inter-core communication and message scheduling effects.
- We define extensible file formats to generally describe SNN-based hardware platforms and spiking neural networks mapped to these platforms.
- We introduce a simulator calibration methodology that hierarchically and systematically characterizes and models the performance and energy of real-world hardware from individual core pipeline stages all the way up to the full on-chip network level.
- We demonstrate SANA-FE by calibrating against Intel's Loihi platform, demonstrating 11.7% and 24.3% average absolute error predicting energy and latency, respectively, of different applications. We further model IBM's TrueNorth architecture, showing over $20\times$ speedup compared to existing TrueNorth simulators. Finally, we explore design trade-offs for a Loihi-based architecture, optimizing the number and size of cores for two applications.

This paper is organized as follows. Sections II and III describe related work and design patterns of various spiking architectures that our tool represents. Sections IV and V then detail the simulator design and simulation algorithms, including input formats and main functionality. Section VI explains our bottom-up calibration methodology, and Section VII details experiments to evaluate SANA-FE. Finally, Section VIII concludes the paper with a summary and outlook.

II. Related Work

There are a range of simulators that either model the functional behavior or dynamics of biological SNNs, or behavior of SNN-based hardware platforms. Simulators such as the NEural Simulation Tool (NEST) [15], CARLsim [16], Brian 2 [17], and SuperNeuro [18] model SNNs at a biological level, potentially in a high level of detail, e.g., using a system of ordinary differential equations to describe properties of a neuron cell. Such simulators do not model any details of dedicated neuromorphic hardware, which generally execute SNNs at an abstracted level far away from biological reality.

Simulators, such as Brian2Loihi [19] and those in the Nengo [8] and Lava tools, model such abstracted, spiking hardware behavior at a purely functional level [9]. Nengo, Brian2Loihi and Lava are frameworks for designing and deploying SNNs. They support compilation, mapping and execution of SNNs on existing hardware platform such as CPUs, GPUs, or Intel's Loihi. In addition, they include functionally accurate simulators to emulate target platform behavior on a host e.g., by emulating Loihi behavior on a CPU. These reproduce the functionality of a spiking chip, modeling state variables and accounting for implementation details such as hardware counters, variable bit-widths and quantization. However, these simulators do not model implementation specific behavior e.g., which functionality is executed on each core, or network activity such as the number of packets sent by a core. Therefore, these simulations are not detailed enough to estimate performance or energy when considering architectural design decisions.

More hardware-focused simulation tools exist as well, but these either model one aspect of the design or are otherwise limited. Approaches that focus on modeling of network behavior on a spiking chip have used NoC simulators with traffic patterns obtained from randomized spike generators [20], real-world hardware measurements [21], or SNN application simulators such as NEST [22] and CarlSIM [23]. These approaches accurately emulate network effects but do not model performance of other hardware components, such as synaptic memory and neuron dynamics. ATHENA [24] is an analytical tool that estimates energy for neuromorphic crossbar-based data-flow accelerators, but is not easily extendable to other spiking architectures. NeMo [10] and SST [25] both offer simulations of dedicated neuromorphic hardware accelerators leveraging discrete-event simulation techniques. SST is designed for simulation of large-scale and heterogeneous high-performance compute (HPC) clusters. SST has limited support for neuromorphic accelerators, featuring one primitive SNN processing element. By contrast, NeMo was specifically designed to simulate spiking hardware accelerators as part of a tool-chain that enables HPC traffic simulation as well as application development. However, NeMo is focused on a single hardware architecture (IBM TrueNorth) and does not provide energy or performance estimation. Both SST and NeMo simulate all hardware events by tracking their exact timings. SANA-FE, by contrast, focuses on simulation at a coarser time-step granularity, which enables significantly faster speed and potentially more flexibility as individual events do not need to be simulated and tracked.

Tools for benchmarking of neuromorphic hardware have been created at both the application and component level. Some application-level frameworks such as NeuroBench [26] can profile high-level performance metrics including spike activity and synaptic connections without simulating hardware. However, such operation-level metrics lack the detail about low-level hardware activity required for accurate energy and latency prediction. Other work has proposed a micro-benchmark suite for Loihi-based architectures [27] that uses Lava processes to characterize coarse-grain CPU and neural core communication timings. However, their micro-benchmarks do not characterize energy costs nor are they detailed enough to break down latency into individual neural hardware operations required for accurate simulation.

III. Spiking Neuromorphic Architectures

In the following, we review the design-space of existing spiking hardware platforms that we aim to model in this work. A number of spiking hardware platforms have been proposed and deployed. These designs all use a common architecture (Fig. 1), but differ in their realization of basic processing units and system-level topology. Table I summarizes the key features and parameters of existing designs.

TABLE I
COMPARISON OF LARGE-SCALE SNN-BASED HARDWARE ARCHITECTURES.

Name	Neural Core Type	Cores Per Tile	Tile Count	Neuron Model	Interconnect
Loihi [28]	Custom Digital	4	32	Leaky-Integrate-and-Fire (LIF)	NoC Mesh
Loihi 2	Custom Digital	4	32	LIF & Microcode-programmable	NoC Mesh
TrueNorth [29]	Custom Digital	1	4096	Augmented Integrate-and-Fire	NoC Mesh
SpiNNaker [30]	CPU-based	18	1	Any software-based	NoC
SpiNNaker2 [31]	CPU-based	4	38	Any software-based	NoC
Tianjic [32]	Custom Digital	1	156	LIF / sigmoid, tanh	NoC Mesh
Novena [33]	Custom Digital	1	16	LIF	NoC Mesh
Neurogrid [34]	Custom Analog	1	16	Ion channel model	Digital tree-based
BrainScaleS-2 [35]	Custom Analog	1	1	Adaptive Exponential	Digital bus

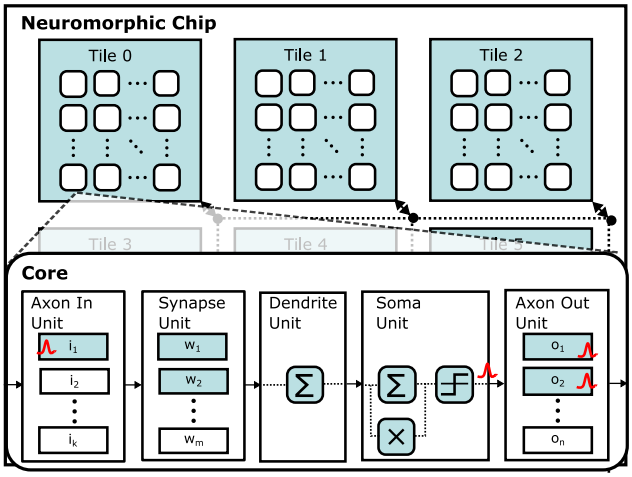


Fig. 1. Generic model of a spiking architecture.

Designs are generally organized as a set of tiles connected over on-chip interconnect, such as cross-bars or a network-on-chip (NoC). Each tile contains one or more cores that share network resources, where each core computes the state of a subset of neurons mapped onto it, and where neurons sharing the same core access hardware in a time-multiplexed manner. Different types of neuromorphic cores are implemented by each platform, using either purely software-based, custom digital or analog realizations, with a range of supported functionalities. However, all designs follow a similar approach to processing spikes.

Each core realizes a custom pipeline to process incoming spikes and groups of spiking neurons mapped to that core. Despite variations in implementation across designs, cores process spikes using a common sequence of neural-inspired operations. Spike messages are received over the network by a core at the input of an axon unit. The axon unit performs a lookup and generates a set of weight addresses. The synaptic unit then loads and processes each weight, filtering them according to a synaptic model before forwarding a current to the dendrite unit. The dendrite unit uses the connectivity and synaptic currents to accumulate and forward a single current to the soma unit. The soma unit in turn performs calculations to simulate behavior of a neuron's membrane potential, applying leaking and integrating the input current over time. If the membrane potential meets a threshold condition, a spike is sent to the axon output unit and the potential is reset. The output axon unit triggers in turn a lookup of all destination cores to send spike messages to. Finally, the core sends one or more messages locally or globally, to be delivered to input axons of connected neurons.

The large-scale SNN-based platforms considered in this work operate in logical time, using a global time-step based execution mechanism to ensure deterministic behavior. Within each global time-step, cores execute a small increment of network time in which each neuron may fire only once. A global synchronization barrier ensures all processing and communication in the current time-step has completed before incrementing logical time and processing the next time-step. This approach allows many neurons to be time-multiplexed over shared hardware, allowing for scalability that is only limited by storage requirements for neuron state. In this approach, there must be a buffer before one of the hardware units, where units before and after the buffer are processed in different time-steps. Time-steps can either run at a fixed rate using an external synchronization mechanism e.g., a low-frequency clock, or dynamically adjust their length using an internal barrier synchronization based on when all cores have completed their computation. Dynamically adjusting the time between synchronization allows computation to scale depending on the amount of spiking activity, i.e., the latency per time-step varies dynamically, limited by the slowest core on the chip.

Another class of SNN-based architectures, not shown in Table I, operates purely in the analog domain based on physical time [36]. In such architectures, neurons cannot share computing resources, limiting scalability. Such architectures are out of the scope of what SANA-FE aims to model.

IV. SIMULATOR DESIGN

Fig. 2 shows an overview of SANA-FE. The simulator requires a description of a hardware platform, a description of an application, i.e. of an SNN mapped to the hardware platform, and command line inputs such as the number of time-steps to simulate. Using these, the kernel simulates the performance of the design in an abstract and coarse-grain time-step loop. In the following sub-sections, we describe each component of the simulator in more detail.

A. Architecture Description Format

We have defined a hierarchical, YAML-based file format to describe SNN-based platforms based on a general architecture template that is derived from existing architectures discussed in Section III. In our file-format, a SNN-based architecture is generally described as a number of connected tiles, each

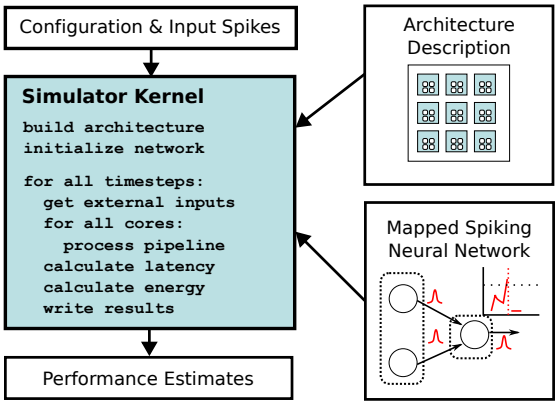


Fig. 2. SANA-FE overview.

```
architecture:
  name: demo
  attributes:
    width: 2
    height: 1
    link_buffer_size: 4
  tile:
    - name: demo_tile[0..1]
      attributes:
        energy_north_hop: 2.0e-12
        latency_north_hop: 1.4e-9
        ...
      core:
        - name: demo_core[0..3]
          attributes:
            buffer_before: soma
          axon_in:
            - name: demo_in
          synapse:
            - name: demo_synapse
              attributes:
                model: current_based
                energy_process_spike: 20.0e-12
                latency_process_spike: 3.0e-9
          dendrite:
            - name: demo_dendrite
          soma:
            - name: demo_soma_default
              attributes:
                model: leaky_integrate_fire
                energy_access_neuron: 20.0e-12
                latency_access_neuron: 3.0e-9
                energy_update_neuron: 10.0e-12
                latency_update_neuron: 1.0e-9
                energy_spike_out: 60.0e-12
                latency_spike_out: 30.0e-9
            - name: demo_soma_alt
              attributes:
                model: leaky_integrate_fire
                ...
          axon_out:
            - name: demo_out
              attributes:
                energy_message_out: 100.0e-12
                latency_message_out: 5.0e-9
```

Listing 1. An SNN-based hardware platform specified in our architecture description format.

containing one or more computational cores. Each core has a fixed sequence of compute units to process spike messages. The user specifies the tiles, cores and compute units in a design as separate sections in the YAML file, and the implementation details of these units using a set of attributes within their sections.

An example of an architecture description is shown in Listing 1. A particular platform is defined under the top-level `architecture` section as one or more `tile` sections. Our architecture description format has keywords that define sections, and all sections contain `name` and `attributes` fields. The `name` is a string description which can optionally include a range in square brackets, indicating multiple instances of a section. Each attribute is a set of name-value pairs, describing features of the hardware or cost metrics. Costs in particular must be specified for the latency and energy of updating each hardware unit.

In this example, there are two tiles defined under the `architecture` section. At the tile level two energy and latency costs specify the energy and latency required to send spike messages one network hop in a given direction. Tiles also specify the size (in messages) of their link buffers. Each tile contains one or more `core` sections, specifying the computational cores in a tile including their individual compute units. Here, tiles have four cores each. The keyword `buffer_before` sets the position of the time-step buffer. The time-step buffer determines the boundary between processing different stages of the pipeline in different time-steps. In this example, the output of the dendrite unit is buffered to be read by the soma unit in the next time-step.

Within each core, we then define the hardware units in that core, starting with the `axon_in` unit which has no attributes in this example. The `synapse` unit specifies a current-based synaptic model, and has an energy and latency cost for processing one inbound spike. The dendrite unit realizes a default behavior of summing weighted inputs from the synapse unit, and in this example also has no attributes. The example defines a list with two alternative soma implementations: `example_soma` and `example_soma_alt`. The first unit in the list is the default, although mapped neurons can optionally specify which soma unit to use in the SNN description file. For both `soma` units, the example specifies that a leaky-integrate-and-fire neuron model is used. The two units differ only in their associated update costs, which include a baseline energy and latency cost for accessing and reading a neuron's state in the soma, and four additional costs associated with writing to a neuron's membrane potential and generating an output spike. Finally, the `axon_out` unit has cost metrics for sending a spike message to the network.

B. Mapped SNN Format

We have further defined a file format to specify SNN applications mapped onto a given architecture description for execution in the simulator. The file format generally describes an SNN as a graph of neurons connected by weighted edges, where each neuron can have attributes controlling its dynamics, and where neurons can be grouped to define common attributes, e.g., per network layer. In addition to the SNN graph itself, the file format also specifies a mapping associating each neuron with a hardware core in the design.

An example is shown in Listing 2, corresponding to the SNN in Fig. 3 mapped into the architecture defined in Listing 1. A mapped SNN file in general has four types of entries. There is one entry per line and each entry is prefixed by a single character denoting its type i.e., `g` (neuron group), `n`

```
g 3 threshold=1.0
g 3 threshold=2.0 soma_hw_name=demo_soma_alt
## Neuron groups
n   0.0    bias=1.0  connections_out=1
n   0.1    bias=0.0  connections_out=1
n   0.2    bias=1.0  connections_out=1
n   1.0    bias=0.0
n   1.1    bias=1.0
n   1.2    bias=0.0
## Edges
e   0.0->1.0   weight=-1.0
e   0.1->1.2   weight=-2.0
e   0.2->1.2   weight=3.0
## Mappings
&   0.0@0.0
&   0.1@0.0
&   0.2@0.1
&   1.0@0.0
&   1.1@0.0
&   1.2@0.1
```

Listing 2. A mapped SNN described in SANA-FE.

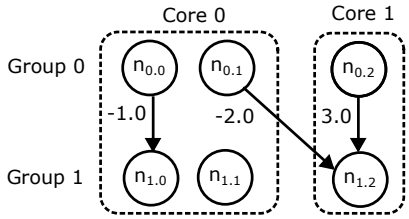


Fig. 3. The mapped SNN from Listing 2.

(neuron), `e` (edge), `&` (mapping) or # (comment). The example in Listing 2 first defines two neuron groups containing three neurons each. The group entry specifies the number of neurons and their shared parameters, including the names of pipeline hardware units to execute on. In the example, the threshold potential is common between neurons, and the second group executes on the `demo_soma_alt` soma unit. We then define six neurons, which are indexed by a group number and group offset e.g., `0.0` for the first neuron in group 0. Next, we define three weighted edges, linking neurons in group 0 to neurons in group 1, where each edge describes a weighted connection between two neurons. Finally, we map the six neurons to two cores, assigning four neurons to core 0 and two to core 1.

C. Simulation Kernel

Using input files, our simulator emulates functionality, latency and energy of the mapped application executing on the given architecture at an abstract time-step granularity. At the core of the simulator is the simulation kernel, which loads its input files and executes a loop that simulates each time-step. During each time-step we simulate neuron behavior, calculate per-step energy and latency estimates, and write these estimates to a trace file.

During loading of the input files, SANA-FE uses the architecture description to initialize class objects for each hardware unit and loads a mapped SNN into a set of neuron objects linked to their mapped core. Neuron objects track SNN state, such as soma membrane potential, synaptic current and outgoing weighted connections.

After initialization, the simulator enters the main time-step loop. The time-step loop emulates the functionality and models the activity in each cores' hardware pipeline. Within this pipeline, we model the updates to hardware units and

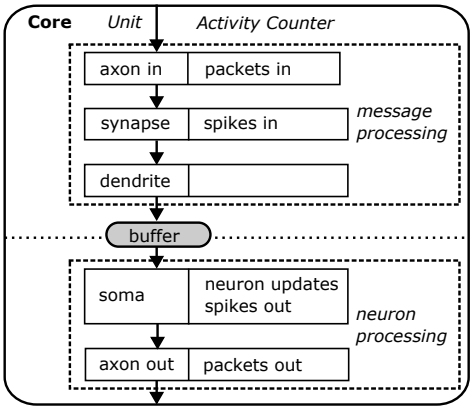


Fig. 4. A simulated core pipeline described in Listing 1.

increment hardware activity counters used later for energy and latency estimation, e.g., the total spikes processed by the synapse unit, or the number of neurons updated by the soma. SANA-FE's energy and latency predictions use these dynamically simulated activity counts, which are integrated with its functional simulation of hardware units. While energy can be estimated purely analytically as a simple weighted sum of hardware activity counts, accurate prediction of latency requires modeling of pipelining and other timing effects.

A pipeline example corresponding to the design in Listing 1 is shown in Fig. 4. As previously discussed, there is a buffer before one of the units in the pipeline. The buffer's location is given in the architecture description, and it defines the time-step boundary. In this example, the time-step boundary is defined between the dendrite and soma units. In the real system, hardware units before the time-step buffer receive and process messages from the network in incoming message order, and then store intermediate updates in the pipeline buffers. In parallel, hardware units after the buffer sequentially read previously buffered values and process neurons in a fixed order. At each time-step boundary, buffered values written in the current time-step are copied into a double buffer for reading as inputs in the following time-step. In the simulator, we can emulate this pipeline behavior by generally estimating time-step latency analytically as the maximum of message and neuron processing delays [14]. However, message dependencies and NoC contentions can introduce additional stalls that need to be accounted for to achieve accurate latency estimates.

V. Simulation Algorithm

In the following, we describe the simulation algorithm of SANA-FE in more detail. The core time-step loop of our simulator kernel is shown in Algorithm 1. A complete list of variables for the time-step and message scheduling algorithms is given in Table II.

The time-step loop has three parts: a neuron processing loop, a message processing loop and an event-based message scheduling algorithm. The neuron processing loop first iterates over all cores and neurons mapped into each core to execute all stages after the time-step buffer, i.e., processing all neuron updates and determining spikes to send for the current time-step. In the example in Fig. 4, neuron processing models

Algorithm 1 Time-step Loop

```
for all timesteps do
    update_external_inputs()
    for all cores c do
        M_c = {}
        for all n_c neurons mapped to c do
            M = process_neuron(n_c)
            M_c.append(M)
    for all cores c do
        for all m in M_c do
            m.d^msg = process_message(m)
    e^step = calculate_energy()
    d^step = schedule_messages(∀c : M_c)
    write e^step, d^step to file
```

TABLE II
SIMULATION AND SCHEDULING ALGORITHM VARIABLES.

Variable	Description
M	A list of messages
M_c	A list of messages sent by core c
d^{nrn}	The neuron processing delay
d^{msg}	The message processing delay
src	A message's source (sending) core
dst	A message's destination (receiving) core
m	A message 4-tuple $\langle d^{nrn}, d^{msg}, src, dst\rangle$
e^{step}	The total time-step energy
d^{step}	The total time-step delay
M^{net}	A list of messages sent to the network
t_m	The time at which message m is sent
t'_m	The time at which message m is received
$\hat{t}_m$	The time at which processing of message m finishes
$\hat{t}_c$	The time at which core c finishes processing its messages
m_{next}	The next message to be scheduled
k	A link between two adjacent network tiles
P	A list of all links in a message's path across the NoC
b_k	The expected buffer utilization for a single link k
b_m	The total buffer utilization for all links traversed by m
d_k^{hop}	The hop delay for link k
d_m^{net}	The network traversal delay for message m
$\hat{b}$	The mean buffer capacity (messages per link)
$\bar{d}^{msg}$	The mean message processing delay for in-flight messages

the behavior, state and activity counter updates of the soma functionality and of producing messages at the output axon. The neuron processing function internally updates activity counters that track the updates in each hardware unit. Additionally, the processing function returns a list M of spike messages generated by neuron n_c in the core c. Each message m is defined as a tuple containing four elements: the neuron processing delay to generate the message in the sending core ($m.d^{nrn}$), the message processing delay in the receiving core ($m.d^{msg}$), and the source ($m.src$) and destination ($m.dst$) cores. Note that the receive delay $m.d^{msg}$ is zero-initialized and will be assigned later. Also, if the destination field is null, the tuple serves as a placeholder for any neuron processing that does not result in a message being sent.

After all neurons have been processed, the message processing loop processes all spike messages in M_c for each core, which updates the time-step buffers in the receiving core for the next time-step. In the given example, this includes all axon input, dendrite and synapse processing including updates of associated incoming packet and spike activity counters on the receiving side. The processing of received messages is performed in a loop across cores and messages. As part of this, the time taken to process each message is calculated, and the message tuple m is extended with the message processing delay, d^{msg}. Messages are processed sequentially and without accounting for message ordering across the chip. Ordering effects are emulated by a separate message scheduling step as part of calculating the final energy and latency estimates for the time-step, e_{step} and d_{step}. We will describe the message scheduling algorithm next.

A. Message Scheduling

Time-step latency generally depends on the time taken to process neuron updates as well as the time to send and process spike messages. The latter may be affected by cross-core interactions and contention in the on-chip network and related hardware. In particular, if there is contention in the network or in a receiving core, the sender of the message may be stalled and its overall processing latency increased accordingly. Whether senders will be stalled and for how long will in turn depend on the precise ordering of messages across the chip.

Fig. 5 shows three scenarios of cores interacting and affecting each other's timing. Fig. 5(a) shows two cores processing neurons and messages without contention. At time t_{start}, Cores 1 and 3 start by processing their first neurons in parallel. After Core 1 processes its neuron 1.1, it sends message m_1 to Core 3 at time t_{m_1}. Message m_1 is received after some network delay at time t'_{m_1}, and Core 3 finishes processing m_1 at time $\hat{t}_{m_1}$. In parallel, Core 1 processes its second and last neuron 1.2. This is the simplest case, and for this example, the time-step latency is simply the sum of Core 1's neuron processing delays.

Fig. 5(b) shows another example where resource contention in the receiving core may delay a message's processing. Core 1 behaves the same as in Fig. 5(a), but Core 2 also sends a message m_2 to Core 3 at time t_{m_2}. Since the message processing pipeline at Core 3 is busy (shown in red), message m_2 must wait in an internal buffer in Core 3 until m_1 is processed at time $\hat{t}_{m_1}$. In this case, we must consider the message processing order to accurately calculate latency.

Finally, Fig. 5(c) shows an example where contention in the network can delay both neuron and message processing. Here, Core 2 sends message m_1 before Core 1 tries to send m_2. Assuming that the network can only transfer a single message simultaneously, while the network is busy transporting m_1, message m_2 is blocked by the network from being sent (shown in red). Consequently, Core 1 is stalled until $t_{m_2} = t'_{m_1}$ before sending m_2 and then processing neuron 1.2. Such cases require modeling of network contention to predict latency.

As the examples show, time-steps may include a combination of such scenarios, leading to complex cross-core dependencies. Acurately estimating the total time-step latency thus requires modeling the order of messages and determining how long the network will delay messages and stall cores, which in turn depends on network parameters such as router buffer sizes. Existing NoC performance models generally either use detailed event-based or cycle-accurate simulations or analytical methods based on queuing theory [37]. Simulation-based

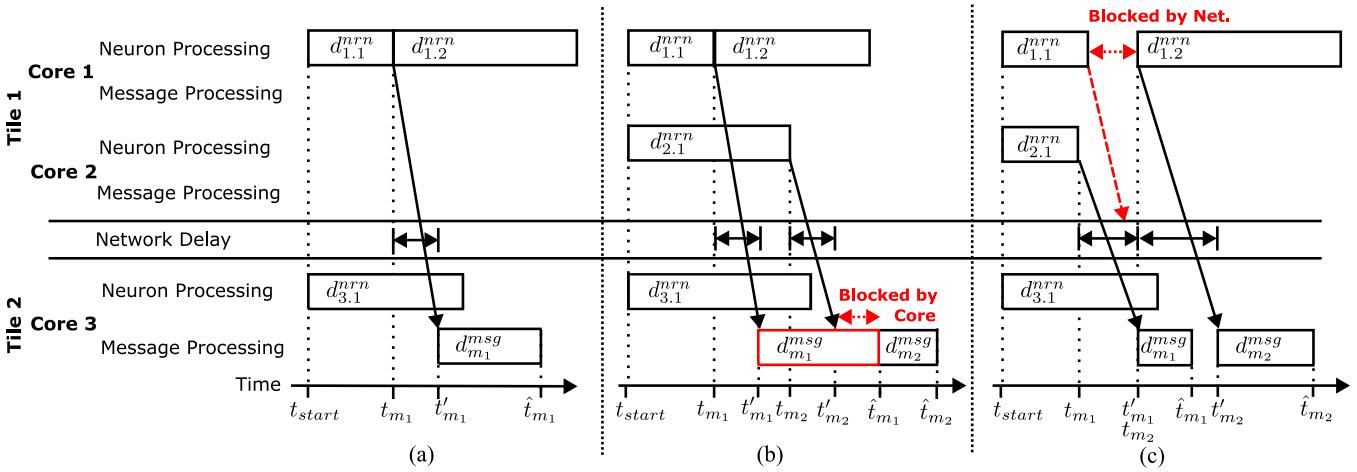


Fig. 5. Example of cross-core interactions on an SNN-based platform.

approaches will accurately predict network timings but are prohibitively slow for design-space exploration. By contrast, queuing-based analytical models can be faster but only predict long-term, steady-state and average behavior of the network, i.e., do not account for short-term transient effects, such as the small bursts of messages in spiking-based platforms that occur between frequent time-step synchronizations barriers.

We propose a semi-analytical model for use in SANA-FE that combines concepts from event-based simulation and queuing-based models to efficiently predict timing interactions. To model timing interactions between cores when exchanging messages and processing neuron updates, we simulate the exact timing of messages using a global scheduling model. Our message scheduling algorithm assigns a global order to all spike messages using latency values previously calculated in the neuron and message processing stages, and uses an analytical model to approximate network delays. In the process, message scheduling determines the timing across the chip and hence the overall time-step latency. Our scheduling algorithm and timing model are described in detail below.

Algorithm 2 Message scheduling

Input: List of messages per core M_c
Output: Total time-step delay d^{step}

1: $M^{net} = \{\}$
2: priority_queue $= \{\}$
3: **for all** cores c **do**
4: $\quad \hat{t}_c = 0$
5: $\quad$ **if** m = pop(M_c) **then**
6: $\quad\quad$ priority_queue.push($\langle m.d^{nrn}, m\rangle$)
7: **while** len(priority_queue) > 0 **do**
8: $\quad \langle t_m, m\rangle =$ priority_queue.pop()
9: $\quad$ **if** $m.dst \neq \emptyset$ **then**
10: $\quad\quad \langle d_m^{snd}, d_m^{net}\rangle$ = net_delays(m, t_m, M^{net})
11: $\quad\quad t_m = t_m + d_m^{snd}$
12: $\quad\quad t'_m = t_m + d_m^{net}$
13: $\quad\quad M^{net}$.append($\langle t'_m, m\rangle$)
14: $\quad\quad \hat{t}_{m.dst} = \max(t'_m, \hat{t}_{m.dst}) + m.d^{msg}$
15: $\quad$ **if** m_{next} = pop($M_{m.src}$) **then**
16: $\quad\quad$ priority_queue.push($\langle t_m + m_{next}.d^{nrn}, m_{next}\rangle$)
17: **return** $\max(t_m, \forall c : \hat{t}_c)$

B. Scheduling Algorithm

Algorithm 2 shows our message scheduling algorithm. It takes a list of messages for cores M_c, created previously in the time-step loop, and returns the total time-step latency, d_{step}. Our scheduler maintains a queue of message events and their time-stamps, and it processes events in time-stamp order to determine dependent events and the global event order. The algorithm starts by initializing time-stamp variables for all cores that will track the times at which they finish processing their last received messages ($\hat{t}_c$). In a loop over all cores, we then initialize the priority queue of time-stamped events such that all cores schedule their first messages after the corresponding delay for processing their first neuron (line 6).

After initialization, we loop over the event queue in time-stamp order until all message events $\langle t_m, m\rangle$ have been processed. As described in Section IV-C, each message m is a 4-tuple containing message timings, a source core and an optionally set destination core. An unset destination core, i.e., $\emptyset$, acts as a placeholder event to indicate the time when processing of a neuron ends that does not produce any message.

The algorithm first checks whether the event is a placeholder or a real message (line 9). If it is an actual message, we estimate and adjust for network delays as follows. We first call a model to calculate the network blocking d_m^{snd} and the network transmission delay d_m^{net}. The model will be later described in Algorithm 3. The message's timestamp is then increased by the blocking delay d_m^{snd}, which estimates how long the network is busy and blocks messages from being sent (line 11). Next, using the estimated network transmission delay d_m^{net}, the time t'_m at which the message is delivered by the network and arrives at the destination core is determined (line 12). Using this arrival time t'_m and the message processing delay $m.d^{msg}$, the time-stamp $\hat{t}$ of the receiving core $m.dst$ is updated to record when it finishes receiving and processing the message, after handling any existing messages (line 14). In addition, the message and its arrival time-stamp are added to a list of messages M^{net} injected into the network, needed by the analytical network delay model as described below. After handling the current message m, we pop the next message

m_{next} of the sending core (if applicable) and schedule m_{next} after its processing delay i.e., at the point the message is ready to be sent to the network (line 16). Once the event queue is empty and all messages have been scheduled, the total time-step latency is then given as the maximum of the time-stamp t_m of the last message or placeholder event and of the time $\hat{t}_c$ when all cores finishing processing their last received message.

C. Network Model

Algorithm 3 describes our analytical model used to estimate network congestion and congestion-induced network delays, i.e., the network blocking time d^{snd} and network traversal latency d^{net} for a given message m. The network model takes as input the message m to be sent, the time at which m is ready to be sent, and the list of previously sent messages passed by the scheduler. In addition, the algorithm takes the maximum link buffer size $\hat{b}$ and the per-link hop delays d^{hop} as fixed calibrated network parameter values that are read from the architecture description file.

The algorithm first determines a subset, M, of sent messages in M^{net} that have not been received by time t_m. These messages are in-flight in the network (line 1). Using the information about in-flight messages, the expected buffer utilization i.e., the expected buffer queue length b_k is calculated for each router link k. To calculate b_k, each current in-flight message in M contributes some proportional utilization to all links along its path P, where P is determined by a given dimension-order routing scheme. Each message is assumed to contribute an equal amount of utilization across all links in its path, and its receiving core i.e., across $|P|+1$ buffers.

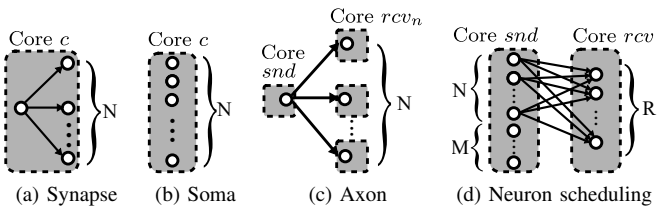
Once b_k are calculated, we estimate the total amount of buffer space b_m expected to be used by message m as sum of the b_k over all links k in the message's path. We further compute the mean processing delay $\bar{d}^{msg}$ of all in-flight messages (line 9) as the expected service time per message and hence per network buffer element. Using b_m and $\bar{d}^{msg}$, we first estimate the network back-pressure to calculate the expected delay d_m^{snd} for which message m is stalled in its sending core. Back-pressure and blocking occurs when the buffers along a message's path are full. We compute the difference between the total buffer utilization b_m and the available buffer capacity along message m's path P, and we multiply a positive difference by the expected service time $\bar{d}^{msg}$ to get the delay due to back-pressure (line 10). Finally, the message's network traversal latency d_m^{net} is calculated as the maximum of the total network hop time (assuming no network congestion), and the total expected queuing delay along the message's path, where the expected queuing delay is estimated as the average queue utilization along the message's path multiplied by the mean service time (line 11).

VI. Simulator Calibration

SANA-FE is designed to accurately model a wide range of neuromorphic system architectures that can be rooted in existing designs serving as exploration baseline. We propose a methodology to calibrate SANA-FE to closely match its energy and latency estimates to real hardware. Our calibration

Algorithm 3 Network blocking and message latency

Input: Message m
Input: Time at which m is sent t_m
Input: Messages sent to the network M^{net}
Output: Network blocking time d_m^{snd}
Output: Message network latency d_m^{net}
1: $M = \{\forall \langle t'_n, n\rangle \in M^{net} : t'_n > t_m\}$
2: $\forall k : b_k = 0$
3: **for all** messages n in M **do**
4: $\quad P =$ message_route(n)
5: $\quad$ **for all** links k in path P **do**
6: $\quad\quad b_k = b_k + \frac{1}{|P|+1}$
7: $P =$ message_route(m)
8: $b_m = \sum_{k \in P} b_k$
9: $\bar{d}^{msg} = \text{mean}(\forall n \in M : n.d^{msg})$
10: $d_m^{snd} = \bar{d}^{msg} \times \max(0, b_m - \hat{b} \times |P|)$
11: $d_m^{net} = \max(\sum_{k \in P} d_k^{hop}, \bar{d}^{msg} \times \frac{b_m}{|P|})$
12: **return** $\langle d_m^{snd}, d_m^{net}\rangle$



(a) Synapse (b) Soma (c) Axon (d) Neuron scheduling

Fig. 6. Four micro-benchmark SNNs to isolate (a) synapse unit, (b) soma unit, (c) axon unit activity and (d) the neuron processing order. Dashed boxes represent cores and circles represent mapped neurons.

methodology follows a systematic and hierarchical approach. We first characterize the performance of each pipeline hardware unit and set metrics for energy and latency described in the architecture description file. We then calibrate the neuron processing pipeline in each hardware core. Finally, we calibrate network-level parameters, such as the size of buffers across the network-on-chip. In the following, we describe the steps of our calibration methodology in more detail.

A. Core-Level Calibration

SANA-FE uses average cost metrics for each hardware pipeline unit to estimate their unit energy and latency at the individual operation level. To calibrate the simulator, these metrics must be set based on measurements of each unit operation in isolation. This can be done by simulating each unit in a low, e.g., circuit-level detailed simulation, which is only feasible if detailed implementation-level (e.g., RTL) hardware models are available. Alternatively, micro-benchmarks can be used to isolate and measure unit-specific activities [27].

Fig. 6(a)–(c) shows three SNN micro-benchmarks we use to characterize the energy and latency of each pipeline unit from hardware measurements.

1) Synapse stage calibration: We first isolate and calibrate synaptic reads by varying the intra-core connectivity while keeping all other activity constant (Fig. 6(a)). In this case, the number of neurons sending spike packets is fixed and each connected to N receiving neurons within the same core. As such, once a spike packet is received within the same core, it is expanded into N synaptic look-ups via weighted connections.

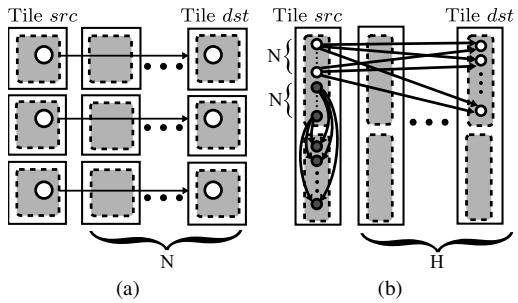


Fig. 7. Micro-benchmark SNNs for measuring network hop costs and link buffer capacity.

The time-step latency and energy usage for varying N are measured by executing this micro-benchmark on a given platform, and linear regression analysis is used to estimate the average incremental cost per synaptic look-up (regression slope). Note that the cost of a synaptic read depends on configurable parameters such as the number of bits per weight, weight sharing in convolutional kernels, or the compression scheme used. Therefore, it is important to characterize the synaptic look-up costs of all the possible synapse types that can be used by simulated applications.

2) Soma stage calibration: Soma updates are isolated using SNNs as shown in Fig. 6(b), where the number of neurons N in all cores is varied from zero to the maximum supported by the core. We measure and calibrate types of soma updates: 1) a neuron that is initialized but does *not* update its potential, 2) a neuron that updates and writes its membrane potential, and 3) a neuron that generates a spike. Each type of neuron update uses different hardware logic and therefore has different energy and latency costs. These update costs can be measured in separate experiments by using one of three different neuron configurations. Configuration (1) requires neurons to remain idle and have no input stimuli or output spikes i.e., a zero membrane potential and a positive threshold voltage. Configuration (2) requires neurons to update their potential every time-step, but this must remain less than the threshold voltage. For example, a leaky-integrate-and-fire neuron could be configured by setting its threshold and leak to the maximum positive value, and adding some input current every time-step. If possible, neuron processing should be triggered by internally biasing neurons rather than receiving spikes. For example, Loihi supports a configurable bias potential added to neurons every time-step and TrueNorth implements a reverse leak voltage. In configuration (3) neurons' thresholds and reset potentials are set to zero, triggering neurons to fire every time-step.

3) Axon stage calibration: Finally, axon reads are characterized by varying core-to-core connectivity, where one core has all of its supported neurons configured to be spiking, and each of these neurons is connected to one neuron in N other cores (Fig. 6(c)). Every connection adds another axon operation per neuron, but also another synaptic read. To isolate the cost of an axon transmitting a spike, we measure the total energy and latency for varying N and subtract the previously characterized cost of the synaptic read.

4) Neuron pipeline calibration: In addition to the individual pipeline units, we also calibrate the overall pipeline processing logic using a micro-benchmark as shown in Fig. 6(d). In particular, the neuron processing order is calibrated by configuring two neuron groups of sizes N and M. The first neuron group has N connected neurons that send spikes to R neurons on another core. The second neuron group has M neurons that are updated but do not send spikes. The two groups are mapped to the same core in two configurations: group N is passed to the compiler either *before* group M or *after* group M. If group M is executed first, spike messages from N are sent only after all neurons in M are processed. If group N is executed first, spikes are sent immediately and then neurons in M are processed in parallel to message processing, leading to faster run-time than the first execution case. Because of this effect, the order that neurons are processed can be determined by observing the run-time of both configurations and observing which neuron mapping executes faster.

B. Network-Level Calibration

Calibrating at the network level requires measuring network unit latency and energy costs per hop and configuring buffer sizes across the NoC.

1) Network hop costs: To measure network hop costs, one neuron in one or more cores is connected to another neuron in a core N hops away in the x or y direction (Fig. 7(a)), where pre-synaptic neurons spike every time-step and destination neurons only receive spikes. By measuring the energy and latency for N between 0 and the maximum hops, we can estimate the minimum cost per network hop for each spike.

2) Buffer sizing: To calibrate the average router buffer size or queue depth, we measure the maximum messages buffered between two tiles for varying hop counts (Fig. 7(b)). To record the maximum buffered messages, we use SNNs where the number of spiking neurons and hence spike packets being created can be configured between 0 to the maximum supported neurons. We map all the spike generating neurons to a core on the source tile, and these neurons send spikes over the network to a core in another tile. Sending neurons are connected to multiple receiving neurons in the destination cores so that the message processing delay in the receiver is much larger than the neuron processing and network delays i.e., the rate at which messages are generated, ensuring that messages will fill up buffers.

The source core on the source tile is configured to first send N spike packets to a destination core on a destination tile that is H hops away from the source tile, where every router hop has its own message buffer of size B and the total buffer capacity between the tiles is $H \times B$. Then, N messages are sent locally to another core on the source tile. By recording when these messages finish processing i.e., the end of the time-step, we can determine if and for what N buffer capacity is exceeded. Specifically, if $N \leq H \times B$, the second set of messages can be sent and processed within the source tile without ever being blocked. By contrast, if $N > H \times B$, messages injected into the network will fill up the buffers between the source and destination tiles, and sending of messages within the source tile will be delayed,

TABLE III
NEUROMORPHIC BENCHMARKS

Benchmark	Net. Sizes (Min.–Max. Neurons)	Cores Used (Min.–Max.)	Net. Type	Platforms Supported	Time-steps per run
Bio-inspired SNN [13]	98–1682	1–4	Connected layers	Loihi	10^5
Latin Square Solver [12]	512–3375	1–4	Winner-takes-all	Loihi	10^4
Gesture Categorization [11]	18678	49	2D-convolutional (4) & connected (1) layers	Loihi	128
Randomly Generated	64–262144	1–1024	Random sparse	Loihi, TrueNorth	10^5 (Loihi), 10 (TrueNorth)

extending the end of the time-step. During calibration, we find the minimum value of N at which the time-step delay starts increasing due to additional blocking.

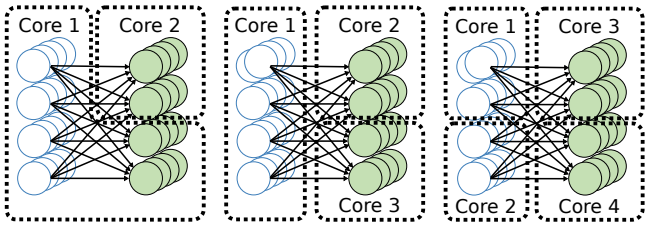
VII. Experiments and Results

We implemented SANA-FE in C and open-sourced all code at [38]. We applied SANA-FE to model Intel's Loihi and IBM's TrueNorth neuromorphic platforms. We validated functional accuracy by comparing spike traces from SANA-FE and Loihi, using Intel's Nahuku platform, which was accessed through Intel's Neuromorphic Research Cloud [12]. We also calibrated SANA-FE's Loihi model against the Nahuku platform using the methodology described in Section VI. For TrueNorth, since hardware was not available, we compared spike traces from NeMo [10], an existing and previously validated TrueNorth simulator. We demonstrated SANA-FE executing both real-world applications and randomized SNNs, which we will now describe in more detail. We will then discuss results for energy and latency prediction, simulator speed and design-space exploration.

A. Experimental Setup

To first calibrate the simulator, we executed the four micro-benchmark setups on Loihi for 10^5 time-steps and measure energy usage and total latency. These micro-benchmarks were executed in parallel across multiple cores and results averaged to improve measurement noise. The neuron scheduling benchmark was executed, with $N = 256$, $M = 768$, and $R = 128$. We then used both real-world neuromorphic applications and randomly generated SNNs, summarized in Table III. We mapped and simulated three real-world neuromorphic benchmarks for Loihi with different network sizes, network topologies and spiking behavior. These included a simplified network inspired by the Dragonfly prey interception neural network [13], a constraint satisfaction problem (CSP) solver mapped to the Latin squares problem [12], and an application that classifies hand-gestures captured from a dynamic vision sensor (DVS) based camera [11].

The bio-inspired benchmark consists of two connected layers of neurons scaled to different numbers of neurons (N) using the equations described in [13]. In this benchmark, the first layer of neurons spikes every time-step and sends messages to a second layer of receiving neurons. We generated networks with N between 98 and 1682, and measured energy and latency over 10^5 time-steps. We tested three mappings of neurons to cores to exercise different core and NoC behavior (Fig. 8). In the first mapping shown in Fig. 8(a), we used Intel's NxSDK framework and compiler for its Loihi platform to generate a default assignment that sequentially maps neurons from both layers to fill the first core, and spills neurons onto a second core once the first core is full. In the second configuration (Fig. 8(b)), we mapped all spiking neurons to the first core and receiving cores to the second and third cores. In the third and final configuration (Fig. 8(c)), we mapped the SNN across four cores.



(a) Default mapping (b) Three core mapping (c) Four core mapping

Fig. 8. Three mappings of the bio-inspired benchmark from [13].

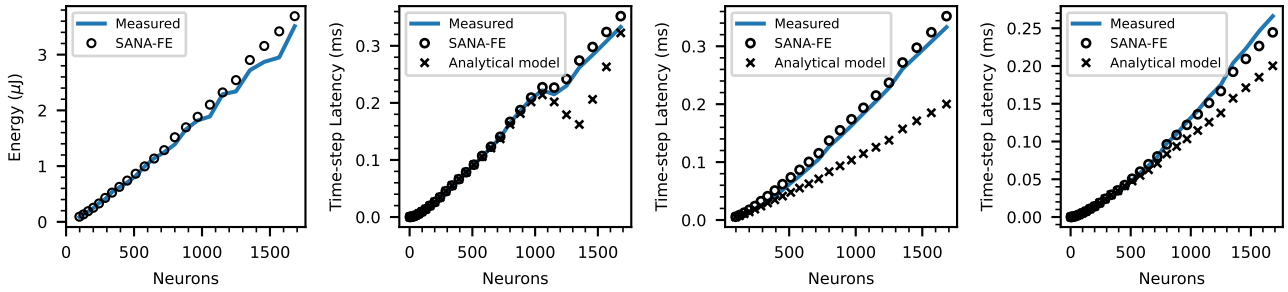
For the CSP solver, we generated a set of SNNs to solve Latin squares problems for N digits on an $N \times N$ grid, using the stochastic SNNs described in [12]. In this benchmark, the solver was run 8 times, ranging the problem size, N, from 8 to 15 digits. We executed the solver for 10^4 time-steps using NxSDK's default mapping, for each value of N.

For the DVS gesture classification, we used a previous implementation for Loihi [11] which uses Intel's neuromorphic deep-learning framework NxTF [39]. It uses a 5-layer convolutional SNN with a final dense layer, which is mapped to 49 cores by the NxTF compiler. We ran 100 test-case inferences for 5 different gestures, where each inference took 100 ms of DVS camera data and was processed for 128 time-steps.

In addition to real-world benchmarks available for the Loihi platform, we also generated randomized SNNs for both Loihi and TrueNorth using a range of network sizes, connectivity and spiking probabilities. Randomized networks were generated using a fixed seed, varying the number of cores, neurons per core, probability of each neuron spiking, packets per neuron and spikes per packet. For every SNN we used a fixed spiking pattern, where every neuron designated as spiking fired every time-step. For Loihi, the number of cores were ranged from 1 to 128, with 64 or 128 neurons instantiated per core and the other parameters varied. The networks were executed for 10^5 time-steps. For TrueNorth, we replicated the randomized SNNs in [10], with up to 1024 cores with 256 neurons per core and with 80% of spikes sent intra-core and 20% of spikes inter-core. SNNs were executed for 10 time-steps.

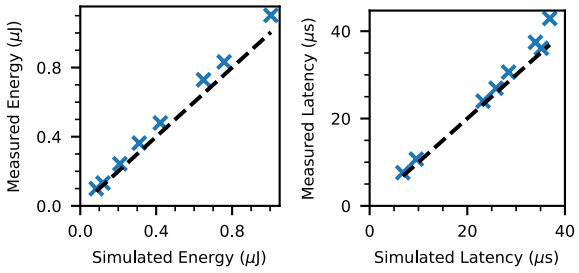
B. Energy and Latency Estimation

To demonstrate the accuracy of energy and latency predictions in SANA-FE, we executed the three real-world applica-



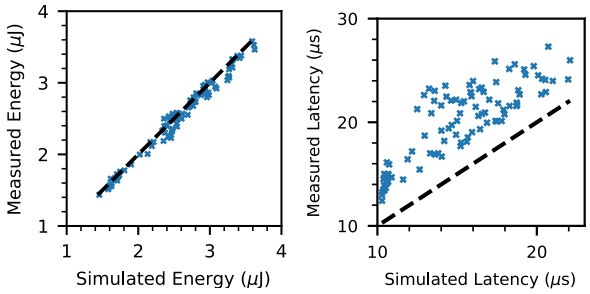
(a) Energy (b) Latency for default mapping (c) Latency for three-core mapping (d) Latency for four-core mapping

Fig. 9. Predicted average energy and latency per time-step for different sized SNNs from a bio-inspired benchmark.



(a) Energy (b) Latency

Fig. 10. Comparison of estimated average energy and latency per time-step against Loihi measurements for a Latin square solver executing different sized problem sets. Points represent values predicted by SANA-FE for different executed runs, and the dashed line represents a perfectly accurate prediction.



(a) Energy (b) Latency

Fig. 11. Comparison of estimated average energy and latency per time-step against Loihi measurements for different gesture categorizations.

tions on SANA-FE's calibrated Loihi model, and compared predictions against measurements taken on Loihi. All energy and latency measurements were taken using built-on probes on Intel's Nahuku platform. Latency measurements were recorded and compared at a time-step granularity. Energy measurements were taken using coarse-grained power probes and were averaged over the entire application execution to obtain per-time-step estimates.

Fig. 9 shows the estimated average energy and latency per time-step for different instances of the bio-inspired benchmark, showing that SANA-FE accurately captures both trends for different SNN sizes and mappings. For this application, the energy consumption does not depend on mappings as all cores are within the same network tile, and therefore the hardware mapping does not affect the total activity counts across the chip. Also, in this experiment, we only model dynamic energy, i.e. we do not model the change in static power due to potential power-gating of cores that are 'turned-off'. There is a dip in time-step latency for the default mapping around 1000 neurons, where the compiler switches from a single to two core mapping. Mapping across more cores results in more parallelization of neuron and message processing leading to faster execution times. However, partitioning across more cores also increases communication costs and network blocking delays. SANA-FE's scheduling algorithm is able to capture the effects of parallelization and communication delays, and can accurately predict the performance of different SNN mappings.

Fig. 9 includes a comparison of latency estimates using a simpler analytical timing model that is based on aggregated hardware activity counts [14]. As results show, such an analytical model is not able to account for cross-core and NoC timing effects in multi-core mappings.

Fig. 10 and Fig. 11 show the correlation of the estimated and measured average energy and latency per time-step, over different executions of the Latin square solver and DVS gesture categorization applications, respectively. The total energy and latency are generally predicted accurately for both applications. SANA-FE consistently underestimates average latency for the DVS gesture application.

Fig. 12 shows a trace of simulated vs. measured DVS latency over time. The simple analytical timing model from [14] does not replicate timing trends and overestimates latency in some cases. By contrast, SANA-FE's scheduling algorithm more reliably tracks latency trends, but is not able to accurately replicate all peaks. Inaccuracies are due to design details not captured by simulation. In particular, we would have to model the NoC in more detail than the semi-analytical model used by SANA-FE. To explore this, we implemented a custom, event-based network model that loads SANA-FE message traces and accurately tracks messages traversing the network, including arbitration, buffer queues and back-pressure at every router link. The dotted line in Fig. 12 shows the trace of a SANA-FE version using this NoC model. As can be seen, our semi-analytical model performs comparably to the detailed event-based model, despite only processing a single event per spike message. The event-based NoC model generates tens to hundreds of events per-message and is orders of magnitude slower with only a limited gain in accuracy. To improve accuracy further, an even more detailed, e.g., cycle-accurate NoC model would be needed, which would, however, likely be too slow for rapid exploration.

Finally, Fig. 13 compares predictions and measurements for the randomized networks. Fig. 13(a) shows that energy

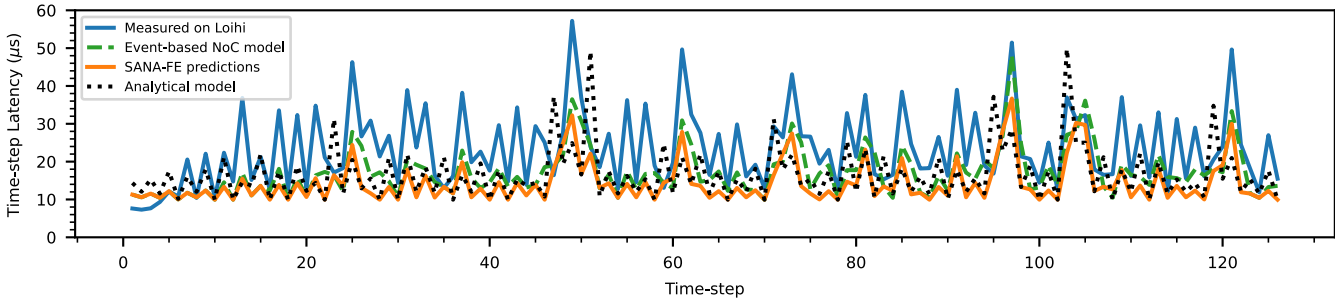
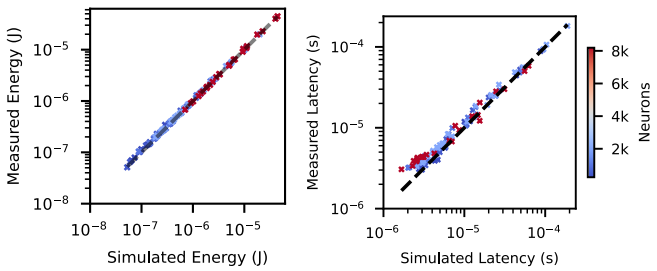


Fig. 12. Results from a time-series comparing simulated vs. measured time-step latency for a single gesture categorization from the DVS gesture data-set. The dotted line shows simulated results for a version of SANA-FE using an event-based NoC model.



(a) Energy (b) Latency

Fig. 13. Comparison of estimated average energy and latency per time-step against Loihi measurements different randomized SNNs. The color of each point indicates the number of neurons in that SNN.

TABLE IV
PREDICTION ERROR.

Benchmark	Energy		Latency	
	Avg. Abs.	Avg.	Avg. Abs.	Avg.
Bio-inspired SNN	5.0%	−4.1%	3.0%	−3.0%
Latin Square Solver	11.7%	10.7%	7.6%	7.5%
Gesture Categorization	2.7%	2.0%	24.3%	24.3%
Randomly Generated	3.9%	1.6%	15.4%	5.9%

is accurately predicted across a range of operating conditions, while Fig. 13(b) shows that latency trends are generally captured. Some of these random networks exercise scenarios that are unlikely in real-world applications e.g., fully connected neurons that are all spiking. Furthermore, each measurement represents only a single spiking pattern repeated over multiple time-steps – the effect of outliers would be averaged out in a real application that has many different spiking patterns during its execution.

SANA-FE's prediction error for different benchmarks is summarized in Table IV. Both the average absolute error and the average error were calculated across different executions of each benchmark. SANA-FE predicts energy and latency with an error margin of 11.7% and 24.3%, respectively.

C. Simulator Speed

To demonstrate simulator speed, we simulated all benchmarks on an Intel i7-13700 and recorded run-times (Table V). We calculated the throughput, i.e., the number of simulated time-steps per second, and show the range of run-times, throughput results and real-time speeds for the smallest and largest SNN sizes. We also compared randomized benchmarks run on SANA-FE against NeMo, an existing TrueNorth simulator (Fig. 14). Due to its abstract, time-step based simulation model SANA-FE executes the random application orders-of-magnitude faster than a discrete event-based simulator such

TABLE V
SIMULATOR SPEED.

Benchmark	Run-time	Throughput	Real-time
Bio-inspired SNN	7–255 s	390–14k steps/s	129–131 ms/s
Latin Square Solver	1–161 s	64–72k steps/s	2–54 ms/s
Gesture Categorization	0.3–2.1 s	61–427 steps/s	2–5 ms/s
Randomly Generated	4–4278 s	23–23k steps/s	2–71 ms/s

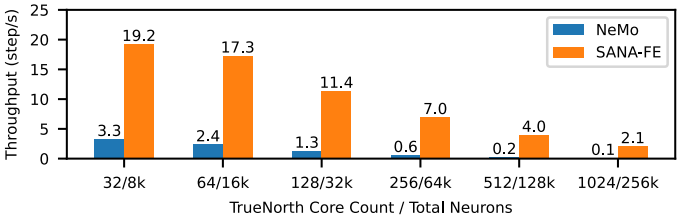


Fig. 14. Simulator speed of SANA-FE compared to the TrueNorth simulator NeMo executing randomized SNNs.

as NeMo for all network sizes. For 1024 out of 4096 cores SANA-FE simulated the random application over 20× faster than NeMo.

D. Design-Space Exploration

To demonstrate SANA-FE's capability for rapid, early design-space exploration, we performed a design-space sweep to predict the effect of design choices, and optimize architectures for gesture categorization and CSP solver applications. We generated a set of designs with different numbers of cores, scaling per-core resources, i.e., the number of neurons per core, synaptic memory and axon memory, such that the total resources in the design remained constant. For example, if the core count was doubled, the synaptic memory, neurons supported and axons per core were halved. We assumed that per-operation energy and latency costs remained the same, using previously calibrated values. For the gesture categorization application, we mapped the SNN from Section VII-B by modifying the SNN mapping code in Intel's NxTF framework. For the Latin square SNNs, we adopted a greedy algorithm that maps neurons to cores until they are full. We executed the mapped SNNs on their corresponding design for 128 time-steps for gesture categorization and 3000 time-steps for the Latin square solver (for a problem size of $N = 15$), measuring the dynamic energy and run-time. The design-space sweep took 29 s for gesture categorization and 473 s for the Latin square solver on an Intel i7-13700.

Fig. 15 and Fig. 16 show performance and energy usage for gesture categorization and the Latin square solver respectively,

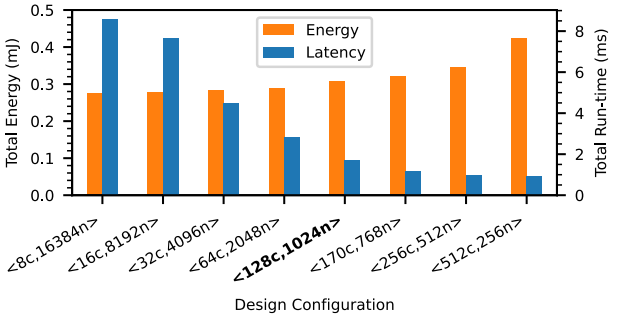


Fig. 15. Energy and latency executing the DVS gesture benchmark on different Loihi-based architectures. Each architecture is labeled by a tuple specifying (c) the number of cores and (n) the maximum neurons per core.

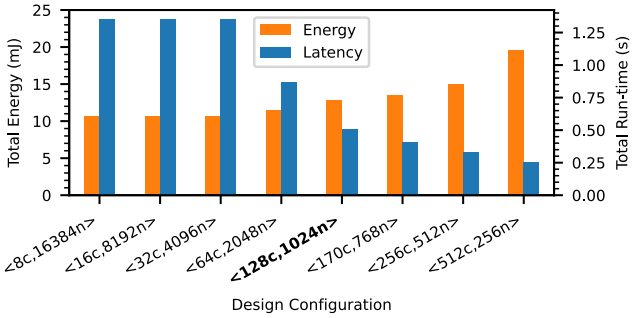


Fig. 16. Energy and latency executing the Latin squares benchmark on different Loihi-based architectures with labels ⟨cores, neurons per core⟩.

with designs labeled ⟨cores, neurons per core⟩. Loihi's configuration of 128 cores with 1024 neurons each is highlighted in bold. Results show that latency decreases while dynamic energy usage increases with higher core counts, as adding cores requires more tiles and incurs higher communication costs. Latency trends are different for the two applications, as the network connectivity is different for convolutional SNN used by gesture categorization and the sparse SNN for constraint solvers. Fig. 15 shows that DVS run-time drops as cores are increased, before plateauing around 170 cores. While Loihi is close to the optimal run-time for gesture categorization, the design with 170 cores executes the gesture benchmark 21% faster than Loihi but with only a 2% increase in dynamic energy. For the Latin square solver (Fig. 16), designs with cores supporting fewer than 4096 neurons are forced to map their SNN across multiple cores, leading to a significant reduction in run-time. Run-time plateaus for designs with core counts between 64 and 170, but is reduced further in designs with more than 170 cores. For designs with more than 170 cores, run-time could be traded off against increased energy usage depending on the needs of the user. Overall, design-space exploration results show that the Loihi base architecture provides a good balance, but there is room for application-specific optimization of latency-energy trade-offs. SANA-FE allows system architects to perform such trade-off analyses and explorations in a very short amount of time at early design stages.

VIII. Summary, Conclusion and Future Work

In this paper we presented SANA-FE, a novel configurable simulator of advanced neuromorphic architectures for fast exploration. SANA-FE uses an architecture description file and a mapped SNN model to simulate activity in different parts of the design. We described a methodology to calibrate SANA-FE against real hardware to accurately estimate real-world hardware energy and latency. On a Loihi platform, SANA-FE estimates energy within 12% for four benchmark applications. Finally, we have developed a timing model that can predict time-step latency within 25%. Using energy and latency estimates from SANA-FE, system architects can rapidly and effectively explore design-spaces and make energy-latency trade-offs for future SNN-based platforms.

In future work, we plan to further improve accuracy and simulation speed of SANA-FE, e.g., by incorporating more advanced NoC models and parallelizing the simulator to run on many-core and cluster platforms. Furthermore, we plan to extend SANA-FE to support analog and mixed-signal components, and to analyze the impact of using such design elements including emerging neuromorphic devices [40] in large-scale neuromorphic designs. Finally, we plan to integrate SANA-FE with other neuromorphic application development frameworks, such as Lava [9], Fugu [41], and SNNTorch [42]. SANA-FE was primarily designed for hardware design-space exploration, but within this context, it could also be useful for development of neuromorphic compilers and application optimization frameworks.

Acknowledgments

This article has been authored by an employee of National Technology & Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan https://www.energy.gov/downloads/doe-public-access-plan.

References

[1] C. Mead, *Analog VLSI and Neural Systems*. Reading, MA: Addison-Wesley, 1989.
[2] G. Indiveri *et al.*, "Neuromorphic silicon neuron circuits," *Front. Neurosci.*, vol. 5, p. 73, 2011.
[3] C. S. Thakur *et al.*, "Large-scale neuromorphic spiking array processors: A quest to mimic the brain," *Front. Neurosci.*, vol. 12, p. 891, 2018.
[4] J. D. Smith *et al.*, "Neuromorphic scaling advantages for energy-efficient random walk computations," *Nat. Electron.*, vol. 5, no. 2, pp. 102–112, 2022.
[5] J. Aimone *et al.*, "A review of non-cognitive applications for neuromorphic computing," *Neuromorphic Comp. Eng.*, 2022.
[6] C. D. Schuman, S. R. Kulkarni, M. Parsa, J. P. Mitchell, P. Date, and B. Kay, "Opportunities for neuromorphic computing algorithms and applications," *Nat. Comput. Sci.*, vol. 2, no. 1, pp. 10–19, 2022.
[7] A. Basu, L. Deng, C. Frenkel, and X. Zhang, "Spiking neural network integrated circuits: A review of trends and future directions," in *IEEE Custom Integr. Circuits Conf. (CICC)*, 2022.
[8] T. Bekolay *et al.*, "Nengo: a Python tool for building large-scale functional brain models," *Front. Neuroinform.*, vol. 7, p. 48, 2014.
[9] Intel's Lava Software Framework for Neuromorphic Computing. [Online]. Available: github.com/lava-nc/lava
[10] M. Plagge, C. D. Carothers, E. Gonsiorowski, and N. Mcglohon, "NeMo: A massively parallel discrete-event simulation model for neuromorphic architectures," *ACM Trans. Model. Comput. Simul. (TOMACS)*, vol. 28, no. 4, sep 2018.
[11] R. Massa, A. Marchisio, M. Martina, and M. Shafique, "An efficient spiking neural network for recognizing gestures with a DVS camera on the Loihi neuromorphic processor," in *Int. Joint Conf. Neural Netw. (IJCNN)*, 2020.

[12] M. Davies *et al.*, "Advancing neuromorphic computing with Loihi: A survey of results and outlook," *Proc. IEEE*, vol. 109, no. 5, pp. 911–934, 2021.
[13] L. Parker, F. Chance, and S. Cardwell, "Benchmarking a bio-inspired SNN on a neuromorphic system," in *Neuro-Inspired Comp. Elements Conf.*, 2022.
[14] J. Boyle, M. Plagge, S. G. Cardwell, F. S. Chance, and A. Gerstlauer, "Performance and energy simulation of spiking neuromorphic architectures for fast exploration," in *Int. Conf. Neuromorphic Syst. (ICONS)*, 2023.
[15] M.-O. Gewaltig and M. Diesmann, "NEST (NEural Simulation Tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007.
[16] L. Niedermeier *et al.*, "CARLsim 6: An open source library for large-scale, biologically detailed spiking neural network simulation," in *Int. Joint Conf. Neural Netw. (IJCNN)*, 2022, pp. 1–10.
[17] M. Stimberg, R. Brette, and D. F. Goodman, "Brian 2, an intuitive and efficient neural simulator," *eLife*, vol. 8, p. e47314, Aug. 2019.
[18] P. Date, C. Gunaratne, S. R. Kulkarni, R. Patton, M. Coletti, and T. Potok, "SuperNeuro: A fast and scalable simulator for neuromorphic computing," in *Int. Conf. Neuromorphic Syst. (ICONS)*, 2023.
[19] C. Michaelis, A. B. Lehr, W. Oed, and C. Tetzlaff, "Brian2Loihi: An emulator for the neuromorphic chip Loihi using the spiking neural network simulator Brian," *Front. Neuroinform.*, vol. 16, pp. 1–13, 2022.
[20] R. Kleijnen, M. Robens, M. Schiek, and S. van Waasen, "A network simulator for the estimation of bandwidth load and latency created by heterogeneous spiking neural networks on neuromorphic computing communication networks," *J. Low Power Electron. Appl.*, vol. 12, no. 2, p. 23, 2022.
[21] R. Kleijnen, M. Robens, M. Schiek, and S. Van Waasen, "Verification of a neuromorphic computing network simulator using experimental traffic data," *Front. Neurosci.*, vol. 16, p. 958343, 2022.
[22] M. Robens, R. Kleijnen, M. Schiek, and S. van Waasen, "NoC simulation steered by NEST: McAERsim and a Noxim patch," *Front. Neurosci.*, vol. 18, p. 1371103, 2024.
[23] A. Balaji, P. Adiraju, H. J. Kashyap, A. Das, J. L. Krichmar, N. D. Dutt, and F. Catthoor, "PyCARL: A PyNN interface for hardware-software co-simulation of spiking neural network," *arXiv:2003.09696*, 2020.
[24] M. Plagge *et al.*, "ATHENA: Enabling codesign for next-generation AI/ML architectures," in *IEEE Int. Conf. Rebooting Comput. (ICRC)*, 2022.
[25] A. F. Rodrigues *et al.*, "The structural simulation toolkit," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 37–42, 2011.
[26] J. Yik *et al.*, "NeuroBench: A framework for benchmarking neuromorphic computing algorithms and syst." *arXiv:2304.04640*, 2023.
[27] W. G. Gomez, A. Pignata, R. Pignari, V. Fra, E. Macii, and G. Urgese, "First steps towards micro-benchmarking the Lava-Loihi neuromorphic ecosystem," in *IEEE 16th Int. Symp. Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2023, pp. 462–469.
[28] M. Davies *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.
[29] M. V. DeBole *et al.*, "TrueNorth: Accelerating from zero to 64 million neurons in 10 years," *Computer*, vol. 52, no. 5, pp. 20–29, 2019.
[30] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker project," *Proc. IEEE*, vol. 102, no. 5, pp. 652–665, 2014.
[31] C. Mayr, S. Hoeppner, and S. Furber, "SpiNNaker 2: A 10 million core processor system for brain simulation and machine learning," *arXiv:1911.02385*, 2019.
[32] J. Pei *et al.*, "Towards artificial general intelligence with hybrid tianjic chip architecture," *Nature*, vol. 572, no. 7767, pp. 106–111, 2019.
[33] V. P. Nambiar *et al.*, "0.5V 4.8 pJ/SOP 0.93μW leakage/core neuromorphic processor with asynchronous NoC and reconfigurable LIF neuron," in *IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, 2020.
[34] B. V. Benjamin *et al.*, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proc. IEEE*, vol. 102, no. 5, pp. 699–716, 2014.
[35] C. Pehle *et al.*, "The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity," *Front. Neurosci.*, vol. 16, p. 795876, 2022.
[36] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *arXiv:1705.06963*, 2017.
[37] Z. Qian, P. Bogdan, C.-Y. Tsui, and R. Marculescu, "Performance evaluation of NoC-based multicore systems: From traffic analysis to NoC latency modeling," *ACM Trans. Des. Automat. Electron. Syst. (TODAES)*, vol. 21, no. 3, pp. 1–38, 2016.
[38] SANA-FE. [Online]. Available: github.com/SLAM-Lab/SANA-FE
[39] B. Rueckauer, C. Bybee, R. Goettsche, Y. Singh, J. Mishra, and A. Wild, "NxTF: An API and compiler for deep spiking neural networks on Intel Loihi," *ACM J. Emerg. Technol. Comput. Syst. (JETC)*, vol. 18, no. 3, pp. 1–22, 2022.
[40] J. Tang *et al.*, "Bridging biological and artificial neural networks with emerging neuromorphic devices: fundamentals, progress, and challenges," *Adv. Mater.*, vol. 31, no. 49, p. 1902761, 2019.
[41] J. B. Aimone, W. Severa, and C. M. Vineyard, "Composing neural algorithms with Fugu," in *Proc. Int. Conf. Neuromorphic Syst. (ICONS)*, 2019, pp. 1–8.
[42] J. K. Eshraghian *et al.*, "Training spiking neural networks using lessons from deep learning," *Proc. IEEE*, 2023.

James A. Boyle is a Ph.D. student in the Chandra Department of Electrical and Computer Engineering at The University of Texas at Austin, USA. He received his Masters of Engineering (M.Eng.) in electronic engineering with computer systems from the University of Southampton, UK in 2016.

Mr. Boyle has five years of industry experience, working with the CPU pre-silicon validation team at Arm, Cambridge, UK. His research interests include computer architecture and neuromorphic computing.

Mark Plagge is a postdoctoral researcher in the Center for Computing Research at Sandia National Laboratories. He received his PhD in computer science from Rensselaer Polytechnic Institute, where he focused on parallel discrete event simulations. Currently, Mark's research involves the application of novel neural networks, optimizing machine learning on advanced hardware, and exploring spiking neural networks. His work aims to push the boundaries of AI and optimization techniques on cutting-edge computational platforms.

Suma George Cardwell is a Principal Member of Technical Staff in the Center for Computing Research at Sandia National Laboratories. She completed her PhD and MS in Electrical and Computer Engineering at Georgia Tech, Atlanta in 2015 and 2011 respectively. Her current research focuses on neuromorphic computing, brain-inspired algorithms, event-based processing, co-design of machine learning hardware and algorithms, AI-guided microelectronics design, and applications of heterogeneous systems from HPC to the edge.

Frances S. Chance is a Principal Member of the Technical Staff in the Department of Cognitive and Emerging Computing of the Center for Computing Research at Sandia National Laboratories. Her research applies knowledge of biological nervous systems and neural circuit operations to develop and constrain novel neural-informed algorithms and brain-based technologies. She received her PhD and MS in Computational Neuroscience from Brandeis University.

Andreas Gerstlauer (SM'11) is a Cullen Trust for Higher Education Endowed Professor and Associate Chair in the Electrical and Computer Department at The University of Texas at Austin. He received the Ph.D. degree in Information and Computer Science from the University of California at Irvine (UCI) in 2004. Prior to joining UT Austin in 2008, he was an Assistant Researcher in the Center for Embedded Computer Systems (CECS) at UCI. His research interests cover system-level design and embedded systems. His work was recognized with several best paper awards, and he serves or has served as Editor for ACM TECS and TODAES journals as well as General or Program Chair for major international conferences such as ESWEEK.