Neuromorphic Computing for the Masses

Shadi Matinizadeh†, Arghavan Mohammadhassani†, Noah Pacik-Nelson‡, Ioannis Polykretis‡,
Krupa Tishbi†, Suman Kumar†, M. L. Varshika†, Abhishek Kumar Mishra†,
Nagarajan Kandasamy†, James Shackleford†, Eric Gallo‡, Anup Das†
†*Drexel University, USA* and ‡*Accenture Labs, USA*
Correspondence Email: {sm4884,anup.das}@drexel.edu

2024 International Conference on Neuromorphic Systems (ICONS) | 979-8-3503-6865-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICONS62911.2024.00014

***Abstract*—Neuromorphic computing describes the hardware implementation of biological neurons and synapses of spiking neural networks (SNNs). We introduce SONIC, a software-defined hardware design methodology to make neuromorphic computing accessible to the general computing community. SONIC is designed using three main components. First, SONIC integrates QUANTISENC, a parameterized SNN hardware written in Verilog HDL. This design consists of leaky integrate-and-fire (LIF) neurons and current-based (CUBA) synapses that are configured using Python to implement different SNN topologies. Second, SONIC integrates PRONTO, a SystemVerilog testbench that can be automatically synthesized using Python to benchmark this hardware against existing designs for different learning tasks and datasets. Finally, SONIC introduces a system software to interface with QUANTISENC, making it programmable and easy to prototype on FPGA and ASIC, starting from SNN specifications written in Python. Overall, SONIC offers a complete framework for simultaneously defining and training SNN models in software, generating its Verilog design, deploying model parameters to hardware, performing inference on live data, evaluating hardware performance, and visualizing inference results. We evaluate SONIC using three spiking datasets. Our results show the scalability and superior performance of SONIC in terms of area, throughput, and power compared to existing designs. SONIC is available as an open-source framework for the neuromorphic community to use without restriction.**

***Index Terms*—spiking neural networks (SNNs), design methodology, neuromorphic computing, PyTorch.**

I. Introduction

Neuromorphic computing describes the hardware implementation of biological neurons and synapses [1]. They are efficient in implementing Spiking Neural Networks (SNNs), which are emerging computing models based on the third generation of neural networks [2]. Over the years, architects and designers have created neuromorphic computing systems using analog and digital components, offering lower power consumption, reduced latency, and several other benefits seen in biological systems [3]–[7]. Although analog designs improve energy efficiency by taking advantage of electronic and physical laws in implementing neurons and synapses, we focus on digital designs because they are faster to implement on FPGA and ASIC due to the maturity of their design flows, while benefiting from technology scaling [8].

Most digital neuromorphic designs are still in the initial design exploration phase within a select few organizations around the world. Recent efforts to commercialize neuromorphic designs include Brainchip's Akida [9] and SynSense DYNAP [10] development boards. In addition to their high cost of ownership, these platforms are not easily accessible to the general computing community. We introduce SONIC (SOftware-defined NeuromorphIC), an open-source software-defined hardware design methodology to make neuromorphic computing accessible to all. The key idea is to enable the user to define and train an SNN model using a high-level language such as Python, generate the corresponding register transfer level (RTL) description of the hardware, and create the hardware-software interface to improve programmability when the RTL is prototyped on FPGA and ASIC. Simultaneously, its open-source design methodology can be used to teach both software and hardware of neuromorphic systems towards building a robust workforce specializing in AI systems.

SONIC consists of three key components. First, SONIC integrates QUANTISENC [7], a layer-based neuromorphic hardware design written in Verilog hardware description language, integrating leaky integrate-and-fire (LIF) neurons with current-based (CUBA) synapses. It allows configuring the number of layers, neurons per layer, and layer-to-layer connectivity using Verilog parameters that can be programmed via software. Second, SONIC integrates PRONTO [11], a SystemVerilog testbench to verify QUANTISENC and benchmark it against existing neuromorphic designs for different learning tasks and datasets. Finally, SONIC introduces a hardware-software interface to improve programmability of QUANTISENC when prototyped on FPGA and ASIC.

The following are our key **contributions**.

1) **Configurability:** SONIC is designed to make neuromorphic hardware configurable via software. Given its flexibility, expressiveness, GPU acceleration of training algorithms, and a large user base, the front-end of SONIC is designed to directly interface with simulators such as snnTorch [12] and SpikingJelly [13], which use the torch dialect to specify and train SNNs. We design an application programming interface (API) to generate the Verilog hardware description language (HDL) of QUANTISENC for a target SNN model specified in Python using torch dialects. This API allows to configure layer architectures, neurons per layer, and connection between layers of QUANTISENC. It also allows to set separate quantization and precision policies for synaptic weights and state variables of neurons.
2) **Verification Architecture:** We engineer a modular SystemVerilog testbench to verify QUANTISENC for different learning tasks and datasets. We design an API

979-8-3503-6865-9/24/$31.00 ©2024 IEEE
DOI 10.1109/ICONS62911.2024.00014

to extract input stimuli for the hardware by interfacing with Python code. This is then driven to QUANTISENC using a design-specific interface designed inside the testbench. The interface is also used to record the output and verify design functionalities. Finally, the testbench compares the performance of the hardware with respect to its software implementation.

3) **Hardware-Software Interface:** We design the software stacks for QUANTISENC neuromorphic hardware consisting of the application software and the system software. The hardware-software interface is used to load synaptic weights in memory, drive input, and visualize hardware output. We architect this interface for both FPGA and ASIC based designs.
4) **Open-Source Release:** SONIC is developed to foster research and education in neuromorphic computing. SONIC is available at https://github.com/drexel-DISCO/SONIC.git under the MIT license to allow academia and industry to access the framework without restriction. We believe that our open-source methodology will allow this framework to undergo public peer review, which will make it comprehensive over time through community contributions and feedback.

We evaluate SONIC using Spiking MNIST, DVS Gesture, and Spiking Heidelberg Digit (SHD) datasets [14]. Our results show scalability and superior performance in terms of area, throughput, and power. Finally, we demonstrate an educational curriculum that integrates SONIC to teach the co-design of applications and hardware for neuromorphic computing.

Overall, our philosophy behind SONIC is to make neuromorphic hardware programmable, efficient, and easy-to-use for the broader scientific community.

II. High-Level Overview of SONIC

Figure 1a shows a high-level overview of SONIC.

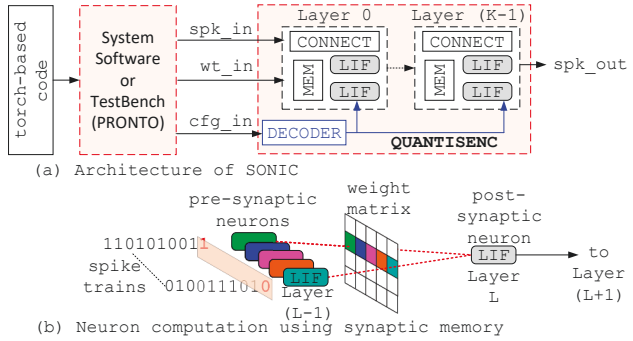

Fig. 1: (a) High-level overview of SONIC integrating QUANTISENC (right), its system software and verification interface (middle), and a torch-based framework to specify and train SNN models (left). (b) Computation using spike trains for synaptic weights programmed inside QUANTISENC memory.

SONIC integrates the following.

1) **QUANTISENC [7],** a layer-based parameterized neuromorphic design written in Verilog HDL.
2) **PRONTO [11],** a modular SystemVerilog testbench to verify and benchmark QUANTISENC.
3) **Software stacks**, including the application software and system software to interface with QUANTISENC.

In our prior work [7], we introduce details of QUANTISENC. Here, we introduce key architectural aspects necessary to understand SONIC.

QUANTISENC consists of layers of LIF neurons. The number of layers, neurons per layer, the connection between layers, and the quantization and precision policy of each layer are Verilog parameters that can be configured via software to implement an SNN model. LIF parameters are stored in the control registers. They can be configured at run-time via system software to explore power and performance trade-offs.

Synaptic memory of QUANTISENC is distributed among its layers such that all pre-synaptic weights are stored in the layer where the post-synaptic neurons are implemented. In this way, once a neuron in a layer fires a spike, all its post-synaptic neurons can start processing the spike at the same time. SONIC exploits this massively parallel processing architecture with distributed memory organization to enable pipelining in processing layer computations through intelligent task scheduling [15], which increases throughput.

We illustrate a fully connected layer in Figure 1b, where a neuron in layer L (post-synaptic neuron) receives input from all neurons in layer $(L-1)$ (pre-synaptic neurons). The output spikes of this neuron are routed to all neurons in the layer $(L+1)$. At a finer granularity, synaptic memory is an $M \times N$ weight matrix. This weight matrix corresponds to a design configuration of N neurons in the layer with M pre-synaptic connections per neuron. Spike trains from pre-synaptic connections are weighed using synaptic weights to generate output spike trains as illustrated in the figure. The access granularity of a layer's synaptic memory is that of a synaptic weight between a pair of pre-synaptic and post-synaptic neurons. Therefore, each weight can be addressed and programmed individually.

The I/O interface to QUANTISENC is as follows.

- `wt_in`: This interface is used to program synaptic memory by specifying weight addresses and data.
- `cfg_in`: This interface is used to configure the neuron parameters and their dynamics. We implement control registers inside the `decoder` module to store these parameters.
- `spk_in/out`: This interface is used for data input and output of SONIC.

III. QUANTISENC Hardware Design

A. Neuron Design

QUANTISENC uses leaky integrated-and-fire (LIF) neurons defined by the first-order ordinary differential equation (ODE)

$$\tau \frac{dU(t)}{dt} = -U(t) + R \cdot I_{in}(t) \quad (1)$$

where $U(t)$ is the membrane potential, $\tau = R \cdot C$ is the neuron's time constant defined using its membrane capacitance C and

resistance R, and $I_{in}(t)$ is the input current. With the forward Euler method, Equation 1 can be solved as

$$U(t+\Delta t) = U(t) + \frac{\Delta t}{\tau}\Big(-U(t) + R \cdot I_{in}(t)\Big) \tag{2}$$

Equation 2 can be rewritten using neuron parameters as

$$U(t+\Delta t) = U(t) - \texttt{decay\_rate} \cdot U(t) + \texttt{growth\_rate} \cdot I_{in}(t), \text{ where} \tag{3}$$

$$\texttt{decay\_rate} = \frac{\Delta t}{\tau} = \frac{\Delta t}{R \cdot C} \text{ and } \texttt{growth\_rate} = R \cdot \frac{\Delta t}{\tau} = \frac{\Delta t}{C} \tag{4}$$

Decay and growth rates are stored in control registers. The input current $I_j(t)$ to the neuron j is calculated as the weighted sum of input spikes from all its pre-synaptic connections for current-based synapse (CUBA) as

$$I_j(t) = \sum_i x_{ij} \cdot w_{ij}, \tag{5}$$

where x_{ij} is the spike from neuron i to neuron j and w_{ij} is the synaptic weight between these neurons. A spike is produced when membrane potential $U(t)$ exceeds the threshold voltage V_{th}, which is stored in the control register. After producing a spike, the following two mechanisms are triggered.

1) Reset Mechanism: It controls the recovery phase of the membrane potential using the following configurations.

$$U(t) = \begin{cases} V_{reset} & \textbf{Reset-to-Constant} \\ 0 & \textbf{Reset-to-Zero} \\ U(t) - V_{th} & \textbf{Reset-by-Subtraction} \\ U(t) - \texttt{decay\_rate} * U(t) & \textbf{Default} \end{cases} \tag{6}$$

2) Refractory Mechanism: It controls the maximum firing frequency of a neuron.

$$f_{\text{Max}} \leq \frac{1}{\texttt{refractory\_period}} \tag{7}$$

Reset mechanism and refractory period are also stored in the control registers.

B. Synapse Design

QUANTISENC supports different architectures to connect its layers. These are defined using the following parameters.

1) Network Topology: QUANTISENC can be configured to implement all-to-all, one-to-one, and Gaussian connection between layers.
2) Synaptic Polarity: Synaptic weights in QUANTISENC can be positive or negative, implementing excitatory and inhibitory effect of synaptic activation, respectively.

These synaptic configurations are defined as Verilog parameters, which can be programmed via software.

C. Variable Quantization and Mixed Precision

QUANTISENC uses fixed-point representation `Qn.q` (integer bits = n and precision bits = q) with variable quantization and mixed decimal precision for synaptic weights, activation, and neuron membrane potential. These settings are provided as Verilog parameters and configured via software.

D. Configuration Summary

Table I summarizes different configurations of QUANTISENC and their corresponding enabler in SONIC. We divide these into static and dynamic configurations. Static configurations are defined as Verilog parameters. We develop an API to configure them for the SNN model defined in the Python code. The dynamic configurations are stored in control registers. We use the proposed system software to program these registers using the QUANTISENC's IO interface.

Configuration	Parameters	Implementation	SONIC Enabler
static	number of layers number of neurons per layer synaptic connections quantization and precision	Verilog parameter	Python API
dynamic	growth rate decay rate threshold voltage reset mechanism refractory period	control register	system software

TABLE I. Configuring QUANTISENC using SONIC.

IV. Hardware-Software Interface

Figure 2a shows the hardware-software interface of SONIC designed for an FPGA-based computing system.

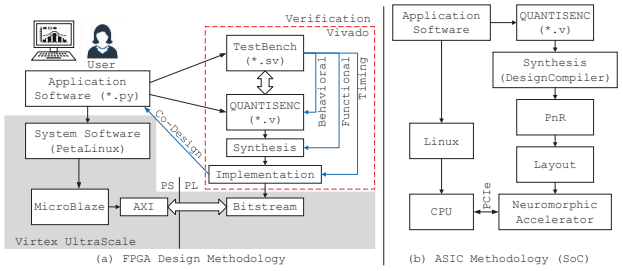

Fig. 2: Hardware-software interface of SONIC.

The Python code of a user forms the application software. This code is used to specify an SNN model, a target dataset, and the training algorithm. Using the torch dialect, users can specify a wide range of SNN models, such as convolution-based [16], Hopfield-based [17], and reservoir-based [18]. Similarly, users can also specify a wide range of learning algorithms [19]–[23]. Without loss of generality, we describe SONIC for the FPGA development boards and the hardware simulation infrastructure of AMD (formerly Xilinx).

The application software is executed on the processing system (PS) of an FPGA board. It interfaces with the QUANTISENC hardware implemented in the programmable logic (PL). We evaluate SONIC for two state-of-the-art FPGA platforms. For AMD's Virtex Ultrascale evaluation board, the PS is composed of the MicroBlaze soft core with the PetaLinux OS. For AMD's Zynq MPSoC products, the MicroBlaze soft core can be replaced with an ARM Cortex core.

The following is the proposed **design flow** of SONIC.

1) We develop a Python API to interface with the SNN model code implemented in the application software

to extract the static design configurations. These settings are then programmed in the Verilog parameter file of QUANTISENC. Once these configurations are programmed, the QUANTISENC hardware becomes equivalent to the SNN model of the application software.

2) We configure PRONTO to create the verification setup for QUANTISENC. This setup includes a modular SystemVerilog testbench to drive stimuli to QUANTISENC and capture the output. PRONTO facilitates (1) behavioral simulation (using the Verilog HDL), (2) functional simulation (using synthesized netlist), and (3) timing simulation (using implemented netlist) in AMD's Vivado tool suite. These simulations can be used to record the switching activities of the internal design nets, which can then be used to estimate the power consumption.
3) The MicroBlaze/ARM software stack including the PetaLinux form the system software. It interfaces with the PL block to program synaptic weights, drive input, and visualize hardware results at run-time using the high-speed AMBA AXI interconnect.

This power estimation framework can be integrated inside a loop to do power/energy-aware co-design.

Figure 2b shows the design flow of SONIC for the ASIC design. This flow consists of a front-end and a back-end design step. The front-end includes synthesis and static timing analysis. The back-end design step includes placement, routing, and layout. After fabrication and testing, a neuromorphic accelerator is connected to the host CPU inside a system-on-chip (SoC) using peripheral interconnect express (PCIe). Brainchip's Akida is an example of a neuromorphic system-on-chip (NSoC) [9]. The hardware-software interface consists of application software and system software, which includes OS, compiler, and task scheduler [24]–[29].

A. Enabling Pipelining via Operation Scheduling

The synaptic memory of QUANTISENC is distributed among its hardware layers such that all post-synaptic neurons in a layer can simultaneously process spikes from their pre-synaptic neurons implemented in its preceding layer. In terms of end-to-end data processing, QUANTISENC is a dataflow architecture, where each layer starts processing spikes when its preceding layer finishes generating spikes.

SONIC takes advantage of the distributed computing and memory architecture of QUANTISENC. As the layers of QUANTISENC do not share resources (neuron logic or synaptic memory), SONIC can schedule these layers to operate simultaneously on different data streams, essentially creating a hardware pipeline using these layers.

We illustrate the hardware pipelining concept in Figure 3. To initialize, SONIC loads the synaptic weights and configuration registers in the QUANTISENC hardware using its IO interface. Subsequently, SONIC schedules processing input data streams in a streaming fashion as illustrated in the figure. At any given time (after the pipeline is full), we see that the layers are all operating in parallel on different data streams. The system software schedules one stream after another with a latency equal to the sum of the processing time of a layer (d) and the waiting time (s). This waiting time ensures that the membrane potential of a neuron resets to its resting potential before it begins processing spike trains from the next input stream. In steady state, the maximum throughput obtained using this pipelined parallelism is $1/(d+s)$.

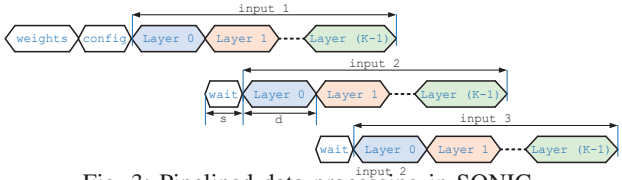

Fig. 3: Pipelined data processing in SONIC.

V. Education in AI Hardware

Figure 4 describes a curriculum that we developed at Drexel University, which integrates SONIC into existing courses to teach neuromorphic computing. This curriculum is structured into undergraduate and graduate content. At the undergraduate level, students are introduced to machine learning (ML) engineering based on the fundamentals of probability and data analytics. Student projects involve ML training and inference using the Python-based PyTorch framework. The Introduction to VLSI course introduces HDL coding using the Verilog HDL language with digital logic as background. Student projects include designing adders and multipliers in Verilog and verifying them using testbenches. On the analog front, courses such as electronic devices introduce students to the basics of semiconductors. Thereafter, the modern transistor design course introduces CMOS technology.

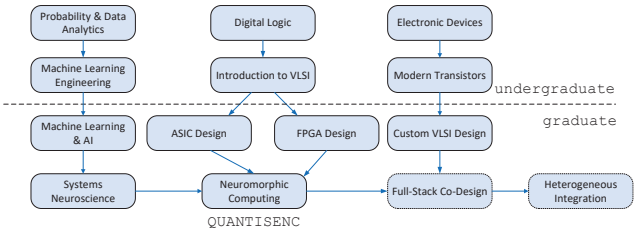

Fig. 4: Educational curriculum integrating SONIC.

At the graduate level, the Machine Learning and AI course covers the mathematical foundation of AI and also introduces different deep learning models. The system neuroscience course is an interdisciplinary course that introduces the neuronal and circuit basis underlying sensory processing and perception. This course also introduces SNNs with simple neuron and synapse models.

On the hardware front, the ASIC and FPGA design courses form the prerequisite to the neuromorphic computing course. For the ASIC design course, we plan to use QUANTISENC as a design reference to introduce the design and automation of digital CMOS Application Specific Integrated Circuit (ASIC) systems. The physical design flow of ASIC will introduce logic synthesis, floorplanning, placement, clock tree synthesis, routing, and verification of QUANTISENC. These back-end physical design flow steps will also be covered through hands-on practice using industrial VLSI CAD tools. The project-based digital FPGA design course will teach students how to

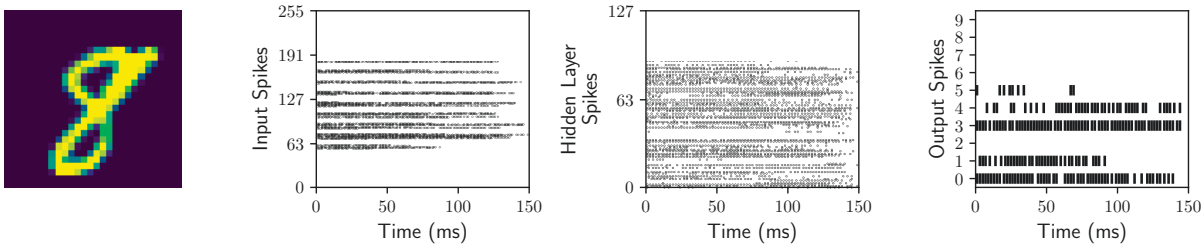

Fig. 5: A classification example with the spiking MNIST dataset using the handwritten digit 8.

implement SONIC on the AMD FPGA. Students will learn to perform simulations using the testbench.

The neuromorphic computing course will utilize all key concepts in prerequisite courses to introduce hardware-software co-design in the area of AI. The course is structured into three sections. In the first section, the course will introduce key concepts in SNNs, including different types of neuron and synapse, and learning approaches using SNNs, e.g., supervised, unsupervised, and reinforcement learning. The projects in this section will include developing time series and image-based applications using SNNs. In the second section, the course will introduce the QUANTISENC and its implementation on FPGA. The projects will include extending QUANTISENC to support different neuron models, exploring the trade-off in hardware and software performance. The third section is planned to be project-based, where students will learn the implementation of application-specific neuromorphic systems. Students will use SONIC to perform full-system emulation for the selected applications.

The neuromorphic computing course will form the prerequisite for a full stack co-design-oriented course that will integrate all compute stacks into the curriculum, teaching students to explore design points that span these stacks. The proposed course will facilitate both a top-down understanding, where new(er) applications can create the need for architectural features/assists, which in turn can drive innovations in circuits leveraging and optimizing newer materials/devices, and a bottom-up understanding, where novel material/device features and challenges can open up new architectural features that can enable/enhance new(er) applications.

We believe that the proposed educational curriculum will address the specific need to build a robust semiconductor workforce that specializes in AI systems.

VI. Results

We use Spiking MNIST, DVS Gesture, and Spiking Heidelberg Digit (SHD) [14] datasets to evaluate SONIC on AMD's Virtex Ultrascale (primary), Virtex 7, and Zynq Ultrascale FPGA boards. Table II summarizes these settings.

SONIC is built using three key components – (1) QUANTISENC hardware [7], (2) a SystemVerilog testbench (PRONTO [11]), and (3) a system software for QUANTISENC. We use snnTorch [12] to specify and train SNN models in software. SONIC's open API is used to configure QUANTISENC to implement the SNN model in hardware. Subsequently, the SONIC's SystemVerilog testbench is used to verify the design for the three evaluated datasets.

For full-system emulation, the trained model weights are programmed in the QUANTISENC hardware (in its synaptic memory) using SONIC's system software. The system software is also used to record the output of SONIC to perform classification and evaluate accuracy.

Datasets	Classes	Training Examples	Test Examples	Evaluation Board	Technology	Resources
Spiking MNIST	10	60,000	100	Virtex UltraScale	16nm FinFET	LUTs: 537,600 FFs: 1,075,200 BRAMs: 1728 DSPs: 768
DVS Gesture	11	1176	288	Virtex 7	28nm	LUTs: 303,600 FFs: 607,200 BRAMs: 1030 DSPs: 2800
Spiking Heidelberg Digit (SHD)	20	8156	2264	Zynq UltraScale	16nm FinFET	LUTs: 230,400 FF: 460,800 BRAM: 312 DSP: 1728

TABLE II. Datasets and FPGA boards used for evaluation.

Figure 5 shows a classification example using the spiking MNIST dataset for the handwritten digit 8. The configuration of SONIC is set to $(256 \times 128 \times 10)$.[1] This baseline configuration gives the best trade-off for the Spiking MNIST dataset in terms of application performance, e.g., accuracy of digit recognition (96.5%), and hardware performance, e.g., area, power, latency, and throughput. In Figure 5, the image is presented to SONIC for a time duration of 150 ms. This is a user-defined parameter and is controlled through the application software. The figure shows the spikes generated by neurons of the three layers (input, hidden, and output). We use a spike counter on the output layer to decode and visualize the classification result.

Output decoding is illustrated in Figure 6. We observe that output neuron 8 has the highest spike count, so the SONIC result is correctly interpreted as 8 corresponding to the image. However, neuron 3 has the second highest number of spikes, followed by neuron 0. This is because of the structural

[1]We scale the original 28×28 images to 16×16 images to reduce model size. This is the maximum scaling that can be performed on handwritten images from the dataset without affecting the classification accuracy.

	Configuration	Neurons	Synapses	Quantization & Precision	LUTs (of 537,600)	Δ	FFs (of 1,075,200)	Δ	BRAMs (of 1728)	Δ	DSPs (of 768)	Δ	Dynamic Power (W)	Δ
1	$256 \times 128 \times 10$	394	34,048	Q5.3	7.6%	–	0.66%	–	3.99%	–	0%	–	0.390	–
2	$256 \times 128 \times 10$	394	34,048	Q9.7	9.38%	1.2×	1.39%	2.1×	3.99%	0%	35.93%	–	0.738	1.9×
3	$256 \times 256 \times 10$	522	68,096	Q5.3	17.44%	2.3×	1.85%	2.8×	7.69%	1.9×	0%	–	1.241	3.2×
4	$256 \times 256 \times 256 \times 10$	778	133,632	Q5.3	34.08%	4.5×	3.55%	5.4×	15.10%	3.8×	0%	–	2.172	5.6×

TABLE III. Resource utilization and dynamic power of SONIC for different SNN architectures.

similarity of the handwritten digit 8 to digits 3 and 0. SONIC allows to perform such analysis for different datasets.

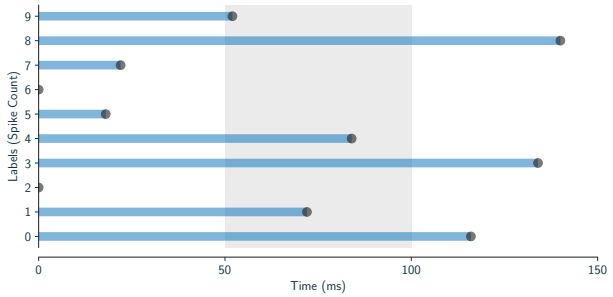

Fig. 6: Counting spikes to decode results.

A. Design Scalability Analysis using SONIC

SONIC can be used to analyze the scalability of neuromorphic designs and perform fast design space exploration. To illustrate this, we provide resource utilization and dynamic power of QUANTISENC for the Spiking MNIST dataset. The key idea is to use the SONIC's API to configure QUANTISENC and synthesize and implement it on a target FPGA board. Subsequently, the SONIC's PRONTO framework is used to simulate the design using the dataset, recording the switching activities in the switching activity interface format (SAIF). This SAIF file is then used to report the dynamic power using AMD's Vivado tool.

Row 1 of Table III reports the resource utilization and dynamic power of QUANTISENC for the baseline configuration of $256 \times 128 \times 10$ and a quantization and precision policy of Q5.3 for neurons and Q1.3 for synaptic weights (reported as Q5.3/Q1.3). The design consists of 394 LIF neurons and 34,048 synapses. It uses BRAMs to implement synaptic memory. The resource utilization for the Virtex UltraScale FPGA is as follows: 40,767 (= 7.6%) LUTs, 7,085 (= 0.66%) FFs, and 69 (= 3.99%) BRAMs. No DSP slices are used. The dynamic power consumption is 390 mW using a spike frequency of 600 KHz. This frequency gives the maximum performance per watt for the design (see Section VI-C). Row 2 reports the utilization using Q9.7/Q1.7. Increasing quantization and precision results in an increase of 1.2× in LUTs and 2.1× in FFs compared to the baseline configuration in row 1. The number of BRAMs remains the same, but the design uses 276 (= 35.93%) DSPs. The dynamic power increases by 1.9×.

Rows 3 & 4 show the utilization and dynamic power for two additional design configurations. The configuration in row 3 has 522 neurons (32.5% higher than baseline) and 68,096 synapses (2× higher than baseline). For this configuration, the design requires 2.3× more LUTs, 2.8× more FFs, and 1.9× more BRAMs than the baseline. The dynamic power is 3.2× higher. On the other hand, the configuration in row 4 has 778 neurons (2× higher than baseline) and 133,632 synapses (4× higher than baseline). For this configuration, the design requires 4.5× more LUTs, 5.4× more FFs, and 3.8× more BRAMs than the baseline. The dynamic power is 5.6× higher.

These results show scalability with the number of neurons and synapses. One way to utilize this scalability result in SONIC is to quickly evaluate the utilization of FPGA resources for a specific configuration of the hardware, without having to synthesize the design, which often takes a considerable amount of time. This is useful when conducting design space exploration, where multiple iterations may be necessary before making a final design choice.

B. Benchmarking State-of-the-Art Designs using SONIC

SONIC can be used to compare and benchmark state-of-the-art neuromorphic designs. Table IV compares SONIC with other designs, for a single neuron and for SNN architectures.

	A Single Neuron (Q3.0)			SNN Architecture		
	Euler [30]	Euler [31]	Ours	Best Accuracy [32]	Best Hardware [33]	Ours
Configuration	–	–	–	784-1024-10	784-2048-10	256-128-10
LUTs	95	76	80	78,679	16,813	40,767
FFs	85	20	23	16,864	7,559	7,085
BRAMs	0	0	0	174	129	69
Norm. Resource	1.43	1.0	1.05	2.02	1.0	0.92
Power (W)	0.25	NR	0.05	3.4	1.03	0.390
Accuracy	–	–	–	98.4%	93.0%	96.5%

TABLE IV. Comparison to state-of-the-art.

We observe that the normalized resource utilization of a single neuron of QUANTISENC is 5% higher than [31] and 26.5% lower than [30]. For the SNN architecture of the Spiking MNIST dataset, QUANTISENC requires 2.2× lower resources than [32] and 8% lower resources than [33]. These improvements are due to the efficient implementation of QUANTISENC using variable quantization and precision policies for its neurons and synaptic weights [7].

C. Throughput Improvement using SONIC

Figure 7 plots the worst setup slack (in ns) of QUANTISENC obtained using SONIC as we increase the spike frequency from 100 KHz to 1.2 MHz using the baseline configuration of $(256 \times 128 \times 10)$ and Q5.3/Q1.3 quantization and precision policies. Setup slack is defined as the difference between the required time and the arrival time of the data at

an endpoint (typically a register). During static timing analysis (STA), a negative setup slack indicates timing violations. The peak frequency is one that results in the least positive setup slack. The subplot reports the dynamic power of SONIC for these memory settings. We make the following observations.

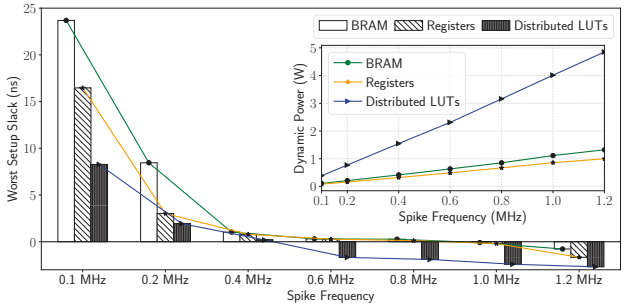

Fig. 7: Maximum frequency and power trade-off.

First, the setup slack is positive for a spike frequency of 100, 200, and 400 KHz for all three memory implementations. As we increase the frequency to 600 KHz, there are multiple timing violations for distributed LUT-based synaptic memory. The peak frequency for this implementation is 450 KHz. Second, the setup slack for distributed LUT-based implementation is 61% lower than for BRAM-based, which means that the latter supports a higher peak frequency. Our results show that the peak spike frequencies for register-based and BRAM implementations are 850 and 925 KHz, respectively. Finally, the register-based implementation has the least dynamic power for all spike frequencies. It is 23% and 79% lower than the BRAM and distributed LUT-based implementations, respectively.

In addition to the peak frequency, we also evaluate QUANTISENC based on its real-time performance, which is measured as the number of images inferred per second, and throughput per watt, which is measured as the number of fixed-point operations performed per watt.

$$\text{Real-time Performance} = \frac{1}{\text{exposure time} + N_{\text{reset}}/f} \text{ (see Sec. IV)} \quad (8)$$

where exposure time is the time interval for which each image is exposed to the SNN model for inference, N_{reset} is the number of clock cycles needed to reset the membrane potential, and f is the spike frequency. The value N_{reset} depends on the membrane time constant (τ). Our empirical studies show that $N_{\text{reset}} = 4$ clock cycles at $f = 1$ KHz for $\tau = 5$ ms. This results in a real-time performance of 41.67 frames per second (fps) for an exposure time of 20 ms. This is the performance obtained by exploiting pipelined parallelism using SONIC. In previous works such as [34], such parallelism has not been exploited. Therefore, real-time performance is $\frac{1}{\text{exposure time}+(K\times L)/f}$, where K is the number of layers and L is the latency (in the number of clock cycles) of each layer. For the baseline design $(256 \times 128 \times 10)$ with three layers, the maximum performance obtained using [34] is 31.25 fps. SONIC improves the real-time performance by 33.3% by exploiting pipelined parallelism.

Figure 8 shows the performance per watt for the three designs analyzed in Table III as the spike frequency ranges from 100 KHz to 1 MHz. We observe that the performance per watt increases with an increase in frequency until it reaches the maximum. This is because at lower frequencies the increase in performance exceeds the increase in dynamic power, resulting in an overall increase in performance per watt for all evaluated designs. With a further increase in frequency, the dynamic power starts to dominate the performance-power ratio, resulting in a reduction in performance per watt. The maximum performance per watt is indicated with a circle, which occurs at a frequency much lower than the highest frequency supported by the respective design.

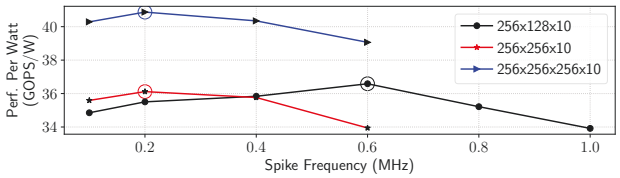

Fig. 8: Performance per watt.

D. Prototyping QUANTISENC Configurations using SONIC

Table V reports the largest QUANTISENC hardware that can be implemented on three evaluated FPGA platforms using SONIC. We report the configuration for a wide design using a single hidden layer and a deep design using multiple hidden layers. We also report the power consumption of these configurations. SONIC allows to easily evaluate and prototype different design configurations on different FPGA platforms.

Platform	Wide Design (Single Hidden Layer)		Deep Design (Multiple Hidden Layers)	
	Configuration	Power (W)	Configuration	Power (W)
Virtex UltraScale	256-1470-10	9.557	256-28(64)-10	6.371
Virtex 7	256-704-10	5.818	256-20(64)-10	4.833
Zynq UltraScale	256-640-10	3.349	256-12(64)-10	1.854

TABLE V. Configuration on three FPGA platforms.

E. Evaluating Different Datasets using SONIC

Table VI summarizes the QUANTISENC results for the three datasets evaluated using SONIC.

		Configuration	Resource Utilization			Accuracy	Dynamic Peak Power (W)	Peak Performance per Watt (GOPS/W)
			LUTs	FFs	BRAMs			
1.	Spiking MNIST	256-128-10	7.6%	0.66%	3.99%	96.5%	0.390	22.91
2.	DVS Gesture	400-300-300-11	60%	15%	18%	85.07%	1.827	24.45
3.	SHD	700-256-256-20	65%	20%	24%	87.8%	1.629	16.09

TABLE VI. Design summary for different datasets.

Row 2 summarizes our design exploration for the DVS Gesture dataset. QUANTISENC requires a configuration of $(400 \times 300 \times 300 \times 11)$, which uses 60% LUTs, 15% FFs, and 18% BRAMs on VirtexUltraScale. The accuracy obtained is 85.07% compared to an snnTorch accuracy of 87.1% [35]. The implemented design has a peak dynamic power of 1.827 W and a performance per watt of 24.45 GOPS/W. Row 3 summarizes our design exploration for the SHD dataset. The design requires a configuration of $(700 \times 256 \times 256 \times 20)$, which uses 65% LUTs, 20% FFs, and 24% BRAMs on

VirtexUltraScale. The accuracy obtained is 87.8% compared to the snnTorch accuracy of 90.3% [36]. The implemented design has a peak dynamic power of 1.629 W and a performance per watt of 16.09 GOPS/W.

F. Early ASIC Synthesis Results using SONIC

Table VII reports the results of the early ASIC synthesis of LIF design of QUANTISENC using the Synopsys Design Compiler for a spike frequency of 100 MHz. The design uses 1,574 nets, 944 combinatorial cells, 35 sequential cells (Flip-Flops), and 309 buffers & inverters, occupying a total area of 2894 μm^2. The design consumes 101.7 μ W of power, divided into 23.2 μ W of switching (dynamic) power and 78.5 μW of leakage power. Our future work will explore other stages of the ASIC design of QUANTISENC.

Technology	Nets	Comb. Cells	Seq. Cells	Buf/Inv	Area	Switching Power	Leakage Power	Total Power
32nm	1574	944	35	309	2894 μm^2	23.2 μW	78.5 μW	101.7 μW

TABLE VII. Early ASIC synthesis results of a neuron.

VII. Conclusion

We propose SONIC, an open-source software-defined neuromorphic design methodology to make neuromorphic computing accessible to all. SONIC is built using three core components. First, it integrates QUANTISENC, a layer-based neuromorphic hardware design with LIF neurons and CUBA synapses. We design API for SONIC to configure layer architectures, neurons per layer, and connection between layers, alongside setting quantization and precision policies for neurons and synaptic weights. Second, it integrates PRONTO, a framework for fast verification and prototyping of SNN hardware on FPGA. Finally, it integrates a system software to perform operation scheduling on QUANTISENC by exploiting its distributed computing and memory architecture. Overall, SONIC is a generalized framework to configure LIF dynamics, deploy model parameters to hardware, perform inference on live data, evaluate hardware performance, and visualize inference results. We evaluate SONIC's capabilities using datasets like Spiking MNIST, DVS Gesture, and Spiking Heidelberg Digit (SHD) on different FPGA development boards. We analyze different design configurations and show SONIC's ability to perform design space exploration. These explorations show superior resource utilization, power efficiency, latency, and throughput of QUANTISENC compared to other designs.

Acknowledgments

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

References

[1] C. Mead, "Neuromorphic electronic systems," *Proc. of the IEEE*, 1990.
[2] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, 1997.
[3] M. Lakshmi Varshika *et al.*, "Design of many-core big little μbrain for energy-efficient embedded neuromorphic computing," in *DATE*, 2022.
[4] D. S. Modha, F. Akopyan, A. Andreopoulos *et al.*, "Neural inference at the frontier of energy, space, and time," *Science*, 2023.
[5] M. Davies, N. Srinivasa, T.-H. Lin *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, 2018.
[6] O. Richter *et al.*, "DYNAP-SE2: A scalable multi-core dynamic neuromorphic asynchronous spiking neural network processor," *NCE*, 2024.
[7] 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 *MWSCAS*, 2024.
[8] A. Joubert, B. Belhadj, O. Temam, and R. Héliot, "Hardware spiking neurons design: Analog or digital?" in *IJCNN*, 2012.
[9] BrainChip. (2022) Akida Neuromorphic System-on-Chip.
[10] SynSense. (2023) DYNAP-SE2.
[11] S. Matinizadeh and A. Das, "An open-source and extensible framework for fast prototyping and benchmarking of spiking neural network hardware," in *FPL*, 2024.
[12] J. Eshraghian, M. Ward, E. Neftci *et al.*, "Training spiking neural networks using lessons from deep learning," *Proc. of the IEEE*, 2023.
[13] W. Fang, Y. Chen, J. Ding, Z. Yu, T. Masquelier, D. Chen, L. Huang, H. Zhou *et al.*, "SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence," *Science Advances*, 2023.
[14] J. Yik *et al.*, "Neurobench: Advancing neuromorphic computing through collaborative, fair and representative benchmarking," *arXiv*, 2023.
[15] A. Das, "Real-time scheduling of machine learning operations on heterogeneous neuromorphic SoC," in *MEMOCODE*, 2022.
[16] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *IJCV*, 2015.
[17] H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, M. Widrich, T. Adler, L. Gruber, M. Holzleitner, M. Pavlović, G. K. Sandve *et al.*, "Hopfield networks is all you need," *arXiv preprint arXiv:2008.02217*, 2020.
[18] 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*, 2018.
[19] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in Neuroscience*, 2016.
[20] Q. Meng, M. Xiao, S. Yan *et al.*, "Towards memory-and time-efficient backpropagation for training spiking neural networks," in *ICCV*, 2023.
[21] A. Paul, S. Wagner, and A. Das, "Learning in feedback-driven recurrent spiking neural networks using full-force training," in *IJCNN*, 2022.
[22] A. Paul *et al.*, "Data driven learning of aperiodic nonlinear dynamic systems using spike based reservoirs-in-reservoir," in *IJCNN*, 2024.
[23] A. Paul and A. Das, "Learning in recurrent spiking neural networks with sparse full-FORCE training," in *ICANN*, 2024.
[24] E. Müller, S. Schmitt, C. Mauch, S. Billaudelle, A. Grübl, M. Güttler, D. Husmann, J. Ilmberger, S. Jeltsch *et al.*, "The operating system of the neuromorphic BrainScaleS-1 system," *Neurocomputing*, 2022.
[25] A. Balaji, A. Das, Y. Wu, K. Huynh, F. G. Dell'Anna, G. Indiveri, J. L. Krichmar, N. D. Dutt, S. Schaafsma, and F. Catthoor, "Mapping spiking neural networks to neuromorphic hardware," *TVLSI*, 2019.
[26] A. Das, "A design flow for scheduling spiking deep convolutional neural networks on heterogeneous neuromorphic system-on-chip," *TECS*, 2023.
[27] G. Tang and K. P. Michmizos, "Real-time mapping on a neuromorphic processor," in *NICE*, 2020.
[28] A. Das, Y. Wu, K. Huynh, F. Dell'Anna *et al.*, "Mapping of local and global synapses on spiking neuromorphic hardware," in *DATE*, 2018.
[29] T. Titirsha, S. Song, A. Das, J. Krichmar, N. Dutt, N. Kandasamy, and F. Catthoor, "Endurance-aware mapping of spiking neural networks to neuromorphic hardware," *IEEE TPDS*, 2021.
[30] W. Guo, H. E. Yantır *et al.*, "Toward the optimal design and FPGA implementation of spiking neural networks," *IEEE TNNLS*, 2021.
[31] W. Ye, Y. Chen, and Y. Liu, "The implementation and optimization of neuromorphic hardware for supporting spiking neural networks with mlp and cnn topologies," *IEEE TCAD*, 2022.
[32] A. M. Abdelsalam *et al.*, "An efficient FPGA-based overlay inference architecture for fully connected DNNs," in *ReConFig*, 2018.
[33] Z. He, C. Shi, T. Wang, Y. Wang, M. Tian, X. Zhou, P. Li, L. Liu *et al.*, "A low-cost FPGA implementation of spiking extreme learning machine with on-chip reward-modulated STDP learning," *TCAS II*, 2021.
[34] F. Corradi *et al.*, "Gyro: A digital spiking neural network architecture for multi-sensory data analytics," in *DroneSE*, 2021.
[35] W. He *et al.*, "Comparing SNNs and RNNs on neuromorphic vision datasets: Similarities and differences," *Neural Networks*, 2020.
[36] I. Hammouamri *et al.*, "Learning delays in spiking neural networks using dilated convolutions with learnable spacings," *arXiv*, 2023.