TNNGen: Automated Design of Neuromorphic Sensory Processing Units for Time-Series Clustering

Prabhu Vellaisamy*, Harideep Nair*, Vamsikrishna Ratnakaram, Dhruv Gupta, and John Paul Shen
Electrical and Computer Engineering Department, Carnegie Mellon University
{pvellais, hpnair, vratnaka, dhruvgup, jpshen}@andrew.cmu.edu

Abstract—**Temporal Neural Networks (TNNs), a special class of spiking neural networks, draw inspiration from the neocortex in utilizing spike-timings for information processing. Recent works proposed a microarchitecture framework and custom macro suite for designing highly energy-efficient application-specific TNNs. This paper introduces *TNNGen*, a pioneering effort towards the automated design of TNNs from PyTorch software models to post-layout netlists. TNNGen comprises a novel PyTorch functional simulator for TNN modeling and application exploration and a Python-based hardware generator for PyTorch-to-RTL and RTL-to-Layout conversions. Seven representative TNN designs for time-series signal clustering across diverse sensory modalities are simulated, and their post-layout hardware complexity and design process runtimes are assessed to demonstrate the effectiveness of TNNGen. We also show TNNGen's ability to forecast silicon metrics accurately without running the hardware process flow.**

Index Terms—**Temporal neural networks, Neuromorphic sensory processing units, Time-series clustering, Design automation.**

I. Introduction and Background

Deep neural networks (DNNs) [1] have emerged as the de facto technology for artificial intelligence (AI), even surpassing human-like sensory processing capabilities. However, this impressive progress comes with an exponential surge in compute demands and energy consumption, raising concerns about long term sustainability of this trend [2]–[4]. Temporal Neural Networks (TNNs), a special class of spiking neural networks (SNNs) employing spike time-based processing with close adherence to biological plausibility [5]–[7], offer a promising alternative path for AI compute, with potential for orders of magnitude improvements on energy efficiency.

Recent research [8] demonstrates that single-layered TNNs excel in unsupervised time-series clustering [9]–[11] and are amenable for resource-constrained edge devices. Further works in advancing TNN research include a microarchitecture framework for TNN implementation [12], and augmenting the ASAP7 [13] 7nm CMOS process design kit (PDK) with TNN-tailored custom macros, called TNN7 [14], for improved energy-efficiency. Recently, the idea of creating an end-to-end framework that can automate the design of specialized TNN chiplets for online sensory processing applications was suggested in [15], [16] as forward looking research proposals.

This work, *TNNGen*, serves as the first attempt at realizing such a framework for the automated design of application-specific TNNs, or *Neuromorphic Sensory Processing Units* (NSPUs), for online clustering of time-series sensory signals. As shown in Fig. 1, it leverages PyTorch [17], PyVerilog [18], Python, Cadence toolchain, and open-source FreePDK45 [19], ASAP7, and TNN7 libraries [14]. Starting from high-level modeling of TNNs in PyTorch, it enables generation of post-layout netlists of the models along with hardware metrics in a single automated flow. It facilitates the development of optimized energy-efficient designs without expert involvement, integrating previously segregated software-only and hardware-only TNN developments. Further, we enable users without EDA access to obtain key hardware results without running the actual process flow, via *forecasting*.

*Both authors contributed equally to this work.

Other similar design frameworks have been proposed for DNNs [20], [21]. ANNA [20] performs application-specific neural architecture search (NAS) followed by translation to High-Level Synthesis (HLS) kernels utilizing pre-defined architectural templates (e.g., convolution units). SODA [21] utilizes multi-level intermediate representation (MLIR) constructs for mapping algorithmic Python models to low-level LLVM intermediate representation, which is then converted to Verilog RTL using HLS. However, both frameworks only support automation until the logic synthesis stage and are targeted towards DNN accelerators. In contrast, our proposed TNNGen generates specialized highly efficient post-layout netlists of neuromorphic TNN implementations starting with TNN functional models in PyTorch. Our key contributions are:

- *TNNGen* - a pioneering attempt at a design framework for automated design of custom TNN-based NSPUs.
- A novel TNNGen functional simulator based on PyTorch for robust modeling and rapid application exploration.
- A novel TNNGen hardware generator based on PyVerilog that translates TNN PyTorch models to layout in conjunction with Cadence EDA tools and a library of finely-tailored TCL scripts for optimizing the TNN designs.
- Time-series clustering performance and post-layout hardware metrics for seven representative TNN designs generated using TNNGen across different technology nodes, extending beyond previous post-synthesis studies.
- Accurate *forecasting* of post-layout die area and leakage power for a quick evaluation of hardware complexity in lieu of the time-consuming EDA runs. It also allows users without expertise in hardware design to focus on software modeling while gaining insights into the silicon cost.

The next section details the TNNGen framework and its key components. Section III describes our experimental setup

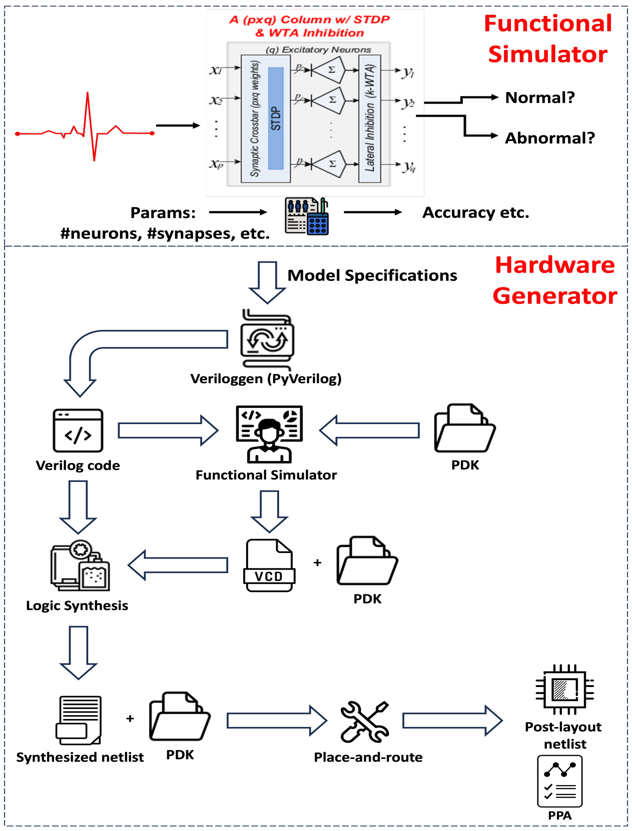

Fig. 1. TNNGen framework for designing TNN-based neuromorphic sensory processing units. It comprises a PyTorch functional simulator and a hardware generator, and automates the entire design flow from PyTorch to chip layout.

and evaluation results, highlighting the optimized TNN designs generated and design flow runtimes, along with *forecasting*. Finally, Section IV summarizes key findings and future work.

II. TNNGen Design Framework

The proposed TNNGen framework orchestrates the entire TNN design flow by providing user-tunable parameters. It facilitates rapid TNN model development and application performance evaluation using a PyTorch-based simulator, which then provides the model specifications to an automated hardware flow that delivers highly optimized physical design netlists.

A. Functional Simulator

A novel PyTorch-based functional simulator is developed as part of TNNGen for swift design space exploration and precise evaluation of application-specific metrics (e.g., classification accuracy, clustering rand index, F1 score, etc.). The simulator is flexible, offering users the ability to quickly explore a vast design space and use the resulting insights to develop optimized TNN models. Some key design space configurations include: (i) single-column TNNs with an arbitrary number of neurons (q) and synapses per neuron (p), and (ii) large multi-layer TNNs with an arbitrary number of layers and columns per layer with configurable inter-layer connectivity. It supports various neuron response function models (including step-no-leak, ramp-no-leak, leaky-integrate-and-fire), winner-take-all inhibition (with customizable winner count and tie-breaking options), and spike timing dependent plasticity (STDP) learning in both supervised and unsupervised modes. Pytorch's tensor operations are utilized to implement all the TNN functionalities for high simulation speed. Further, it also supports GPU acceleration through PyTorch's CUDA API.

TNNGen simulator models time precisely, aligning with the direct implementation methodology in [12] wherein spike times are dictated by precise hardware clock cycles. It performs cycle-accurate temporal modeling for time windows around onset of spikes, and dynamically switches to an event-driven approach in windows where spikes are absent to speed up the simulation. TNNGen employs a modular and parameterized approach, leveraging key functional blocks of TNNs.

B. Hardware Generator

TNNGen automates the hardware design process flow by facilitating automated RTL generation, RTL simulation, logic synthesis, and place-and-route while ensuring a smooth design flow. It leverages Veriloggen package built on top of PyVerilog [18] to provide a Python interface to the user for converting PyTorch model specifications to Verilog RTL codes.

Table I specifies the process flows within TNNGen, the Cadence tools utilized during each flow, and the various libraries currently supported by the framework. Cadence EDA tools are specifically chosen as the ASAP7 and TNN7 libraries are primarily supported in the Cadence toolchain. However, TNNGen is built with huge focus on flexibility and modularity to enable easy integration of other toolchains and libraries. We plan to open-source TNNGen for the research community to not only leverage it for their custom TNN design flow but also enhance it with additional capabilities.

In the TNNGen backend, to enable PyTorch-to-RTL conversion, we implemented all the TNN functionalities in PyVerilog, ensuring the generated RTL is highly optimized and aligns with the microarchitecture in [12]. TNN7 custom macros are incorporated to help accelerate runtime. [14] reports a 3x speedup for logic synthesis; we go a step further and report the place-and-route speedup in Section III. Further, TNNGen contains a library of specifically tailored TCL scripts and templates for automating the various design flows and PDKs, while providing end-user with complete freedom to configure the flow as needed. Note that TNNGen does not cover DRC & LVS checks for signoff as it requires expert intervention.

TABLE I
Industry EDA tools supported by TNNGen.

Design Flow Stage	Cadence Tool
RTL simulation	Xcelium
Logic synthesis	Genus
Place-and-route	Innovus
Library support	FreePDK45, ASAP7, TNN7

III. Results and Evaluation

A. Experimental Setup

We adopt the same seven single-column designs in [15], targeting different sensory modalities within the time series

TABLE II

Seven different TNN configurations for various sensory modality applications used for the experimental setup. Clustering performance (rand index) for TNNs vs state-of-the-art DTCR, normalized to k-means is provided.

Column Size (*pxq*)	UCR Benchmark Name	Sensory Modality	DTCR Rand Index	TNN Rand Index
65x2	SonyAIBORobotSurface2	Accelerometer	0.8354	0.6066
96x2	ECG200	ECG	0.6648	0.6648
152x2	Wafer	Fabrication process	0.7338	0.555
343x2	ToeSegmentation2	Motion sensor	0.8286	0.6683
637x2	Lightning2	Optical + RF sensor	0.5913	0.577
470x5	Beef	Food spectrograph	0.8046	0.731
270x25	WordSynonyms	1D word outlines	0.8984	0.8473

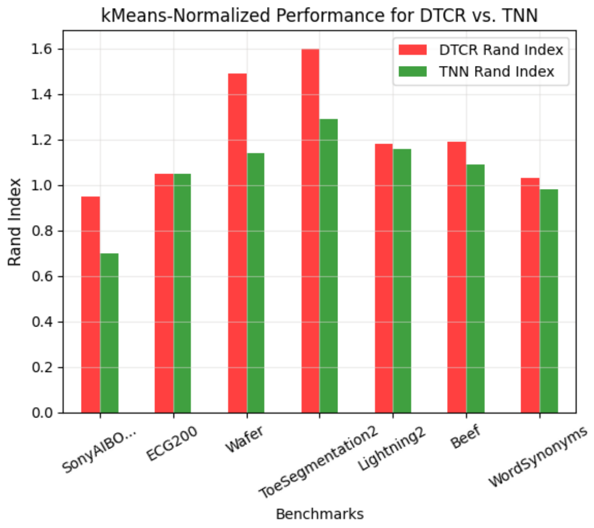

Fig. 2. Clustering performance (rand index) for TNNs vs state-of-the-art deep learning method DTCR, normalized to k-means. Four of the seven TNN designs closely match the performance of DTCR. The other three TNNs fall a bit short. Note that these are all single-column TNNs that are significantly smaller compared to multi-layer RNNs as used in DTCR.

archive from University of California, Riverside (UCR) [22], as representative benchmarks to showcase TNN designs. As shown in Table II they include: 1) *SonyAIBORobotSurface2* - classify two types of walking surfaces based on accelerometer data; 2) *ECG200* - classify normal heartbeat vs. myocardial infarction using ECG signals; 3) *Wafer* - classify normal vs. abnormal silicon wafers based on fabrication process control sensor signals; 4) *ToeSegmentation2* - classify normal vs. abnormal walking using motion sensor data, and 5) *Lightning2* - detect presence or absence of lightning based on power-density series derived from optical and RF sensor spectrogram; 6) *Beef* - classify adulteration levels into five classes using food spectrograph data; and 7) *WordSynonyms* - classify 25 different words based on 1D series from word outlines. The designs are evaluated using three approaches:

- Develop PyTorch TNN models as per p, q parameters in Table II and report corresponding clustering performance.
- Use TNNGen to generate post-layout hardware metrics across multiple cell libraries and technology nodes.
- Predict the hardware metrics without actually running any of the process flows, using TNNGen's *forecasting* feature.

TABLE III

Post-place-and-route leakage power results for the seven TNN columns targeting the seven UCR benchmarks, using three PDK cell libraries: FreePDK45, ASAP7, TNN7.

UCR Benchmark Name	Synapse Count	FreePDK45 Leakage (mW)	ASAP7 Leakage (μW)	TNN7 Leakage (μW)
SonyAIBORobotSurface2	130	0.299	0.961	0.57
ECG200	192	0.442	1.41	0.84
Wafer	304	0.717	2.26	1.34
ToeSegmentation2	686	1.59	5.09	3.14
Lightning2	1274	2.95	9.81	5.84
Beef	2350	5.452	17.4	11.06
WordSynonyms	6750	15.66	46.69	31.13

TABLE IV

Post-place-and-route die area results for the seven TNN columns targeting the seven UCR benchmarks, using three PDK cell libraries: FreePDK45, ASAP7, TNN7.

UCR Benchmark Name	Synapse Count	FreePDK45 Area (μm^2)	ASAP7 Area (μm^2)	TNN7 Area (μm^2)
SonyAIBORobotSurface2	130	14284.466	1028.67	692.06
ECG200	192	21036.08	1513.05	1015.8
Wafer	304	33868.98	2394.01	1608.52
ToeSegmentation2	686	75654.82	5388.72	3682.63
Lightning2	1274	140,502.84	10184.45	6860.68
Beef	2350	259,167.4	18298.1	12634.83
WordSynonyms	6750	744,422.4	51158.20	35303.88

This research extends beyond previous post-synthesis studies on TNN implementations by presenting post-layout results using multiple PDK cell libraries. Further, we provide layout runtime comparisons and assess the *forecasting* feature of TNNGen that can predict post-layout hardware metrics without time-consuming EDA runs. Our runtime simulations are run on a server with 8 Intel Xeon(R) E5-2680 CPU cores.

B. TNNGen Design Performance and Hardware Complexity

TNNGen simulator is used for modeling and rapidly simulating different single-column TNN designs targeting seven different sensory modalities. To evaluate the unsupervised clustering performance, *rand index* is utilized, following the method outlined in [8]. Table II and Fig. 2 show the rand index results normalized to k-means for TNN and a state-of-the-art deep learning algorithm called Deep Temporal Clustering Representation (DTCR) [11]. It can be seen that a single TNN column performs nearly as well as DTCR for four of the seven benchmarks but underperforms for the remaining three. The performance on those three benchmarks can be potentially improved by adopting more complex TNNs with multiple columns and layers. On average, DTCR outperforms TNNs by nearly 12%, aligning with the results in [8]. However, it is essential to note that in contrast to integer-based single-column TNN models, DTCR employs a significantly more complex multi-layer RNN model performing high dimensional floating point tensor algebra, rendering it impractical for deployment in mobile and edge devices due to its computational demands. Thus, the simulator results presented demonstrate the efficacy of small TNN designs for time-series clustering.

TNNGen hardware generator translates the above software models to layouts. We employ three open cell libraries,

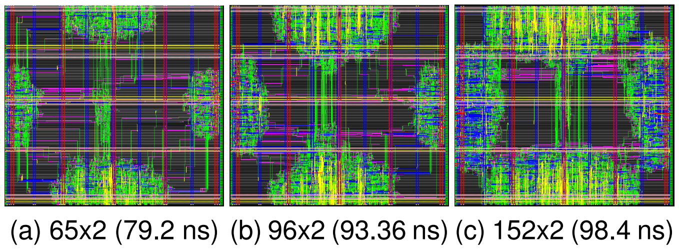

Fig. 3. Layouts of three generated column configurations fitted onto the same floorplan size. $p \times q$ column configurations are provided with computation latencies inside parentheses.

namely, 45nm CMOS FreePDK45 [19], 7nm CMOS predictive ASAP7 PDK [13], and TNN-tailored custom TNN7 library with nine macros as proposed in [14]. The resulting hardware metrics are reported in Tables III and IV.

With TNN7, there is a 32.1% and 38.6% decrease in area and leakage power compared to ASAP7, respectively, aligning with the findings in [14]. We report only leakage power here as total power requires fine-tuned physical rules specific to each design, including clock tree synthesis. Nevertheless, we do report the total power specifically for the largest column (6750 synapses) with TNN7 library for evaluation purposes.

Using the TNN7 macros, the largest column results in just 0.035 mm^2 area and consumes only 0.067 mW total (leakage + dynamic) power after layout, going beyond the post-synthesis area and total power reported in [14]. The corresponding FreePDK45 and ASAP7 area/leakage values are 0.744 mm^2/15.66 mW and 0.051 mm^2/0.047 mW respectively. The advantage of 7nm designs vs. 45nm is clear. We also see from Tables III and IV that TNN7 (with custom macro cells) achieves better area and leakage than ASAP7.

For computation latency (i.e., per sample inference) evaluation, we first consider three smaller columns (65×2, 96×2, 152×2) fitted for the same floorplan size, as shown in the layouts in Fig. 3. The resulting computation times are 79.2 ns, 93.36 ns, and 98.4 ns, respectively. For the largest 270×25 column, the resulting latency is 180 ns. We can see that these TNN designs are extremely fast in performing inference and thereby ideal for low power real-time edge AI deployment.

C. Runtime Evaluation

Authors in [14] report an approximate 3x speedup during logic synthesis due to the use of TNN7 macros during the mapping and optimization phases. We further validate and extend their empirical results by taking a step further and evaluating runtime speedup during place-and-route using Innovus.

Fig. 4 illustrates the runtime for place-and-route for increasing column sizes using only ASAP7 cells vs. TNN7 macros. As depicted, runtime scales with increasing column sizes, but TNN7 macros yield a slower trend. On average, the layout runtime using TNN7 in Innovus place-and-route is roughly 32% better than ASAP7. For the largest column, the entire hardware process flow (synthesis + place-and-route runtime) is reduced by almost 47%, indicating larger designs benefit more in runtime speedup with TNN7's custom macros.

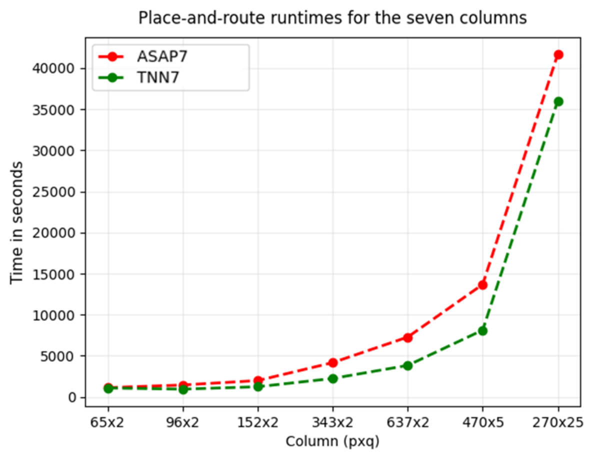

Fig. 4. Innvous place-and-route runtime (in seconds) for baseline (ASAP7) and TNNGen (TNN7) column designs.

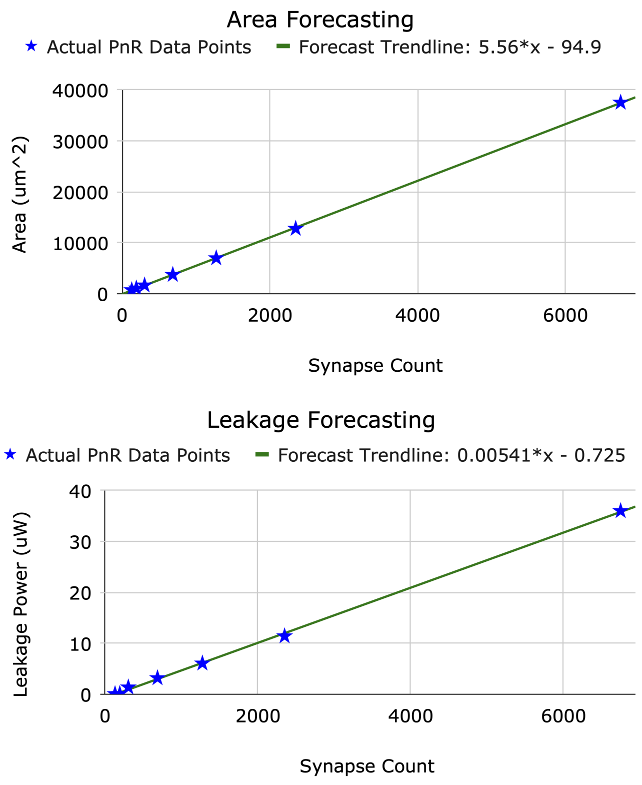

Fig. 5. Area and Leakage power forecasting illustrating actual data points ('stars') and the forecasting trendline equations.

D. Area and Leakage Power Forecasting

Hardware development is typically time-consuming. Many researchers may not have access to commercial EDA tools. Hence, we integrate a *forecasting* feature for predicting silicon die area based on TNN synapse count without actually running the TNNGen hardware flow. This feature leverages the linear trends of area and leakage power with respect to total synapse count to build a linear regression model, which is trained on many TNNGen runs with varying TNN sizes.

The corresponding regression models for area and leakage

TABLE V
FORECASTED (FC) POST-PLACE-AND-ROUTE 7NM PPA FOR SEVEN REPRESENTATIVE UCR COLUMN DESIGNS USING TNNGEN.

UCR Benchmark Name	Syn. Count	FC Area (μm^2)	FC Area Error	FC Leakage (μW)	FC Leakage Error
SonyAIBORobot...	130	627.9	+10.36%	-	-
ECG200	192	972.62	+6.07%	-	-
Wafer	304	1595.34	+2.25%	0.92	+32.9%
ToeSegmentation2	686	3719.26	-0.33%	2.98	+6.14%
Lightning2	1274	6988.54	-0.25%	6.16	-1.72%
Beef	2350	12971.1	-1.7%	11.98	-5.1%
WordSynonyms	6750	37435.1	+0.2%	35.77	+0.52%

power follow the equations:

$$Area = 5.56 * SynapseCount - 94.9 \quad (1)$$

$$Leakage = 0.00541 * SynapseCount - 0.725 \quad (2)$$

Fig. 5 along with Table V report the forecasting (FC) results for area and leakage power, along with their forecasting errors. It can be seen that area can be predicted very accurately within 1% of the original values for large designs. Leakage power, although inaccurate for small designs (omitted for the two smallest designs), is also highly accurate for large designs (the largest design only incurs 0.52% error). Fig. 5 illustrates the efficacy of the linear trendline. The forecasting regression model is part of the TNNGen framework and can be continually refined with more actual design data points.

IV. CONCLUSION & FUTURE WORKS

This paper serves as the first effort in creating an automated design framework (from PyTorch model to chip layout) for the design of application-specific TNN-based neuromorphic sensory processing units (NSPUs). TNNGen confirms the feasibility of such a framework. Initial results indicate automated designs are highly efficient. Post-layout 7nm results show our largest benchmark design requires only 0.035 mm^2 die area and 0.067 mW total power, with a compute latency of 180 ns. This work demonstrates the benefits of leveraging custom macros in improving both hardware metrics and design flow runtimes. We plan to develop a full library of custom macros that can be smoothly integrated into the framework. The current framework stands as an important milestone for demonstrating the feasibility and effectiveness of an end-to-end toolchain for the automated design of application-specific TNNs for online sensory signal processing.

The current TNNGen has some limitations that can be alleviated with further enhancements. TNNGen currently only supports the design of single-column TNNs. We plan to extend the current framework to support more diverse applications and much more complex multi-layer TNN designs. Furthermore, currently only very limited sensory types are supported. We plan to incorporate more diverse sensory signal encoders in order to support a wider range of sensory modalities. With these enhancements, it will allow TNNGen to cover a much larger design space for implementing a wider range of application-specific NSPUs. We plan to open source this framework to facilitate experimentation and further enhancements by the research community at large.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] OpenAI. (2018) Ai and compute. [Online]. Available: https://openai.com/blog/ai-and-compute/

[3] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning," *arXiv preprint arXiv:2007.05558*, 2020.

[4] Numenta. (2022) Ai is harming our planet: addressing ai's staggering energy cost. [Online]. Available: https://www.numenta.com/blog/2022/05/24/ai-is-harming-our-planet/

[5] J. E. Smith, "Space-time algebra: A model for neocortical computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 289–300.

[6] ——, "Space-time computing with temporal neural networks," *Synthesis Lectures on Computer Architecture*, vol. 12, no. 2, pp. i–215, 2017.

[7] ——, "A temporal neural network architecture for online learning," *arXiv preprint arXiv:2011.13844*, 2020.

[8] S. Chaudhari, H. Nair, J. M. Moura, and J. P. Shen, "Unsupervised clustering of time series signals using neuromorphic energy-efficient temporal neural networks," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 7873–7877.

[9] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah, "Time-series clustering–a decade review," *Information systems*, vol. 53, pp. 16–38, 2015.

[10] Q. Zhang, J. Wu, P. Zhang, G. Long, and C. Zhang, "Salient subsequence learning for time series clustering," *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 9, pp. 2193–2207, 2018.

[11] Q. Ma, J. Zheng, S. Li, and G. W. Cottrell, "Learning representations for time series clustering," *Advances in neural information processing systems*, vol. 32, 2019.

[12] H. Nair, J. P. Shen, and J. E. Smith, "A microarchitecture implementation framework for online learning with temporal neural networks," in *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2021, pp. 266–271.

[13] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[14] H. Nair, P. Vellaisamy, S. Bhasuthkar, and J. P. Shen, "Tnn7: A custom macro suite for implementing highly optimized designs of neuromorphic tnns," in *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2022, pp. 152–157.

[15] J. P. Shen and H. Nair, "Cortical columns computing systems: Microarchitecture model, functional building blocks, and design tools," in *Neuromorphic Computing*. IntechOpen, 2023, ch. 8. [Online]. Available: https://doi.org/10.5772/intechopen.110252

[16] P. Vellaisamy and J. P. Shen, "Towards a design framework for tnn-based neuromorphic sensory processing units," *arXiv preprint arXiv:2205.14248*, 2022.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[18] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040. Springer International Publishing, Apr 2015, pp. 451–460. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16214-0_42

[19] C. H. Oliveira, M. T. Moreira, R. A. Guazzelli, and N. L. Calazans, "Ascend-freepdk45: An open source standard cell library for asynchronous design," in *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2016, pp. 652–655.

[20] C. Li, K. Zhang, Y. Li, J. Shang, X. Zhang, and L. Qian, "Anna: Accelerating neural network accelerator through software-hardware co-design for vertical applications in edge systems," *Future Generation Computer Systems*, vol. 140, pp. 91–103, 2023.

[21] 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, no. 5, pp. 78–88, 2022.

[22] H. A. Dau, E. Keogh, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, Yanping, B. Hu, N. Begum, A. Bagnall, A. Mueen, G. Batista, and Hexagon-ML, "The UCR time series classification archive," October 2018, https://www.cs.ucr.edu/~eamonn/time\_series\_data\_2018/.