



NeuroEngine: A Hardware-Based Event-Driven Simulation System for Advanced Brain-Inspired Computing

Hunjun Lee*

Department of Electrical and Computer Engineering
Seoul National University
Republic of Korea
hunjunlee7515@snu.ac.kr

Yujin Chung

Department of Electrical and Computer Engineering
Seoul National University
Republic of Korea
yujin.chung@snu.ac.kr

Chanmyeong Kim*

Department of Electrical and Computer Engineering
Seoul National University
Republic of Korea
chanmyeong@snu.ac.kr

Jangwoo Kim[†]

Department of Electrical and Computer Engineering
Seoul National University
Republic of Korea
jangwoo@snu.ac.kr

ABSTRACT

Brain-inspired computing aims to understand the cognitive mechanisms of a brain and apply them to advance various areas in computer science. Deep learning is an example to greatly improve the field of pattern recognition and classification by utilizing an artificial neural network (ANN). To exploit advanced mechanisms of a brain and thus make more great advances, researchers need a methodology that can simulate neural networks with higher computational capabilities such as advanced spiking neural networks (SNNs) with two-stage neurons and synaptic delays. However, existing SNN simulation methodologies are too slow and energy-inefficient due to their software-based simulation or hardware-based but time-driven execution mechanisms.

In this paper, we present NeuroEngine, a fast and energy-efficient hardware-based system to efficiently simulate advanced SNNs. The key idea is to design an accelerator to enable event-driven simulations of the SNNs at a minimum cost. NeuroEngine achieves high speed and energy efficiency by carefully architecting its datapath and memory units to take the best advantage of the event-driven mechanism while satisfying all the important requirements to simulate our target SNNs. For high performance and energy efficiency, NeuroEngine applies a simpler datapath, multi-queue scheduler, and lazy update to minimize its neuron computation and event scheduling overhead. Then, we build an end-to-end simulation system by implementing a programming interface and a compilation toolchain for NeuroEngine hardware. Our evaluations show that NeuroEngine greatly improves the harmonic mean performance

and energy efficiency by 4.30× and 2.60×, respectively, over the state-of-the-art time-driven simulator.

CCS CONCEPTS

• **Hardware** → **Neural systems**; • **Computer systems organization**;

KEYWORDS

brain-inspired computing, neuromorphic accelerators, event-driven simulation

ACM Reference Format:

Hunjun Lee, Chanmyeong Kim, Yujin Chung, and Jangwoo Kim. 2021. NeuroEngine: A Hardware-Based Event-Driven Simulation System for Advanced Brain-Inspired Computing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, April 19–23, 2021, Virtual, USA. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3445814.3446738>

1 INTRODUCTION

Spiking neural networks (SNNs) incorporate the concept of time to the internal dynamics of neurons and the communication between them [31]. In SNNs, a neuron keeps an internal state called voltage which changes with both the incoming spikes and time. If the voltage reaches a certain condition, the neuron transfers information to the connected neurons by firing a spike. The spike is transferred to destination neurons after a dedicated delay for each connection (i.e., synaptic delay). The internal dynamics and the spike-based data transfer closely mimic the mechanisms of the brain and are assumed to play a critical role in modeling the brain's cognitive functions [33, 57].

Previous studies simplify the SNNs to enable brain-inspired computing at low cost [16, 51]. They adopt a one-stage neuron to model the internal dynamics in a highly abstract form. Also, they transfer spikes without considering the synaptic delays. Using the simplified model, they successfully use SNNs for several tasks (e.g., object recognition [9]); however, their applicability is limited due to their highly simplified working model.

Other studies adopt the complex neuronal dynamics with synaptic delays to enable *advanced brain-inspired computing*. They rely on

*Both authors contributed equally to this work.

[†]Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '21, April 19–23, 2021, Virtual, USA

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8317-2/21/04...\$15.00

<https://doi.org/10.1145/3445814.3446738>

a two-stage neuron with relatively complex dynamics to model the internal dynamics and consider synaptic delays in the spike transfer. The complex models allow SNNs to process information with higher temporal resolution and exhibit richer dynamics [27, 56]. Therefore, recent studies apply them for various workloads including: image classification using precise spike time [12, 16, 50], feature extraction [62], satisfaction problem [18], and path planning [43].

Despite the potential benefits of supporting advanced SNNs, existing works fail to efficiently support them due to their software-based frameworks or hardware-based but time-driven execution mechanisms. To support complex dynamics, existing frameworks rely on general purpose processors such as CPUs [22, 38, 39, 42, 61] and GPUs [5, 46, 65]. Unfortunately, their underlying processors are slow and inefficient compared to specialized datapaths [30]. Other works use FPGAs [11, 60, 63] or ASICs [14, 19, 30, 56] for high speed simulations. Such works achieve higher performance over the software-based counterparts, but they still suffer from the intense computational overhead to update the internal state of the neurons. The reason is that they adopt a time-driven mechanism where the simulator updates all neurons at every time quantum of the simulation called timestep (e.g., 1 ms).

To mitigate the overhead, there are some efforts to adopt event-driven mechanisms to simulate SNNs [6, 32, 37, 48, 49]. Instead of updating all neurons for each timestep, they compute when a neuron receives or fires a spike. For each computation, they calculate the state change over several timesteps at once. By aggregating the updates, they reduce the number of computations compared to the time-driven counterparts. Considering the sparse nature of SNNs, event-driven mechanisms are potentially more efficient. However, designing an event-driven simulator for SNNs with two-stage neurons and synaptic delays encounters the following challenges:

- **Complex Computations:** To aggregate the voltage change, the simulator requires a datapath to calculate the closed-form solution of the neuron model. The two-stage neurons complicate the closed-form solution, and the simulator suffers from the high computational overhead accordingly. Unlike one-stage neurons, two-stage neurons do not fire instantly after receiving a spike; therefore, it requires a complex computation to predict when a neuron fires a spike.
- **Complex Scheduling:** The event-driven simulator schedules events to indicate that a neuron receives or fires a spike, and performs computation for each event. It schedules a new event after 1) a spike transfer indicating the arrival of a spike or after 2) a computation that predicts the firing time. The prediction of the firing time in two-stage neurons and the spike transfer with synaptic delays complicate the scheduling and incur redundant computations when not handled properly.
- **Resource Contention:** The simulator schedules a new computation after each computation and spike transfer; therefore, it suffers from the resource contention in the scheduler when parallelizing the two operations.

In this paper, we present *NeuroEngine*, a hardware-based event-driven simulator for fast and efficient simulations for advanced brain-inspired computing. To the best of our knowledge, NeuroEngine is the first work to *faithfully implement the event-driven simulation for advanced SNNs* in hardware. We propose novel schemes

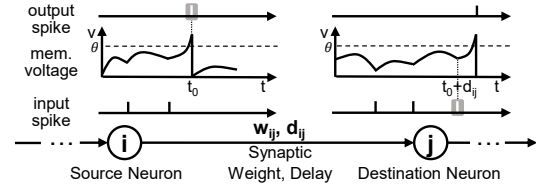


Figure 1: Example working model of an SNN.

ranging from the specialized datapath design to memory units to maximize the benefits of the event-driven simulations:

- **Optimized Datapath:** We propose a datapath to efficiently support the closed-form solution by splitting a complex computation into multiple sub-computations and pipelining them. Then, we approximate the existing prediction model to further reduce the hardware cost without affecting the functionality.
- **Composite Queue:** We propose a composite queue structure by combining two different queues with auxiliary data structures. The composite queue detects the computations that become redundant while scheduling and automatically removes them from the queue.
- **Lazy Update:** NeuroEngine successfully parallelizes the computation and the spike transfer by using the lazy update method. Lazy update identifies the computations that can bypass the event scheduling and parallelizes them with the spike transfers.

To build and evaluate an end-to-end system, we integrate our architecture with 2D mesh-based NoC as in the state-of-the-art brain simulation systems [1, 14]. Then, we add a programming interface and a compilation toolchain to deploy SNNs on NeuroEngine. We evaluate the performance of NeuroEngine by writing Verilog code and synthesizing it at Register-Transfer-Level (RTL) using Samsung 65 nm standard cell library. Our synthesis result indicates that NeuroEngine incurs only 13.8% additional area over the state-of-the-art time-driven baseline. To evaluate the overall performance and energy efficiency, we use six benchmarks collected from the existing publications. The result shows that NeuroEngine achieves 4.30× harmonic mean speedup and 2.60× harmonic mean energy efficiency over the time-driven baseline.

2 BACKGROUND & MOTIVATION

2.1 Spiking Neural Networks

SNNs model the brain using a directed graph whose nodes indicate neurons and edges represent connections between the neurons called synapses. Each neuron keeps an internal state called voltage, which changes with spikes and time. The weight and the delay dedicated to each synapse are called the synaptic weight and the synaptic delay. They determine how much and when a spike affects the destination. Figure 1 illustrates a working model of the SNNs where a source neuron (i) is connected to a destination neuron (j) through a synapse with weight (w_{ij}) and delay (d_{ij}). Each neuron updates the states with respect to the incoming spikes and time (i.e., neuron computation). The neuron i fires a spike at t_0 . Then, it propagates the spike (highlighted in the grey box) to the neuron j , and the synaptic weight is accumulated to the neuron j after the delay (at $t_0 + d_{ij}$) (i.e., weight accumulation). If the source neuron

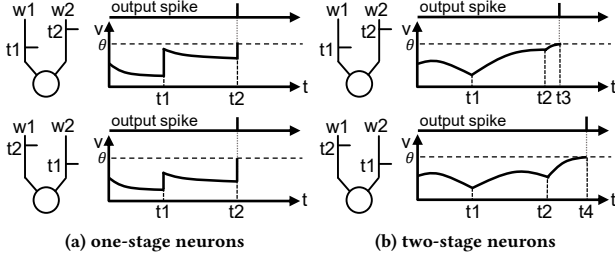


Figure 2: Behavior difference of the one-stage and two-stage neurons for different input patterns.

is an excitatory neuron, the spike increases the voltage while an inhibitory neuron decreases the voltage.

The spike transfer (i.e., weight accumulation) with synaptic delays plays a critical role in the network’s capability to process complex information and exhibit temporal dynamics. A previous work by Izhikevich identifies that the networks with synaptic delays allow much diverse firing patterns called polychronous group [27]. Moreover, Smith points out the importance of synaptic delays when processing the temporally-encoded information [57]. Therefore, the synaptic delay should be precisely modeled to enable an advanced form of computing using the SNNs.

2.2 Neuron Models

The neuron models which mimic the internal dynamics of the spiking neurons can be classified into two categories depending on how they respond to the input spikes: the one-stage and the two-stage neuron [56]. The one-stage neuron changes its membrane voltage with an infinite slope, and thus, fires a spike only when it receives a spike. On the other hand, the two-stage neuron gradually changes its voltage and takes time to fire a spike after receiving a spike. By reacting to the input spikes with a non-instant slope, the two-stage neurons can process more complex relationships between input spikes. Quantitatively, a previous study [56] identifies that a two-stage neuron processes information with 50% higher resolution than the one-stage neurons. Figure 2 illustrates an example of the one-stage neuron and the two-stage neuron receiving two different input spike patterns and shows how the two-stage neuron enables richer temporal dynamics. In the first case, a neuron receives a former spike at t_1 with larger weight w_1 and the latter spike at t_2 with smaller weight w_2 . The order of the two spikes is reversed in the second case. The one-stage neuron fires a spike only when it receives a spike, and the timing of the output spike is the same in both cases (t_2 in Figure 2a); whereas the two-stage neuron takes time to fire a spike, and the timing of the output spikes differ (t_3, t_4 in Figure 2b).

To that end, emerging studies focus on using two-stage neuron models for various tasks. For example, studies by Smith [57–59] emphasize the computational power of the two-stage neuron models and use them on spike-based image classification workloads. Other works apply a spike-based backpropagation on two-stage neuron models to achieve the state-of-the-art accuracy on classification workloads [36, 54]. Recent studies are exploring the potential

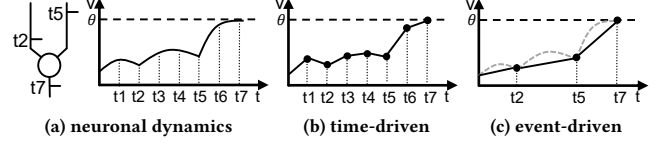


Figure 3: Neuron computations in time-driven simulation and event-driven simulation.

of two-stage SNNs on diverse use cases. Ponulak et al. reproduce the brain’s navigating function using the two-stage neuron [43]. As another example, two-stage neurons are adopted for feature extraction (i.e., LASSO) [62] or solving satisfaction problem [18].

Among the two-stage neurons, we mainly focus on biexponential model for its high applicability in recent brain-inspired computing tasks. It is defined as:

$$\begin{aligned} \frac{dg_{type}}{dt} &= -\frac{g_{type}}{\tau_{type}} + \sum_i \delta(t - (t_i + d_{ij})) \cdot w_{ij} \quad (type = exc, inh) \\ \frac{dv}{dt} &= -\frac{v}{\tau_v} + g_{exc} - g_{inh} \end{aligned} \quad (1)$$

if $v > \theta$, then fire a spike and $v = v_0$

where v is a membrane voltage which decays with the time constant τ_v and θ indicates the threshold voltage. The $\sum_i \delta(t - (t_i + d_{ij})) \cdot w_{ij}$ indicates the accumulated weight using the firing time of the source neuron (t_i), the delay (d_{ij}) and weight (w_{ij}) between the source and the destination neuron. Also, g_{exc} and g_{inh} refer to the current, which enables the gradual increase in the voltage by low-pass filtering the accumulated weights. The current decays with the time constants τ_{exc} and τ_{inh} . Biexponential model is both computationally powerful and efficient, making it widely used in various studies [21, 56]. Also, a recent neuromorphic benchmark study [13] emphasizes its high applicability to various workloads.

2.3 Time-Driven & Event-Driven Simulation

To incorporate the concept of time into its operation, SNN simulation divides the biological time into short discrete timesteps (e.g., 1 ms) and proceeds to the next timestep after evaluating one. On each timestep, the simulation performs two main operations: neuron computation and weight accumulation. The neuron computation updates the internal states (e.g., v) and checks if a neuron fires a spike. The weight accumulation iterates over the destination neurons for each generated spikes. Then, it accumulates the weight for the destination neurons using the synaptic weight and delay.

There are two representative approaches, time-driven and event-driven, to simulate SNNs. The time-driven mechanism performs neuron computation for all neurons every timestep (Figure 3b). Algorithm 1 illustrates an example of the time-driven simulation. The simulator iterates over all the neurons for each timestep and updates the states based on the previous states and the accumulated weight (Line 10–12). If the neuron fires a spike, the simulator stores the spike in the buffer. Concurrently, the simulator performs weight accumulation for each spike in the buffer (Line 13–14).

On the other hand, the event-driven mechanism calculates the change in the internal states over several timesteps at once (Figure 3c). Instead of updating all neurons every timestep, the simulator

Algorithm 1: Time-driven SNN simulation

```

1 # t_cur: the current simulation timestep
2 # neurons: a list of all neurons
3 # n.fanout: a list of outgoing edges (destination,weight,delay)
4 # n.states: internal states of neuron n
5 # n.acc[t]: accumulated weight for neuron n at timestep t
6 # spike_buf: a buffer for the generated spikes
7
8 Func simulate(neurons, t_max):
9   While (t_cur < t_max):
10     For (n in neurons): # neuron computation
11       fired = state_update(n.states, n.acc[t_cur]) # Eq.1
12       If (fired): spike_buf.append(n)
13     For (src in spike_buf): # weight accumulation
14       For ((dst,w,d) in src.fanout): dst.acc[t_cur + d] += w
15     t_cur += 1, spike_buf.reset()

```

Algorithm 2: Event-driven SNN simulation

```

1 # event_queue: the event queue (target_neuron, event_time)
2 # n.last: timestep of the last update for neuron n
3
4 Func simulate(neurons, t_max):
5   While (t_cur < t_max):
6     While ((n,t) = event_queue.top() and t == t_cur):
7       event_queue.pop() # neuron computation
8       fired = state_update(n.states, n.acc[t], n.last) # Eq.2
9       n.last = t
10      If (fired): spike_buf.append(n)
11      t_pred = predict(n.states) # spike prediction (Eq.3)
12      event_queue.schedule(n, t_pred)
13    For (src in spike_buf): # weight accumulation
14      For ((dst,w,d) in src.fanout):
15        dst.acc[t_cur + d] += w
16        event_queue.schedule(dst, t_cur + d)
17    t_cur += 1, spike_buf.reset()

```

performs neuron computations only when a neuron receives or fires a spike. The simulator uses the term *events* to indicate such occasions and adopts an event queue to handle the events. Algorithm 2 illustrates an example of the event-driven simulation. For each timestep, the simulator checks the event queue where the events are stored in chronological order. Then, it performs neuron computation for each event using an equation (i.e., Equation 2) and updates the last time the neuron was updated (Line 6–12). After that, the event-driven simulator calculates the time when a neuron fires a spike and schedule an event (i.e., *out-event*) at the calculated timestep (Line 10–11). Ideally, the simulator calculates the exact time when the neuron will fire a spike; however, existing works use approximation methods to conservatively predict the firing time due to the high complexity of the calculation. Note that the one-stage neurons do not schedule an out-event as they fire a spike only when it receives a spike. After processing all the events for the timestep, the simulator performs weight accumulation (Line 13–16). The simulator schedules an event for the destination neuron at the timestep where it receives the spike ($t_{\text{cur}} + d$) to ensure that it performs neuron computation for the spike (Line 16). We refer to such an event as an *in-event*.

The event-driven simulation exhibits higher efficiency than the time-driven simulation by updating the internal states only when

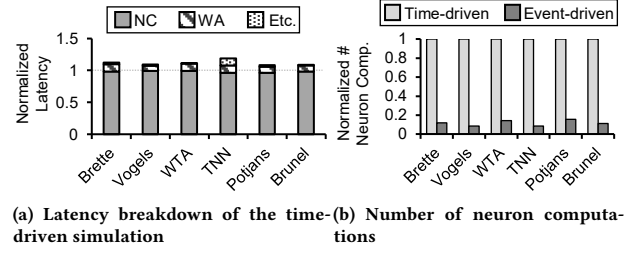


Figure 4: Potential of the event-driven simulation.

an event exists. To examine the potential of the event-driven simulation, we breakdown the simulation latency of the baseline time-driven simulator on six SNN benchmarks (Refer to Section 6.1 for the evaluation setup). The result shows that the time-driven simulator suffers from the excessive neuron computation overhead (Figure 4a). The NC and WA refer to the time taken in neuron computation and weight accumulation, respectively, and Etc indicates subordinate functions, including routing and synchronization overhead. The latency is normalized to the total latency of each benchmark. Note that the total latency is smaller than the combined value as the computations can be parallelized. The result indicates that NC is a critical performance bottleneck taking up to 99.3% of the total latency. Then, we profile the number of neuron computations required in the event-driven simulator (Figure 4b). Our results show that the event-driven simulator can greatly reduce the neuron computation overhead with up to 91.4% fewer computations than the time-driven counterpart.

3 CHALLENGES

The event-driven simulation is potentially highly efficient over the time-driven simulation by reducing the number of neuron computations; however, we identify three challenges in realizing an efficient event-driven simulation for advanced SNNs with two-stage neurons and synaptic delays.

3.1 Complex Computations

Unlike the time-driven method which updates the states over a fixed interval, the event-driven simulation updates the states over arbitrary time intervals. Thus, it requires a closed-form solution for the target model which is highly complex for two-stage neurons. The closed-form solution for biexponential model is defined as:

$$\begin{aligned}
 g_{type,t} &= g_{type,t'} \cdot e^{-\Delta t / \tau_{type}} + \overbrace{\sum_i w_{ij} \cdot s_i(t)}^{I_{type,t}} \\
 \tau_{vg_{type}} &= (\tau_v \cdot \tau_{type}) / (\tau_v - \tau_{type}) \\
 v_t &= v_{t'} \cdot e^{-\Delta t / \tau_v} + \sum_{type} \tau_{vg_{type}} \cdot g_{type,t'} \cdot (e^{-\Delta t / \tau_v} - e^{-\Delta t / \tau_{type}})
 \end{aligned} \tag{2}$$

where the $g_{type,t}$ (current) and v_t (voltage) illustrate the state variables of the neuron j at time t (Equation 1). $s_i(t)$ denotes the occurrence of a spike of a neuron i at time t . Δt is the time interval between the current time t and the last update time t' . As the

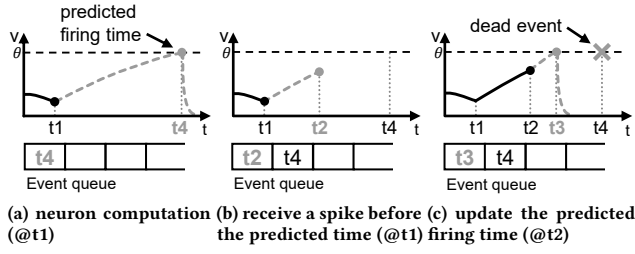


Figure 5: Occurrence of a dead event.

equation illustrates, the datapath needs to support complex computations with varying intervals, Δt . As SNNs are sparse in events, the Δt leads to high bit precision requirements for the computation.

Moreover, the event-driven simulations require complex operations to predict when a neuron fires a spike. Unlike the one-stage neuron which fires right after receiving a spike, the two-stage neuron takes time to fire a spike after receiving a spike (Figure 2). To calculate the firing time, some approaches rely on the LUTs where the precalculated results are stored [20, 35, 37, 48]; however, the LUT demands too large memory capacities to be implemented in hardware. To mitigate the complexity of solving the equation, some other works propose a linear approximation model to conservatively predict the firing time [15, 32]. The prediction for the biexponential model is:

$$A_v = v_t + \sum_{type} g_{type,t} \cdot \tau_{v_{type}}, \quad A_{type} = -g_{type,t} \cdot \tau_{v_{type}} \quad (3)$$

$$slope = \begin{cases} -A_v/\tau_v - A_{exc}/\tau_{exc} - A_{inh}/\tau_{inh}, & \text{if } A_v > 0 \\ -A_v/\tau_v - A_{exc}/\tau_{exc} - A_{inh}/\tau_v, & \text{if } A_v < 0 \end{cases}$$

$$t_{pred} = (\theta - v_t)/slope$$

where $slope$ and t_{pred} refer to an approximate slope and the resulting predicted time, respectively. The prediction model requires power- and area-consuming divide operations and incur high computational overhead.

3.2 Complex Scheduling

We identify two challenges in designing an efficient event scheduler for advanced SNNs. First, both the in-events and the out-events are scheduled at an arbitrary timestep as shown in Algorithm 2 (Line 12, 16). Therefore, the scheduler should sort the events at a low cost to process the events in sequential order. Second, the in-events and the out-events interact with each other and may become redundant. In this case, the scheduler should detect the redundant events and remove them from the event queue. For example, an out-event becomes redundant if a neuron receives a spike before the predicted firing time. The predicted firing time changes after receiving the spike, and the scheduled out-event becomes useless. Figure 5 illustrates how the out-event becomes a useless event which we refer to as a *dead event*. After the neuron computation at t_1 , the neuron predicts the next firing time as t_4 and schedules an out-event (Figure 5a). Right after scheduling the out-event, a new in-event which indicates that a neuron receives a spike at t_2 is scheduled (Figure 5b). Due to the event, the simulator performs

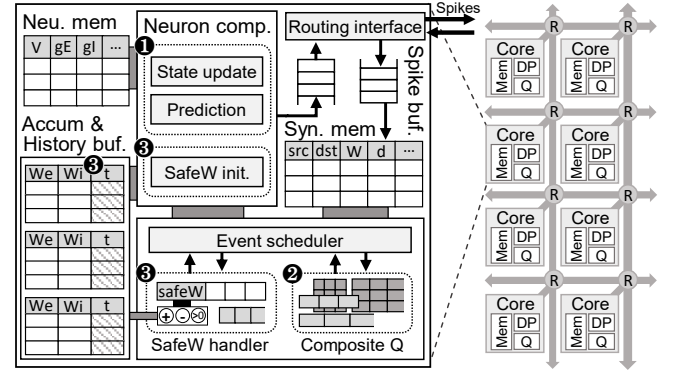


Figure 6: Overview of the NeuroEngine architecture.

neuron computation at t_2 , and predicts a new firing event, indicating that the neuron is predicted to fire a spike at t_3 (Figure 5c). As the firing time is updated to t_3 , the previously scheduled out-event at t_4 becomes useless, turning into a dead event.

3.3 Resource Contention

In the event-driven simulation, the simulator suffers from the resource contention in event scheduling when processing neuron computation and weight accumulation simultaneously. Using the event-driven mechanism to update the states, the number of neuron computations is highly reduced, and the weight accumulation also becomes a major source of performance bottleneck; therefore, it is highly demanding to parallelize neuron computation and weight accumulation. However, as described in Algorithm 2, the event-driven simulator schedules events for both neuron computation (Line 12) and weight accumulation (Line 16). To that end, the concurrent processing of the two operations is a challenging task with the resource contention in the event scheduler.

4 NEUROENGINE ARCHITECTURE

In this section, we present NeuroEngine, a fast and efficient hardware-based event-driven simulation system. We apply simple datapaths, composite queue scheduler, and lazy update to minimize the computation and event scheduling overhead. Figure 6 illustrates the overall architecture of NeuroEngine. First, our simple datapaths with reduced bit precision and approximate divider offer fast and efficient neuron computations (①). Second, our composite queue structure for scheduling the events prevents redundant event scheduling (②). Lastly, we propose a novel method called lazy update to bypass event scheduling and parallelize operations (③).

4.1 Optimized Datapath

4.1.1 Pipelined Datapath with Reduced Bit Precision. We design an optimal datapath for neuron computation by reducing the integer-part-bit precision for representing the Δt . The key idea is to shift and split the time interval into multiple sub-intervals and pipeline the computations. First, we identify that the Δt can be shifted as it is multiplied by the inverse of the time constants, $1/\tau$ (i.e., $1/\tau_v$, $1/\tau_{type}$) (Equation 2). By performing the right shift on the Δt and the left shift on the $1/\tau$, we can reduce the precision for representing

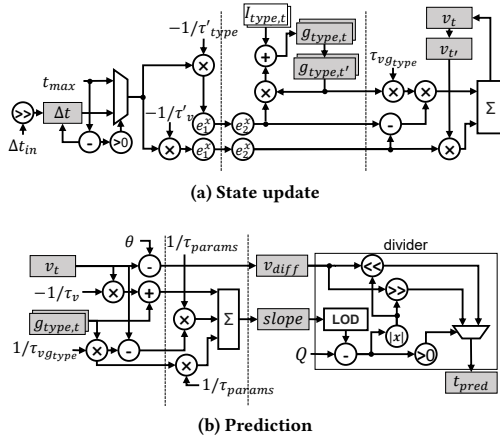


Figure 7: Optimized datapath for neuron computation.

the interval ($\Delta t \cdot 1/\tau = (2^{-n} \cdot \Delta t) \cdot (2^n \cdot 1/\tau)$). We further reduce the bits by splitting the Δt into smaller sub-intervals so that the Δt does not exceed the supporting bit precision. Although splitting increases the number of computations, the overhead is minimal (Section 6.3). After that, we fully pipeline the datapaths into three stages with forwarding paths to prevent stalls when consecutively updating the sub-intervals. Figure 7a illustrates our state update datapaths. The $g_{type,t}$ and v_t refers to the internal states in Equation 2. Also, the datapaths for the exponential function is divided into two parts (i.e., e_1^x and e_2^x) for efficient pipeline. The $1/\tau$'s are the inverse of the time constants after performing the left shift. The datapaths shift and split the time interval into sub-intervals, t_{max} for each event. Then, the computations for each sub-interval pass through the pipeline stages. If the two consecutive computations target the same neuron, the updated states at each stage are forwarded.

4.1.2 Efficient Prediction with an Approximate Divider. We design an approximate divider which shifts the dividend by $\lceil \log_2 \text{divisor} \rceil$ instead of performing a complex division. Such an abstraction achieves high efficiency by eliminating the iterations required in the conventional divider. The approximate divider avoids the loss of functionality as the shift-based division produces pessimistic results. The result may lead to additional false-positive predictions; however, the resulting overhead is minimal (Section 6.3). Then, we pipeline the datapaths to reduce the critical path latency. Figure 7b illustrates our prediction datapaths including the approximate divider. The Q denotes the fractional-part-bit precision, and τ_{params} is a constant determined by the sign of A_v . The LOD module returns the index of the first '1' in the input value. Our divider allows the fixed-point division to operate in a single clock cycle, improving latency as well as the power and area consumption of the prediction.

4.2 Composite Queue

We design a composite queue structure composed of heterogeneous data structures to detect and remove redundant events. First, we design an optimal queue structure for the in-event and the out-event, respectively, after analyzing their unique features. Then, we

identify three conditions where different events interact and become redundant. Next, we propose two auxiliary data structures to detect the interaction and remove an event if it becomes redundant. By combining the optimal queue with auxiliary data structures, we propose our composite queue-based event scheduler.

From our analysis, we identify the key features of the in-event and the out-event. We identify two different features of the in-event which indicates the arrival of the input spikes. First, multiple in-events for a single neuron coexist if a neuron receives spikes with different synaptic delays. Second, the range of the synaptic delay is fixed, and the in-events reside only within a fixed time window. In contrast, only one out-event is scheduled for a neuron, and an out-event can occur in the distant future without any time constraint. Also, it turns into a dead event and should be removed in some cases (Figure 5). Using the identified features, we design an optimal queue structure to schedule each event:

In-Event Queue: In-event queue consists of multiple FIFO queues to efficiently handle multiple coexisting events in a time window. Each FIFO queue handles in-events for each timestep within the maximum synaptic delay, d_M ; therefore, it keeps only d_M queues. It uses the queues in a circular manner and each queue holds in-events for $t_0, t_0 + d_M, t_0 + 2d_M, \dots$ as timestep goes on ($0 \leq t_0 < d_M$). Then, NeuroEngine sequentially iterates over the FIFO queues, naturally processing the events in the time order.

Out-Event Queue: We adopt a Structured Heap Queue (SHQ) [10] as the out-event queue for efficient deletion of events and scheduling of events over a long time range. The SHQ is a modified version of the pipelined heap queue (PHQ) [26] which stores only a single event for each neuron and provides an arbitrary deletion of a neuron. Such characteristics highly fit well with the identified features of the out-events. Note that popping the event and deletion of an arbitrary event induces a stall cycle.

There are three conditions where an event becomes redundant as a result of the interaction between the in-events and the out-events. First, even if a neuron receives two spikes at the same timestep, a single event-driven computation can process the aggregated information of the two spikes. Thus, the in-event for the second spike becomes a duplicate event and do not need to be scheduled. Second, an out-event becomes useless if an in-event is scheduled before the out-event (Figure 5). In this case, the scheduler should remove the out-event. Lastly, an out-event does not need to be scheduled if an in-event is already scheduled before the predicted time of the out-event. We propose two auxiliary data structures and mechanisms to handle the interactions efficiently:

In-Event Bitmap: The in-event queue is equipped with a bitmap that tracks the existence of the scheduled in-events. The entry indicates whether an in-event exists in each FIFO queue. The in-event bitmap provides an overview of the existence in the in-event and enables NeuroEngine to identify redundant events and prevent scheduling them. Before scheduling a new in-event, the scheduler uses the bitmap to check if an in-event is already scheduled at the timestep. Also, before scheduling a new out-event, the scheduler checks if an in-event is already scheduled at an earlier timestep than the predicted time.

Out-Event Time Vector: The out-event queue holds an additional data structure called an out-event time vector that stores the scheduled time of the out-events. Using the out-event time vector, the

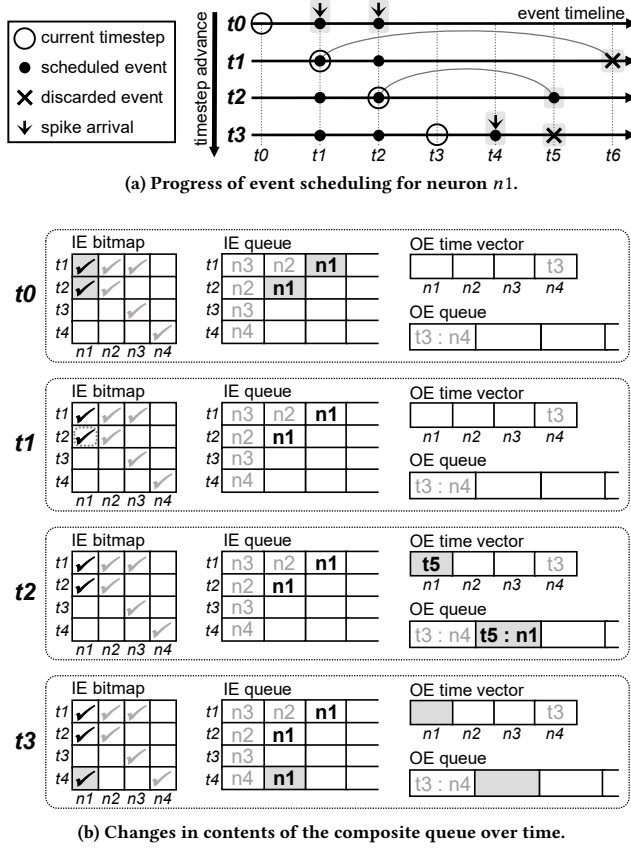


Figure 8: Redundant event handling in the composite queue.

scheduler can detect the dead event without accessing the out-event queue. Before NeuroEngine schedules a new in-event, the scheduler uses the out-event time vector to examine if an out-event is scheduled at an earlier timestep. Then, the scheduler removes the out-event from the out-event queue.

Figure 8 illustrates an example of how NeuroEngine handles redundant events using the composite queue. In this example, there are four neurons ($n1 \sim n4$) and we focus only on how the composite queue handles the events for $n1$. Figure 8a shows the progress of the event scheduling for $n1$ from timestep $t0$ to $t3$, while Figure 8b illustrates the contents of the composite queue at each timestep. At $t0$ (Figure 8-($t0$)), $n1$ receives two input spikes arriving at $t1$ and $t2$. NeuroEngine schedules the two new in-events by updating the in-event bitmap and inserting the in-events into the in-event queue (marked with the black text in the figure). Then, at $t1$ (Figure 8-($t1$)), NeuroEngine performs neuron computation for the scheduled in-event for $n1$ and predicts the firing time ($t6$). However, NeuroEngine does not schedule the new out-event at $t6$ because the in-event bitmap indicates that it is a dead event due to the in-event scheduled at $t2$. Next, at $t2$ (Figure 8-($t2$)), $n1$ performs neuron computation again and predicts the next firing time ($t5$). In this case, there is no in-event scheduled at an earlier timestep than the predicted time, thus NeuroEngine schedules the new out-event at $t5$ by updating the

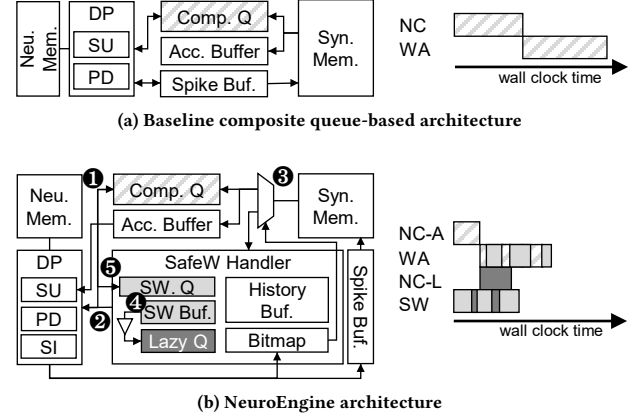


Figure 9: The working example of the event-driven simulator with and without the lazy update.

out-event time vector and out-event queue. Lastly, at $t3$ (Figure 8-($t3$)), an input spike arriving at $t4$ is scheduled as an in-event. At the same time, NeuroEngine detects that the out-event at $t5$ becomes a dead-event by checking the out-event time vector. Therefore, NeuroEngine deletes the corresponding entries in the out-event queue and out-event time vector.

4.3 Lazy Update

The event-driven simulator suffers from the low parallelization opportunity induced by the resource contention when scheduling the events. Figure 9a illustrates the resource contention in the baseline architecture with the composite queue. Two types of operations exist in the baseline: neuron computation (NC) and weight accumulation (WA). First, the simulator sequentially pops the events from the composite queue. The simulator performs NC to update the states (SU) and predict an out-event (PD). Then, the simulator accesses the out-event queue in the composite queue to store the out-event. If the neuron fires a spike, the simulator stores it in the spike buffer. After NC, the simulator performs WA by iterating over the spikes in the spike buffer, searching for the destination neurons using the synapse memory (Syn. Mem.), and accumulating the weights to the buffer (Acc. Buffer). At the same time, the simulator accesses the out-event queue to delete a dead event if necessary and schedule an in-event. When concurrently processing NC and WA, the out-event queue causes a resource contention. As the out-event queue is based on a priority queue, the event schedulings are serialized, resulting in low parallelism.

To maximize parallelism, we propose a novel method called *lazy update*. The idea is to detect the neurons that would not fire a spike after neuron computation and bypass event scheduling for the computations. Then, NeuroEngine defers the computations until the neuron is expected to fire a spike. When flushing the deferred computations, NeuroEngine parallelizes them with weight accumulation as they do not incur event scheduling; thereby, reducing the overall latency.

The key observation is that computations of a neuron can be deferred as long as the neuron does not fire a spike. We refer to such

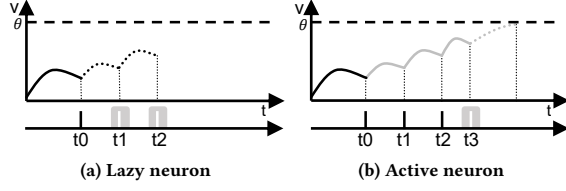


Figure 10: Example of the lazy update with a lazy neuron turning into an active neuron.

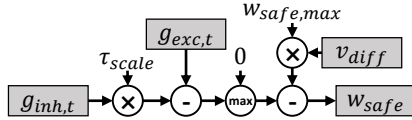


Figure 11: Datapath for safe weight initialization.

neurons as lazy neurons and use the term active neurons to indicate the neurons that are likely to fire a spike. Figure 10 illustrates an example of a lazy neuron turning into an active neuron and how the computations are deferred in NeuroEngine. In the example, the neuron is not expected to fire a spike even after receiving two events at t_1 and t_2 (Figure 10a). Thus, NeuroEngine identifies the neuron as a lazy neuron and defers the computations. On the other hand, the neuron is expected to fire a spike after an incoming spike at t_3 (Figure 10b); therefore, the neuron turns into an active neuron, and NeuroEngine no longer defers the computations. Then, NeuroEngine flushes the deferred computations to update the internal states before processing the in-event at t_3 . We refer to the whole process as lazy update.

We devise a new concept called *safe weight* to determine if a neuron is a lazy neuron with minimal computational overhead. The safe weight indicates the total amount of weight a neuron withstands before turning into an active neuron. NeuroEngine initializes the safe weight of a neuron when performing neuron computation. If the resulting safe weight is positive, the neuron turns into a lazy neuron, and NeuroEngine starts to defer the computations. When a lazy neuron receives a spike, NeuroEngine updates the safe weight of the neuron instead of scheduling neuron computation. If the updated safe weight becomes negative, the neuron becomes active, and NeuroEngine schedules the neuron.

We define simple functions to initialize and update the safe weight and design a datapath (Figure 11). First, we calculate the maximum voltage a neuron reaches after receiving a spike of weight w . Using the value, we derive how much weight a neuron should receive to fire a spike at the resting voltage and define it as a reference safe weight ($w_{safe,max}$):

$$w_{safe,max} = \theta / (\tau_v g_{exc} \cdot (\frac{\tau_v}{\tau_{exc}} - \frac{\tau_v g_{exc}}{\tau_v} - \frac{\tau_v}{\tau_{exc}} - \frac{\tau_v g_{exc}}{\tau_{exc}})) \quad (4)$$

Then, we scale the reference safe weight to take the initial voltage v_t into account. If the current (i.e., g_{exc} and g_{inh}) is not zero, we calculate the effective current (g_{eff}) by scaling g_{inh} and subtract it with g_{exc} . We subtract the effective current from the scaled safe

weight to initialize the value. The algorithm is defined as:

$$\begin{aligned} \tau_{scale} &= \min(\tau_{inh}/\tau_{exc}, 1) \\ g_{eff} &= \max(g_{exc,t} - g_{inh,t} \cdot \tau_{scale}, 0) \\ v_{diff} &= \theta - \max(v_t, 0) \\ w_{safe} &= w_{safe,max} \cdot v_{diff} - \max(g_t, 0) \end{aligned} \quad (5)$$

where τ_{scale} indicates the scaling factor for the inhibitory current and w_{safe} indicates the safe weight. To update the safe weight, we simply subtract the weight of the spike from the safe weight. If the spike is from an inhibitory neuron, we make a scaling factor using the decay constant of the currents and scale the weight before updating the safe weight:

$$\begin{aligned} w_{safe} &= w_{safe} - w_{spike,type} \text{ if } type = exc \\ w_{safe} &= w_{safe} + w_{spike,type} \cdot \tau_{scale} \text{ if } type = inh \end{aligned} \quad (6)$$

where $w_{spike,type}$ indicates the weight of the incoming spike. Note that we make a conservative calculation of the safe weight value not to affect the functionality of the simulation.

To implement lazy update, we define four different operations and add additional datapaths, memory modules, and controllers called a safe weight handler (SafeW Handler) (Figure 10b). The four operations include: neuron computation for active neurons (NC-A, ①) and lazy neurons (NC-L, ②), weight accumulation (WA, ③), and safe weight update (SW, ④) for lazy neurons. NC-A in NeuroEngine is similar to the baseline (Figure 10a), but with additional computation to initialize the safe weight (SI). If the initialized safe weight is positive, the neuron becomes a lazy neuron and skips the out-event scheduling. Then, NeuroEngine flips the bitmap (Bitmap) in the scheduler to indicate that the neuron is a lazy neuron (①). NC-L (i.e., flushing the deferred computations), on the other hand, does not require prediction and safe weight initialization. Instead, NeuroEngine updates the states for the neurons in the lazy queue (Lazy Q) which stores the neurons that turn into an active neuron as a result of the SW (②).

In WA, NeuroEngine checks the bitmap of the destination neurons. If the bitmap indicates that the neuron is an active neuron, NeuroEngine schedules the event using the composite queue. On the other hand, NeuroEngine bypasses the composite queue and accumulates the weight to the history buffer (History Buffer) and schedule a safe weight update event to the safe weight update queue (SW Q) if the destination neuron is a lazy neuron. The history buffer is essentially an extended version of the accumulation buffer with a time index to store the accumulated weight with time. Also, the safe weight update queue handles the safe weight update events using the same structure as the in-event queue (③). Lastly, SW updates the safe weight of the lazy neurons in the safe weight update queue. If the safe weight overflows and becomes negative, the neuron turns into an active neuron, and the bitmap is flipped. Then, the handler pushes the neuron to the Lazy Q and flush the lazy queue in NC-L (④).

Then, we organize the operations in a way that does not incur resource contention while maximizing parallelism. The example Gantt chart (Figure 9) illustrates how we organize the operations. First, NeuroEngine concurrently performs NC-A and SW as the two operations do not incur resource contention. The other two operations (i.e., WA and NC-L) are blocked as NC-A occupies the

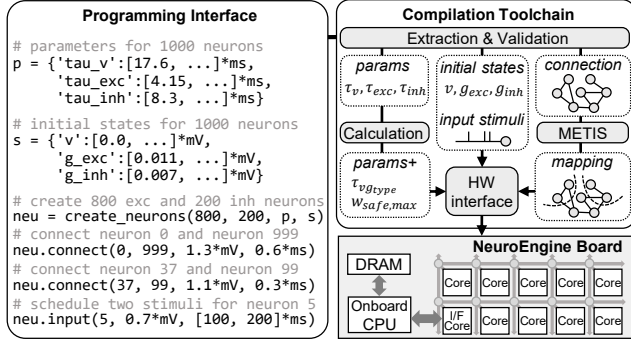


Figure 12: Programming interface and compilation toolchain.

state update datapaths and the composite queue. When NC-A finishes, NeuroEngine starts WA and NC-L operations by iterating over the spike buffer and the Lazy Q in parallel. After NeuroEngine completes WA, NC-L, and SW, NeuroEngine moves on to the next timestep. By maximizing the parallelism, NeuroEngine reduces the overall latency compared to the baseline.

There are two corner cases in the lazy update. If a neuron turns into a lazy neuron after NC-A, the remaining events for the neuron in the composite queue should update the safe weight instead of triggering the neuron computation. NeuroEngine detects such events by checking the bitmap after it pops an event from the composite queue and pushes the event to the safe weight queue (6). Also, if a neuron turns into an active neuron after SW, the remaining events in the safe weight queue should perform neuron computation instead of updating the safe weight. Thus, NeuroEngine checks the bitmap before updating the safe weight and perform neuron computation if the target neuron is an active neuron. Such cases incur stall cycles in the simulation; however, the overhead is minimal.

5 IMPLEMENTATION

5.1 NeuroEngine Hardware

To build an end-to-end simulation system, we implement NeuroEngine architecture on top of the scalable Network-on-Chip (NoC) similar to the existing SNN simulation systems [1, 14]. The NoC supports up to 32 cores using the 2D mesh interconnection network. Each NeuroEngine core consists of neuron and synapse memory, 32-bit fixed-point datapaths, the composite queue to handle the events, and accumulation buffers to store the accumulated weights and history. A single core can process up to 512 neurons and 262,144 connections using a 0.60-MB per-core SRAM, which serves as neuron and synapse memory and an additional 0.03-MB per-core SRAM for event queues and history buffers. For the weight accumulation unit, we adopt a memory architecture which supports up to eight synaptic delays as in the existing architecture [14]. NeuroEngine communicates with the host via an onboard CPU and interface core. The onboard CPU supervises the operation of the NeuroEngine chip, and the interface core (I/F Core) in the chip acts as an interface between the onboard CPU and NeuroEngine cores. Also, multiple NeuroEngine chips can be connected together for larger-scale simulations.

5.2 Software Stack

5.2.1 Programming Interface. We implement a programming interface and compilation toolchain to deploy and evaluate SNNs on NeuroEngine (Figure 12). The programming interface is designed to describe arbitrary network topologies. Using the provided API, users can create neurons, connect them with synapses, and set the network parameters (e.g., synaptic weights, delays, θ , τ_v , and τ_g) and initialize the states. Following are the essential APIs our interface provides:

```
create_neurons(exc_neu, inh_neu, params, init_states)
connect(src_neu_id, dst_neu_id, weight, delay)
input(neu_id, weight, timing)
```

`create_neurons` generates a group of biexponential neurons and initializes them with `params` and `init_states`. The number of the excitatory and inhibitory neurons are specified by `exc_neu` and `inh_neu`. `connect` creates a synapse between the two neurons with the given weight and delay. Lastly, using `input`, the user can inject a series of input stimuli into a neuron at specific timesteps. We also provide additional APIs to mitigate the programming burden in network construction such as `random_connect` which connects a group of neurons with a given probability. An example code for SNN creation is shown in Figure 12.

5.2.2 Compilation Toolchain. Using the program, our compiler checks if the network parameters violate the hardware constraints (i.e., maximum synaptic delay and the size of the network). Then, the compiler calculates additional simulation parameters to enable event-driven computations and our optimization schemes (i.e., τ_{vgtype} and $w_{safe,max}$) using the user-provided parameters and Equation 2 and 4. Next, the compiler extracts the connectivity of the network in the form of an adjacency matrix. Then we partition the matrix to distribute the neurons to the physical cores in a way that minimizes inter-core connections using an open-source partitioning program called Metis [29]. After that, the toolchain sends all parameters and mapping information to NeuroEngine Board through the PCIe interface. Lastly, the onboard CPU sends the data to NeuroEngine's interface-core via an AXI interface, which then transfers the data to the dedicated target cores. We verify the operation of our full system by deploying NeuroEngine on the Xilinx VCU1525 FPGA board and running SNN simulation with the toolchain.

6 EVALUATION

6.1 Experimental Setup

To evaluate NeuroEngine, we compare our 32-core NeuroEngine system against the hardware-based time-driven simulator. For the baseline design, we carefully implement a simulator based on existing time-driven architecture, Loihi [14]. We use the term Time to indicate the Loihi-like baseline with the time-driven execution mechanism for brevity. To measure the energy efficiency and area of NeuroEngine and Time, we write Register-Transfer-Level (RTL) code in Verilog and synthesize it using Samsung 65-nm technology. The SRAM and NoC costs are measured using CACTI 7 [3] and Orion 2.0 [28], respectively. The synthesis results in each simulator

Table 1: Chip area of Time and NeuroEngine.

Accelerator	Component	Area [mm ²]
Time	Neuron Memory & Datapath	4.14
	Synapse Mem.	151.20
	Accumulation Buf.	51.56
	Router + Link	15.31
	<i>Total</i>	222.21
NeuroEngine	Neuron Memory & Datapath	9.47
	Synapse Mem.	151.20
	Accumulation & History Buf.	68.72
	Router + Link	16.14
	Event Queue	7.38
	<i>Total</i>	252.91

Table 2: Evaluated SNN benchmarks.

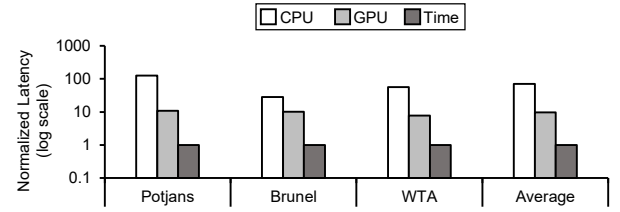
Name	Neurons	Synapses
Brette [7]	10,000	2,000,000
Brunel [8]	5,000	2,500,000
Vogels [64]	4,000	320,000
Potjans [44]	8,000	3,000,000
WTA [16]	3,984	3,814,400
TNN [57]	10,656	86,400

operate at 500 MHz. Table 1 shows the area consumption after the synthesis. NeuroEngine greatly improves the performance with only 13.8% area overheads.

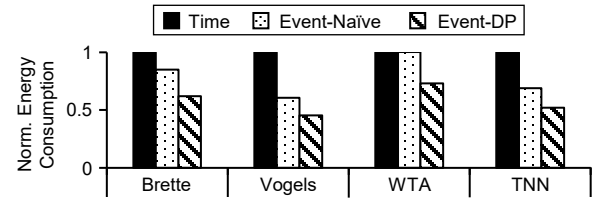
We collect six different benchmarks from the previous studies using two-stage SNNs (Table 2). The benchmarks are of various scales with different numbers of neurons and synapses. The three benchmarks, Brette, Vogels, and Brunel, are variations of a random balanced network that is widely used to exploit mechanisms of the brain. Potjans is a biologically realistic SNN modeled using anatomical data. Each benchmark is configured to use a timestep of 0.1 ms and to run for 10,000 timesteps. We also use WTA [16] and TNN [57] which perform image classification or clustering workloads. WTA is the most basic type of network to perform classifications using the SNNs [17]. When porting WTA, we replace the neuron model with biexponential model and achieve 91.87% accuracy on the MNIST dataset which is slightly lower than the reported accuracy (91.9%). TNN adopts a new computational paradigm using two-stage neurons proposed in recent publications [57, 59]. When porting the network to simulators, we reduce the time precision of TNN to 3-bit following the recent findings [33, 59]. We also replace the neuron model with the biexponential model.

6.2 Efficiency of the Hardware-Based Solutions

First, we evaluate the efficiencies of the hardware-based solution over the software-based counterparts. For evaluation, we profile the latencies of the CPU- (Intel Xeon E5-2650 v4 CPU) and GPU-based (NVIDIA Titan X (Pascal)) solutions and compare them against Time on three SNN benchmarks. For evaluation, we use NEST to simulate the SNNs on the CPUs and GeNN to simulate on the GPUs. The profile result indicates the efficiencies of HW-based solutions over the SW-based solutions (Figure 13). CPU, GPU, and Time in the Figure indicates the CPU, the GPU, and the time-driven baseline, respectively. CPU suffers from an average of 70.27× higher latency

**Figure 13: SNN simulation latency profiling of CPU, GPU, and Time.****Table 3: Area and power consumption of the datapath.**

	Time	Event-Naïve	Event-DP
Power [mW]	12.02	83.37	60.80
Area [mm ²]	0.02	0.17	0.10

**Figure 14: Total energy consumption of the datapath.**

overhead compared to Time. GPU mitigates the overhead to some extent using a high-throughput processor, but the latency overhead is still 9.66× higher. To that end, the SNN simulations should rely on ASIC-based platforms to simulate SNNs at high speed.

6.3 Low Event-Driven Computation Costs

For the remaining evaluation sections (Section 6.3– 6.6), we use three variants of event-driven baseline in addition to Time: a basic event-driven simulator (Event-Naïve), an event-driven simulator with optimized datapath (Event-DP), and an event-driven simulator equipped with both optimized datapath and the composite queue (Event-CQ). Event-Naïve and Event-DP use the PHQ [26] as its event scheduler.

To evaluate the efficiency of the datapath, we compare the computation costs of our low-cost datapath against two baselines: Time and Event-Naïve. To identify the bit precision of the datapaths of Event-Naïve, we profile the benchmarks. The profile results indicate that one of our benchmarks, WTA, requires up to 12 bits to fully represent the time intervals and results in 40-bit fixed-point datapaths, while the optimized datapath requires only 32 bits.

Table 3 shows the area and power consumption of each datapath and Figure 14 illustrates the total energy consumption of the datapath normalized to Time on four benchmarks. The result indicates that the event-driven execution mechanism complicates the datapath to some degree; however, it executes the datapath far less than the time-driven counterpart to achieve higher energy efficiency. Event-Naïve suffers from 6.94× larger datapath power consumption and 8.47× larger area consumption compared

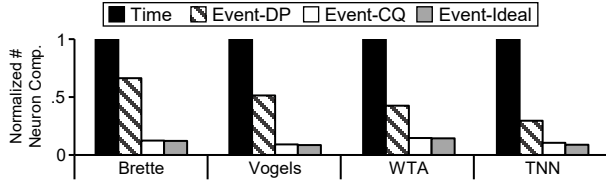


Figure 15: Reduced number of neuron computations.

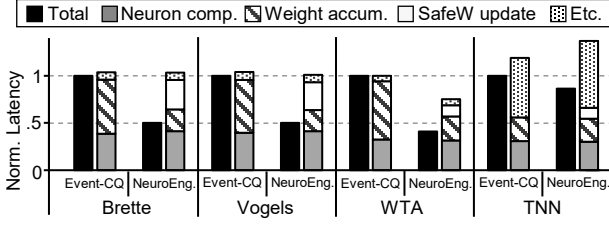


Figure 16: Reduced latency with increased parallelism.

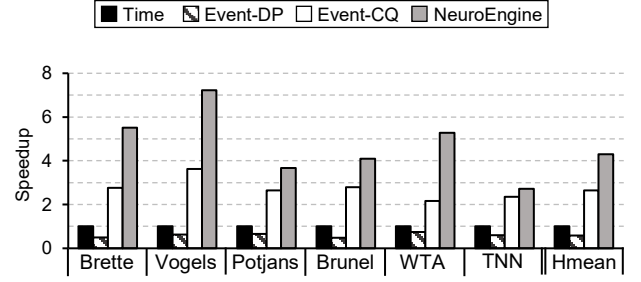
to Time. Event-DP achieves $1.37\times$ lower datapath energy consumption compared to Event-Naïve. Although Event-DP suffers from high datapath power consumption, it reduces the energy consumed by 42% on average with a smaller number of computations. Compared to Event-Naïve, Event-DP increases the energy efficiency of datapath by $1.35\times$. The gap between the energy efficiency ($1.35\times$) and power consumption ($1.37\times$) of datapath is due to the slight increase in the number of computations from sub-intervals and the approximate divider (Section 4.1).

6.4 Eliminated Redundant Event Scheduling

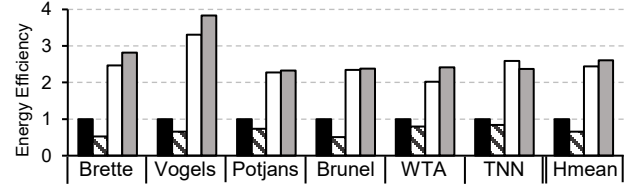
To examine the effectiveness of the composite queue (Section 4.2), we profile the total number of neuron computations in the simulation (Figure 15). Event-Ideal indicates the ideal number of computations excluding duplicate or dead events. The profiling results show that the composite queue properly avoids the scheduling of redundant events. Compared to Time, Event-DP reduces the number of computations by 53% on average. Event-CQ further decreases the number of computations by 76% in comparison to Event-DP by removing redundant computations. Remarkably, Event-CQ achieves near-ideal computation reduction with only 6% overhead due to the approximation in the datapath (Section 4.1).

6.5 Speedup Using Lazy Update

In this experiment, we evaluate the parallelization opportunities introduced with lazy update (Section 4.3). For evaluation, we profile the active cycles of neuron computation datapath, weight accumulation unit, and safe weight handler (Figure 16). The Etc represents the routing and synchronization overhead. We examine the degree of parallelism (i.e., DOP) by dividing the sum of active cycles of each component by the overall latency. Compared to Event-CQ which achieves only 1.06 DOP on average, NeuroEngine achieves 1.80 DOP. With the high parallelization opportunity, NeuroEngine achieves $1.76\times$ speedup over Event-CQ. We also observe that lazy update further reduce the overall latency by reducing the weight



(a) Speedup normalized to the time-driven simulator



(b) Energy efficiency normalized to the time-driven simulator

Figure 17: Improvements in the SNN simulation efficiency of NeuroEngine over time-driven and event-driven baselines.

accumulation overhead for two reasons. First, the weight accumulation for lazy neurons bypass the composite queue and do not suffer from slow priority queue accesses. Second, lazy update performs neuron computation for neurons that are about to fire a spike; therefore, the spikes are generated in a shorter time interval and fully utilize the pipeline stages for weight accumulation. Note that TNN introduces a large load imbalance between the cores after the feedforward inhibition [57]; therefore, both NeuroEngine and Event-CQ suffer from large load imbalance (Etc) overhead.

6.6 Fast & Efficient Event-Driven Simulations

In this experiment, we evaluate the overall speedup and energy efficiency of NeuroEngine. We employ two evaluation metrics: simulation speed and energy consumption. For comparison, we measure the latency and the energy consumption of the hardware with the six benchmarks (Table 2). Our results indicate the significant performance and energy efficiency improvements of NeuroEngine (Figure 17). Using the composite queue, Event-CQ achieves $2.65\times$ harmonic mean speedup over Time while Event-DP is slower compared to Time. Using the lazy update, NeuroEngine improves the performance and energy efficiency, achieving $4.30\times$ and $2.60\times$ harmonic mean speedup and energy efficiency over Time.

6.7 Iso-area Comparisons

We conduct an iso-area comparison of NeuroEngine as the increased complexity in the datapath and the memory units consume a larger area (Table 1). For fairness, we increase the number of datapaths with additional memory ports to increase parallelism in Time. The modified Time baseline consumes 253.02 mm^2 which is slightly larger than NeuroEngine. Then, we evaluate the modified version under the same benchmarks and compare it against NeuroEngine. Figure 18 illustrates the normalized speedup and energy efficiency of NeuroEngine over the iso-area Time. The result

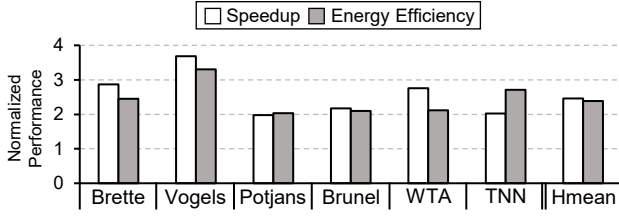


Figure 18: The speedup and energy efficiency of NeuroEngine over the iso-area Time baseline.

indicates that NeuroEngine still achieves 2.62× harmonic mean speedup and 2.38× energy efficiency over the iso-area baseline.

7 RELATED WORK

7.1 Time-Driven SNN Simulators

There is a wide range of studies focused on simulating SNNs using the time-driven approach. Among the simulators, NEST [22], Brian [23], Auryn [66], and NEURON [24] are the representative CPU-based simulators. GeNN [65], Carlsim [5], and BSIM [46] improve the performance of the CPU-based simulators using GPUs. Spinnaker [42] improves scalability by integrating low-power CPUs on top of custom communication and memory architecture. Some simulators adopt FPGAs for computations [11]. To improve performance over the SW-based and FPGA-based approaches, prior studies design ASIC-based accelerators (e.g., TrueNorth [1], Loihi [14], TianjiC [41], ROLLS [45]) achieving high scalability and performance over the SW solutions. Despite the high performance over SW, the accelerators suffer from intensive computations of the time-driven simulations.

7.2 Event-Driven SNN Simulators

To minimize the computational overhead in the time-driven simulators, some simulators adopt event-driven simulation methodology. NEVESIM [40], ED-LUT [48], PUNNETS [32], and MVASpike [6, 47] are the CPU-based simulators supporting event-driven computations. Among them, ED-LUT supports two-stage neurons using a LUT table. Also, the NEVESIM, PUNNETS, and MVASpike support two-stage neurons using a linear approximation for the prediction. To improve the performance over the CPU-based approach, some simulators adopt FPGAs or ASICs for the event-driven simulation. But, none of the existing works effectively handles the two-stage neurons and incoming spikes with synaptic delays. Mini-taur [37] and PEASE [49] are hardware-based event-driven simulators; however, they only support one-stage neurons. Also, other FPGA-based simulators supporting simplified two-stage neurons using the SHQ [10, 52] do not effectively handle incoming spikes with synaptic delays. To the best of our knowledge, NeuroEngine is by far the first work to effectively handle event-driven simulations of two-stage neurons with synaptic delays.

7.3 Spatially-Expanded SNN Simulators

Some SNN simulators adopt spatially expanded architecture where each neuron is computed with different physical units. Unlike existing simulators which use a single computing unit (e.g.,

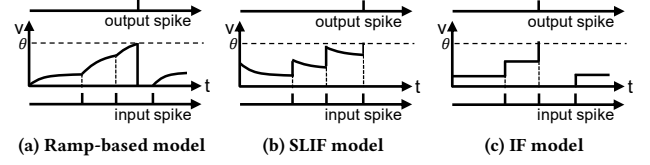


Figure 19: Simpler neuron models supported by NeuroEngine.

digital circuit) to simulate multiple neurons, they have a dedicated computing unit for each simulating neuron. For example, Nebula [55] and Inxs [34] use memristor-based brain-inspired circuits to simulate each neuron. Neurogrid [4] and BrainScaleS [53] use a dedicated analog circuit for each simulating neuron. Nair et al. [33] directly implement each neuron using digital CMOS circuits. These simulators cannot adopt event-driven computation as all the computations are done in parallel; however, they incur large chip area consumption and are not scalable to simulate large scale SNNs.

8 DISCUSSION

8.1 Flexibility in Neuron Models

The disadvantage of simulating SNNs in an event-driven manner is that it complicates the datapath due to the complex state update and prediction mechanisms. To that end, the NeuroEngine's architecture is tailored to only biexponential model and do not offer flexibility to support different neuron models. However, we can specialize biexponential model to emulate the behaviors of simple two-stage and one-stage neurons. Our schemes are also applicable to other models with even more complex dynamics as long as they have the behavioral properties we exploit.

8.1.1 Generality of the Target Model. Biexponential model is capable of emulating simpler neuron models (e.g., IF, SLIF, Ramp-based) by using appropriate parameters. Accordingly, NeuroEngine can simulate such models without any modification on the datapath. Instead, NeuroEngine compiler emulates other models by carefully selecting three model parameters (i.e., $1/\tau_v$, $1/\tau_{gtype}$, and τ_{vgtype}). For example, the Ramp-based model is a two-stage model without membrane voltage leakage (Figure 19a). Taking the membrane time constant as infinity (i.e., $1/\tau_v = 0$) results in the absence of the leakage and emulates the ramping behavior. As another example, Simple Leaky Integrate-and-Fire (SLIF) model is a one-stage neuron model with exponential membrane voltage leakage (Figure 19b). To generate the infinite slope of the one-stage neurons, the compiler sets the current-related parameters to make the current decay instantly (i.e., $1/\tau_{gtype} = \infty$, $\tau_{vgtype} = 1$). For the simplest non-leaky Integrate-and-Fire (IF) model (Figure 19c), we additionally set the membrane time constant to infinity.

8.1.2 Versatility of the Schemes. To maximize the efficiency of the event-driven simulation, we propose three schemes by exploiting the behavioral properties of biexponential model. Most of the widely used neuron models share the properties and can adopt our solutions. Each technique is applicable when the target neuron

model meets the following constraints. First, our datapath optimization technique assumes that the differential equation for the target neuron model has a closed-form solution and a pessimistic linear approximation model such as Equation 3. Some highly complex forms of neuron models (e.g., Hodgkin-Huxley model [25]) use multiple state variables with a super-linear increase in the membrane voltage. Such models are too complex to derive closed-form solutions and are not suitable for the pessimistic linear approximation. Second, the composite queue structure is applicable to the arbitrary type of neuron models. Lastly, the lazy update method applies to the neuron models which have a bounded increase in the membrane voltage and whose voltage increases less when the initial voltage is smaller. Most of the models fall into this category; however, some exceptions exist. For example, Exponential Integrate and Fire (EIF) model starts to exponentially increase the voltage when the voltage reaches a certain level. In this case, the voltage increases more with higher voltage value; therefore, we cannot initialize and update the safe weight using Equation 5–6.

8.2 Integrating Learning Rules

In the presence of the recent works emphasizing the need for brain-like learning rules (e.g., STDP), some researchers demand learning rule support in the system. We focus on accelerating neuron computations using our system, but we can easily extend it to support learning rules by adopting existing architectures. For example, FlexLearn [2] provides a flexible datapath supporting various learning rules. Also, Loihi [14] presents a programmable learning engine using 16-bit microcodes. The architectures are orthogonal to the execution mechanisms of neurons, and thus, NeuroEngine can seamlessly integrate the architectures on top of it.

9 CONCLUSION

In this paper, we present NeuroEngine which enables fast and efficient advanced brain-inspired computing using the event-driven simulation system. The idea of NeuroEngine is to optimize the key components in event-driven simulation ranging from the datapaths to the event scheduling units. We further improve the simulation by proposing a novel method called lazy update to increase the parallelism. The resulting design, NeuroEngine, achieves significant performance improvement and energy efficiency compared to both state-of-the-art time-driven and naive event-driven simulators.

ACKNOWLEDGMENTS

This work was partly supported by Samsung Electronics, Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Korean Government (NRF-2017R1A2B3011038, NRF-2019R1A5A1027055, NRF-2020M3H6A10-84857, NRF-2019-Global Ph.D. Fellowship Program), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.17110809-72). The EDA tool and the standard cell library was supported by the IC Design Education Center (IDEC), Korea. We also appreciate the support from Automation and Systems Research Institute (ASRI), Inter-university Semiconductor Research Center (ISRC) and Neural Processing Research Center (NPRC) at Seoul National University.

REFERENCES

- [1] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar, William P. Risk, Bryan Jackson, and Dharmendra S. Modha. 2015. TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 10 (2015). <https://doi.org/10.1109/TCAD.2015.2474396>
- [2] Eunjin Baek, Hunjun Lee, Youngsok Kim, and Jangwoo Kim. 2019. FlexLearn: fast and highly efficient brain simulations using flexible on-chip learning. In *Proc. 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. <https://doi.org/10.1145/3352460.3358268>
- [3] Rajeev Balasubramanian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017). <https://doi.org/10.1145/3085572>
- [4] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R. Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V. Arthur, Paul A. Merolla, and Kwabena Boahen. 2014. Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations. *Proc. IEEE* 102, 5 (2014). <https://doi.org/10.1109/JPROC.2014.2313565>
- [5] Michael Beyeler, Kristofer D. Carlson, Ting-Shuo Chou, Nikil Dutt, and Jeffrey L. Krichmar. 2015. CARLsim 3: A User-Friendly and Highly Optimized Library for the Creation of Neurobiologically Detailed Spiking Neural Networks. In *Proc. International Joint Conference on Neural Networks (IJCNN)*. <https://doi.org/10.1109/IJCNN.2015.7280424>
- [6] Romain Brette. 2006. Exact simulation of integrate-and-fire models with synaptic conductances. *Neural Computation* 18, 8 (2006). <https://doi.org/10.1162/neco.2006.18.8.2004>
- [7] Romain Brette, Michelle Rudolph Ted Carnevale, Michael Hines, David Beeman, James M. Bower, Markus Diesmann, Abigail Morrison, Philip H. Goodman, Jr. Frederick C. Harris, Milind Zirpe, Thomas Natschlager, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P. Davison, Sami El Boustani, and Alain Destexhe. 2007. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience* 23, 3 (2007). <https://doi.org/10.1007/s10827-007-0038-6>
- [8] Nicolas Brunel. 2000. Dynamics of Sparsely Connected Networks of Excitatory and Inhibitory Spiking Neurons. *Journal of Computational Neuroscience* 8, 3 (2000). <https://doi.org/10.1023/A:1008925309027>
- [9] Yongqiang Cao, Yang Chen, and Deepak Khosla. 2015. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision* (2015). <https://doi.org/10.1007/s11263-014-0788-3>
- [10] Louis-Charles Caron, Michiel D'Haene, Frédéric Mailhot, Benjamin Schrauwen, and Jean Rouat. 2013. Event management for large scale event-driven digital hardware spiking neural networks. *Neural networks* 45 (2013). <https://doi.org/10.1016/j.neunet.2013.02.005>
- [11] Kit Cheung, Simon R Schultz, and Wayne Luk. 2016. NeuroFlow: a general purpose spiking neural network simulation platform using customizable processors. *Frontiers in neuroscience* 9 (2016), 516. <https://doi.org/10.3389/fnins.2015.00516>
- [12] Iulia M Comsa, Thomas Fischbacher, Krzysztof Potempa, Andrea Gesmundo, Luca Versari, and Jyrki Alakuijala. 2020. Temporal coding in spiking neural networks with alpha synaptic function. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. <https://doi.org/10.1109/ICASSP40776.2020.9053856>
- [13] Mike Davies. 2019. Benchmarks for progress in neuromorphic computing. *Nature Machine Intelligence* 1, 9 (2019), 386–388. <https://doi.org/10.1038/s42256-019-0097-1>
- [14] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steve McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan, Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. 2018. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro* 38, 1 (2018). <https://doi.org/10.1109/MM.2018.112130359>
- [15] Michiel D'Haene, Benjamin Schrauwen, Jan Van Campenhout, and Dirk Stroobandt. 2009. Accelerating event-driven simulation of spiking neurons with multiple synaptic time constants. *Neural computation* 21, 4 (2009). <https://doi.org/10.1162/neco.2008.02-08-707>
- [16] Peter U. Diehl and Matthew Cook. 2015. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience* 9 (2015). <https://doi.org/10.3389/fncom.2015.00099>
- [17] Zidong Du, Daniel D Ben-Dayana Rubin, Yunji Chen, Liqiang He, Tianshi Chen, Lei Zhang, Chengyong Wu, and Olivier Temam. 2015. Neuromorphic Accelerators: A Comparison Between Neuroscience and Machine-Learning Approaches. In *Proc. 48th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. <https://doi.org/10.1145/2830772.2830789>

- [18] Gabriel A Fonseca Guerra and Steve B Furber. 2017. Using stochastic spiking neural networks on spinnaker to solve constraint satisfaction problems. *Frontiers in neuroscience* 11 (2017), 714. <https://doi.org/10.3389/fnins.2017.00714>
- [19] Charlotte Frenkel, Martin Lefebvre, Jean-Didier Legat, and David Bol. 2018. A 0.086-mm² 12.7-pJ/SOP 64k-Synapse 256-Neuron Online-Learning Digital Spiking Neuromorphic Processor in 28nm CMOS. *IEEE transactions on biomedical circuits and systems* 13, 1 (2018), 145–158. <https://doi.org/10.1109/TBCAS.2018.2880425>
- [20] Jesus A Garrido, Richard R Carrillo, Niceto R Luque, and Eduardo Ros. 2011. Event and time driven hybrid simulation of spiking neural networks. In *Proc. International Work-Conference on Artificial Neural Networks (IWANN)*. Springer, 554–561. https://doi.org/10.1007/978-3-642-21501-8_69
- [21] Wulfram Gerstner and Werner M. Kistler. 2002. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511815706>
- [22] Marc-Oliver Gewaltig and Markus Diesmann. 2007. NEST (NEural Simulation Tool). *Scholarpedia* 2, 4 (2007).
- [23] Dan F. M. Goodman and Romain Brette. 2009. The Brian simulator. *Frontiers in Neuroscience* 3, 2 (2009). <https://doi.org/10.3389/fnins.2009.0026.2009>
- [24] M. L. Hines and N. T. Carnevale. 2001. NEURON: A Tool for Neuroscientists. *The Neuroscientist* 7 (2001). <https://doi.org/10.1177/107385840100700207>
- [25] A. L. Hodgkin and A. F. Huxley. 1952. A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve. *The J. of Physiology* 117 (1952). <https://doi.org/10.1113/jphysiol.1952.sp004764>
- [26] Aggelos Ioannou and Manolis G. H. Katevenis. 2007. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks. *IEEE/ACM Transactions on Networking* 15, 2 (2007). <https://doi.org/10.1109/TNET.2007.892882>
- [27] Eugene M. Izhikevich. 2006. Polychronization: Computation with Spikes. *Neural Computation* 18 (2006). <https://doi.org/10.1162/089976606775093882>
- [28] Andrew B Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. 2009. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proc. Design, Automation & Test in Europe (DATE)*. <https://doi.org/10.1109/DATE.2009.5090700>
- [29] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392. <https://doi.org/10.1137/s1064827595287997>
- [30] Dayeol Lee, Gwangmu Lee, Dongup Kwon, Sunghwa Lee, Youngsok Kim, and Jangwoo Kim. 2018. Flexon: A Flexible Digital Neuron for Efficient Spiking Neural Network Simulations. In *Proc. 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. <https://doi.org/10.1109/ISCA.2018.00032>
- [31] Wolfgang Maass. 1997. Networks of Spiking Neurons: The Third Generation of Neural Network Models. *Neural Networks* 10, 9 (1997). [https://doi.org/10.1016/S0893-6080\(97\)00011-7](https://doi.org/10.1016/S0893-6080(97)00011-7)
- [32] Takaki Makino. 2003. A Discrete-Event Neural Network Simulator for General Neuron Models. *Neural Computing & Applications* 11, 3-4 (2003). <https://doi.org/10.1007/s00521-003-0358-z>
- [33] Harideep Nair, John Paul Shen, and James E Smith. 2020. Direct CMOS Implementation of Neuromorphic Temporal Neural Networks for Sensory Processing. (2020). [arXiv preprint arXiv:2009.00457](https://arxiv.org/abs/2009.00457).
- [34] Surya Narayanan, Ali Shafiee, and Rajeev Balasubramanian. 2017. INXS: Bridging the throughput and energy gap for spiking neural networks. In *Proc. International Joint Conference on Neural Networks (IJCNN)*. <https://doi.org/10.1109/IJCNN.2017.7966154>
- [35] Francisco Naveiros, Jesus A Garrido, Richard R Carrillo, Eduardo Ros, and Niceto R Luque. 2017. Event-and time-driven techniques using parallel CPU-GPU co-processing for spiking neural networks. *Frontiers in Neuroinformatics* 11 (2017). <https://doi.org/10.3389/fninf.2017.00007>
- [36] Emre O Neftci, Hesham Mostafa, and Friedemann Zenke. 2019. Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks. *IEEE Signal Processing Magazine* 36 (2019), 61–63. <https://doi.org/10.1109/MSP.2019.2931595>
- [37] Daniel Neil and Shih-Chii Liu. 2014. Minitaur, an event-driven FPGA-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 12 (2014). <https://doi.org/10.1109/TVLSI.2013.2294916>
- [38] Eustace Painkras, Luis A. Plana, Jim Garside, Steve Temple, Simon Davidson, Jeffrey Pepper, David Clark, Cameron Patterson, and Steve Furber. 2012. SpiN-Naker: A Multi-Core System-on-Chip for Massively-Parallel Neural Net Simulation. In *Proc. IEEE Custom Integrated Circuits Conference (CICC)*. <https://doi.org/10.1109/CICC.2012.6330636>
- [39] Eustace Painkras, Luis A Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R Lester, Andrew D Brown, and Steve B Furber. 2013. SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits* 48, 8 (2013). <https://doi.org/10.1109/JSSC.2013.2259038>
- [40] Dejan Pecevski, David Kappel, and Zeno Jonke. 2014. NEVESIM: event-driven neural simulation framework with a Python interface. *Frontiers in neuroinformatics* 8 (2014). <https://doi.org/10.3389/fninf.2014.00070>
- [41] Jing Pei, Lei Deng, Sen Song, Mingguo Zhao, Youhui Zhang, Shuang Wu, Guanrui Wang, Zhe Zou, Zhenzhi Wu, Wei He, Feng Chen, Ning Deng, Si Wu, Yu Wang, Yujie Wu, Zheyu Yang, Cheng Ma, Guoqi Li, Wentao Han, Huanglong Li, Huaqiang Wu, Rong Zhao, Yuan Xie, and Luping Shi. 2019. Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature* 572, 7767 (2019). <https://doi.org/10.1038/s41586-019-1424-8>
- [42] Luis A. Plana, David Clark, Simon Davidson, Steve Furber, Jim Garside, Eustace Painkras, Jeffrey Pepper, Steve Temple, and John Bainbridge. 2011. SpiNNaker: Design and Implementation of a GALS Multicore System-on-Chip. *ACM Journal on Emerging Technologies in Computing Systems* 7, 4 (2011). <https://doi.org/10.1145/2043643.2043647>
- [43] Filip Jan Poulak and John J Hopfield. 2013. Rapid, parallel path planning by propagating wavefronts of spiking neural activity. *Frontiers in computational neuroscience* 7 (2013), 98. <https://doi.org/10.3389/fncom.2013.00098>
- [44] Tobias C. Potjans and Markus Diesmann. 2014. The Cell-Type Specific Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Model. *Cerebral Cortex* 24 (2014). <https://doi.org/10.1093/cercor/bhs358>
- [45] Ning Qiao, Hesham Mostafa, Federico Corradi, Marc Osswald, Fabio Stefanini, Dora Sumislawska, and Giacomo Indiveri. 2015. A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses. *Frontiers in neuroscience* 9 (2015). <https://doi.org/10.3389/fnins.2015.00141>
- [46] Peng Qu, Youhui Zhang, Xiang Fei, and Weimin Zheng. 2020. High Performance Simulation of Spiking Neural Network on GPGUs. *IEEE Transactions on Parallel and Distributed Systems* 31, 11 (2020), 2510–2523. <https://doi.org/10.1109/TPDS.2020.2994123>
- [47] Olivier Rochel and Dominique Martinez. 2003. An event-driven framework for the simulation of networks of spiking neurons. In *11th European Symposium On Artificial Neural Networks (ESANN)*.
- [48] Eduardo Ros, Richard Carrillo, Eva M Ortigosa, Boris Barbour, and Rodrigo Agis. 2006. Event-driven simulation scheme for spiking neural networks using lookup tables to characterize neuronal dynamics. *Neural Computation* 18, 12 (2006). <https://doi.org/10.1162/neco.2006.18.12.2959>
- [49] Arnab Roy, Swagath Venkataramani, Neel Gala, Sanchari Sen, Kamakoti Veezhnathan, and Anand Raghunathan. 2017. A programmable event-driven architecture for evaluating spiking neural networks. In *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. <https://doi.org/10.1109/ISLPED.2017.8009176>
- [50] Bodo Rueckauer and Shih-Chii Liu. 2018. Conversion of analog to spiking neural networks using sparse temporal coding. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*. <https://doi.org/10.1109/ISCAS.2018.8351295>
- [51] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. 2017. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in neuroscience* 11 (2017), 682. <https://doi.org/10.3389/fnins.2017.00682>
- [52] Guillaume Séguin-Godin, Frederic Mailhot, and Jean Rouat. 2015. Efficient event-driven approach using synchrony processing for hardware spiking neural networks. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*. <https://doi.org/10.1109/ISCAS.2015.7169242>
- [53] Abhronil Sengupta, Yong Shim, and Kaushik Roy. 2016. Proposal for an all-spin artificial neural network: Emulating neural and synaptic functionalities through domain wall motion in ferromagnets. *IEEE transactions on biomedical circuits and systems* 10, 6 (2016), 1152–1160. <https://doi.org/10.1109/TBCAS.2016.2525823>
- [54] Sumit Bam Shrestha and Garrick Orchard. 2018. Slayer: Spike layer error reassignment in time. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [55] Sonali Singh, Anup Sarma, Nicholas Jao, Ashutosh Pattnaik, Sen Lu, Kezhou Yang, Abhronil Sengupta, Vijaykrishnan Narayanan, and Chita R Das. 2020. NEBULA: a neuromorphic spin-based ultra-low power architecture for SNNs and ANNs. In *Proc. 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. <https://doi.org/10.1109/ISCA45697.2020.00039>
- [56] James E Smith. 2014. Efficient digital neurons for large scale cortical architectures. In *Proc. 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*. <https://doi.org/10.1145/2678373.2665707>
- [57] James E. Smith. 2017. *Space-Time Computing with Temporal Neural Networks*. Morgan & Claypool. <https://doi.org/10.2200/S00771ED1V01Y201704CAC039>
- [58] James E Smith. 2018. Space-time algebra: a model for neocortical computation. In *Proc. 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. <https://doi.org/10.1109/ISCA.2018.00033>
- [59] James E. Smith. 2020. A Neuromorphic Paradigm for Online Unsupervised Clustering. (2020). [arXiv preprint arXiv:2005.04170](https://arxiv.org/abs/2005.04170).
- [60] Athul Sripad, Giovanni Sanchez, Mireya Zapata, Vito Pirrone, Taho Dorta, Salvatore Cambria, Albert Marti, Karthikeyan Krishnamourthy, and Jordi Madrenas. 2018. SNAVA—A real-time multi-FPGA multi-model spiking neural network simulation architecture. *Neural Networks* 97 (2018), 28–45. <https://doi.org/10.1016/j.neunet.2017.09.011>

- [61] Marcel Stimberg, Romain Brette, and Dan FM Goodman. 2019. Brian 2, an intuitive and efficient neural simulator. *Elife* 8 (2019). <https://doi.org/10.7554/eLife.47314>
- [62] Ping Tak Peter Tang, Tsung-Han Lin, and Mike Davies. 2017. Sparse coding by spiking neural networks: Convergence theory and computational results. (2017). arXiv preprint arXiv:1705.05475.
- [63] David Thomas and Wayne Luk. 2009. FPGA accelerated simulation of biologically plausible spiking neural networks. In *Proc. 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*. <https://doi.org/10.1109/FCCM.2009.46>
- [64] Tim P Vogels and Larry F Abbott. 2005. Signal propagation and logic gating in networks of integrate-and-fire neurons. *Journal of neuroscience* 25, 46 (2005). <https://doi.org/10.1523/JNEUROSCI.3508-05.2005>
- [65] Esin Yavuz, James Turner, and Thomas Nowotny. 2016. GeNN: a code generation framework for accelerated brain simulations. *Scientific Reports* 6 (2016). <https://doi.org/10.1038/srep18854>
- [66] Friedemann Zenke and Wulfram Gerstner. 2014. Limits to high-speed simulations of spiking neural networks using general-purpose computers. *Frontiers in Neuroinformatics* 8 (2014). <https://doi.org/10.3389/fninf.2014.00076>