2024 34th International Conference on Field-Programmable Logic and Applications (FPL) | 979-8-3315-3007-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/FPL64840.2024.00042

An Open-Source and Extensible Framework for Fast Prototyping and Benchmarking of Spiking Neural Network Hardware

Shadi Matinizadeh and Anup Das
Drexel University
Philadelphia, Pennsylvania, USA
Email: {sm4884,anup.das}@drexel.edu

***Abstract*—Spiking neural networks (SNNs) are bioplausible machine learning models that use discrete spikes to encode, compute, and transmit information. Combined with event-driven low-power hardware, SNNs can improve the energy efficiency of learning tasks. Although there have been several efforts to build SNN hardware, there is no uniform framework to verify and benchmark these designs in terms of key hardware performance metrics such as inference accuracy, area, power consumption, and throughput. We propose PRONTO, an open-source and extensible framework to verify SNN hardware for different learning tasks and datasets. Given the ubiquity of PyTorch in the machine learning community and for demonstration purposes, the frontend of PRONTO is integrated with a torch-based SNN simulator for model specification and training. Its backend is integrated with an open-source quantized SNN hardware. PRONTO interfaces with a torch code to generate input stimuli which are then driven to SNN hardware through a configurable SystemVerilog testbench, verifying the design across various SNN-specific configurations. PRONTO utilizes a dataflow-based approach to validate SNN models that are segmented and run on a mix of software and hardware platforms. We describe PRONTO and evaluate it using six datasets spanning image, audio, and text classification. We present benchmark results for various input settings. PRONTO is available under an open-source licensing to provide a platform to evaluate all current and future SNN hardware designs. We believe PRONTO will substantially reduce the design verification effort, thus facilitating fast design prototyping.**

***Index Terms*—spiking neural networks (SNNs), verification, testbench, SystemVerilog, PyTorch.**

I. Introduction

Unlike classical machine learning algorithms, where neurons are characterized by single-, static-, and continuous-valued activation, biological neurons use discrete spikes to compute and transmit information. Spiking Neural Networks (SNNs), which use these neurons, are therefore more biologically realistic and efficient mechanisms for spatio-temporal information processing [1]. Like other machine learning models, SNNs can run in software on general-purpose hardware such as a CPU or GPU. However, SNNs achieve energy efficiency when implemented on event-driven hardware such as μBrain [2], DYNAPs [3], Loihi [4], TrueNorth [5], NeuroGrid [6], NeuRRAM [7], and SpiNNaker [8]. These systems are not easily accessible to the general computing community. Therefore, FPGAs remain the de facto platform for fast prototyping and benchmarking of SNN designs [9].

Recently, several SNN designs have been proposed targeting FPGA platforms [10]–[20]. There are also High-Level Synthesis (HLS) approaches to implement SNN models on an FPGA [21]–[24]. Each design comes with its own verification and evaluation methodology to compare against other designs for a select few classification datasets. For instance, Table I compares existing designs for the MNIST dataset.

TABLE I. SNN FPGA hardware for MNIST dataset.

	[16]	[17]	[18]	[19]	[20]
SNN Hardware	784-1024-10	784-2048-10	256-128-10	784-500-500-10	784-1024-1024-10
Power (W)	3.4	1.03	0.623	1.5	0.477
Accuracy (%)	98.4	93.0	96.5	94.2	97.06

A limitation of such an evaluation is that the verification framework that includes SNN-specific settings is typically not exposed, meaning that they may not be uniform across these designs. We put the spike encoding mechanism into perspective, which defines how each input is encoded to generate spike trains that are driven to SNN hardware. Rate coding, which encodes an input value as the frequency of spikes, generates more spikes than inter-spike interval (ISI) coding, which encodes it as the latency between spikes. Designs that use ISI coding consume less power/energy as a result of processing sparse spike events. This sparsity depends on (1) the time duration for which an input is presented to hardware (called time steps) and (2) the number of spikes generated in a time interval (called spike rate). We evaluate these input parameters for an SNN hardware design targeted for FPGA [18] and show that these parameters significantly impact hardware performance, such as inference accuracy, area, power, and throughput per watt (see Section V).

Furthermore, verification of SNN designs is a crucial step, as it directly influences production performance on ASIC and FPGA, and ultimately determines product functionality and customer perception. Currently, design verification accounts for a substantial portion of the product development life cycle and is expected to become a critical bottleneck as the complexity of SNNs increases. The SNN hardware community lacks an engineering setup that can be easily customized and used for fast verification and prototyping of SNN designs.

Contributions – To address these research needs, we propose PRONTO, an open-source and extensible framework for the interface between software and hardware, allowing fast prototyping and benchmarking of SNN hardware on FPGA.

The following are our key contributions.

1) Given its flexibility, expressiveness, GPU acceleration of training algorithms, and a large user base, the frontend of PRONTO is designed to directly interface with simulators such as snnTorch [25] and SpikingJelly [26], which use the torch dialect to specify and train SNNs. PRONTO facilitates simulating several different SNN

979-8-3315-3007-5/24/$31.00 ©2024 IEEE
DOI 10.1109/FPL64840.2024.00042

models, information encoding techniques, and classification datasets. While PRONTO natively supports torch-based SNN simulators, we provide hooks to extend support for other standalone SNN simulators such as CARLsim [27], ENLARGE [28], and Brian [29].

2) Given its open-source development methodology and for demonstration purposes, the backend of PRONTO is integrated with QUANTISENC [18], a software-defined SNN hardware written in the Verilog hardware description language (HDL). PRONTO can be used to benchmark other designs in terms of hardware performance for different SNN settings, such as the spike encoding technique, time steps, and classification dataset.
3) We engineer a highly modular testbench written in SystemVerilog to interface with a Python-based torch code to extract model parameters (synaptic weights) and generate input stimuli. PRONTO uses this testbench to program synaptic weights in hardware memory, e.g., in BRAM, and drives stimuli to hardware, one input at a time via a design-under-test (DUT)-specific interface. This interface can be easily customized for a target SNN hardware. The testbench simulates an SNN design for a certain time duration (a user configuration) and decodes the hardware output for performance estimation.
4) Finally, we propose a dataflow-based design methodology to validate SNN models that are segmented and run on a mix of software and hardware platforms. In this way, PRONTO provides support for SNN models that perform certain operations such as batch normalization and text embedding in software, due to lack of hardware support [30]–[32]. Using this hardware-software partitioning methodology, we show that PRONTO can perform rapid hardware/software co-design and explore performance and power trade-offs.

We describe PRONTO and evaluate it using six datasets spanning vision, speech, and text classification. Our detailed analysis shows that resource utilization is independent of input SNN settings. Therefore, this metric can be directly *reported* from published works when comparing different SNN designs. However, accuracy, power, and throughput per watt vary significantly based on the spike encoding mechanism and time steps. Therefore, these performance metrics must be *individually evaluated* for each design for a given SNN setting. Furthermore, we show that PRONTO can be used to perform Pareto analysis for a given SNN model and a target dataset.

PRONTO is available at https://github.com/drexel-DISCO/PRONTO.git under the MIT license to allow academia and industry to access the framework without restriction.[1]

II. PRONTO: High-Level Overview

PRONTO does not introduce yet another SNN hardware or a Python toolchain to automatically synthesize hardware. Instead, PRONTO is a framework to benchmark current and future SNN designs. Its automated testbench-based simulation methodology can simplify the effort required to verify a new design, allowing rapid prototyping and deployment.

Figure 1 shows the building blocks of PRONTO. At a high level, PRONTO consists of a frontend, middleware, and backend. The frontend of PRONTO uses a Python code with torch dialect to specify SNN model architectures, learning rules, datasets, and SNN-specific input settings, which include (1) spike encoding mechanism and (2) time steps. The middleware, which is the core of PRONTO, consists of an SNN simulator such as snnTorch [25] ❶, workload synthesis and stimuli generation ❷, dataflow analysis for model partitioning ❸, and a testbench along with its interface to the DUT ❹. Finally, the backend consists of SNN hardware platforms (the DUT). We demonstrate PRONTO for three different backends – (1) a conventional SNN hardware, e.g., QUANTISENC [18], (2) a many-core SNN hardware with a shared interconnect, e.g., μBrain [2], and (3) an SNN hardware generated using a high-level synthesis (HLS) approach, e.g., SODA [21].

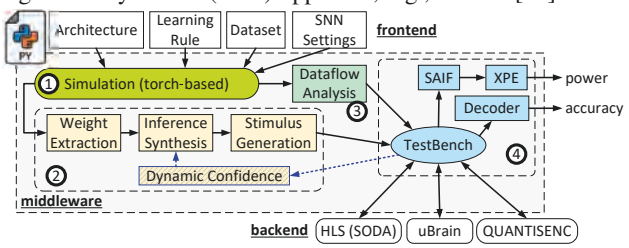


Fig. 1: High-level overview of PRONTO's design consisting of a frontend, middleware, and backend.

PRONTO supports the following configurations.

- SNN architecture: Currently, PRONTO supports multi-layer perceptron and convolution-based SNN models. Our future work will address other models such as liquid state machines [33], Reservoir-in-Reservoir [34] and Bayesian networks [35].
- Learning algorithms: PRONTO supports learning algorithms that can be implemented using the torch dialect. This includes spike-timing-dependent plasticity (STDP) [36], surrogate gradient descent [37], FORCE [38], full-FORCE [39], and spatial learning through time (SLTT) [40], among others.
- Dynamic confidence: PRONTO can be extended to implement mechanisms such as dynamic confidence [41], which terminates the inference based on the confidence of the hardware output decoded by the testbench.

III. Detailed Design of PRONTO

Without loss of generality, we describe PRONTO considering grayscale image input. A grayscale image is a two dimensional matrix of pixels with width W and height H as shown in Figure 2. A batch is formed by stacking B images ❶.

Each pixel is encoded into spike trains for a time duration T equal to the value of time steps ❷. Therefore, spike encoding converts a $(B \times W \times H)$ input matrix to a binary $(T \times B \times W \times H)$ matrix. We represent the image dimension $D = (W \times H)$.[2] In the analog domain, a spike is represented by a voltage

[1]PRONTO: A Framework for fast **PRO**totyping and benchmarking of S**N**N hardware using **TO**rch-based machine learning dialects.

[2]In this way, we extend the representation to RGB images with 3 dimensional input matrices encoding intensities for red, green, and blue channels.

waveform ❸. For digital computations, spike trains are strings of 0's and 1's, where a binary 1 indicates a spike ❹.

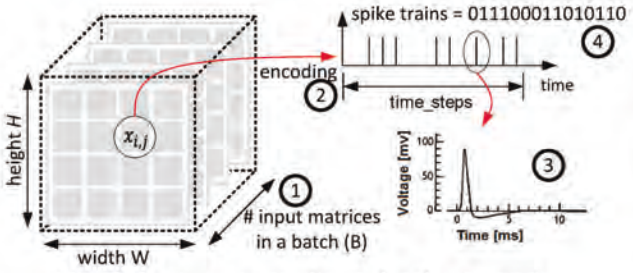


Fig. 2: Input representation and spike conversion.

PRONTO supports two spike encoding mechanisms.

1) Deterministic Rate Coding: It involves creating spike trains for an input with the number of spikes in time steps proportional to the analog value of the input.
2) Inter-Spike Interval (ISI) Coding: It involves encoding the analog value of an input as both the timing of spikes and the distance between spikes.

To illustrate the difference between these spike encoding mechanisms, Figure 3 shows the spike events generated for an image pixel with an intensity of 0.8 (chosen arbitrarily) on a scale of 0 to 1 using (1) rate coding with 50 time steps, (2) ISI coding with 50 time steps, and (3) rate coding with 100 time steps. These input settings result in different spike trains that lead to performance differences (see Section V).

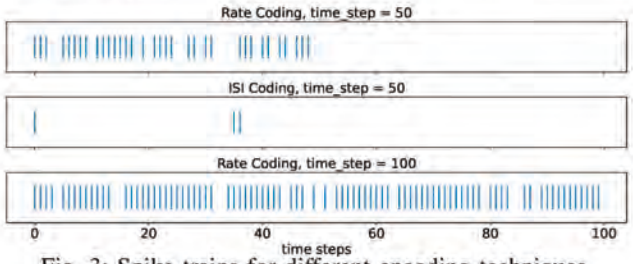


Fig. 3: Spike trains for different encoding techniques.

A. Synaptic Weight Extraction

PRONTO uses a data structure to store synaptic weights extracted from an SNN model using the torch command `model.parameters`. For each synaptic weight, we record the layer index, the pre- and post-synaptic neuron indices, and the raw floating point value of the weight. The weight value is converted into binary representation using the quantization and precision settings of the hardware. Layer and neuron indices are used as a memory address to program a binary weight value in a synaptic memory of the hardware.

Equation 1 shows the memory address and data for a synaptic weight connecting presynaptic neuron i in layer $(l-1)$ to postsynaptic neuron j in layer l. The weight value is encoded in $Qn.q$ representation with $(n+q)$ binary bits [18].

$$w^l_{i,j} = \begin{cases} \text{Memory address:} & <l><j><i> \\ \text{Memory data:} & |w^l_{i,j}|_{Qn.q} \end{cases} \quad (1)$$

B. Inference Synthesis and Spike Trains Generation

Model inference is performed on a data batch using the trained model parameters that are programmed to the hardware. Figure 4 illustrates the spike train generation mechanism from an input batch of images. Typically, the fanin of an SNN hardware is equal to the dimension D of each input image. Therefore, PRONTO generates separate spike trains for each pixel and stacks them together as shown in Figure 4.

To stream images consecutively to an SNN hardware for inference, PRONTO drives a string of 0's (i.e., no spikes) for a time duration equal to the *reset time*, after presenting the spike trains of an image to the hardware. This is done to reset the residual neuron membrane potential from an image to the resting potential before processing the next image.

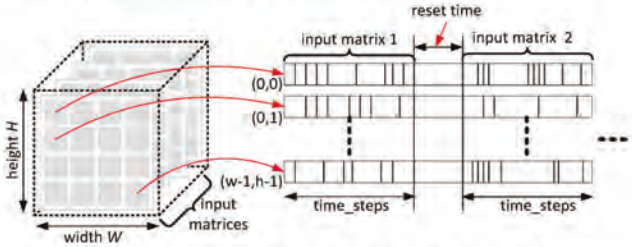


Fig. 4: Data batch to spike trains for hardware inference.

C. Testbench and Performance Estimation

Figure 5 shows the internal architecture of the proposed SystemVerilog testbench of PRONTO. Its DUT interface consists of a weight driver to drive synaptic weights and a stimulus driver to drive input spikes to the DUT. The design of the weight driver is specific to the DUT instantiated in the testbench. For open source QUANTISENC [18], the weight driver simply writes the address and weight on its output port connected to the DUT and asserts the write enable signal to enable writing of this weight to the corresponding synaptic memory address inside the DUT. The stimulus driver drives the input stimulus to the input port of the DUT.

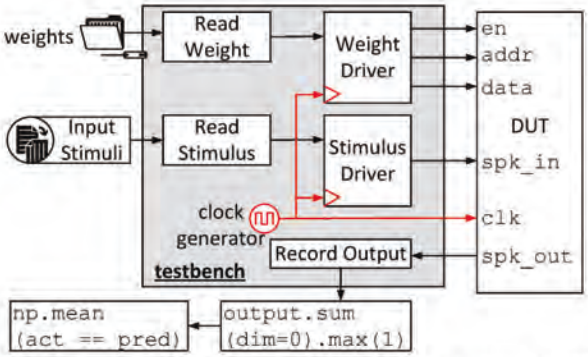


Fig. 5: The proposed testbench and its interface with the DUT.

PRONTO uses file exchanges between the Python-based implementation of inference synthesis and spike train generation (Section III-B) and the activation of DUT through the SystemVerilog testbench. At the end of the simulation, PRONTO estimates the following hardware performance.

1) Power consumption: The DUT power is estimated considering switching activities of internal nets in the DUT.
2) Real-time classification result: PRONTO facilitates observing the real-time classification of each input by counting the number of spikes for each output neuron in the time interval T (time steps) and identifying the winner (predicted class) as the neuron with the most spikes. This is performed in Python using the command `output.sum(dim=0).max(1)` and by reading the DUT output which is saved in a file.
3) Overall classification accuracy: PRONTO can also report overall classification accuracy by calculating the fraction of inputs where the hardware output label matches the target label. This can be performed in Python using the command `np.mean(act== pred)`.

Figure 6 shows the flowchart describing the working of the proposed testbench of PRONTO to verify an SNN design. Simulation settings such as runtime and power switches are setup at the start of the simulation ❶. A finite state machine (FSM) ❷ is designed to read synaptic weights (address and data) from a file (output from Section III-B) and drive them to the DUT using the weight driver of the DUT interface.

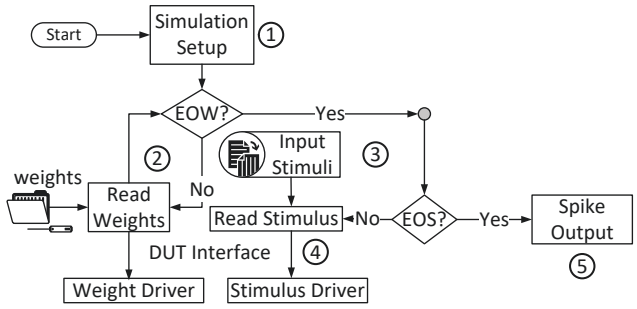


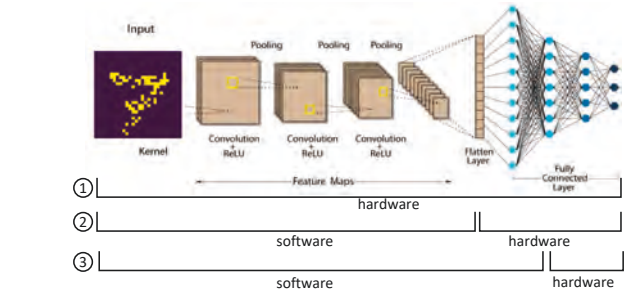
Fig. 6: Flowchart describing the proposed testbench.

Once synaptic weights are loaded into the synaptic memory of the DUT, a second FSM ❸ is initiated to read spike trains (stimulus) from a file (also an output from Section III-B) and drive them to the DUT using the interface ❹, one input per clock instance. The DUT output is monitored and recorded in a SystemVerilog array. This is then saved in a file at the end of the simulation to estimate performance ❺.

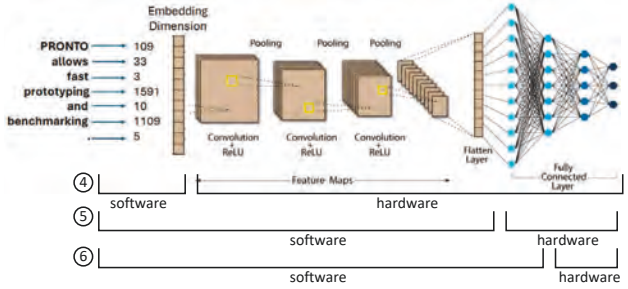
IV. Co-Design and Model Partitioning

SNN hardware comes in a wide variety of flavors in terms of operations that are supported on them. For example, the design in [16] implements fully connected layers, while that in [17] implements both convolution and fully connected layers. For PRONTO to work with different SNN architectures, we propose an automated model partitioning approach, as illustrated in Figure 7 for (a) image classification and (b) text classification. In Figure 7a, we illustrate three mapping scenarios: (1) mapping the entire model to hardware ❶, (2) mapping convolution layers to software and fully connected layers to hardware ❷, and (3) mapping all convolution layers and a subset of fully connected layers to software and the remaining layers to hardware ❸. The last scenario is for an SNN hardware that cannot map all fully connected layers due to limited on-chip neurons and synaptic memory.

In Figure 7b, we also illustrate three mapping scenarios: (1) text embedding in software with all other layers mapped to hardware ❹, (2) text embedding and feature extraction (convolution layers) in software and classification in hardware ❺, and (3) text embedding, feature extraction (convolution layers), and a few of the classification layers in software and the remaining classification layers in hardware ❻.



(a) software/hardware partitioning of image classification



(b) software/hardware partitioning of text classification

Fig. 7: Model partitioning between software and hardware for classifying (a) images and (b) texts.

Let $G = (V, E)$ be an SNN graph [32]. Let the first κ layers are mapped to software, i.e., $V_S = \{v_0, v_1, \cdots, v_{\kappa-1}\}$, and the remaining layers to hardware, i.e., $V_H = \{v_\kappa, v_{\kappa+1}, \cdots, v_{|V|-1}\}$, with $V = V_S \cup V_H$. The input stimulus for the hardware is the output of layer $v_{\kappa-1}$, while the synaptic weights programmed to the hardware are those belonging to layers $v_i \in V_H$. Algorithm 1 shows the pseudocode for co-design.

Algorithm 1: Partitioning SNN graph for co-design.

Input: SNN graph $G = (V, E)$
Output: Pareto points $\mathcal{P}$
1 $\mathcal{P} = \emptyset$; /* Initialize Pareto points. */
2 $V_H = V$ and $V_S = \emptyset$; /* Assign all layers to hardware. */
3 (sti, wts) = `partition`(V_S, V_H); /* Partition the model. */
4 (acc, area, lat, pwr) = `simulate`(sti, wts); /* Simulate and evaluate performance. */
5 $\mathcal{P}$.append ({acc, area, pwr, thr}) ; /* Insert Pareto point. */
6 **for** $v_i \in V$ **do** /* For each layer of SNN */
7 $V_S = V_S \cup \{v_i\}$ and $V_H = V \setminus V_S$; /* Assign v_i layer to software. */
8 (sti, wts) = `partition`(V_S, V_H); /* Partition the model. */
9 (acc, area, lat, pwr) = `simulate`(sti, wts); /* Simulate and evaluate performance. */
10 $\mathcal{P}$.append ({acc, area, lat, pwr}) ; /* Insert Pareto point. */
11 **end**
12 **return** $\mathcal{P}$

The algorithm starts by (1) defining an empty Pareto array (line 1), (2) assigning all layers to hardware (line 2), (3)

partitioning the model and generating stimulus, (4) simulating the model using the testbench (Sec. III), recording accuracy, area, power, and throughput (line 4), and adding the result to the Pareto array (line 5). Next, for each layer $v_i \in V$ (line 6), the algorithm inserts the layer into the software and the remaining layers into the hardware (line 7). The model is partitioned (line 8). The model is simulated (line 9) and a Pareto point is created and appended to the Pareto array (line 10). All Pareto points are returned at the end of the exploration (line 11). The Pareto points of $\mathcal{P}$ can be used to explore the trade-offs between accuracy, area, latency, and power.

V. Results and Discussions

We evaluate PRONTO using six datasets that involve image, text, and audio classification. These datasets are summarized in Table II. For each dataset, we report a reference SNN model. We use snnTorch [25] to train the model on a Lambda workstation, which has AMD Threadripper 3960X with 24 cores, 128 MB cache, 128 GB RAM, and 2 RTX3090 GPUs. We also report the baseline accuracy using snnTorch.

TABLE II. Evaluated datasets and corresponding model and accuracy.

	input/output	training/test	model	accuracy
MNIST [42]	input: (28×28) output: 10	training: 60,000 test: 10,000	MLP [18]	99.8%
Fashion MNIST [43]	input: (28×28) output: 10	training: 60,000 test: 10,000	CNN & MLP [44]	71.0%
CIFAR10 [45]	input: (32×32) output: 10	training: 50,000 test: 10,000	CNN [45]	90.2%
AG_NEWS [46]	dimension: 128 output: 4	training: 114,000 test: 7600	MLP [47]	72.0%
Speech CMD [48]	input: (1×8000) output: 35	training: 84,843 test: 11,005	CNN [48]	89.1%
Speech-to-Spike [49]	input: (28×28) output: 10	training: 60,000 test: 10,000	MLP [49]	87.5%

We use AMD's Vivado (2023.2 ML Edition) for all hardware simulations and Virtex Ultrascale VCU108 FPGA development board for prototyping. It consists of 537,600 Lookup Tables (LUTs), 1,075,200 Flipflops (FFs), and 1728 Block RAMs (BRAMs). We evaluate the following SNN designs.

- QUANTISENC [18]: This is an open-source SNN hardware implemented as a layer-based architecture. Each layer implements leaky integrate-and-fire neurons. SNN models are mapped using SONIC [50].
- SODA [21]: This is a torch-based framework to synthesize a given model using the Bambu HLS [51].
- μBrain [2]: A many-core design built using small $M \times N$ cores, configured to implement N neurons per core with a maximum of M pre-synaptic connections per neuron. SNN models are mapped using SpiNeMap [52].

A. Impact of SNN Parameters

Table III reports the hardware performance (inference accuracy, area, power, and throughput per watt) of QUANTISENC [18] for all evaluated datasets using rate and ISI coding with time steps of 50, 100, and 200. The design is synthesized and implemented using Vivado's default strategies. We observe the variation in performance metrics for different SNN settings, which motivates the need for a common framework to benchmark SNN hardware designs. We also provide the following **critical insights** for system designers.

1) For each dataset, the **resource utilization** remains the same across all SNN settings. Therefore, this metric can be compared between designs using published data.
2) The **inference accuracy** varies significantly for these input settings. In general, ISI coding leads to lower accuracy because it uses fewer spikes than rate coding. This is directly correlated with biology [53]. Time steps play a critical role in determining the inference accuracy. With smaller time steps (say 50), fewer spikes are generated, which results in lower accuracy because biological neurons cannot learn the output classes with fewer spike events representing each class.
3) The **power consumption** varies depending on the stage at which the power simulation is performed. We observe that the power reported after synthesis is usually higher than the power reported after implementation. Both of these power measures use default switching activities for internal nets in the design. PRONTO can also extract realistic switching activities from the input stimulus and use them to compute power. We see a variation in the power measure when using these methodologies.
4) The **performance per watt** varies with time steps and spike encoding techniques. This is because throughput varies with time steps within each technique, while power consumption varies across different techniques.

B. Design Benchmarking

Figure 8 illustrates the benchmarking of three (open source) designs in terms of throughput/watt using PRONTO for the six datasets using rate coding and a time step of 100. Results are normalized with respect to QUANTISENC.

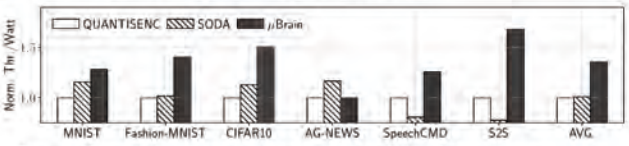


Fig. 8: Benchmarking designs in terms of throughput/watt.

Our objective here is ***not*** to prove one design to be better than the others. Instead, PRONTO offers a platform for system designers to critically evaluate their design against others in terms of key performance metrics such as accuracy, area, power, and throughput. For example, from Fig. 8 we see that the SODA design is efficient for simple models and datasets (MNIST, CIFAR10, and AG-NEWS). However, for larger models (SpeechCMD and S2S), QUANTISENC offers higher throughput and lower power, possibly due to their distributed memory and pipelined design. On the other hand, the many-core μBrain design offers scalability in terms of both throughput and power, resulting in higher throughput/watt.

C. Model Partitioning and Pareto Analysis

PRONTO can be used to explore the best architecture for a given dataset. Table IV reports the exploration in terms of inference accuracy and throughput/watt for the following three different configurations of the Fashion-MNIST dataset.

1) Trained in snnTorch as an MLP $(256 \times 256 \times 10)$ and inference in FPGA with a similar architecture.

TABLE III. Evaluated datasets and corresponding hardware performance. Spike frequency = $100KHz$.

	MNIST (LUTs=8%, FFs=1%, BRAM=4%)					**Fashion MNIST** (LUTs=15%, FFs=2%, BRAM=8%)					**CIFAR10** (LUTs=21%, FFs=2%, BRAM=11%)				
	Rate Coding			**ISI Coding**		**Rate Coding**			**ISI Coding**		**Rate Coding**			**ISI Coding**	
	50	**100**	**200**	**100**	**200**	**50**	**100**	**200**	**100**	**200**	**50**	**100**	**200**	**100**	**200**
Accuracy	96.0%	97.6%	99.2%	0	89.8%	46.8%	60.1%	73.4%	68.75%	69.0%	80.6%	81.1%	84.4%	60.7%	60.7%
Power (Post Synthesis)	0.128W	0.128W	0.128W	0.128W	0.128W	0.232W	0.232W	0.232W	0.232W	0.232W	0.390W	0.390W	0.390W	0.390W	0.390W
Power (Post Implementation)	0.105W	0.118W	0.145W	0.086W	0.086	0.189W	0.264W	0.183W	0.178W	0.178W	0.260W	0.260W	0.260W	0.260W	0.260W
Power (Using SAIF)	0.086W	0.086W	0.086W	0.057W	0.057W	0.153W	0.153W	0.154W	0.137W	0.137W	0.200W	0.200W	0.200W	0.189W	0.189W
Throughput/Watt (GOPS/W)	29.6	14.8	7.8	44.7	22.3	16.6	8.3	4.4	18.6	9.3	12.7	6.3	3.4	13.5	6.8
	AG_NEWS (LUTs=7%, FFs=1%, BRAM=4%)					**Speech CMD** (LUTs=9%, FFs=1%, BRAM=5%)					**S2S** (LUTs=28%, FFs=2%, BRAM=15%)				
	Rate Coding			**ISI Coding**		**Rate Coding**			**ISI Coding**		**Rate Coding**			**ISI Coding**	
	50	**100**	**200**	**100**	**200**	**50**	**100**	**200**	**100**	**200**	**50**	**100**	**200**	**100**	**200**
Accuracy	70.0%	70.8%	71.2%	55.8%	59.0%	80.6%	81.2%	87.9%	54.0%	54.5%	79.1%	80.2%	80.2%	68.5%	70.0%
Power (Post Synthesis)	0.119W	0.119W	0.119W	0.119W	0.119W	0.174W	0.174W	0.174W	0.174W	0.174W	0.500W	0.500W	0.500W	0.500W	0.500W
Power (Post Implementation)	0.081W	0.081W	0.081W	0.081W	0.081W	0.122W	0.122W	0.122W	0.122W	0.122W	0.332W	0.332W	0.332W	0.332W	0.332W
Power (Using SAIF)	0.064W	0.064W	0.064W	0.022W	0.022W	0.077W	0.077W	0.077W	0.050W	0.050W	0.287W	0.287W	0.287W	0.200W	0.200W
Throughput/Watt (GOPS/W)	39.8	20.0	10.5	116.0	58.0	33.1	16.6	8.8	51.0	25.5	8.8	4.4	2.2	12.7	6.3

2) Trained in snnTorch as CNN architecture and using FPGA for only the dense layers while executing the convolution layers in software.
3) Trained in snnTorch as CNN architecture and executing both convolution and dense layers on FPGA.

We observe that the accuracy is higher when using a CNN (columns 3 & 4 vs. column 1). Throughput/watt is the highest when the entire model is executed on the hardware (column 4). PRONTO's frontend and backend interfaces allow to prototype and benchmark different hardware and model configurations.

TABLE IV. Model partitioning for Fashion-MNIST.

	snnTorch training: MLP inference: FPGA	**snnTorch training: CNN inference: FPGA & software**	**snnTorch training: CNN inference: FPGA**
Accuracy	60.1%	89.7%	89.7%
Throughput/watt	8.3	3.8	10.2

Figure 9 shows the Pareto exploration for two datasets – SpeechCMD and CIFAR10. For these explorations, we plot the normalized performance (throughput/watt) on the y-axis and the reciprocal of the normalized utilization of FPGA resources on the x-axis. We make the following observations.

The highest performance is obtained when an entire model is implemented on FPGA. However, this configuration also results in the highest FPGA utilization (area). On the other hand, when the entire model (except the last classification layer) is executed in software, we achieve the lowest performance. However, the utilization of FPGA resources is the least. Designers can use PRONTO to retain Pareto-optimal design points (shown in the figure) and find a sweet spot in terms of FPGA utilization and performance.

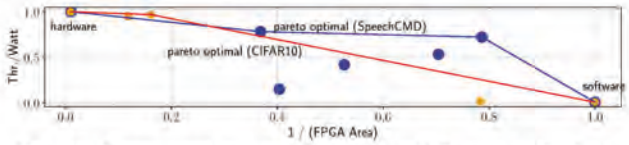


Fig. 9: Pareto exploration for SpeechCMD and CIFAR10.

D. Algorithmic Complexities

The complexity of Algorithm 1 is guided by the for loop of lines 5-11 with unit operations. The complexity is $\mathcal{O}(|V|)$.

VI. Conclusions

We propose PRONTO, a verification framework that can be used for rapid prototyping and benchmarking of spiking neural network (SNN) hardware designs. PRONTO consists of a frontend to interface with a torch-based Python code for model specification and training. Its backend can interface with several SNN hardware platforms, including those generated using high-level synthesis. The core of PRONTO is its middleware, which consists of a SystemVerilog testbench to extract synaptic weights and input stimuli from a torch code. Synaptic weights are programmed into hardware memory using a weight driver, while input stimuli are driven to the hardware using a stumulus driver. These drivers can be easily customized for the hardware design under test. PRONTO's dataflow-based model partitioning approach can be used to perform hardware-software co-design to explore different design trade-offs. Using six datasets spanning vision, speech, and text classification, we thoroughly evaluate the capabilities of PRONTO in evaluating different performance metrics. PRONTO's Python interface and SystemVerilog testbench code is available at https://github.com/drexel-DISCO/PRONTO.git.

Acknowledgments

This work is supported by Accenture LLP, the DOE DE-SC0022014, and NSF OAC-2209745 & CCF-1942697.

References

[1] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.
[2] M. L. Varshika, A. Balaji, F. Corradi, A. Das *et al.*, "Design of many-core big little μbrains for energy-efficient embedded neuromorphic computing," in *Design Automation and Test in Europe (DATE)*, 2022.
[3] O. Richter, C. Wu, A. M. Whatley, G. Köstinger, C. Nielsen, N. Qiao, and G. Indiveri, "DYNAP-SE2: A scalable multi-core dynamic neuromorphic asynchronous spiking neural network processor," *Neuromorphic Computing and Engineering*, vol. 4, no. 1, p. 014003, 2024.
[4] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.
[5] D. S. Modha, F. Akopyan, A. Andreopoulos, R. Appuswamy, J. V. Arthur, A. S. Cassidy, P. Datta, M. V. DeBole *et al.*, "Neural inference at the frontier of energy, space, and time," *Science*, vol. 382, 2023.

[6] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary *et al.*, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014.

[7] W. Wan, R. Kubendran, C. Schaefer, S. B. Eryilmaz, W. Zhang, D. Wu, S. Deiss, P. Raina *et al.*, "A compute-in-memory chip based on resistive random-access memory," *Nature*, vol. 608, no. 7923, pp. 504–512, 2022.

[8] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, 2014.

[9] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[10] W. Guo, H. E. Yantır, M. E. Fouda, A. M. Eltawil, and K. N. Salama, "Toward the optimal design and FPGA implementation of spiking neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 8, pp. 3988–4002, 2021.

[11] J. Li, G. Shen, D. Zhao, Q. Zhang, and Y. Zeng, "FireFly: A high-throughput hardware accelerator for spiking neural networks with efficient DSP and memory optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[12] M. T. L. Aung, D. Gerlinghoff, C. Qu, L. Yang, T. Huang, R. S. M. Goh *et al.*, "DeepFire2: A convolutional spiking neural network accelerator on FPGAs," *IEEE Transactions on Computers*, no. 99, pp. 1–11, 2023.

[13] H. Liu, Y. Chen, Z. Zeng, M. Zhang, and H. Qu, "A low power and low latency FPGA-based spiking neural network accelerator," in *International Joint Conference on Neural Networks (IJCNN)*, 2023.

[14] A. Carpegna, A. Savino, and S. Di Carlo, "Spiker+: A framework for the generation of efficient Spiking Neural Networks FPGA accelerators for inference at the edge," *arXiv preprint arXiv:2401.01141*, 2024.

[15] S. Panchapakesan, Z. Fang, and J. Li, "SyncNN: Evaluating and accelerating spiking neural networks on FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, 2022.

[16] A. M. Abdelsalam, F. Boulet *et al.*, "An efficient FPGA-based overlay inference architecture for fully connected DNNs," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2018.

[17] Z. He, C. Shi, T. Wang, Y. Wang, M. Tian, X. Zhou, P. Li, L. Liu, N. Wu, and G. Luo, "A low-cost FPGA implementation of spiking extreme learning machine with on-chip reward-modulated STDP learning," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, 2021.

[18] S. Matinizadeh, A. Mohammadhassani, N. Pacik-Nelson, I. Polykretis, A. Mishra, J. Shackleford, N. Kandasamy, E. Gallo, and A. Das, "A fully-configurable digital spiking neuromorphic hardware design with variable quantization and mixed precision," in *International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2024.

[19] D. Neil and S.-C. Liu, "Minitaur, an event-driven FPGA-based spiking network accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2621–2628, 2014.

[20] J. Han, Z. Li, W. Zheng, and Y. Zhang, "Hardware implementation of spiking neural networks on FPGA," *Tsinghua Science and Technology*, vol. 25, no. 4, pp. 479–486, 2020.

[21] S. Curzel, N. B. Agostini, S. Song, I. Dagli, A. Limaye, C. Tan, M. Minutoli, V. G. Castellana *et al.*, "Automated generation of integrated digital and spiking neuromorphic machine learning accelerators," in *International Conference on Computer-Aided Design (ICCAD)*, 2021.

[22] H. Fang, Z. Mei *et al.*, "Encoding, model, and architecture: Systematic optimization for spiking neural network in FPGAs," in *International Conference on Computer-Aided Design (ICCAD)*, 2020.

[23] I. A. Svoboda, T. Adegbija *et al.*, "Design space exploration of sparsity-aware application-specific spiking neural network accelerators," *arXiv preprint arXiv:2310.16745*, 2023.

[24] N. B. Agostini, S. Curzel, J. J. Zhang, A. Limaye, C. Tan, V. Amatya, M. Minutoli, V. G. Castellana, J. Manzano, D. Brooks *et al.*, "Bridging Python to silicon: The SODA toolchain," *IEEE Micro*, vol. 42, 2022.

[25] J. K. Eshraghian, M. Ward, E. O. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun *et al.*, "Training spiking neural networks using lessons from deep learning," *Proceedings of the IEEE*, 2023.

[26] W. Fang, Y. Chen, J. Ding, Z. Yu, T. Masquelier *et al.*, "SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence," *Science Advances*, vol. 9, no. 40, p. eadi1480, 2023.

[27] L. Niedermeier, K. Chen, J. Xing, A. Das, J. Kopsick, E. Scott, N. Sutton *et al.*, "CARLsim 6: an open source library for large-scale, biologically detailed spiking neural network simulation," in *International Joint Conference on Neural Networks (IJCNN)*, 2022.

[28] P. Qu, H. Lin, M. Pang, X. Liu, W. Zheng, and Y. Zhang, "ENLARGE: An efficient SNN simulation framework on GPU clusters," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[29] D. F. Goodman and R. Brette, "The brian simulator," *Frontiers in Neuroscience*, vol. 3, p. 643, 2009.

[30] A. Das, "Real-time scheduling of machine learning operations on heterogeneous neuromorphic SoC," in *International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2022.

[31] BrainChip. (2022) Akida Neuromorphic System- on-Chip.

[32] A. Das, "A design flow for scheduling spiking deep convolutional neural networks on heterogeneous neuromorphic system-on-chip," *ACM Transactions on Embedded Computing Systems (TECS)*, 2023.

[33] A. Das, P. Pradhapan, W. Groenendaal, P. Adiraju, R. T. Rajan, F. Catthoor, S. Schaafsma, J. L. Krichmar, N. Dutt, and C. Van Hoof, "Unsupervised heart-rate estimation in wearables with liquid states and a probabilistic readout," *Neural Networks*, vol. 99, pp. 134–147, 2018.

[34] A. Paul, N. Kandasamy *et al.*, "Data driven learning of aperiodic nonlinear dynamic systems using spike based reservoirs-in-reservoir," in *International Joint Conference on Neural Networks (IJCNN)*, 2024.

[35] N. Skatchkovsky *et al.*, "Bayesian continual learning via spiking neural networks," *Frontiers in Computational Neuroscience*, vol. 16, 2022.

[36] N. Caporale and Y. Dan, "Spike timing–dependent plasticity: a hebbian learning rule," *Annu. Rev. Neurosci.*, vol. 31, pp. 25–46, 2008.

[37] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks," *IEEE Signal Processing Magazine*, vol. 36, no. 6, pp. 51–63, 2019.

[38] A. Paul, S. Wagner, and A. Das, "Learning in feedback-driven recurrent spiking neural networks using full-force training," in *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2022, pp. 1–10.

[39] A. Paul and A. Das, "Learning in recurrent spiking neural networks with sparse full-FORCE training," in *International Conference on Artificial Neural Networks (ICANN)*, 2024.

[40] Q. Meng, M. Xiao, S. Yan, Y. Wang, Z. Lin, and Z.-Q. Luo, "Towards memory-and time-efficient backpropagation for training spiking neural networks," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 6166–6176.

[41] C. Li, E. G. Jones, and S. Furber, "Unleashing the potential of spiking neural networks with dynamic confidence," in *International Conference on Computer Vision (ICCV)*, 2023.

[42] M. Fatahi, M. Ahmadi, M. Shahsavari, A. Ahmadi, and P. Devienne, "evt_MNIST: A spike based version of traditional MNIST," *arXiv preprint arXiv:1604.06751*, 2016.

[43] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[44] X. Cheng, Y. Hao, J. Xu, and B. Xu, "LISNN: Improving spiking neural networks with lateral interactions for robust object recognition." in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.

[45] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, and L. Shi, "Direct training for spiking neural networks: Faster, larger, better," in *AAAI Conference on Artificial Intelligence*, 2019.

[46] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[47] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," *Conference on Neural Information Processing Systems (NeurIPS)*, 2015.

[48] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[49] K. M. Stewart, T. Shea, N. Pacik-Nelson, E. Gallo *et al.*, "Speech2spikes: Efficient audio encoding pipeline for real-time neuromorphic systems," in *Neuro-Inspired Computational Elements Conference (NICE)*, 2023.

[50] S. Matinizadeh, A. Mohammadhassani, N. Pacik-Nelson, I. Polykretis *et al.*, "Neuromorphic computing for the masses," in *International Conference on Neuromorphic Systems (ICONS)*, 2024.

[51] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo, "Bambu: An open-source research framework for the high-level synthesis of complex applications," in *Design Automation Conference (DAC)*, 2021.

[52] A. Balaji, A. Das, Y. Wu, K. Huynh, F. G. Dell'Anna *et al.*, "Mapping spiking neural networks to neuromorphic hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, 2019.

[53] S. A. Prescott and T. J. Sejnowski, "Spike-rate coding and spike-time coding are affected oppositely by different adaptation mechanisms," *Journal of Neuroscience*, vol. 28, no. 50, pp. 13 649–13 661, 2008.