Preprint of paper from AICAS 2024, Abu Dhabi, April 2024.

TNN-CIM: An In-SRAM CMOS Implementation of TNN-Based Synaptic Arrays with STDP Learning

Harideep Nair, David Barajas-Jasso, Quinn Jacobson, and John Paul Shen
Electrical and Computer Engineering Department, Carnegie Mellon University
{*hpnair, dbarajas, qjacobso, jpshen*}*@andrew.cmu.edu*

***Abstract*—Temporal Neural Networks (TNNs), a special class of spiking neural networks (SNNs), process information via spike timings, analogous to the brain. Recent works have proposed microarchitecture and custom macros for directly implementing extremely energy-efficient TNNs with standard CMOS. However, prior works implement synapses using expensive register-based counters. Since synapses constitute most of the hardware complexity in TNNs, it is imperative to optimize their implementation. This work proposes TNN-CIM, an in-SRAM implementation of synaptic arrays, as a first attempt towards compute-in-memory (CIM) solution for TNNs. In TNN-CIM, not only are the synaptic weights stored in SRAM, but synaptic response function generation (inference) and spike timing dependent plasticity (learning) are also embedded directly within the SRAM array. Our results in 45nm CMOS demonstrate 1.3x and 1.7x reduction in power and area respectively while improving latency performance by 1.4x. Further, we provide transistor count equations to assess hardware complexity scaling of arbitrary TNN-CIM synaptic arrays.**

***Index Terms*—Temporal Neural Networks, Synaptic Arrays, STDP, SRAM, Compute-In-Memory, Neuromorphic Computing**

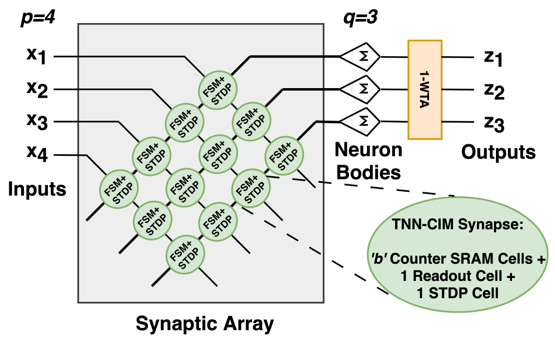


Fig. 1: A 4x3 TNN column containing: 4 synaptic inputs per neuron and 3 neurons with its 4x3 synaptic crossbar, and 1-WTA lateral inhibition [*adapted from [12]*]. Each of the 12 synapses in the crossbar stores a b-bit weight value, performs local inference via counter-based FSM, and updates its weight locally via STDP learning, concurrently every compute cycle.

I. Introduction and Background

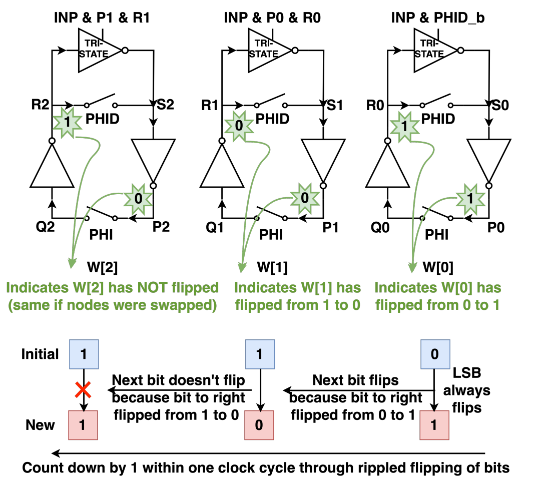
Neuromorphic computing employing spiking neural networks (SNNs) derives its inspiration from brain's computational principles and has been touted as a promising approach for future AI compute systems [1]. Temporal Neural Networks (TNNs) [2]–[4] are a special class of SNNs that operate on spike timings and utilize a biologically plausible learning mechanism called Spike Timing Dependent Plasticity (STDP). Recent works demonstrated the efficacy of TNNs in delivering state-of-the-art performance for time-series signal clustering [5] and handwritten digit recognition [4] applications while consuming merely tens of uW to tens of mW power [6].

TNNs, structurally and functionally inspired by the neocortex, adhere to a well-defined organizational hierarchy consisting of synapses, neurons, columns (or minicolumns), and layers [7]. A column forms the fundamental operational building block for TNNs. Authors in [5] demonstrated that a single TNN column is powerful enough to outperform other significantly more complicated approaches for time-series clustering. Figure 1 shows a pxq TNN column consisting of q=3 neurons sharing a set of p=4 synaptic inputs through a 4x3 synaptic crossbar, followed by winner-take-all (WTA) inhibition across the neuron outputs. The synaptic crossbar constitutes bulk of the compute and thereby hardware complexity in a column.

Recent works have proposed a microarchitecture framework [8] and a set of custom macro building blocks [6] for efficient hardware implementation of TNNs. However, these works rely on expensive flip-flops to implement the synaptic crossbar. Our work is a first attempt at implementing the entire synaptic crossbar array in SRAM, to significantly reduce the hardware complexity. This serves as the first step towards compute-in-memory (CIM) implementation of TNNs, hence named *TNN-CIM*. SRAM is chosen in this work due to its compatibility with standard CMOS logic and maturity [9], [10] as compared to other emerging approaches such as Resistive RAM [11].

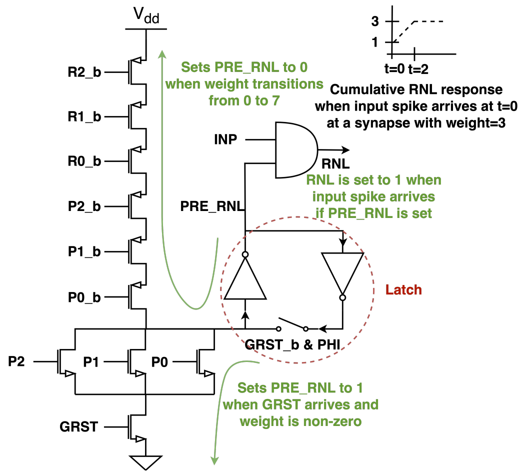
Our SRAM design follows the very recent work titled FAST [13], which proposed a 10T SRAM cell with two intra-cell NMOS switches and one inter-cell transmission gate switch. In contrast to prior SRAM-based CIM works [14]–[19] that are limited by serial row-by-row access, FAST enables concurrent row compute through multi-phased shifting. Here, we repurpose the FAST SRAM cell to store sequential states across switches and use two-phased switching to ripple states through a series of cells, thus implementing a finite state machine (FSM). We then leverage this FSM to perform *in-situ* synaptic inference and STDP learning rules. Our key contributions are:

- We present *TNN-CIM*, the first work towards compute-in-memory implementation of TNNs, wherein the synaptic crossbar with STDP is fully implemented in SRAM.
- As part of synaptic inference, TNN-CIM proposes a novel *Ripple-Flip Counter* to implement binary counting within SRAM, with arbitrary powers-of-2 count values. We



INP & P1 & R1
INP & P0 & R0
INP & PHID_b
TRI-STATE
R2
S2
R1
S1
R0
S0
PHID
PHI
Q2
P2
Q1
P1
Q0
P0
W[2]
W[1]
W[0]
Indicates W[2] has NOT flipped (same if nodes were swapped)
Indicates W[1] has flipped from 1 to 0
Indicates W[0] has flipped from 0 to 1
Initial
New
Next bit doesn't flip because bit to right flipped from 1 to 0
Next bit flips because bit to right flipped from 0 to 1
LSB always flips
Count down by 1 within one clock cycle through rippled flipping of bits

Fig. 2: Ripple-Flip Counter: 3-bit weight is shown where each bit is implemented in an SRAM cell consisting of additional two transmission gate switches controlled by PHI/PHID signals and a tri-state inverter that flips the cell value based on the "flip" status of the cell to the right. Example shows weight counting down from 6 to 5 through rippled flipping of bits.



V_{dd}
R2_b
R1_b
R0_b
P2_b
P1_b
P0_b
P2
P1
P0
GRST
Sets PRE_RNL to 0 when weight transitions from 0 to 7
Cumulative RNL response when input spike arrives at t=0 at a synapse with weight=3
t=0 t=2
INP
RNL
RNL is set to 1 when input spike arrives if PRE_RNL is set
PRE_RNL
Latch
GRST_b & PHI
Sets PRE_RNL to 1 when GRST arrives and weight is non-zero

Fig. 3: RNL Readout Cell: Pull-down circuit sets *PRE_RNL* to 1 at the beginning of gamma period for non-zero synaptic weight. Pull-up network resets *PRE_RNL* to 0 when weight transitions from 0 to 7 by leveraging the decoupled states across all 3 counter cells. *PRE_RNL* is latched using a custom SRAM-type cell with one switch. Final *RNL* output is set to 1 if *PRE_RNL* is high and input spike is asserted.

believe such sequential counting implemented in SRAM without any adder is novel, to the best of our knowledge.

- Along with synaptic inference compute, TNN-CIM implements *in-situ* STDP-based unsupervised learning with which all the synapses in the crossbar can be simultaneously and locally updated in every compute cycle.
- Compared to previous TNN implementations [6], [8], TNN-CIM provides improvements of 1.3x, 1.4x and 1.7x, in power, performance and area, respectively.
- Parameterized equations to assess transistor count complexity scaling of TNN-CIM synaptic arrays are provided.

Section II describes circuit-level design of the key components of the proposed SRAM-based TNN-CIM. Experimental framework and hardware complexity analysis are presented in Section III, followed by conclusions in Section IV.

II. TNN-CIM: SRAM Synapse Implementation

This section presents the key components of TNN-CIM, starting with the Ripple-Flip Counter design, followed by implementation of synaptic inference utilizing this counter and finally synaptic learning algorithm (STDP) implementation. Minor circuit details are omitted from schematics for brevity.

A. Ripple-Flip Counter FSM

Prior SRAM CIM works have focused exclusively on implementing combinational logic such as search, shift, add, and multiply. Here, we propose an SRAM design that leverages FAST [13] switches to store different states and implement sequential counting (Figure 2). FAST SRAM uses a peripheral 1-bit adder to perform addition in N clock cycles (where N is the bitwidth of the stored value). While this can be used to implement counting, it takes multiple cycles to update its count by 1. In contrast, the key mechanism in TNN-CIM is to perform upcount (or downcount) by rippling bit flips from LSB to MSB *within a single clock cycle*. For example, downcount by 1 simply flips all '0's to '1's from LSB to MSB until a '1' is reached at which point that '1' is flipped to '0' and no further rippling occurs (see Figure 2). Similarly, upcount by 1 entails flipping all '1's to '0's from LSB to MSB until a '0' is reached. Note that this mechanism easily enables upcount/downcount by arbitrary powers-of-2 by just changing the starting bit for rippling (count by 2 starts from the second bit, i.e., bit to the left of LSB and so on). To implement this in SRAM, a bit has to know whether its neighbor to the right has been flipped or not. This can be done by carefully re-purposing the decoupled states across the switches within each cell, as explained next.

Figure 2 shows a 3-bit weight counter where each counter cell incorporates two transmission gate switches and a tri-state inverter in addition to the traditional 6T configuration. In contrast to FAST that uses three phases, we use only two phases wherein the transmission gate switches are controlled by *PHI* and delayed *PHID* (ϕ_2 and ϕ_{2d} as in FAST), which are asserted in the first half of clock cycle and de-asserted in the second half. The switches are opened before performing the operation ("shift" in case of FAST; "downcount" in our case), and once finished, *PHI* switch closes first followed by the *PHID* switch. To downcount by 1, the LSB cell W[0] is flipped via the tri-state inverter when an input spike *INP* arrives and *PHID* switch is open (*PHID_b* is set). This triggers a chain reaction that ripples to the left, all within the same clock cycle.

Now, W[1] only needs to flip if W[0] flipped from 0 to 1. Note, if the bit to the right didn't flip or flipped from 1 to 0, it implies the end of rippling. This "flip" status is indicated by both *P0* and *R0* nodes of W[0] simultaneously having a value of 1. Also, this is possible only if the switches are open. This

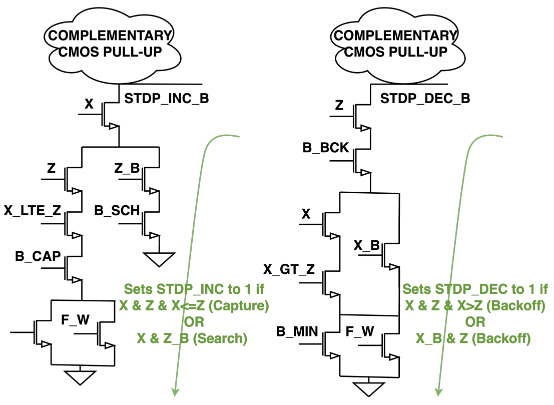

COMPLEMENTARY CMOS PULL-UP
STDP_INC_B
X
Z
Z_B
X_LTE_Z
B_SCH
B_CAP
F_W
Sets STDP_INC to 1 if X & Z & X<=Z (Capture) OR X & Z_B (Search)
COMPLEMENTARY CMOS PULL-UP
STDP_DEC_B
Z
B_BCK
X
X_B
X_GT_Z
B_MIN
F_W
Sets STDP_DEC to 1 if X & Z & X>Z (Backoff) OR X_B & Z (Backoff)

Fig. 4: STDP: Custom CMOS gates to increment/decrement as per STDP cases 1 (capture), 2 (backoff), 3 (search), 4 (backoff) in Table I. *B_CAP*, *B_SCH*, *B_BCK*, *B_MIN* are Bernoulli random variables to regulate corresponding stochastic updates. *F_W* is stabilization function [8]. X/Z are input/output spikes.

Case	Input Conditions		Weight Update
1. Capture	$X \neq \infty$;	$X \leq Z$	$\Delta w = +B\_CAP * max(F\_W, B\_MIN)$
2. Backoff	$Z \neq \infty$	$X > Z$	$\Delta w = -B\_BCK * max(F\_W, B\_MIN)$
3. Search	$X \neq \infty$; $Z = \infty$		$\Delta w = +B\_SCH$
4. Backoff	$X = \infty$; $Z \neq \infty$		$\Delta w = -B\_BCK * max(F\_W, B\_MIN)$
-	$X = \infty$; $Z = \infty$		$\Delta w = 0$

TABLE I: STDP update rules (from [8]) directly implemented in TNN-CIM, consisting of four cases depending on the presence/absence/relative timings of input (X)/output (Z) spikes.

TABLE II: TNN-CIM Transistor Count (TC) evaluation of a single synapse with bitwidth b based on its key components.

Component	b-bit Synapse	pxq Synaptic Array
Ripple-Flip Counter	$18b + 6$	$(18b + 6) * pq$
RNL Readout	$3b + 13$	$(3b + 13) * pq$
STDP	$2^{b+2} + 2b + 76$	$(2^{b+2} + 2b + 76) * pq$
Total	$2^{b+2} + 23b + 95$	$(2^{b+2} + 23b + 95) * pq$

is depicted in the control signals of W[1]'s tri-state inverter. A similar mechanism applies for all the bits to the left. Note that this counter naturally wraps around to 7 from 0, as is needed for RNL response function generation. This is described next.

B. Synaptic Inference (Response Function Readout)

As proposed in [8], synaptic inference generates a unary readout of ramp-no-leak (RNL) response function based on the weight counter state (i.e., the weight value). RNL output is set to 1 when an input spike arrives at a synapse with non-zero weight (at which point weight counter also starts decrementing), and it then gets reset to 0 when the weight counter wraps around to 7 from 0 (for 3-bit weight). This sets the RNL output to 1 for as many cycles as the synaptic weight value. We implement this as a "readout" cell as follows.

For every 3-bit synaptic weight consisting of 3 counter cells (Figure 2), there exists one readout cell (Figure 3). As in [8], a *gamma* period denotes the duration of one synaptic learning plus inference compute. *GRST* is a signal that is asserted only for one cycle at the beginning of every gamma period. As shown in Figure 3, it consists of a custom tri-state circuit that sets or resets *PRE_RNL* which is latched in a set of two cross-coupled inverters with a transmission gate switch. *PRE_RNL* is set to 1 at the beginning of gamma period only if the weight value is non-zero (indicated by *P2*, *P1* and *P0*), else it remains at zero. In the presence of an input spike, *RNL* output is set to 1 if *PRE_RNL* is asserted. *PRE_RNL* and thereby *RNL* are both de-asserted when weight counter transitions from 0 to 7 which is implemented by the pull-up circuit, leveraging the decoupled states across the switches within the 3 counter cells. Once *PRE_RNL* is reset to 0, it can only be set in the next gamma period. This ensures RNL output is asserted only until weight counter counts down to 0 in presence of an input spike.

C. Synaptic Learning (STDP)

The STDP unsupervised learning logic (Figure 4) generates two control signals, *STDP_INC* and *STDP_DEC*, to increment or decrement the weight counter by 1. TNN-CIM directly implements the four STDP cases listed in Table I as noted by the green text in Figure 4 (fifth case is implicit). Compared to previous TNN STDP implementations, our STDP implementation incorporates a key optimization. Each counter cell in Figure 2 that natively performs decrement needs additional transistors to support increment due to STDP. This overhead is avoided by ensuring the counter wraps around to its (original value + 1) during RNL readout (i.e., downcount for only 7 instead of 8 cycles for 3-bit weight). As a result, during the STDP cycle, high *STDP_INC* implies no change, high *STDP_DEC* implies decrement by 2, and neither set high implies decrement by 1 (this restores original weight from previous gamma period before current gamma's RNL compute begins). Thus, we avoid having to perform any explicit increment. Note that both STDP_INC and STDP_DEC cannot be set high simultaneously. In addition to the circuit shown in Figure 4, we also use custom latches (highlighted in red in Figure 3) to generate signals such as X_LTE_Z ($X \leq Z$) and X_GT_Z ($X > Z$). Thus, this STDP logic implementation localized for every set of 3 counter cells (representing 3-bit weight) enables TNN-CIM to perform *in-situ online continuous learning within the SRAM array.*

III. Evaluation and Results

This section evaluates the proposed TNN-CIM SRAM-based synaptic array against the baseline flipflop-based microarchitecture in [8]. Two types of evaluation are presented: 1) transistor count analysis and parameterized equations to assess hardware complexity scaling of TNN-CIM, and 2) 45 nm CMOS power-performance-area (PPA) for three synaptic array sizes - two sizes 64x8, 128x10 are adopted from [8] and 96x2 from [7] targeting unsupervised ECG signal clustering.

A. Transistor Count Evaluation

We derive transistor count equations based on the bitwidth b of each synapse, and the synaptic array size pxq with total synapse count $p*q$ (Figure 1). Table II provides the parameterized equations for a single synapse and a pxq synaptic array, segregated into the three key components. These equations

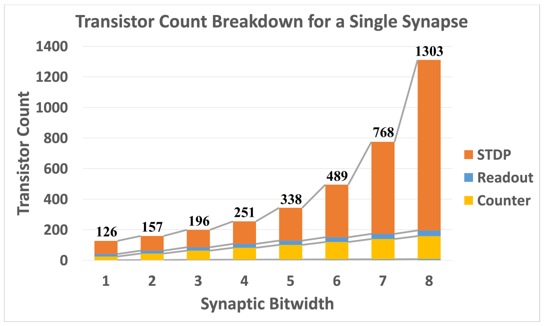


Fig. 5: Transistor Count Breakdown for a Synapse: STDP (60%), counter (30%), and readout (10%). Total transistor count is shown at the top of the bar for each bitwidth.

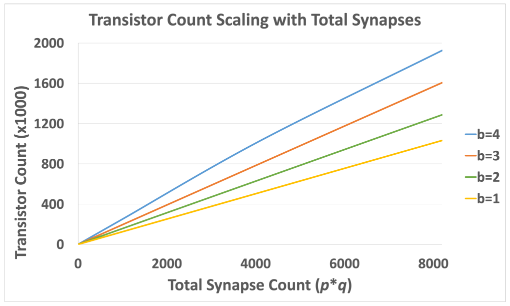


Fig. 6: Transistor Count scaling relative to total synapse count (pxq) and bitwidth (b) of synaptic weights.

can be used to assess transistor count, and thereby area and leakage power, for arbitrarily configured TNN-CIM array.

Figures 5 illustrates the breakdown of transistor count for a single synapse across varying bitwidths (1 to 8 bits). Note that prior TNN works only present hardware complexity for 3-bit synapses. Each 3-bit synapse in TNN-CIM consumes 196 transistors (less than half of the transistor count in [8]). It can be seen from Figure 5 that STDP scales exponentially with bitwidth and experiences a sharp increase for 7 and 8 bits, thereby suggesting a reasonable maximum bitwidth of 6. In contrast, counter and readout scale much better (linear) throughout 1 to 8 bits. Figure 6 illustrates the linear scaling of transistor count with increasing total synapse count across 1 to 4 bits (from neuroscience, only 3-4 bits are needed).

Key Takeaway: STDP consumes majority (60%) of synaptic complexity, followed by counter (30%) and readout (10%). STDP's overhead (esp., dynamic power) can be substantially mitigated once weights converge (due to infrequent updates).

B. PPA Evaluation and ECG Signal Clustering Performance

Table III provides 45nm PPA for three TNN-CIM synaptic arrays (b=3 bits) and compares them against corresponding flipflop-based baselines in [8]. Baseline area, power values are scaled to reflect just the synaptic complexity alone. Computation time is not scaled as neuron body incurs the critical path in baseline. Baseline $96x2$ values are derived using characteristic scaling equations from [8]. TNN-CIM schematics and layouts are designed with 1V supply voltage and 100kHz clock using Cadence Virtuoso with SPICE simulations to get PPA values.

TABLE III: 45nm PPA Comparison of TNN-CIM vs. Baseline [8] for three synaptic array sizes (b=3 bits), including an application-specific configuration for ECG signal clustering.

	Synapses x Neurons	**Total Synapses**	**Area** [mm^2]	**Comp. Time** [ns]	**Power** [mW]
TNN-CIM	64 × 8	512	0.026	22.5	0.177
	128 × 10	1280	0.066	22.5	0.443
	96 × 2 (ECG)	192	0.010	22.5	0.066
Baseline [8]	64 × 8	512	0.045	28.95	0.225
	128 × 10	1280	0.117	32.40	0.558
	96 × 2 (ECG)	192	0.017	30.6	0.084

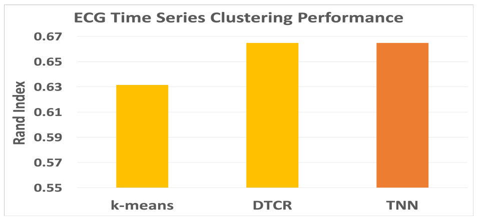


Fig. 7: ECG Clustering Performance (rand index) of TNN with 96x2 synaptic array vs. k-means and state-of-the-art DTCR.

Table III shows that TNN-CIM reduces area and power by 1.7x and 1.3x respectively. In contrast to baseline implementations, TNN-CIM incurs majority of the critical path in synaptic rippling compute (body accumulation can be simply implemented as analog addition over shared bitline [20]), and hence it stays constant with fixed bitwidth. Computation time is improved by 1.4x. Area and power for TNN-CIM and baseline scale linearly with $p*q$. Figure 7 shows PyTorch [21] results (clustering rand index) on ECG200 dataset [22] for a TNN column with $96x2$ synaptic array, illustrating its efficacy over baseline k-means and much more complex DTCR [23].

Key Takeaway: TNN-CIM significantly improves all three PPA metrics compared to flipflop-based synaptic arrays. Compared to complex ML algorithms running on CPUs and GPUs consuming 10's-100'sW power, TNN-CIM incurs just 66 μW, enabling competitive ECG clustering within sub-mW power.

IV. Conclusion and Future Work

Previous works have proposed microarchitecture and custom macros for efficient implementation of TNNs using standard CMOS and flipflop-based synapses. This work proposes an in-SRAM implementation of TNN synaptic arrays, *TNN-CIM*, to optimize the major source of hardware complexity in TNNs. As part of TNN-CIM, a novel Ripple-Flip Counter is presented that can also be used as a regular CIM counter outside the context of a synapse. This counter is almost 2x more efficient than a flipflop-based counter. TNN-CIM enables significant improvement in power (1.3x), performance (1.4x) and area (1.7x) compared to flipflop-based synapses. This work serves as a first step towards a CIM solution for TNNs and can be extended to incorporate well-researched analog addition over bitline for neuron body accumulation and WTA inhibition, in order to fully implement TNN columns entirely in SRAM.

References

[1] C. D. Schuman, S. R. Kulkarni, M. Parsa, J. P. Mitchell, P. Date, and B. Kay, "Opportunities for neuromorphic computing algorithms and applications," *Nature Computational Science*, vol. 2, no. 1, pp. 10–19, 2022.

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

[3] ——, "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.

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

[5] 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.

[6] 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.

[7] 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

[8] 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.

[9] C.-J. Jhang, C.-X. Xue, J.-M. Hung, F.-C. Chang, and M.-F. Chang, "Challenges and trends of sram-based computing-in-memory for ai edge devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 5, pp. 1773–1786, 2021.

[10] S. Mittal, G. Verma, B. Kaushik, and F. A. Khanday, "A survey of sram-based in-memory computing techniques and applications," *Journal of Systems Architecture*, vol. 119, p. 102276, 2021.

[11] S. Yu, W. Shim, X. Peng, and Y. Luo, "Rram for compute-in-memory: From inference to training," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 7, pp. 2753–2765, 2021.

[12] H. Nair, J. P. Shen, and J. E. Smith, "Direct cmos implementation of neuromorphic temporal neural networks for sensory processing," *arXiv preprint arXiv:2009.00457*, 2020.

[13] Y. Chen, Y. Fu, M. Lee, S. George, Y. Liu, V. Narayanan, H. Yang, and X. Li, "Fast: A fully-concurrent access sram topology for high row-wise parallelism applications based on dynamic shift operations," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 4, pp. 1605–1609, 2022.

[14] K. Lee, J. Jeong, S. Cheon, W. Choi, and J. Park, "Bit parallel 6t sram in-memory computing with reconfigurable bit-precision," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[15] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester, "14.2 a compute sram with bit-serial integer/floating-point operations for programmable in-memory vector acceleration," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 224–226.

[16] X. Si, J.-J. Chen, Y.-N. Tu, W.-H. Huang, J.-H. Wang, Y.-C. Chiu, W.-C. Wei, S.-Y. Wu, X. Sun, R. Liu *et al.*, "A twin-8t sram computation-in-memory unit-macro for multibit cnn-based ai edge processors," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 1, pp. 189–202, 2019.

[17] A. Biswas and A. P. Chandrakasan, "Conv-sram: An energy-efficient sram with in-memory dot-product computation for low-power convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 217–230, 2018.

[18] A. Agrawal, A. Jaiswal, C. Lee, and K. Roy, "X-sram: Enabling in-memory boolean computations in cmos static random access memories," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4219–4232, 2018.

[19] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[20] K. Lee, J. Kim, and J. Park, "Low-cost 7t-sram compute-in-memory design based on bit-line charge-sharing based analog-to-digital conversion," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–8.

[21] 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, pp. 8026—8037, 2019.

[22] H. A. Dau, A. Bagnall, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, and E. Keogh, "The ucr time series archive," *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 6, pp. 1293–1305, 2019.

[23] Q. Ma, J. Zheng, S. Li, and G. W. Cottrell, "Learning representations for time series clustering," in *Advances in Neural Information Processing Systems*, 2019, pp. 3781–3791.