LoAS: Fully Temporal-Parallel Dataflow for Dual-Sparse Spiking Neural Networks

Ruokai Yin
Yale University
New Haven, USA
ruokai.yin@yale.edu

Youngeun Kim
Yale University
New Haven, USA
youngeun.kim@yale.edu

Di Wu
University of Central Florida
Orlando, USA
di.wu@ucf.edu

Priyadarshini Panda
Yale University
New Haven, USA
priya.panda@yale.edu

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00084

***Abstract*—Spiking Neural Networks (SNNs) have gained significant research attention over the past decade due to their potential for enabling resource-constrained edge devices. While existing SNN accelerators efficiently process sparse spikes with dense weights, the opportunities for accelerating SNNs with sparse weights, referred to as dual-sparsity, remain underexplored. In this work, we focus on accelerating dual-sparse SNNs, particularly on their core operation: sparse-matrix-sparse-matrix multiplication (spMspM). Our observations reveal that executing a dual-sparse SNN on existing spMspM accelerators designed for dual-sparse Artificial Neural Networks (ANNs) results in sub-optimal efficiency. The main challenge is that SNNs, which naturally processes multiple timesteps, introducing an additional loop in ANN spMspM, leading to longer latency and more memory traffic. To address this issue, we propose a fully temporal-parallel (FTP) dataflow that minimizes data movement across timesteps and reduces the end-to-end latency of dual-sparse SNNs. To enhance the efficiency of the FTP dataflow, we introduce an FTP-friendly spike compression mechanism that efficiently compresses single-bit spikes and ensures contiguous memory access. Additionally, we propose an FTP-friendly inner-join circuit that reduces the cost of expensive prefix-sum circuits with negligible throughput penalties. These innovations are encapsulated in LoAS, a Low-latency inference Accelerator for dual-sparse SNNs. Running dual-sparse SNN workloads on LoAS demonstrates significant speedup (up to $8.51\times$) and energy reduction (up to $3.68\times$) compared to prior dual-sparse accelerators.**

***Index Terms*—SNNs, dual-sparsity, dataflow, accelerator**

I. Introduction

Spiking Neural Networks (SNNs) have attracted considerable interest as potential energy-efficient alternatives to Artificial Neural Networks (ANNs) [5], [11], [41]. Inspired by biological neurons, SNNs utilize highly sparse unary-coded [52] ($\{0,1\}$) spikes to compute and communicate information. Consequently, running SNNs on hardware significantly reduces computation and data movement, making them well-suited for edge computing. As a result, SNNs have been widely employed in computer vision tasks such as image classification [44], [54], optical flow estimation [27], semantic segmentation [21], and object detection [20].

Opportunity. With the growing demand for edge devices with limited memory capacity, recent research on SNNs has underscored the importance of dual-sparsity, where both spikes and weights are sparse, which can be achieved through pruning techniques [5], [22]. Pruning weight connections in SNNs has been explored both during training [4], [47] and inference [37]. Some studies have achieved approximately 98% weight sparsity and 90% spike sparsity for SNNs at iso-accuracy [22], [57] by leveraging the lottery ticket hypothesis [13]. These works highlight the potential of dual-sparse SNNs to achieve unprecedented energy efficiency and memory footprint savings with minimal to no loss in accuracy.

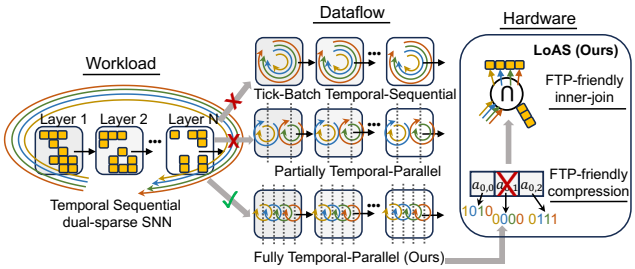

Fig. 1. An illustrative example of the FTP dataflow and LoAS architecture. The FTP dataflow is compared with prior dataflow designs for SNNs. The temporal-sequential tick-batch approach is adapted from SpinalFlow [35], while the partially temporal-parallel approach is adapted from PTB [28]. Each arrow loop represents the processing of one timestep, and the vertical line indicates parallel processing.

Challenge. Despite the algorithmic advancements in dual-sparse SNNs, hardware development has not yet caught up to fully exploit such dual-sparsity. Generally, existing SNN accelerators can be classified into two main categories. First, multi-core neuromorphic systems[1] employ a plethora of cores, and even chips, leverage the inherent parallelism in spiking neuron dynamics [1], [7], [14], [46]. While these systems are capable of capturing the extensive parallelism and sparse activity among neurons, they require all neurons (including all weight connections) to be mapped on-chip. This approach inevitably leads to a significant waste of hardware resources on neurons that are not involved in any computations due to dual-sparsity [39]. Second, dataflow-based SNN accelerators are inspired by dataflow-oriented ANN accelerators, taking advantage from the extensive data reuse across an array of

This work was supported in part by CoCoSys, a JUMP2.0 center sponsored by DARPA and SRC, the National Science Foundation (CAREER Award, Grant #2312366, Grant #2318152), the DoE MMICC center SEA-CROGS (Award #DE-SC0023198), and the University of Central Florida.

[1]We do not compare with these systems due to our focus on single-core dataflow SNN accelerator designs in this work.

979-8-3503-5057-9/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00084

TABLE I
COMPARISON OF LOAS WITH PRIOR SNN ACCELERATORS. S AND T DENOTE THE SPATIAL AND TEMPORAL DIMENSIONS, RESPECTIVELY. SPATIAL PARALLELISM REFERS TO PE-LEVEL PARALLELISM.

Accelerator	Spike Sparsity	Weight Sparsity	Parallel support	Neuron support
SpinalFlow [35]	✔	✘	S	LIF [8]
PTB [28]	✔	✘	S+partial-T	LIF
Stellar [32]	✔	✘	S+fully-T	FS [50]
LoAS (ours)	✔	✔	S+fully-T	LIF

processing elements [28], [32], [35]. However, these designs have primarily focused on processing dense SNN workloads. Currently, there is a lack of dataflow architectures specifically designed to target dual-sparsity in SNNs. Table I summarizes existing dataflow SNN accelerators.

Insight. Although operands in dual-sparse SNNs have varying bitwidths, their interactions follow the pattern of sparse-matrix-sparse-matrix multiplication (spMspM), a concept extensively studied in ANNs [9], [15], [18], [19], [38]–[40], [49], [60], [62]. However, running dual-sparse SNNs on existing spMspM accelerators is inefficient for several reasons. First, the presence of timesteps in SNNs complicates the dataflow design for existing spMspM accelerators. In ANNs, spMspM operations are typically implemented as triple-nested for-loops [39], [53], with different spMspM dataflows being derived by permuting the order of these loops. However, in SNNs, the inclusion of timesteps adds an additional level of looping, resulting in increased latency and memory traffic. Furthermore, this extra loop introduces dataflow dependencies and effectively doubles the dataflow design space, thereby delaying time-to-solution. Second, the asymmetric bitwidths of spikes and weights in SNNs make it inefficient to employ conventional compression formats used in ANN spMspM accelerators. Existing ANN spMspM accelerators typically store sparse matrices using popular compression formats like compressed sparse row (CSR), which require multiple bits to encode the coordinates of non-zero values. Consequently, using multiple bits to compress single-bit spikes (which can only be 0 or 1) is highly inefficient in the context of dual-sparse SNNs.

Proposal. To address these challenges and unleash the potential of dual-sparse SNNs in the context of spMspM, we propose a fully temporal-parallel (FTP) dataflow, as illustrated in Figure 1. FTP dataflow parallelizes all timesteps, thereby avoiding complex dataflow dependencies and minimizing both latency and memory traffic. To further enhance the efficiency of FTP dataflow in terms of memory and computation, we introduce an FTP-friendly spike compression mechanism and an inner-join mechanism. The proposed compression method packs spikes across timesteps, allowing for contiguous memory access. Additionally, the proposed inner-join mechanism nearly halves the cost of the cumbersome prefix-sum circuits while maintaining throughput comparable to prior inner-join designs. To validate the effectiveness of FTP dataflow, we developed LoAS, a **Lo**w-latency Inference **A**ccelerator for Dual-Sparse **S**piking Neural Networks. Our contributions are outlined below:

1) We observe that SNNs with rich dual-sparsity from both input spikes and weight connections perform sub-optimally on existing hardware. Current SNN hardware does not support dual-sparsity, while ANN spMspM hardware struggles to efficiently process timesteps in SNNs, leading to high latency and memory traffic.
2) To enhance the efficiency of processing timesteps, we propose a fully temporal-parallel (FTP) dataflow. FTP eliminates extra memory traffic across timesteps and minimizes the latency associated with processing timesteps sequentially.
3) To fully leverage FTP, we introduce FTP-friendly spike compression for efficient, contiguous memory access, and an FTP-friendly inner-join mechanism that offers low-cost computation with negligible latency penalties.
4) We developed LoAS, a novel architecture that embodies the FTP dataflow. With FTP dataflow and both FTP-friendly compression and inner-join, LoAS achieves significant speedup and energy efficiency compared to other sequential spMspM baselines.

The remainder of this paper is organized as follows: Section II reviews the background and justifies the motivation. Sections III and IV articulate our proposed FTP dataflow and LoAS architecture. Sections V and VI then evaluate our design. Finally, Sections VII and VIII present our discussion and conclusions.

II. BACKGROUND AND MOTIVATION

A. Preliminary of SNNs

1) Leaky-Integrate-and-Fire Neuron: The Leaky-Integrate-and-Fire (LIF) neuron is a classical neuron model [8], widely adopted in prior SNN research [22], [23], [61], [63] due to its biological plausibility and high accuracy. In this work, we focus on accelerating the workloads of dual-sparse SNNs that utilize LIF neurons. During inference, each layer has a sparse input spike tensor $A \in \mathbb{U}^{M\times K\times T}$, where $\mathbb{U} \in 0, 1$, and a sparse weight matrix defined as $B \in \mathbb{Z}^{K\times N}$. Here, T represents the number of total timesteps, while M, N, and K correspond to the spatial dimensions of the input and weight matrix, respectively. The behavior of an SNN layer can be described as follows:

Step 1: Sparse Matrix Multiplication Sparse matrix multiplication (spMspM) is performed across all timesteps to obtain the full output matrix $O \in \mathbb{Z}^{M\times N\times T}$:

$$O_{m,n}[t_i] = \sum_{k=0}^{K} A_{m,k}[t_i]B_{k,n}, \tag{1}$$

where t_i represents the current timestep.

Step 2: LIF Firing LIF neurons take a snapshot of O at timestep t_i and generate a snapshot of the output spike tensor $C \in \mathbb{U}^{M\times N\times T}$ for the current timestep t_i:

$$C_{m,n}[t_i] = \begin{cases} 1 & X_{m,n}[t_i] > v_{th} \\ 0 & \text{else,} \end{cases} \tag{2}$$

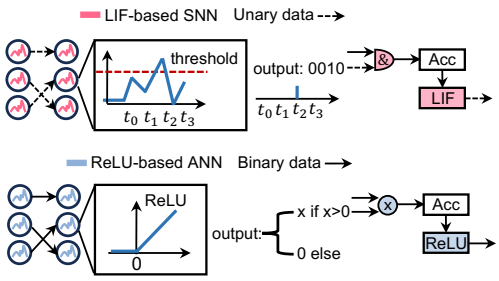

Fig. 2. Comparison between LIF-based SNN neurons and ReLU-based ANN neurons, highlighting their behavior and hardware implementations.

where $X_{m,n}[t_i] = O_{m,n}[t_i] + U_{m,n}[t_{i-1}]$. Here, $U[t_{i-1}]$ is the membrane potential that carries over temporal information from the previous timestep t_{i-1}, and v_{th} is the firing threshold, a predefined scalar value.

Step 3: Membrane Potential Update After the output spikes are generated, the membrane potential is updated to carry residual information to the next timestep, according to the following equation:[2]

$$U_{m,n}[t_i] = \tau X_{m,n}[t_i](1 - C_{m,n}[t_i]), \tag{3}$$

where $\tau \in (0, 1)$ is the leaky factor. From the above equations, we observe that generating the output spike matrix C for timestep t_i requires information from the previous timestep $U[t_{i-1}]$. This introduces temporal dependency between output spike matrices across timesteps. The behavior of a LIF neuron is illustrated in Figure 2.

2) Spike Encoding and SNN Training: A critical step in utilizing SNNs for conventional machine learning tasks is encoding the input source data (e.g., image pixels) into spike trains across multiple timesteps. These input spike trains are then sequentially fed into the SNN for processing. Recent SNN research has adopted direct encoding (a special form of rate encoding) to achieve high accuracy on conventional computer vision tasks with very few timesteps ($\leq$4) [22], [24], [55], [57], [63]. In this work, we focus on accelerating dual-sparse SNNs that use direct encoding. These SNNs are trained using backpropagation-through-time (BPTT) [51] with surrogate gradients [36], achieving performance very close to that of ANNs on many complex tasks [55], [63].

B. Distinctive Features and Challenges of SNNs

Several distinctive features make SNNs well-suited for low-power edge deployment with a potential challenge.

Feature 1: Unary Activation One of the most distinctive features of SNNs is their unary spike activation. Specifically, SNNs utilize single-bit, non-weighted activations to propagate information through layers. The primary advantage of unary activation is the simplified, low-power arithmetic units it enables. As illustrated in Figure 2, compared to the multiply-accumulate (MAC) operations in ANNs, SNNs only require simple bitwise-AND and accumulate (AC) operations.[3] Without the need for expensive multipliers [16], the computations in SNNs demand extremely low power and area.

Feature 2: Sparse Spike Activity Another key feature of SNNs is their highly sparse spike-firing activity. In ANNs, after the MAC operations are completed, a ReLU unit filters out non-positive outputs. In contrast, SNNs use an AC operation, followed by the Leaky-Integrate-and-Fire (LIF) unit, which only fires (producing an output of 1) when the input exceeds a preset threshold. Consequently, the output sparsity in SNNs is typically much higher (around 90%) [57], [58], [61], [63] compared to that in ReLU-based ANNs (around 50%) [39], [43]. This higher sparsity translates into greater computation and memory savings in the context of spMspM acceleration.

Challenge: Repeated Timesteps Despite the aforementioned hardware-friendly features, a significant challenge in deploying SNNs on hardware is their intrinsic reliance on repeated timesteps. A timestep is the smallest discrete unit of time in SNNs.[4] Within each timestep, every neuron must complete the AC operations for all non-zero inputs, fire a spike if necessary, and update its membrane potential. To capture the temporal dynamics of the input data, SNNs must operate across multiple timesteps, as illustrated in Figure 2. However, running across multiple timesteps increases latency and reduces energy efficiency, diminishing the benefits of low-power circuits unless a specialized architecture is employed [35].

C. spMspM Dataflows in SNNs

There are various ways to map spMspM onto hardware, each with distinct efficiency [30], [33]. Three different spMspM dataflows have been proposed in existing dual-sparse ANN accelerators: Inner-product (**`IP`**) [15], [18], [19], [40], Outer-product (**`OP`**) [9], [38], [39], [62], and Gustavson's (**`Gust`**) [49], [60]. Figure 3 illustrates these three dataflows in SNNs for two input matrices, A and B, and an output matrix, C. Their abstract loop nests are shown on the right-hand side. As discussed in Section II-B, it is essential to account for timesteps when performing spMspM operations in SNNs. Inside the black box in Figure 3, the dataflow corresponds to one timestep and is thus identical to ANN dataflow. Outside the black box, the multiple input matrices A (blurred) represent the input spike matrices across different timesteps. Simultaneously, multiple output spike matrices C, which have temporal dependencies, are generated. To accommodate the timesteps in SNNs, an additional loop dimension (t dimension) must be considered in the original triple-nested for-loop. The t dimension (highlighted in the blue box) introduces temporal dependency to each output pixel in SNNs. For example, when processing an SNN using the **`IP`** dataflow, as shown in Figure 3, we first calculate the output cell at position

[2]In this work, we focus on the hard reset scheme, where the membrane potential is reset to zero if an output spike of one is generated. Although other reset schemes exist, using one of them does not affect the generality of the hardware design.

[3]Other implementations using multiplexers exist [28], [35].

[4]Timestep is also referred to as a tick [35] or time-point [28] in other works. We follow the naming convention used in the latest SNN algorithm research.

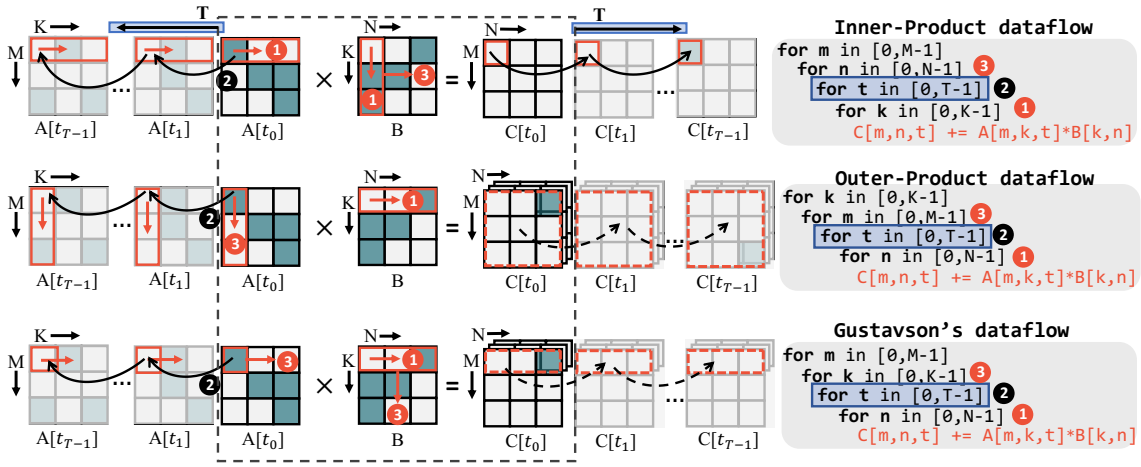

Fig. 3. Comparison of different spMspM dataflows for SNNs. For illustration, $C[t_i]$ represents the spMspM result between $A[t_i]$ and B, similar to spMspM in ANNs. In reality, an additional LIF step (Equation (2)) is required to obtain $C[t_i]$. Circled numbers indicate the computation order for each spMspM dataflow. The `t` dimension is fixed here for illustration; in practice, 16 possible permutations of spMspM dataflow in SNNs exist, discussed in Section III.

(0,0) for timestep 0 (C[0,0,0]). Then, instead of moving to position (0,1), we proceed to calculate the output cell at (0,0) for timestep 1 (C[0,0,1]). Since the output cell C[0,0,1] is temporally dependent on the result of C[0,0,0], we must process C[0,0,0] before C[0,0,1].

D. ANN spMspM Hardware for Dual-Sparse SNNs

We review existing ANN spMspM accelerators to understand why naively running dual-sparse SNNs on these accelerators is sub-optimal.

Inner-join Design: For the **`IP`** dataflow, prior accelerators typically employ an inner-join-based design [9], [15]. In these designs, non-zero values in the rows of matrix A and the columns of matrix B are compressed using bitmask representation, where a bit string has 1's for positions with non-zero values and 0's otherwise. An inner-join unit scans two bitmasks on the fly to determine if there is a matching position (where both multiplicands are non-zero) and then sends the matched pairs to the compute units. Running dual-sparse SNNs on an inner-join-based design does not require additional bitmasks for the input spike matrix A, as the unary spike train itself can be viewed as a bitmask. However, as shown in Figure 4, the timesteps introduce multiple extra rounds of running the expensive inner-join units (which can occupy roughly 46% of the system-level power [15]), leading to high energy costs. Furthermore, since the spike trains are used as bitmasks, all spikes—whether 1 or 0—must be fetched from off-chip DRAM, resulting in no memory traffic savings for the sparse spike matrix A.

Merger-based Design: Unlike **`IP`** dataflow designs that exhibit full output matrix (C) reuse, **`OP`** and **`Gust`** dataflow designs focus on reusing input matrices A and B. In **`OP`**, each column of A and each row of B are traversed only once, leading to efficient input data reuse. However, only one partial sum is generated at a time and is merged later. While these two dataflows achieve better data reuse for the

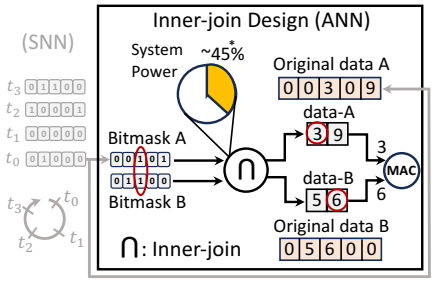

Fig. 4. An example of the inner-join design. The difference between the behavior of ANN and SNN is shown. *Data from SparTen [15].

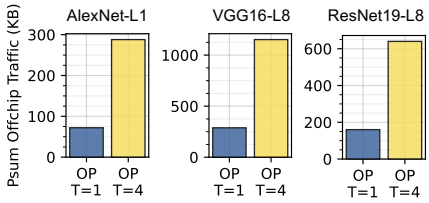

Fig. 5. Off-chip traffic of partial sum matrices across different SNN layers, with SNNs running on GoSPA [9], an OP dataflow spMspM accelerator, using 1 and 4 timesteps.

input matrices, the partial sum matrices (rows) potentially cause increased off-chip data traffic. To mitigate the high memory traffic associated with partial sums, some designs implement large and costly mergers (e.g., 38× more area than multipliers [62]) to merge as many partial sum matrices (rows) as possible before sending them back to off-chip DRAM. Due to the additional t dimension, running dual-sparse SNNs on a merger-based design either requires a more complex merger capable of handling the extra partial sum traffic or results in increased off-chip memory traffic. As shown in Figure 5, for a timestep of four, on average, 4× more partial sum traffic is induced compared to a single timestep.

E. Dataflow Architecture for SNNs

SpinalFlow: Temporal-Sequential Design. SpinalFlow [35] is the first SNN-specific accelerator designed to exploit the

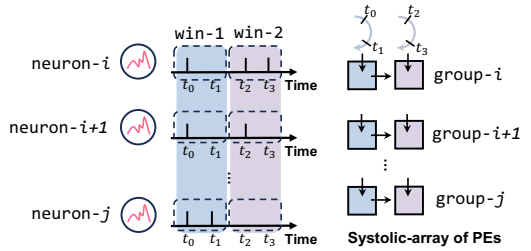

Fig. 6. Example of PTB's partially temporal-parallel design. Each column of the PE array processes a time-window of multiple timesteps. Time-windows run in parallel, but timesteps within each window are processed sequentially.

efficiency of single-bit activation and extremely sparse spike activity. The authors identified the challenge of sequentially processing the entire SNN network through timesteps. To address this challenge, SpinalFlow processes all timesteps for one layer before moving on to the next layer, as illustrated in Figure 1. SpinalFlow dispatches LIF neurons across different processing elements (PEs) and parallelizes the computation. However, within each layer, the timesteps are still processed sequentially, as shown in Figure 1. SpinalFlow is optimized specifically for temporally-coded SNNs, which may lag in accuracy performance compared to rate-coded SNNs [28]. In this work, we focus on accelerating spMspM for general rate-coded SNNs, which achieve accuracy competitive with ANNs across various tasks.

PTB: Partially Temporal-Parallel. While SpinalFlow's design is tailored to temporally-coded SNNs, PTB [28] proposes a general architecture design for rate-coded SNNs. By leveraging the high data-reuse pattern across different PEs in a systolic array architecture [26], PTB breaks the processing of all timesteps into multiple time-windows, each consisting of several contiguous timesteps, and runs these time-windows in parallel, as shown in Figure 1. PTB parallelizes the mapping of multiple time-windows across different columns of the systolic array, while the computation of different LIF neurons is parallelized across the rows of the array. This hardware mapping strategy is illustrated in detail in Figure 6. Although PTB attempts to parallelize the processing of timesteps, the parallelization occurs at the granularity of the time-window. Within each time-window (column of PEs), timesteps are still processed sequentially. Consequently, we categorize PTB as a partially temporal-parallel design. One unique aspect of LoAS compared to PTB is that LoAS leverages a different loop ordering (Section III and VII), enabling comprehensive optimizations.

Prior SNN accelerators with LIF neurons process timesteps in a sequential or partially parallel manner. As discussed in Sections II-C and II-D, it is challenging for these existing SNN designs to achieve high performance in spMspM SNN acceleration. Therefore, a spMspM-friendly strategy for processing timesteps is needed.

Stellar: Temporal-Parallel with non-LIF Neurons. Stellar [32] is another systolic array SNN accelerator that attempts to process timesteps in a fully parallel manner. However, Stellar focuses on optimizing for the Few Spikes (FS) neuron [50], as shown in Table I. FS neurons differ from LIF neurons by decoupling the spike accumulation and firing stages. As a result, FS neurons naturally lack temporal dependency during the spike accumulation stage, making fully parallel temporal processing straightforward in Stellar. In contrast, as discussed in Section II-A, temporal dependency is inherent in the input data for LIF neurons, which makes their design space different from that of Stellar for fully temporal-parallel processing. Unlike the widely adopted LIF neurons, supporting FS neurons requires non-trivial algorithm-hardware co-design, which is beyond the scope of this work.

Algorithm 1 Fully Temporal-Parallel Dataflow (**FTP**)

Input:
Input spike matrix $A \in \mathbb{U}^{M\times K\times T}(\mathbb{U} \in \{0,1\})$
Weight matrix $B \in \mathbb{Z}^{K\times N}$
Output:
Output spike matrix $C \in \mathbb{U}^{M\times N\times T}$

for $m \in M$ **do**
  for $n \in N$ **do**
    for $k \in K$ **do**
      parallel-for $t \in T$ **do** ▷ Spatially unrolled
        $O[m,n,t]$ += $A[m,k,t] \times B[k,n]$
    end for
      parallel-for $t \in T$ **do** ▷ Spatially unrolled
        $C[m,n,t]$ = LIF$(O[m,n,t])$
  end for
end for

III. Fully Temporal-Parallel Dataflow

We propose a fully temporal-parallel dataflow (FTP) designed to mitigate the negative effects of repeatedly processing timesteps on spMspM accelerators (Section II-D). The proposed FTP is outlined in Algorithm 1.

An SNN-friendly spMspM dataflow should satisfy three key objectives: (1) minimize data refetching across timesteps; (2) generate as few partial sums as possible in the temporal dimension (timesteps); and (3) reduce latency in the temporal dimension to minimize the additional cost of sparsity-handling units.

Our first observation is that for all three spMspM dataflows (Section II-C), unless the temporal dimension (t-dim) is placed at the innermost loop, it will result in at least T times more data refetching in the dimensions below, compared to the original dataflow. For example, in **OP**, if the t-dim is placed between m and n, T times more access to B's rows is required. If the t-dim is placed between k and m, T times more access to A's columns and B's rows is required. Depending on the on-chip buffer capacity, repeated memory access may lead to more expensive off-chip memory access, which contradicts objective (1).

Our second observation is that both **OP** and **Gust** dataflows are unsuitable for dual-sparse SNNs because they conflict with objective (2). In the **OP** dataflow, we find that regardless

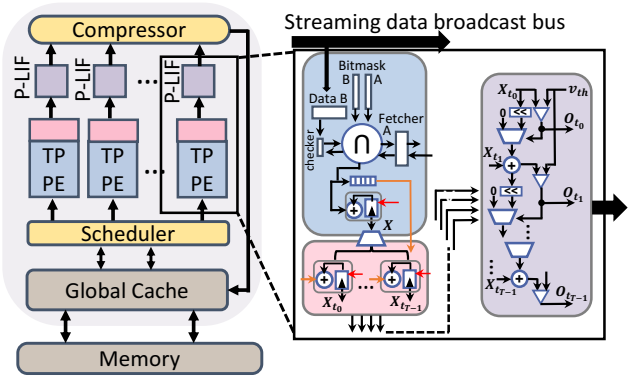

Fig. 7. Architecture of LoAS and the microarchitecture of the TPPE. Red arrows indicate enable signals that skip computation on 0 spikes [59].

of where the t-dim is inserted into the original triple-nested loop, it always produces T times more partial sum matrices compared to the original **OP** dataflow. These partial sums need to be stored in an on-chip cache until all partial sums along both the spatial (k) and temporal dimensions (t-dim) are accumulated, which adds extra memory overhead in **OP**. The same issue arises in the **Gust** dataflow. The t-dim will either generate T times more partial sum rows or require T times more access to both the k and n dimensions. Our final observation is that, regardless of the position of the t-dim, processing it sequentially will always result in T times more processing latency, which conflicts with objective (3).

Our solution is straightforward yet effective. We position the t-dim at the innermost level of the **IP** dataflow and fully parallelize it, as detailed in Algorithm 1. This design choice offers several advantages. Firstly, placing the t-dim at the innermost loop ensures that no additional data movement is incurred, satisfying objective (1). Secondly, since the **IP** dataflow efficiently reuses outputs, no extra partial sums are generated along the t-dim, addressing objective (2). Lastly, by fully parallelizing the t-dim, we eliminate the latency associated with sequentially processing timesteps, achieving objective (3). This is equivalent to transforming the *for-loop* of t-dim into a *parallel-for* loop [53]. The *parallel-for* loop parallelizes operations across different spatial instances, requiring minimal hardware overhead due to the duplication of only inexpensive accumulators, and the number of timesteps in direct-coded SNNs is small (Section II-A). We later demonstrate in the ablation studies that FTP scales well with increasing timesteps.

IV. LoAS

An overview of LoAS is shown in Figure 7. LoAS consists of multiple temporal-parallel processing elements (TPPEs) and parallel Leaky-Integrate-Fire units (P-LIFs) that are specifically designed to execute the FTP dataflow. It also includes a scheduler that distributes workloads across the TPPEs and a compressor that compresses the output spikes from the P-LIFs and writes them back to the on-chip memory. An on-chip SRAM is integrated to facilitate data reuse.

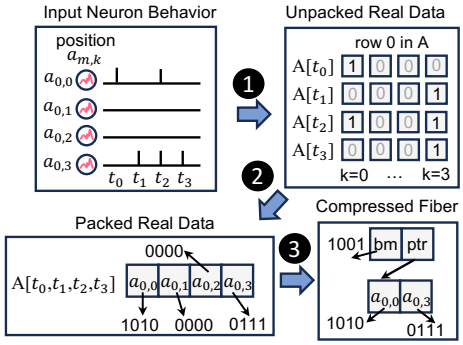

Fig. 8. Example of input spike compression in LoAS. 'bm' stands for bitmask, and 'ptr' stands for pointer.

A. Spikes Compression

We first discuss how sparse input spikes (matrix A) across timesteps are compressed in LoAS. Efficiently compressing matrix A in SNNs involves addressing two key challenges:

How to maximize compression efficiency of 1-bit spikes? Assume that the input spike matrix A has a size of 128×128 for each timestep. In either CSR or CSC format, two 7-bit coordinates are needed to compress each 1-bit non-zero spike.[5] Furthermore, since SNNs naturally operate across multiple timesteps, different spike values may occur at the same coordinate across different timesteps (e.g., 0 for T=1&3, and 1 for T=2&4). To accurately capture all non-zero spikes, separate coordinate values would be required for each timestep.

How to maintain contiguous memory access of non-zero spikes across timesteps? The FTP dataflow proposed in Section III requires spatial unrolling of the input spike matrix A across all timesteps beneath the k dimension. Consequently, a non-contiguous memory layout of A along the t dimension would lead to fragmented memory access at all levels of the memory hierarchy, resulting in higher data movement costs.

To illustrate these two challenges, consider the example in Figure 8. Suppose that the input spikes sent to the system have the pre-synaptic neuron $a_{0,0}$ (the first element of row-0 in matrix A) firing a spike at t_0 and t_2. As shown in step ❶, to represent this pre-synaptic neuron behavior, a single-bit 1 needs to be stored at row-0, column-0 of matrix A for both timesteps 0 and 2, as depicted in the "unpacked real data" box. For each non-zero spike in row-0 of matrix A at each timestep, if a coordinate value (e.g., 4-bit for CSR) is used to record its position, then $2 \times 4 = 8$ bits are needed to compress 2 bits (2 spikes). The compression efficiency in this case is only 25%. Furthermore, memory access to spikes across different timesteps is discontinuous (sequentially accessing different t rows of the spatial row-0 of A).

We propose the following spike compression format for LoAS to address these two challenges. As shown in step ❷, we pack all the spikes (both 0 and 1) across all timesteps into

[5]For 128 columns, $\log_2(128) = 7$ bits are required for the coordinates. We neglect the offsets in this discussion, which would further increase the number of bits used for coordinates.

a single continuous data block for each pre-synaptic neuron. In the example in Figure 8, we store a 4-bit value 1010 at the first position of row-0 in matrix A for $a_{0,0}$ and 0111 at the fourth position for $a_{0,3}$. Since neurons $a_{0,1}$ and $a_{0,2}$ do not spike at any timestep, their packed value would be 0000 (as shown in the "packed real data" box). We define these neurons as **silent neurons**.[6] With this strategy, only the non-silent neurons are treated as non-zero values and stored in memory for matrix A, as shown in step ❸. To accommodate our FTP dataflow, we compress the input spike matrix A in a row-wise manner and use the bitmask format [9], [15], [40] to represent the coordinates of the non-zero values. The bitmask format uses a 1-bit coordinate value for each position in the row. In our example, the bitmask is 1001, indicating that the first and fourth elements in the row are non-zero, while the second and third elements are silent neurons (represented by 0 in the bitmask). Following the bitmask, a pointer is stored to indicate the starting location of the non-zero values in the row. We refer to this compressed row as a fiber [33], [60]. In our example, we use 4 bits to compress 5 bits (5 non-zero spikes), resulting in a compression efficiency of 125%.

The key to our compression method is the ratio of silent neurons in the SNN. Fortunately, empirical studies have shown that SNNs have a significant fraction of silent neurons (60%–70%, as shown in Table II). We also apply a similar bitmask-based technique to compress weights in a column-wise manner, with each compressed weight column also referred to as a fiber.

B. Temporal-Parallel Processing Elements

The fundamental building blocks of LoAS's compute engine are the Temporal-Parallel Processing Elements (TPPEs) and Parallel Leaky-Integrate-Fire units (P-LIFs), which we describe next. Figure 7 details the design of the TPPE. Each TPPE computes the full sum for one output neuron across all timesteps (Line 5 in Algorithm 1). Before computation begins, the bitmask (bm-B) of a fiber from weight matrix B (fiber-B) and its non-zero data are read from SRAM and broadcast into the small bitmask buffers (128 bits in our design) within each TPPE. The bitmask (bm-A) of the fiber from the input spike matrix A (fiber-A) is also fetched and sent to the TPPEs. Each TPPE holds the bitmask for a distinct fiber along the row of A. Once the data are loaded, an *inner-join* operation [9], [15], [18] is performed between the two bitmasks. Depending on the inner-join result, the matched non-zero data of fiber-A are fetched from the global cache and sent to the *pseudo-accumulator* (to be discussed shortly) to perform the accumulation (AC) operation. After the TPPE completes the full computation for one output neuron, it sends the result to the P-LIF unit, which generates output spikes for all timesteps in one operation.

C. Inner-join Unit

The inner-join operation has been extensively studied in prior works [9], [15], [18] for spMspM acceleration in ANNs.

[6]We follow the same terminology used in [28].

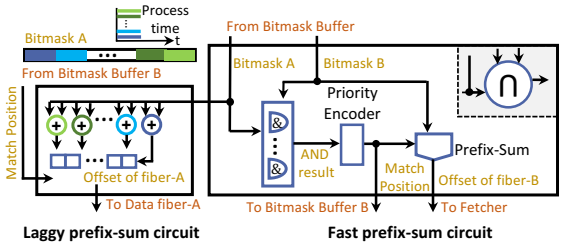

Fig. 9. Illustration of the proposed FTP-friendly inner-join unit.

The inner-join mechanism, coupled with a prefix-sum circuit, has been efficiently implemented using bitmask representation [15]. In [15], a logical AND operation is first applied to two bitmasks to obtain the *AND-result*, which represents the locations where both data are non-zero. This *AND-result* is then sent to a priority encoder to convert the *matched positions* into integer values. The *matched positions* are subsequently sent to two separate prefix-sum circuits to determine the number of 1s preceding each *matched position* in each bitmask, which provides the offsets for each non-zero data in memory.

During this process, the use of two *fast prefix-sum* circuits[7] is a costly operation, consuming more than 45% of the power and area in [15]. To reduce the overhead associated with the prefix-sum circuits, we propose an FTP-friendly inner-join unit, which is detailed in Figure 9.

We first observe that in ANNs, the MAC operation requires both inputs to be explicitly known at computation time. Therefore, two *fast prefix-sum* circuits are necessary to match the processing speed of both inputs. However, this is not the case with SNNs. In SNNs, the input has only two possible states (1 or 0), meaning we either accumulate or discard the weight. This distinction allows for an imbalanced processing speed between the two inputs at the prefix-sum stage.

In our design, instead of using two *fast prefix-sum* circuits as in ANNs, we employ one fast and one *laggy prefix-sum* circuit (soon be explained), as shown in Figure 9. Recall that our compression method only fetches the non-silent neurons (those that fire at least once across timesteps) from DRAM for A. Thus, as soon as we find a matched position in the *AND-result*, we can confidently accumulate the corresponding non-zero value in fiber-B at least once, without immediately knowing the exact spike information from fiber-A. This approach ensures that the throughput of processing fiber-B remains high, regardless of the processing speed of fiber-A.

In our efficient inner-join unit, each time the *fast prefix-sum* circuit generates an offset, the corresponding non-zero value of fiber-B is directly sent to a *pseudo-accumulator* for accumulation. This mechanism opportunistically assumes that the matched non-zero value in fiber-A is all 1s (i.e., the pre-synaptic neuron fires at all timesteps) to fully leverage the throughput of the *fast prefix-sum* circuit. Since the non-zero

[7]In [15], the design of the prefix-sum circuit is not described in detail. We assume it to be a tree-like prefix-sum circuit with O(log(n)) complexity that can run in one clock cycle. We define this design as the *fast prefix-sum* circuit. The size of the input and output for the prefix-sum circuit is set to 128 in both [15] and our work.

value in fiber-A is not always all 1s, we need a mechanism to ensure that the accumulation results are accurate. Instead of using the expensive *fast prefix-sum* circuit to access and verify the matched non-zero value in fiber-A, we use a much simpler circuit to generate the offset for fiber-A. We refer to this simpler prefix-sum circuit as the *laggy prefix-sum* circuit, illustrated on the left side of Figure 9. We use a group of adders to sequentially add up the prefix-sum results and store them in a small buffer. These adders operate in parallel, so the latency of generating all the offsets is equal to $\frac{\text{len(bm-A)}}{\#\text{of adders}}$.

We provide a simple walk-through example in Figure 10. First, we run the *fast prefix-sum* circuit; in every cycle, we accumulate the matched non-zero value of fiber-B and buffer it together with the matched position in small FIFOs. When the *laggy prefix-sum* circuit finishes running, a ready signal is sent out. Next, we check the non-zero value in fiber-A according to the buffered position from FIFO-mp. If the matched value is all 1s, we simply discard the current value in FIFO-B. Otherwise, we send the buffered non-zero values of fiber-B from FIFO-B to the correction accumulators. As illustrated in Figure 10, at cycle 4, we check a_2 and find its value is 1111, so we discard b_2. At cycle 5, we check a_4 and find its value is 1010, so we send b_4 to the correction accumulator for t_1 and t_3.

This example highlights the motivation and benefits of using a combination of *fast* and *laggy prefix-sums*. By utilizing a *fast prefix-sum*, we can consume fiber-B as early as possible by first accumulating it into the *pseudo-accumulator*. While waiting for the *laggy prefix-sum* to correct the accumulation results, we can proceed to fetch the next fiber-B's data into the buffer. This approach allows the latency of fetching fiber-B to be overlapped with the *laggy prefix-sum* and correction, thereby improving overall throughput. Additionally, replacing one *fast prefix-sum* with a *laggy* one reduces the overall power and area of our TPPE.

D. Other Units

After the *pseudo-accumulator* completes its computation, the accumulation results are duplicated and sent to each correction accumulator. The correction value inside each accumulator is then subtracted from the pseudo accumulation results for each timestep. Finally, the corrected results are sent to the P-LIF units to generate the output spikes. As shown in the purple box in Figure 7, we spatially unroll the LIF operations so that the output spikes for all timesteps are generated simultaneously.

LoAS utilizes a unified global buffer to hold the compressed fiber-A and fiber-B along with their bitmask representations. We adopt a FiberCache design [60], as a unified shared cache offers better utilization compared to separate ones. Each line in the global cache consists of two parts: the first part contains the bitmask representation of a fiber followed by a pointer, and the second part holds the non-zero values of that fiber. If the line manages to hold all the non-zero values, the pointer will be a NULL pointer; otherwise, it points to the location of the line where the remaining data are stored. Each PE is responsible for generating one output neuron, so we use a highly banked

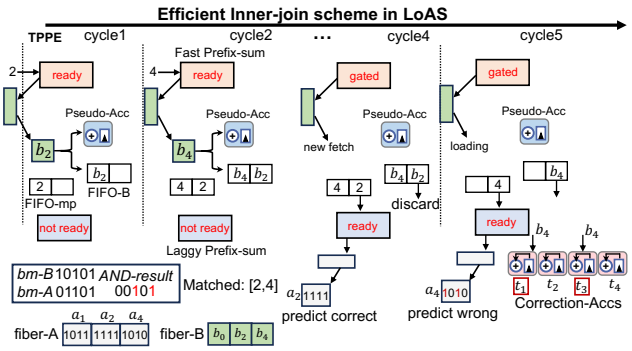

Fig. 10. Walk-through example of the proposed FTP-friendly inner-join unit. This example assumes the *laggy prefix-sum* circuit is ready after 2 cycles. FIFO-mp buffers the matched position, FIFO-B buffers the matched non-zero value of B, and 'Acc' stands for accumulator.

global cache to ensure multiple PEs can access their data concurrently. Within each bank, we fetch as many chunks as possible for one fiber in matrix A and hold them as long as possible to maximize data reuse. This can be achieved by adopting a replacement policy for the global cache, as in [30], [60]. Only one compressed row fiber of matrix B is fetched into the global cache and broadcasted to all TPPEs. We employ a compression unit similar to the one in [15], where an inverted prefix-sum circuit is used to compress the output spikes and generate their bitmask representations. As observed in [15], this compression step does not need to be performed quickly, so we equip an inverted *laggy prefix-sum* circuit to handle the compression. The scheduler is responsible for distributing the data to each TPPE through a simple swizzle-switch-based crossbar [45].

V. Experimental Methodology

Software Configuration[8] **:** For the dual-sparse SNNs, we train and compress AlexNet [25], VGG16 [48], and ResNet19 [17]. We use open-source toolchains for lottery-ticket-hypothesis (LTH)-based SNN pruning [13], [22]. The default number of timesteps T is set to 4 across all experiments. We perform 15 rounds of LTH searching, and all SNNs are trained to convergence, achieving accuracy comparable to state-of-the-art dense baselines [22]. Additionally, we select representative layers from each network to provide single-layer insights. A summary of the workloads is provided in Table II.

We also employ a simple yet effective preprocessing technique: zeroing out all presynaptic neurons that exhibit low firing activity to further increase the number of silent neurons. Specifically, we take the trained SNN and mask the neurons that produce only one output spike across all timesteps. We find that with minimal fine-tuning ($\leq$5 epochs), the original accuracy can be fully recovered, as shown in Figure 11. It is important to note that this preprocessing technique is designed to maintain the accuracy of the original workload, not to improve it. During hardware execution, the compressor will discard output neurons that have 0 or only 1 output spike. As

[8]Source codes: https://github.com/Intelligent-Computing-Lab-Yale/LoAS

TABLE II
SNN WORKLOADS. NL = NUMBER OF LAYERS. T = TIMESTEPS. AVSP{A, B} = AVERAGE SPARSITY OF MATRICES {A, B} IN (%). AVSPA-ORIGIN = ORIGINAL SPIKE SPARSITY ACROSS TIMESTEPS. AVSPA-PACKED = DENSITY OF SILENT NEURONS. AVSPA-PACKED+FT = DENSITY AFTER FINE-TUNED PREPROCESSING. M/N/K DENOTES MATRIX DIMENSIONS.

SNN	NL	T	AvSpA origin	AvSpA packed(+**FT**)	AvSpB
AlexNet(A)	7	4	81.2	71.3(**76.7**)	98.2
VGG16(V)	14	4	82.3	74.1(**79.6**)	98.2
ResNet19(R)	19	4	68.6	59.6(**66.1**)	96.8
Layer	T,M,N,K				
A-L4	4,64,256,3456		75.8	63.2(**69.7**)	98.9
V-L8	4,16,512,2304		88.1	76.5(**86.8**)	96.8
R-L19	4,16,512,2304		57.9	51.4(**55.7**)	99.1
T-HFF	4,784,3072,3072		-	- (**86.8**)	96.8

shown in Table II, this preprocessing effectively increases the number of silent neurons by up to $1.1\times$.

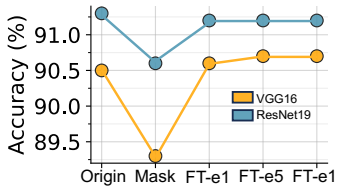

Fig. 11. Accuracy trends with fine-tuned preprocessing. 'Mask' refers to masking out all presynaptic neurons that fire only once during inference. FT-e{x} denotes fine-tuning for x epochs.

Hardware Configuration: We evaluate LoAS using the configuration detailed in Table III. In our experiments, LoAS is configured to support SNNs running with 4 timesteps. We use 16 TPPEs, each equipped with 5 accumulators (1 12-bit *pseudo-accumulator* and 4 10-bit correction accumulators) and 1 inner-join unit. Each inner-join unit contains 1 *fast prefix-sum* circuit and 1 *laggy prefix-sum* circuit. The *fast prefix-sum* circuit can generate offsets in a single cycle, while the *laggy prefix-sum* circuit, which contains 16 adders and a 128-bit buffer, generates offset results in 8 cycles. The TPPE also includes 2 depth-8 FIFOs (for correction purposes) and 2 128-bit buffers (for holding bitmasks). Additionally, a 128-byte buffer is integrated into the TPPE to hold the non-zero weights from fiber-B. We allocate 256 KB (double-buffered) for the global cache. For our workloads, this memory size is sufficient to capture good on-chip data reuse and keep all TPPEs busy.

Baseline: As discussed earlier, there are currently very few spMspM accelerators available for dual-sparse SNNs. To construct our baselines, we take the following approach: We first select three popular ANN spMspM accelerators that use the **`IP`**, **`OP`**, and **`Gust`** dataflows: SparTen [15], GoSPA [9], and Gamma [60]. We then envision a scenario where a dual-sparse SNN (with 4 timesteps and 8-bit weights) is naively run (sequentially processing its timesteps) on these accelerators. To be conservative, we place the t dimension at the innermost loop of the original **`IP`**, **`OP`**, and **`Gust`** dataflows.[9] We then make essential modifications to the three accelerators, such as removing the multipliers in these designs. To ensure a fair comparison, we configure all designs to have 16 PEs and the same global SRAM size. We refer to these three baselines as SparTen-SNN, GoSPA-SNN, and Gamma-SNN.

[9]Adding the t dimension anywhere else would increase data traffic, thus worsening performance.

TABLE III
CONFIGURATION OF THE LOAS SYSTEM.

TPPEs	16 TPPEs, 8-bit weight
Inner-join unit	16 Inner-join units
Global cache	256 KB, 16 banks, 16-way associative
Crossbars	16×16 and 16×16, swizzle-switch based
Main memory	128 GB/s over 16 64-bit HBM channels

We implemented the key components of LoAS and our hardware baselines in RTL and synthesized them using the Synopsys Design Compiler at 800MHz with 32 nm technology. A 128 GB/s High-Bandwidth Memory (HBM) module is connected to LoAS as the off-chip memory. We used CACTI 7.0 [34] to model the memory components. Additionally, we built a simulator in Python to model the cycle-level behavior of LoAS and the baselines by tiling and ordering the spMspM loop and mapping it to hardware.

VI. EXPERIMENTAL RESULTS

A. Hardware Evaluation

Overall Performances: Figure 12 compares the performance of the three dual-sparse SNN accelerator baselines (SparTen-SNN, GoSPA-SNN, and Gamma-SNN) and LoAS (with and without fine-tuned preprocessing). The speedup is measured with respect to the cycle numbers of SparTen-SNN.

The first observation is that LoAS significantly outperforms the other three accelerator baselines in all cases, achieving average speedups of $6.79\times$ (vs. SparTen-SNN), $5.99\times$ (vs. GoSPA-SNN), and $3.25\times$ (vs. Gamma-SNN). This is due to LoAS 's use of the FTP dataflow, which eliminates the intra-PE latency penalty associated with sequentially processing timesteps. Additionally, LoAS reduces both on-chip and off-chip data communications across timesteps.

The second observation is that LoAS's performance gain is highly correlated with the sparsity of matrix A. This relationship is expected since our workloads are extremely sparse in matrix B, making the overall computation matrix-A-bounded. As a result, the performance of the three baselines suffers more when sequentially processing timesteps through matrix A with lower sparsity. However, LoAS does not experience this penalty, resulting in speedups ranging from $4.08\times$ (vs. SparTen-SNN) on VGG16 (highest A sparsity) to $8.51\times$ (vs. SparTen-SNN) on ResNet19 (lowest A sparsity).

Finally, we observe that with the help of preprocessing (removing neurons that spike only once), LoAS further improves performance by an average of 20%. This improvement occurs because the preprocessing technique increases the density of silent neurons (Section IV-A), allowing LoAS to completely avoid data communications and computations associated with those neurons. Figure 12 also compares the energy efficiency of LoAS and the three baselines on different SNN workloads. It is observed that LoAS (with preprocessing) achieves ($3.68\times$,

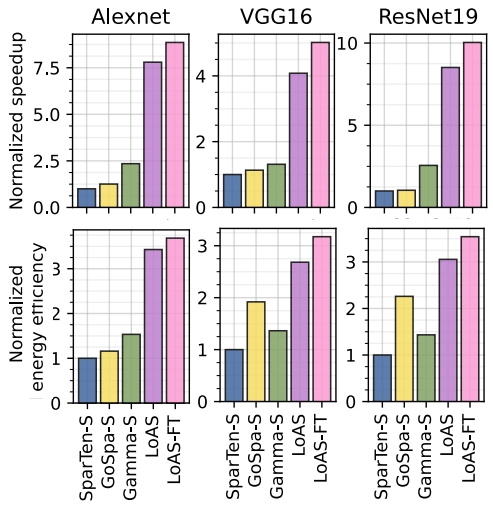

Fig. 12. Performance and efficiency comparison between SparTen-SNN, GoSPA-SNN, Gamma-SNN, and LoAS (with and without fine-tuned (FT) preprocessing) across three SNN workloads. All values are normalized to the SparTen-SNN baseline.

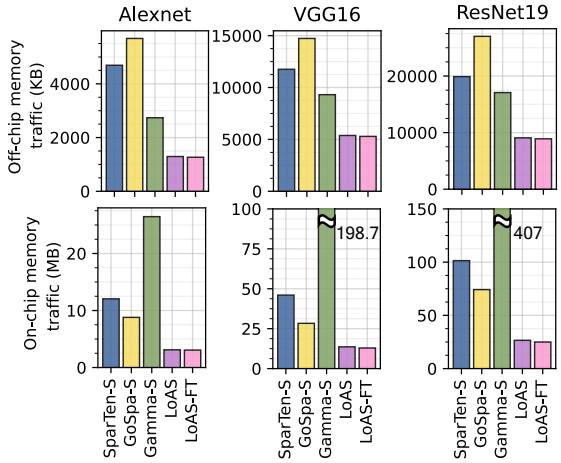

Fig. 13. Off-chip traffic (KB) and on-chip memory traffic (MB) for SparTen-SNN, GoSPA-SNN, Gamma-SNN, and LoAS (with and without preprocessing) across three SNN workloads.

3.09×, 2.40×), (3.17×, 1.50×, 2.33×), and (3.54×, 1.34×, 2.47×) higher energy efficiency over SparTen-SNN, GoSPA-SNN, and Gamma-SNN on AlexNet, VGG16, and ResNet19, respectively.

Detailed Analysis: We now explain the performance gains of LoAS. Thanks to the FTP dataflow, LoAS experiences significantly less on-chip and off-chip memory traffic compared to the three baselines. As shown in Figure 13, compared to SparTen-SNN (**IP**), LoAS achieves 3.93× (3.70×), 3.57× (2.22×), and 4.07× (2.24×) less on-chip SRAM (off-chip DRAM) access on AlexNet, VGG16, and ResNet19, respectively. This outcome is expected since **IP** dataflow designs like SparTen are known for poor input data reuse, a problem exacerbated by the extra temporal dimension (t-dim) in SNN workloads. While the FTP dataflow is a variant of the inner-product dataflow, it does not incur additional executions on the

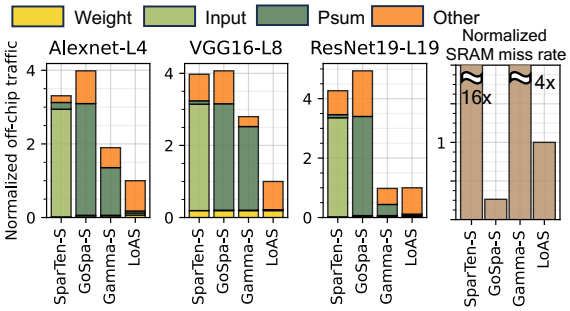

Fig. 14. Normalized off-chip traffic with breakdown for SparTen-SNN, GoSPA-SNN, Gamma-SNN, and LoAS (with preprocessing) across three SNN layer workloads. The normalized SRAM cache miss rate is also provided for the ResNet19 layer workload. All values are normalized to LoAS.

t-dim because it parallelizes the t-dim at the innermost loop.

Not surprisingly, compared to GoSPA-SNN (**OP**), LoAS still achieves 2.87× (4.49×), 2.19× (2.78×), and 2.98× (3.03×) less on-chip SRAM (off-chip DRAM) access on AlexNet, VGG16, and ResNet19, respectively. Although **OP** dataflow designs like GoSPA-SNN are known for excellent input data reuse (on average, GoSPA-SNN has 1.45× less SRAM traffic than SparTen-SNN), their inefficiency arises from the partial sum (psum) matrices. Due to the extra t-dim in SNNs, the size of psum matrices expands with the number of timesteps. GoSPA's design allocates limited on-chip memory for psum, so the psum matrices that cannot fit on-chip must be written to off-chip DRAM and later read back for reduction, leading to significant off-chip memory traffic.

Finally, compared to Gamma-SNN (**Gust**), LoAS achieves 2.16×, 1.76×, and 1.91× less DRAM access on AlexNet, VGG16, and ResNet19, respectively. This result aligns with **Gust** dataflow's ability to reduce off-chip partial row access through on-chip SRAM and mergers. However, while reducing DRAM access, Gamma-SNN's SRAM traffic is exacerbated by the t-dim in SNNs, resulting in an average of 13.4× more SRAM traffic than LoAS.

To better visualize the analysis above, we provide a memory traffic breakdown in Figure 14 for the three SNN layers in Table II. As shown in the figure, SparTen-SNN has the largest input off-chip traffic, while GoSPA-SNN has the largest psum off-chip traffic across all workloads. Among the three baselines, Gamma-SNN has the smallest off-chip traffic footprint due to the **Gust** dataflow's on-chip reuse of partial rows. GoSPA-SNN also has the largest off-chip traffic for the compressed format due to its use of the CSR format for each spike. We note that LoAS has slightly larger (2.1×) off-chip traffic for the compressed format compared to SparTen-SNN, as we need extra bitmasks to mark the positions of non-silent neurons, whereas SparTen-SNN can directly leverage the input spike trains. However, this overhead is negligible compared to LoAS's savings on off-chip traffic for other quantities.

Figure 14 also provides the normalized SRAM cache miss rate for the layer workload in ResNet19. SparTen-SNN has a 16× higher miss rate (1.47%) compared to LoAS. GoSPA-SNN has the lowest miss rate due to its Output-stationary

TABLE IV
AREA AND POWER BREAKDOWN OF LOAS (LEFT) AND ONE TPPE (RIGHT).

Components	Area (mm^2)	Power (mW)	TPPE units	Area	Power
16 TPPEs	0.96	45.1	Accumulators	2e-3	0.16
16 PLIFs	0.02	1.2	Fast Prefix	0.04	1.46
Global cache	0.80	124.5	Laggy Prefix	5e-3	0.32
Others	0.30	18.1	Others	0.01	0.88
Total	**2.08**	**188.9**	**TPPE total**	**0.06**	**2.82**

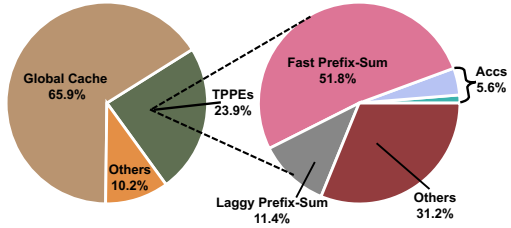

Fig. 15. On-chip power breakdown for LoAS. 'Accs' refers to the accumulators, including 1 pseudo-accumulator and 4 correction accumulators.

dataflow, though this comes at the cost of higher off-chip traffic for `psums`. Gamma-SNN has a higher SRAM miss rate than GoSPA-SNN and LoAS due to the extra t-dim enlarging the partial row traffic by t times, causing some of the extra traffic to be evicted from the on-chip SRAM. Overall, the cache miss rate results align with the off-chip traffic trends. Since all baselines have the same global cache size, the reduction in memory traffic reflects LoAS's improvements in both speedup and energy efficiency.

Area and Power: Table IV shows the area and power breakdown of LoAS with the configuration detailed in Table III. Inside each TPPE, a single *fast prefix-sum* circuit dominates both area (66.7%) and power (51.8%). The original SparTen [15] requires two *fast prefix-sum* circuits for both inputs and weights.[10] Thanks to the *laggy prefix-sum* circuits we proposed (which account for 8.3% of area and 11.4% of power), LoAS requires only one *fast prefix-sum* circuit inside each TPPE. At the system level, the global SRAM cache dominates both power and area, consistent with previous works [30], [33], [60]. Figure 15 provides a visualization of the power distribution.

B. Ablation Studies

Temporal Scalability Studies: In our experimental settings, we configured the TPPE inside LoAS to run SNNs with 4 timesteps. Most state-of-the-art SNN algorithms [10], [12] typically use a decently small timestep ($\leq$8). Therefore, we aim to understand how TPPE scales with the number of timesteps. Figure 16(a) demonstrates that TPPE scales well with the timesteps. This scalability is because all TPPE components, aside from the accumulators and the input data buffer, are agnostic to the number of timesteps. Even at 16 timesteps, the TPPE only increases its area (power) by 1.37× (1.25×) compared to 4 timesteps. We also examine how the ratio of

[10]This is not the case in SparTen-SNN. Since the input spikes function as both bitmasks and data, SparTen-SNN only requires one *fast prefix-sum* circuit.

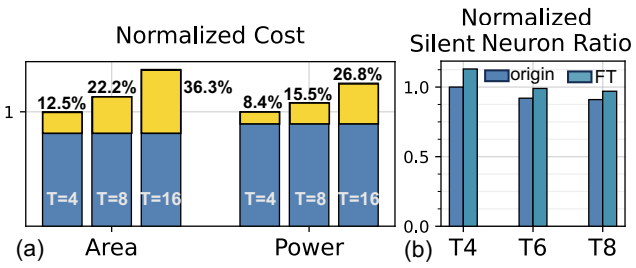

Fig. 16. (a) Scalability of TPPE with increasing timesteps. The yellow region indicates the portion that grows with the timesteps. (b) Scalability of the ratio of silent neurons (sparsity of matrix A) with increasing timesteps. All values are normalized to the original silent neuron ratio at 4 timesteps.

silent neurons in VGG16 scales with the number of timesteps. Figure 16(b) shows that with the aid of the preprocessing technique, even at 8 timesteps, we can maintain a similar ratio of silent neurons as with 4 timesteps. However, it is likely that the number of silent neurons will decrease when using even larger timestep counts (>8). This presents a challenge for LoAS when scaling up to srelatively large number of timesteps.

Scalability Study: Figure 17 further illustrates how the overall performance of LoAS scales with different factors. We first test LoAS running on VGG16 with average sparsity levels of B (weight) at 98.2% (High), 68.4% (Medium), and 25% (Low). The results show that LoAS's performance is highly sensitive to the sparsity level of B. When the sparsity scales from 98.2% to 25%, LoAS's performance decreases by approximately 88%. We also observe that LoAS's performance scales well with the number of timesteps, with only a 14% reduction in performance when the number of timesteps is doubled. Finally, we assess LoAS's scalability with layer size by comparing one layer from VGG16 to the hidden feed-forward (HFF) layer from SpikeTransformer [56] (V-L8 and T-HFF in Table II). The results demonstrate that LoAS scales effectively, even with layers of larger parameter sizes.

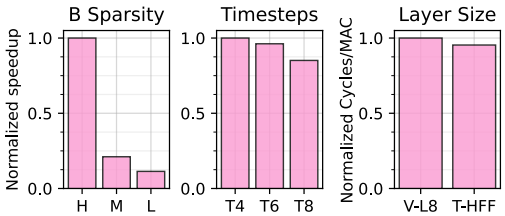

Fig. 17. Scalability of LoAS across different sparsity patterns of matrix B, number of timesteps, and layer size.

Dual-sparse SNN vs. Dual-sparse ANN:

In this work, we aim to provide insights for the community on how to accelerate spMspM for SNNs. However, it's also important to discuss the performance comparison between SNNs and ANNs. Figure 18 shows the comparison of normalized energy efficiency and memory traffic between SNNs (LoAS) and ANNs (SparTen [15] and Gamma [60]) running the VGG16 workload. We use the VGG16 workload in Table II for LoAS. The ANN version of VGG16 has 8-

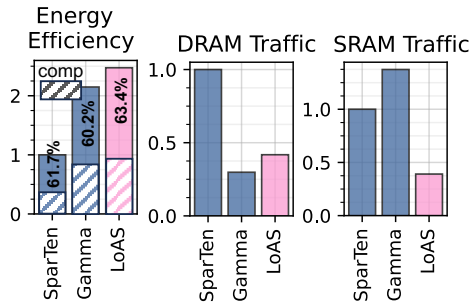

Fig. 18. Normalized energy efficiency and memory traffic comparison between SNNs (LoAS, T=4) and ANN baselines (SparTen, Gamma).

bit weights (98.2% sparsity) and activations (43.9% sparsity). Overall, the SNN running on LoAS achieves roughly 2.5× and 1.2× greater energy efficiency compared to the ANNs running on SparTen and Gamma, respectively. We observe that around 60% of energy consumption is due to data movement in both networks. Therefore, we also include a comparison of DRAM and SRAM traffic in Figure 18. The figure shows that SNNs, on average, have approximately 60% less memory traffic compared to SparTen-ANN. This reduction in memory traffic is due to the lower input bitwidth (4-bit vs. 8-bit) and higher input sparsity (79.6% vs. 43.9%), thanks to SNN's features of unary activation and sparse spike activity (Section II-B). Not surprisingly, Gamma-ANN has lower overall DRAM access compared to LoAS due to its **Gust** dataflow [60]. The tradeoff, however, is 3.5× more SRAM traffic, which explains why LoAS has slightly higher overall energy efficiency.

Dual-sparse SNN vs. Dense SNN: To demonstrate the benefits of dual-sparsity in SNNs, we compare LoAS with prior dense SNN systolic-array accelerators, PTB [28] and Stellar [32], running dense VGG16 with 4 timesteps. For a fair comparison, we set the array size for PTB to 16×4, which generates 16 full-sum outputs for 4 timesteps in parallel, the same as LoAS. We also configure Stellar with the same array size. We leverage ScaleSim [42] to estimate both baselines' memory traffic and cycle counts. The comparison is shown in Figure 19.

We first observe that LoAS has roughly 6× higher energy efficiency compared to PTB, primarily due to 3× (12.5×) less DRAM (SRAM) traffic. Compared to Stellar, LoAS achieves approximately 2.5× higher energy efficiency, along with 2.7× (6.6×) less DRAM (SRAM) traffic. Additionally, LoAS shows a 46.9× speedup over PTB, mainly due to the data sparsity and the difference between PTB's partially temporal-parallel mechanism (Section II-E) and LoAS's fully temporal-parallel mechanism. We also observe that Stellar outperforms PTB across all matrices, primarily due to Stellar's optimized spatiotemporal row-stationary dataflow and spike-skipping technique. However, compared to Stellar, LoAS still achieves approximately 7.1× speedup due to its ability to leverage dual-sparsity. It is important to note that we do not compare with SpinalFlow [35] due to its temporal encoding achieving limited accuracy on challenging tasks [6], [28].

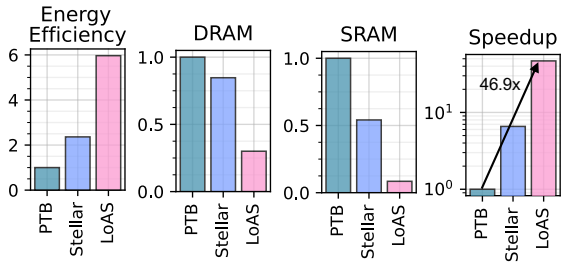

Fig. 19. Normalized performance comparison between the dual-sparse SNN accelerator (LoAS) and dense SNN accelerator baselines (PTB, Stellar).

VII. Related Work

In addition to the prior dense SNN accelerator works discussed in Section II-E, there are also prior works that attempt to leverage sparsity in SNNs. In [3], a neuron filter unit is used to fetch weights only when there is a 1-spike. However, dual-sparsity (both spike and weight sparsity) is not considered. In [2], dual-sparsity in SNNs is considered to skip unmatched computations, but the weights and spikes are fetched in dense format without any compression from off-chip memory, thereby failing to reduce data movement costs. In contrast, LoAS leverages dual-sparsity in SNNs for both computation and data movement.

As discussed earlier, PTB processes timesteps in a partially parallel manner. Even if PTB is reconfigured to run all timesteps in parallel (time-window=1), it still differs from LoAS in loop ordering. In PTB's loop ordering, the t-dim is placed between the m-dim and n-dim, while LoAS places the t-dim in the innermost loop. As discussed in Section III, LoAS's loop ordering brings greater efficiency in spMspM operations. Moreover, PTB is designed to accelerate workloads with time-series data from DVS sensors [29], where the number of timesteps is typically large (>100). For our workloads, where the number of timesteps is small (≤8), PTB experiences low hardware utilization. In [31], processing timesteps in parallel is also explored. However, the focus is on temporal-coded SNN workloads, and loop ordering is not discussed. Finally, as mentioned in Section II-E, Stellar [32] is another work that attempts to process timesteps in parallel. However, it targets non-LIF, FS-coded SNNs and does not support dual-sparsity.

VIII. Conclusion

In this work, we observed that naively running dual-sparse SNNs on existing spMspM accelerators results in suboptimal efficiency due to the latency and memory traffic penalties associated with processing timesteps. To address this issue, we proposed a fully temporal-parallel dataflow (FTP) that eliminates these problems. To maximize the benefits of FTP, we introduced FTP-friendly spike compression and an inner-join mechanism. Additionally, we developed LoAS, a novel architecture that exemplifies the FTP dataflow. With the support of both FTP-friendly compression and inner-join, LoAS achieves significant speedup (up to 8.51×) and energy reduction (up to 3.68×) compared to prior dual-sparse accelerator baselines.

APPENDIX

A. Abstract

This artifact evaluation focuses on the key results related to silent neurons and the fine-tuning algorithm discussed in Section VI, specifically in Table II and Figure 11. The code required to reproduce the results presented in the paper can be found at https://github.com/RuokaiYin/LoAS/tree/main/artifact.

B. Artifact check-list (meta-information)

- **Algorithm:** Profiling and fine-tuning SNN models
- **Dataset:** CIFAR-10 (downloadable from PyTorch)
- **Model:** VGG16, ResNet19, and AlexNet (all provided within the source code)
- **Hardware:** x86_64 machine. GPU: NVIDIA RTX 2080 Ti or NVIDIA V100
- **Output:** Table II and Figure 11.
- **Experiments:** Ratio of silent neurons in SNNs, classification accuracy recovery after fine-tuning
- **How much disk space required (approximately)?:** 1GB (excluding software dependencies)
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes
- **How much time is needed to complete experiments (approximately)?:** 40 minutes (2080Ti)
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** https://doi.org/10.6084/m9.figshare.27012058.v1
- **Code licenses (if publicly available)?:** MIT License

C. Description

1) How to Access: First, clone the source code from https://github.com/RuokaiYin/LoAS.git. Then, install the dependencies listed in the `requirements.txt` file. Finally, navigate to the `artifact` sub-directory. Detailed instructions can be found in the `README.md` files within this sub-directory.

2) Hardware Dependencies: The code has been tested on an x86_64 machine. Reproducing Table II does not require a GPU, but reproducing Figure 11 does. The code has been tested using NVIDIA RTX 2080 Ti and V100 GPUs.

3) Software Dependencies: The required packages are listed in the `requirements.txt` file. `Python` 3.9.7 and `CUDA` 11.1+ are also required.

4) Datasets: The dataset used for fine-tuning is CIFAR-10, which is available from PyTorch. Automatic downloading is enabled in the source code, so CIFAR-10 will be automatically downloaded and stored in the `./dataset` directory.

5) Models: We use deep SNN models based on the VGG16, ResNet19, and AlexNet architectures. Users do not need to download the models separately, as we have provided the model definitions, checkpoints, and post-processed models.

D. Installation

After cloning the GitHub repository, navigate to the repository and install all the package dependencies listed in `requirements.txt`. Below is an example of how to install the dependencies using `pip` on a Linux command line:

```
pip install -r requirements.txt
```

To reproduce the results for Table II, navigate to the following sub-directory and follow the instructions in the `README.md` file:

```
cd artifact/Table-II
```

Similarly, to reproduce Figure 11, navigate to the following sub-directory and follow the instructions in the `README.md` file:

```
cd artifact/Fig-11
```

E. Experiment Workflow

We use bash scripts to automate the workflow for generating results for artifact evaluation. Within each sub-directory (e.g., `artifact/Table-II` and `artifact/Fig-11`), there is a `run.sh` script. To run the script, simply execute:

```
/bin/bash run.sh
```

After the script completes, the results will be available in the corresponding sub-directory. Additional details can be found in the `README.md` file within each sub-directory.

F. Evaluation and Expected Results

After running the scripts as described in Section E, the generated figures and text files will be available locally within each sub-directory. For Table II, the generated result is saved as `tableII_artifact.txt`. The layer-wise and global sparsity information is recorded in this text file for all three networks. Each network will output two sections of results, corresponding to conditions with and without fine-tuning. Additional details are provided in the `README.md` file under the section **How to Interpret the Generated Results**, which explains how to interpret the results and match them with Table II in the original paper. For Figure 11, the generated figure is named `ft_accuracy.pdf`. More information can also be found in the `README.md` file.

G. Notes

There may be minor rounding deviations (approximately 0.1%) in the reproduced results for TableII. Additionally, some variations in the fine-tuned accuracy may occur when reproducing Figure11. However, the overall trends will remain consistent with those presented in the original paper. For more details, please refer to the `README.md` files.

The codes submitted for evaluation are CUDA GPU-agnostic. The specific GPU type is provided only to minimize fluctuations between fine-tuning runs.

H. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

REFERENCES

[1] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla *et al.*, "TrueNorth: design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[2] Q. Chen, C. Gao, and Y. Fu, "Cerebron: a reconfigurable architecture for spatiotemporal sparse spiking neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 10, pp. 1425–1437, 2022.

[3] Q. Chen, G. He, X. Wang, J. Xu, S. Shen, H. Chen *et al.*, "A 67.5 μJ/prediction accelerator for spiking neural networks in image segmentation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 2, pp. 574–578, 2021.

[4] Y. Chen, Z. Yu, W. Fang, T. Huang, and Y. Tian, "Pruning of deep spiking neural networks through gradient rewiring," *arXiv preprint arXiv:2105.04916*, 2021.

[5] D. V. Christensen, R. Dittmann, B. Linares-Barranco, A. Sebastian, M. Le Gallo, A. Redaelli *et al.*, "2022 roadmap on neuromorphic computing and engineering," *Neuromorphic Computing and Engineering*, vol. 2, no. 2, p. 022501, 2022.

[6] I. M. Comsa, K. Potempa, L. Versari, T. Fischbacher, A. Gesmundo, and J. Alakuijala, "Temporal coding in spiking neural networks with alpha synaptic function," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 8529–8533.

[7] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday *et al.*, "Loihi: a neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[8] P. Dayan and L. F. Abbott, *Theoretical neuroscience: computational and mathematical modeling of neural systems*. MIT press, 2005.

[9] C. Deng, Y. Sui, S. Liao, X. Qian, and B. Yuan, "Gospa: an energy-efficient high-performance globally optimized sparse convolutional neural network accelerator," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1110–1123.

[10] S. Deng, Y. Li, S. Zhang, and S. Gu, "Temporal efficient training of spiking neural network via gradient re-weighting," *arXiv preprint arXiv:2202.11946*, 2022.

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

[12] W. Fang, Z. Yu, Y. Chen, T. Huang, T. Masquelier, and Y. Tian, "Deep residual learning in spiking neural networks," *Advances in Neural Information Processing Systems*, vol. 34, pp. 21 056–21 069, 2021.

[13] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *arXiv preprint arXiv:1803.03635*, 2018.

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

[15] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.

[16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz *et al.*, "EIE: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[18] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik *et al.*, "ExTensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[19] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 674–687.

[20] S. Kim, S. Park, B. Na, and S. Yoon, "Spiking-yolo: spiking neural network for energy-efficient object detection," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 07, 2020, pp. 11 270–11 277.

[21] Y. Kim, J. Chough, and P. Panda, "Beyond classification: Directly training spiking neural networks for semantic segmentation," *Neuromorphic Computing and Engineering*, vol. 2, no. 4, p. 044015, 2022.

[22] Y. Kim, Y. Li, H. Park, Y. Venkatesha, R. Yin, and P. Panda, "Exploring lottery ticket hypothesis in spiking neural networks," in *European Conference on Computer Vision*. Springer, 2022, pp. 102–120.

[23] Y. Kim and P. Panda, "Revisiting batch normalization for training low-latency deep spiking neural networks from scratch," *Frontiers in neuroscience*, 2021.

[24] Y. Kim, H. Park, A. Moitra, A. Bhattacharjee, Y. Venkatesha, and P. Panda, "Rate coding or direct coding: Which one is better for accurate, robust, and energy-efficient spiking neural networks?" in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 71–75.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[26] H.-T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, 1982.

[27] C. Lee, A. K. Kosta, A. Z. Zhu, K. Chaney, K. Daniilidis, and K. Roy, "Spike-flownet: event-based optical flow estimation with energy-efficient hybrid neural networks," in *European Conference on Computer Vision*. Springer, 2020, pp. 366–382.

[28] J.-J. Lee, W. Zhang, and P. Li, "Parallel Time Batching: Systolic-array acceleration of sparse spiking neural computation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 317–330.

[29] H. Li, H. Liu, X. Ji, G. Li, and L. Shi, "CIFAR10-DVS: an event-stream dataset for object classification," *Frontiers in neuroscience*, vol. 11, p. 309, 2017.

[30] Z. Li, J. Li, T. Chen, D. Niu, H. Zheng, Y. Xie *et al.*, "Spada: Accelerating sparse matrix multiplication with adaptive dataflow," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 747–761.

[31] F. Liu, W. Zhao, Z. Wang, Y. Chen, T. Yang, Z. He *et al.*, "SATO: spiking neural network acceleration via temporal-oriented dataflow and architecture," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1105–1110.

[32] R. Mao, L. Tang, X. Yuan, Y. Liu, and J. Zhou, "Stellar: Energy-efficient and low-latency SNN algorithm and hardware co-design with spatiotemporal computation," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 172–185.

[33] F. Muñoz-Martínez, R. Garg, M. Pellauer, J. L. Abellán, M. E. Acacio, and T. Krishna, "Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 252–265.

[34] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, 2009.

[35] S. Narayanan, K. Taht, R. Balasubramonian, E. Giacomin, and P.-E. Gaillardon, "Spinalflow: An architecture and dataflow tailored for spiking neural networks," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 349–362.

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

[37] E. O. Neftci, B. U. Pedroni, S. Joshi, M. Al-Shedivat, and G. Cauwenberghs, "Stochastic synapses enable efficient brain-inspired learning machines," *Frontiers in neuroscience*, vol. 10, p. 241, 2016.

[38] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[39] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH computer architecture news*, vol. 45, no. 2, pp. 27–40, 2017.

[40] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das *et al.*, "SIGMA: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 58–70.

[41] K. Roy, A. Jaiswal, and P. Panda, "Towards spike-based machine intelligence with neuromorphic computing," *Nature*, 2019.

[42] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of dnn accelerators using scale-sim," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 58–68.

[43] A. Sarma, S. Singh, H. Jiang, A. Pattnaik, A. K. Mishra, V. Narayanan *et al.*, "Exploiting activation based gradient output sparsity to accelerate backpropagation in cnns," *arXiv preprint arXiv:2109.07710*, 2021.

[44] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, "Going deeper in spiking neural networks: Vgg and residual architectures," *Frontiers in neuroscience*, vol. 13, p. 95, 2019.

[45] K. Sewell, R. G. Dreslinski, T. Manville, S. Satpathy, N. Pinckney, G. Blake *et al.*, "Swizzle-switch networks for many-core systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 2, no. 2, pp. 278–294, 2012.

[46] L. Shi, J. Pei, N. Deng, D. Wang, L. Deng, Y. Wang *et al.*, "Development of a neuromorphic computing system," in *2015 IEEE international electron devices meeting (IEDM)*. IEEE, 2015, pp. 4–3.

[47] Y. Shi, L. Nguyen, S. Oh, X. Liu, and D. Kuzum, "A soft-pruning method applied during training of spiking neural networks for in-memory computing applications," *Frontiers in neuroscience*, vol. 13, p. 405, 2019.

[48] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.

[49] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 766–780.

[50] C. Stöckl and W. Maass, "Optimized spiking neurons can classify images with high accuracy through temporal coding with two spikes," *Nature Machine Intelligence*, vol. 3, no. 3, pp. 230–238, 2021.

[51] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[52] D. Wu, J. Li, R. Yin, H. Hsiao, Y. Kim, and J. San Miguel, "UGEMM: Unary computing architecture for gemm applications," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 377–390.

[53] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, "Sparseloop: An analytical approach to sparse tensor accelerator modeling," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1377–1395.

[54] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, "Spatio-temporal backpropagation for training high-performance spiking neural networks," *Frontiers in neuroscience*, vol. 12, p. 331, 2018.

[55] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, and L. Shi, "Direct training for spiking neural networks: Faster, larger, better," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 1311–1318.

[56] M. Yao, J. Hu, Z. Zhou, L. Yuan, Y. Tian, B. Xu *et al.*, "Spike-driven transformer," *Advances in neural information processing systems*, vol. 36, 2024.

[57] R. Yin, Y. Kim, Y. Li, A. Moitra, N. Satpute, A. Hambitzer *et al.*, "Workload-balanced pruning for sparse spiking neural networks," *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2024.

[58] R. Yin, Y. Li, A. Moitra, and P. Panda, "MINT: Multiplier-less integer quantization for energy efficient spiking neural networks," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 830–835.

[59] R. Yin, A. Moitra, A. Bhattacharjee, Y. Kim, and P. Panda, "SATA: Sparsity-aware training accelerator for spiking neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 6, pp. 1926–1938, 2022.

[60] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 687–701.

[61] W. Zhang and P. Li, "Temporal spike sequence learning via backpropagation for deep spiking neural networks," *NeurIPS*, 2020.

[62] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.

[63] H. Zheng, Y. Wu, L. Deng, Y. Hu, and G. Li, "Going deeper with directly-trained larger spiking neural networks," in *AAAI*, 2021.