SATA: Sparsity-Aware Training Accelerator for Spiking Neural Networks

Ruokai Yin, Abhishek Moitra, Abhiroop Bhattacharjee, Youngeun Kim, and Priyadarshini Panda
Department of Electrical Engineering, Yale University, USA

***Abstract*—Spiking Neural Networks (SNNs) have gained huge attention as a potential energy-efficient alternative to conventional Artificial Neural Networks (ANNs) due to their inherent high-sparsity activation. Recently, SNNs with backpropagation through time (BPTT) have achieved a higher accuracy result on image recognition tasks than other SNN training algorithms. Despite the success from the algorithm perspective, prior works neglect the evaluation of the hardware energy overheads of BPTT, due to the lack of a hardware evaluation platform for this SNN training algorithm. Moreover, although SNNs have long been seen as an energy-efficient counterpart of ANNs, a quantitative comparison between the training cost of SNNs and ANNs is missing. To address the aforementioned issues, in this work, we introduce SATA (Sparsity-Aware Training Accelerator), a BPTT-based training accelerator for SNNs. The proposed SATA provides a simple and re-configurable systolic-based accelerator architecture, which makes it easy to analyze the training energy for BPTT-based SNN training algorithms. By utilizing the sparsity, SATA increases its computation energy efficiency by $5.58\times$ compared to the one without using sparsity. Based on SATA, we show quantitative analyses of the energy efficiency of SNN training and make a comparison between the training cost of SNNs and ANNs. The results show that, on Eyeriss-like systolic-based architecture, SNNs consume $1.27\times$ more total energy with considering sparsity (spikes, gradient of firing function, and gradient of membrane potential) when compared to ANNs. We find that such high training energy cost is from time-repetitive convolution operations and data movements during backpropagation. Moreover, to guide the future SNN training algorithm design, we provide several observations on energy efficiency with respect to different SNN-specific training parameters.**

***Index Terms*—Neuromorphic computing, Spiking neural networks, Computer architecture, Energy-efficiency analysis, Artificial neural networks.**

I. Introduction

Recent advances in deep learning have made artificial neural networks (ANNs) better candidates than humans for many tasks involving the processing of images, videos, and natural language [1]. Besides ANNs, Spiking neural networks (SNN), inspired by the processing paradigm of the human brain, are gaining popularity [2]–[4]. SNNs primarily bring benefits to deep learning applications from two aspects: (1) the capture of both temporal and spatial information, whereas most ANNs lack the information from the time domain due to their spatial feedforward characteristics, (2) the energy-efficient implementations on hardware, since SNNs do not require multipliers for Multiply and Accumulate (MAC) operations during inference time. The inherent single-bit resolution of spikes also reduces the cost of memory communication.

Recently, there has been a growing interest in the field of SNN training algorithms. Works such as [5], [6] have shown that Back Propagation Through Time (BPTT) [7] can achieve higher accuracy performance than Spike Time Dependent Plasticity (STDP) [8], [9] and faster convergence than ANN-SNN conversion methods [2], [10]–[12]. Despite the success from the algorithm perspective, these works neglect the evaluation of the hardware energy overheads of BPTT and thus fail to build the connection between the algorithm superiority and hardware efficiency in SNN training.

However, evaluating the hardware efficiency of SNN algorithms is not a direct task for algorithm researchers. Prior SNN algorithm works [10] use analytical methods to evaluate the hardware energy overheads of BPTT that neglect the underlying hardware architectural details leading to inaccurate estimations. In fact, a hardware evaluation platform for BPTT-based SNN training is missing in the SNN research community. Moreover, despite the fact that SNNs have been long treated as an energy-efficient counterpart of ANNs, there are very limited prior works comparing the energy difference between the two types of networks.

In [13], a rate encoding-based inference accelerator has been proposed and the inference energy for SNNs has been provided. However, the focus of the work is to optimize the NoCs for mapping SNNs onto the chip and has not given an energy comparison between SNNs and ANNs. In the prior work [14], another inference accelerator for SNN has been proposed, however, the accelerator is based on temporal encoding which is different from the rate encoding that BPTT relies on. The work also provides the inference energy difference between SNNs and ANNs, however, a training energy comparison between two types of neural networks is still missing in the community. Recently, the work [15] has proposed a custom-tailored hardware architecture for SNN training that is highly SNN-tailored and targets performance boosting. For example, it utilizes LUT-based convolutions and has complex engines to compress the memory. With the complex and tailored design, it becomes difficult for researchers to make energy analyses of the different SNN training topologies on it. A fair comparison of the training energy cost between SNNs and ANNs is also hard to make on SNN-crafted architecture design. Hence, the work is unsuitable for general-purpose hardware evaluation of BPTT training. Moreover, they merely consider spike and spike gradient level sparsity that insufficiently captures the

Ruokai Yin, Abhishek Moitra, Abhiroop Bhattacharjee, Youngeun Kim, and Priyadarshini Panda are with the Department of Electrical Engineering, Yale University, New Haven, CT, USA. (e-mail: ruokai.yin@yale.edu).

TABLE I
COMPARISON BETWEEN SATA AND OTHER SNN ACCELERATORS WORK. S DENOTES THE SPIKE ACTIVATION, ∇f DENOTES THE GRADIENT OF FIRING FUNCTION AND ∇U DENOTES THE GRADIENT OF MEMBRANE POTENTIAL.

Accelerator	**Type**	**Sparsity**	**Arch-design**
Spinalflow [14]	Inference	S	Systolic array-based
Shenjing [13]	Inference	S	SNN-crafted
H2Learn [15]	Training	∇f	SNN-crafted
SATA	Training	S, ∇f, ∇U	Systolic array-based

repercussions of BPTT on hardware.

Motivated by the aforementioned problem, we propose SATA (Sparsity-Aware Training Accelerator), an Eyeriss-inspired [16] general-purpose training accelerator for BPTT-based SNNs. The focus of SATA is to simulate a simple and reconfigurable accelerator design, which simplifies the analysis of the training energy for BPTT-based training algorithms. Compared to prior works, SATA has several differences. Firstly, unlike prior works [15], the SNN training architecture is more general and not overly optimized to a particular SNN architecture. This enables scalable hardware evaluation across a wide range of SNN models. Secondly, we show that sparsity in the gradients of membrane potential (∇U) can be leveraged to further improve the energy efficiency of SNN training. Moreover, Our general-purpose implementation approach additionally enables us to perform a fair comparison between ANN and SNN training. Finally, our training accelerator can be used as a benchmarking tool to evaluate the hardware training cost of SNNs. Table I summarizes our contributions with respect to prior digital SNN accelerator works that are most related to our works [13]–[15].

Another key point to optimize the energy efficiency of SNNs is to use the energy as a metric directly in training algorithm design. But today, a platform that can make a sparsity-aware estimation of the energy cost for SNN training is missing. We, therefore, propose a framework to estimate the computation and data movement energy in SNN training based on the architecture of SATA. The framework extends the energy estimation method proposed in [17] to further consider the impact of various groups of sparsity (S, ∇f, and ∇U) and SNN-specific training parameters, for example, the number of timesteps.

We summarize our contributions as follows:

1) We present SATA, a sparsity-aware BPTT-based training accelerator for SNNs. The simple and highly reconfigurable design makes it easy to perform a training energy analysis on SATA. The systolic array-based architecture also makes SATA the right baseline to make energy cost comparisons between SNN and ANN training. SATA also comprehensively captures three groups of sparsity (spike S, the gradient of firing function ∇f, and the gradient of membrane potential ∇U) to optimize the training energy efficiency. By utilizing those sparsities, SATA increases its computation energy efficiency by $5.58\times$ compared to the one without using sparsity. Along with SATA, we also propose an energy estimation framework for SNN training based on SATA, which is publicly available [18].
2) We provide a training energy cost comparison between SNNs on SATA and ANNs on our baseline modified from the 8-bit version of Eyeriss [16]. Our result shows that, on Eyriss-like systolic-based architecture, without considering sparsity for both SNNs and ANNs, SNNs consume $1.35\times$ more energy in total training energy compared to ANNs. Specifically, non-sparse SNNs consume $3.28\times$ more energy on computation and $1.28\times$ more energy on memory access compared to non-sparse ANNs. By further considering the sparsity (S, ∇f, ∇U), SNNs now consume $1.27\times$ more total training energy compared to ANNs. Specifically, sparse SNNs consume $1.19\times$ more energy on computation and $1.27\times$ more energy on memory access compared to sparse ANNs.
3) We also showcase various ablation studies on how the three groups of sparsity (S, ∇f, ∇U) change with different SNN training settings (for example, datasets, timestep, and network depth) and the training energy of SNNs resulting from the change of sparsity. We show that the total SNN training energy exponentially increases in a large timestep regime ($T > 32$). We also show that by having more sparsity in ∇U, we can finally achieve less computation energy for SNN training compared to ANNs.

II. RELATED WORK

There has been a wide range of works that have proposed accelerator designs to carry out SNN inference showing a high degree of parallelism, throughput, and energy-efficiency [19]–[23]. These include accelerators with a fully-digital architecture, such as IBM's TrueNorth processor [19], as well as ones in which synaptic computational cores comprise of analog memristive crossbars, such as Resparc [21]. While most of the works focus on inference-only accelerator designs, some like Intel's Loihi processor account for SNN training using STDP learning rule [2], [24]. Furthermore, the TrueNorth and Loihi processors are highly optimized to facilitate asynchronous spike communications with the objective of improving the performance of the deployed SNNs having a specific type of architecture, different from the conventional ones. However, they lack general applicability since they do not have support to benchmark a wide variety of SNNs, particularly SNNs trained by standard BPTT learning rules. Thus, it is imperative to have a general-purpose SNN training accelerator framework that can support the training and inference of a plethora of SNN architectures that is emerging from recent SNN algorithm studies.

There is also a huge volume of work centered around SNNs that claim SNNs to be an energy-efficient alternative to ANNs due to high sparsity in input spikes [2], [10], [20], [21], [25]. But recently, an inference framework implemented in an *Eyeriss*-like systolic-array hardware tailored for SNNs called SpinalFlow [14] has shown that standard rate-coded SNNs with modest spike-rates exhibit significantly lower efficiency than corresponding accelerators for ANNs. Note, *Eyeriss* [16] follows a von-Neumann mode of neural computation widely

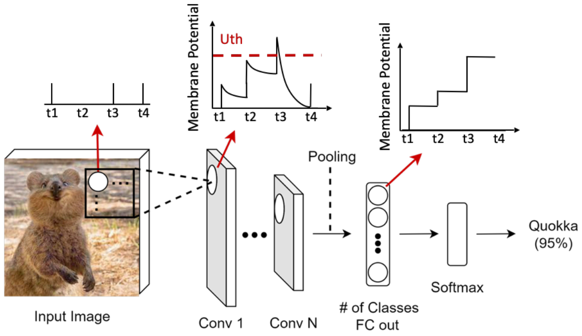


Fig. 1. An illustrative example of SNN. All intermediate neurons output rate-coded spikes and leak potential at each timestep t. Output neurons at the last fully connected layer will not generate spikes and only accumulate potential without leaking.

adopted in modern accelerators and enables us to optimize over different design choices such as type of dataflow, computation reuse, and skipping zero computations. The primary cause behind the inefficiency of SNNs can be attributed to the storage and movement of membrane potentials over multiple timesteps during inference. With this in mind, the next steps include developing a similar hardware evaluation framework that can yield a realistic estimation of hardware energy and latency associated with training a wide range of SNN architectures over multiple timesteps.

To this end, our SATA framework is the first to show that the inherent sparsity in SNNs associated with the spikes and their gradients are alone insufficient to yield training energy efficiency with respect to baseline ANN models. SNN training for conventional architectures, in fact, incurs huge overheads in terms of memory accesses and computations compared to ANNs, thereby making them highly energy-inefficient. Based on the conclusion and discussion posed in this work through the extensive study conducted on SATA and the energy-analysis tool that we propose, we hope that the future SNN algorithm research can be directed towards enhancing specific forms of sparsity (that impact computation cost largely) and avoiding certain values of SNN-specific training parameters (that impact memory cost largely) during training that can enable SNNs to be energy-efficient.

III. Background

A. SNN Basics

The network architectures for SNNs are very similar to that of ANNs, except that all ReLU-based neurons are replaced by simple neuron models to emulate biological neuron behaviors. This includes the update of membrane potential and the firing of spikes. Each pixel of input image fed to SNNs is encoded into a spike train that extends across the total timesteps T. Poisson distribution-based rate encoding scheme [26] is primarily used [5], [6], where each pixel fires a spike train with a frequency proportional to its intensity. Noted that throughout the text, we refer to a timestep to the minimum time unit in SNN, in which a neuron updates the membrane potential according to the input and produces a spike if the threshold is reached.

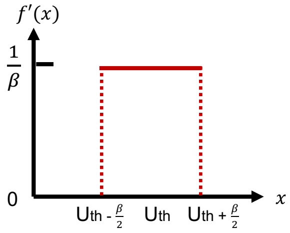


Fig. 2. Illustration of the curve to approximate the derivative of spike firing function. The derivative equals $\frac{1}{\beta}$ when the membrane potential is inside the β range around the firing threshold U_{th} during the forward propagation and equals zero otherwise. We name β as the firing width.

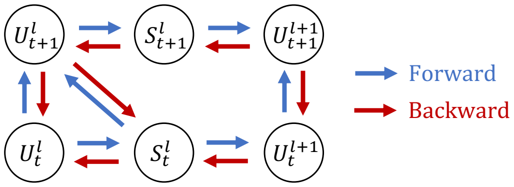


Fig. 3. Illustration of how BPTT works. During the forward propagation, the neuron of layer l at timestep $t+1$ will retain the potential U^l_t and receive the spike S^l_t from the previous timestep, which is considered as the propagation in the temporal domain. In the spatial domain, the neuron at layer $l+1$ receives the spike S^l from the previous layer. Forward paths are shown in blue arrows. For the backpropagation, the paths are reversed and shown in red arrows.

One of the most popular neuron models is a Leaky-Integrate-and-Fire (LIF) model. The LIF neuron receives binary spike inputs at every timestep t. After receiving the spikes, synaptic weights corresponding to each input spike are accumulated in the neuron's membrane potential U. The potential leaks at every timestep, based on the leaking factor α. When the potential reaches the pre-set threshold U_{th}, the neuron fires an output spike and resets its membrane potential. We model LIF using the explicit iterative expression:

$$U^l_t = \alpha U^l_{t-1}(1 - S^l_{t-1}) + W^{l-1}S^{l-1}_t, \tag{1}$$

$$S^l_t = f(U^l_t - U_{th}), \tag{2}$$

where U^l_t and S^l_t represent the potential and spike matrices of layer l at timestep t. Also, W^{l-1} is the weight matrix from previous layer $l-1$. And $f(\cdot)$ is the Heaviside step function, where $f(x) = 1$ when $x > 0$, otherwise $f(x) = 0$. Fig. 1 shows an example SNN for an image classification task.

B. BPTT for SNNs

Recently, backpropagation through time (BPTT) algorithm [7], [27] has become popular to train SNN models from scratch. BPTT shrinks the training accuracy gap between SNNs and ANNs by backpropagating gradients from both spatial and temporal domains, illustrated by Fig. 3. The spike gradient ∇S and potential gradient ∇U at layer l and time t with respect to loss function L are expressed as:

$$\nabla S^l_t = \nabla U^l_{t+1}(-\alpha U^l_t) + \nabla H^{l+1}_t, \tag{3}$$

$$\nabla U^l_t = \nabla U^l_{t+1}\alpha(1 - S^l_t) + \nabla S^l_t f'(U^l_t), \tag{4}$$

where ∇H^{l+1}_t represents the gradients backpropagated from the layer $l+1$ at timestep t, which can be formulated as:

$$\nabla H^{l+1}_t = W^{l+1}\nabla U^{l+1}_t. \tag{5}$$

We use the function proposed in [5] to approximate the derivative of Heaviside step function, where $f'(x) = \frac{1}{\beta}$ when $|x - U_{th}| < \frac{\beta}{2}$, otherwise $f'(x) = 0$. The approximated derivative of step function is illustrated in Fig. 2. The weight update for layer l with learning rate γ follows the rule below:

$$W^l = W^l - \gamma \sum_t \nabla U_t^l S_t^{l-1}. \tag{6}$$

IV. Pitfalls and Opportunities in BPTT

A. Pitfalls in Memory Access and Computation

Although the BPTT training algorithm boosts accuracy performance for SNNs, it deteriorates the hardware performance of the learning process by attaching memory consumption overheads. Since BPTT requires the information of spikes (S) and membrane potential (U) for every timestep during the forward propagation to conduct backpropagation, it introduces time-steps-related memory storage and communication overheads. As we will show in Section VII, the overhead scales exponentially with larger timesteps.

Besides memory overhead, energy overheads also exist in the SNN training computations. BPTT requires extra computations for updating the gradients (∇H) through layers by carrying out the same multi-bit multiply-accumulate (MAC) operation as ANNs and repeating it across all timesteps. The update of learnable parameters W also repeats for each timestep to accumulate the temporal information. Besides the gradients of learnable parameters and activation, SNN also needs ancillary computations for gradients of membrane potential (∇U) as shown in Eqn. (4).

B. Opportunities in Sparsity

Fortunately, SNNs naturally exhibit high sparsity. By leveraging the sparsity in spikes S, we can reduce $\sim 94\%$ of the MAC operations (reduced to accumulation-only operation in SNNs) in Eqn. (1) during the forward propagation of training (shown in Table VI in Section VII). A similar number of gradients accumulated through timesteps in Eqn. (6) will also decrease.

As we discussed above, $f'(U_t^l) = 0$ if U_t^l is out of the β-width U_{th} centered region. If $f'(U_t^l) = 0$, we can skip Eqn. (3) that is the computation of ∇S_t^l. Further, the add operation (corresponding to the second term in RHS) in Eqn. (4) as well as the fetch of U_t^l can be eliminated. We define this as the sparsity in the gradient of the firing function (∇f).

Finally, if $\nabla U_{t+1}^l = 0$, we can skip the convolution computation of ∇H_t^l. We define this as the sparsity in the gradient of potentials (∇U). We summarize the sparsity-aware version of gradient calculation for membrane potential below:

$$\nabla U_t^l = \begin{cases} \alpha \nabla U_{t+1}^l (1 - S_t^l) & \text{if sparsity in } \nabla f \\ \nabla U_{t+1}^l \alpha (1 - S_t^l) + \nabla S_t^l f'(U_t^l) & \text{otherwise} \end{cases} \tag{7}$$

We will utilize these opportunities to guide the architecture design in the next section.

V. Architecture Design

A. Architecture and Dataflow of SATA

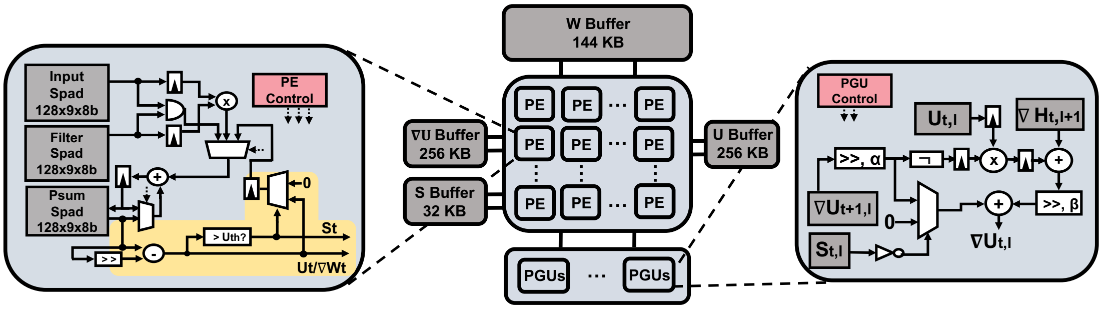
Similar to ANN training, convolution accounts for the majority of the computation workload in SNN training. Thus, we follow the spatial architecture design (doing MAC operations inside a processing element (PE) array) utilized by previous ANN accelerator works [16], [28] for SATA. However, the PE design for SATA needs to consider the difference in data representation and computation units across distinct convolution stages of SNN training, which will be explained later. Separate computation units for updating the gradients of membrane potential called potential gradient units (PGUs) are attached to simplify the design of PEs. Further, since computations in SNNs repeat for multiple timesteps, spatial dataflow that suits previous ANN accelerators, for example, row-stationary dataflow will no longer be energy efficient due to the repeated data communication cost between computation units and memory. To this end, SATA adopts a tailored temporal dataflow (namely, the combination of weight-stationary in [16] and tick-batch in [14]) for SNN-training to reduce the total energy overhead. We call this dataflow *temporal weight stationary*.

In Fig. 5, we illustrate the temporal weight-stationary dataflow. The PE array has K PEs and they first generate T (total timesteps) outputs for all K neurons that share the position (0,0) across K output channels in the output feature map. Each PE only works on one output neuron. To maximally reuse the filters, each PE has a scratchpad to hold all the C filters that participate in the computation at the corresponding output channel. First, C input receptive fields (sized $R \times R$) for all the timesteps are fetched at once and shared by K PEs. After the first computation cycle is done, all temporal and spatial computations required by those K output neurons are completed. We will write them back to memory and fetch the next $C \times T$ input receptive fields to compute the $K \times T$ outputs for the next K neurons. Notice that the same filters will stay in each PE and be reused until all the outputs are generated for the output feature map. By utilizing this dataflow, SATA fully reuses the filters across T timesteps and reduces the repeated accumulation cost of each output neuron compared to non-temporal weight stationary dataflow.

The overall architecture for SATA is shown in Fig. 4(b). The example configuration considers training a VGG5 SNN with 8-bits resolution for all parameters in 8 timesteps [6]. We use 128 PEs and 128 PGUs in our design to facilitate the maximum number of feature maps in a single layer in VGG5. Generally, for other larger convolutional networks, the number of feature maps per layer is often a multiple of 128. We use a 144KB weight buffer to fit in the maximum number of 8-bit filters between two layers. U and ∇U buffers are set to 256KB for holding 8-bit potentials and gradients for 128 neurons across all timesteps. Similarly, the S buffer is set to 32KB due to the single-bit resolution of spikes.

B. PE Design for Different Computation Stages

There are three convolution stages in a complete cycle of SNN training: forward convolution, backpropagate convolution, and weight update convolution. The PEs are designed to



(a) Details of PE design (b) Overall Architecture of SATA (c) Details of PGUs design

Fig. 4. Overview of SATA's architecture. (a) The dashed line indicates the signal for PE control. Each PE is equipped with a multiplier for the MAC operation during backpropagation and is attached to the circuit to carry out LIF computation during the forward computation (shaded in yellow). (b) The SATA architecture is composed of 128 PEs and 128 PGUs to facilitate the maximum number of output feature maps among any single layer in VGG5. And the different global buffers (GLBs) are also set to the corresponding size to facilitate maximum storage requirements among all layers in VGG5. (c) Each PGU composes of the circuits to carry out the computation of ∇U as in Eqn. (3, 4).

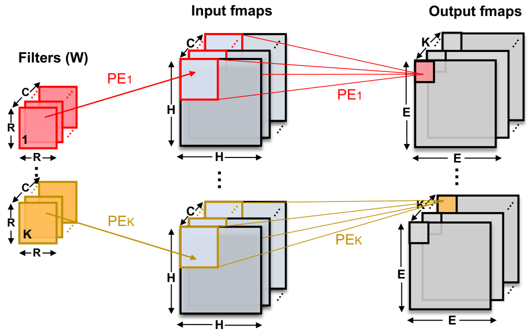


Fig. 5. Illustration of SATA's dataflow. The filters stay stationary in PEs for maximum filter reuse across timesteps. Further, each PE will only focus on the computation for one neuron in one output feature map at a time. For example, in the figure above, the pink-colored filters will be stored in PE$_1$ and PE$_1$ will be responsible for processing the pink-colored output pixel at the output feature map for all timesteps T.

be able to carry out the computations among all three stages as shown in Fig. 4(a). The filter scratch pads are set to the size of $128 \times 9 \times 8$ bits to be compatible with SATA's dataflow (considering most modern SNN architectures, like VGG, have 3×3 sized kernels). The other two scratch pads are set to the same size for making compatible computation with the filter's size.

1) Forward Stage: During the forward propagation, spike activations S will be convolved with filters W for all timesteps. Due to the 1-bit resolution of spikes, the multiplication will be simplified to and operations. At each timestep, after all the convolution partial sums are computed, the outputs go through the LIF computation units (yellow-shaded components in Fig. 4(a)) to generate spikes and update the membrane potential. If the input spike equals zero, the accumulation and the scratch pad read of filters will be elided.

2) Backpropagation Stage: To backpropagate gradients ∇H through the convolutional layers, convolutions are performed between 8-bit potential gradients ∇U and 8-bit filters W. Notice that during the backpropagation, W needs to be transposed into W^T. This convolution is identical to the MAC-based convolution in ANN except for the repetition across all timesteps. Thus, we need an extra multiplier (see Fig. 4(a)) in the PE to accomplish the operation. The multiplier will be gated to save energy during the other two stages (namely, forward convolution and weight update stages). if the sparsity condition for ∇U is met, the MAC operation and scratch pad read of filters will be elided.

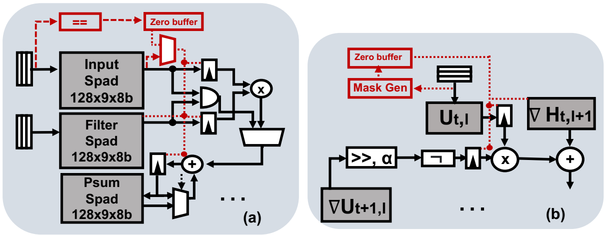


Fig. 6. Sparsity handling units inside PE and PGU. The red dash lines indicate the signal for sparsity handling units. We only show the units related to sparsity handling. (a) Leveraging input (S and ∇U) sparsity to save MAC and filter scratch pad reading. (b) Leveraging ∇f sparsity to skip the related computations of ∇S.

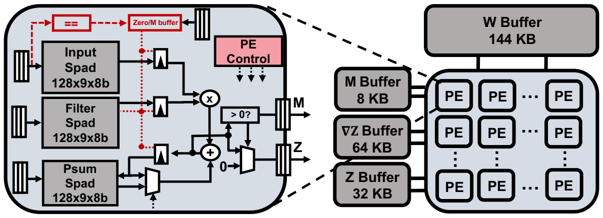
3) Weight Update Stage: Finally, spike activations S stored during the forward propagation are convolved with potential gradients ∇U to generate the gradients for updating parameters W. This convolution reuses the computation units and the sparsity handling units from the forward propagation due to the identical data resolution. Again, this convolution needs to be repeated for all timesteps.

C. Potential Gradient Units

We use PGUs to accomplish the computation in Eqns. (3) and (4). The computation itself is straightforward, and we show the computation unit design in Fig. 4(c)). PGUs will first fetch $U_{t,l}$ to check whether there is sparsity in ∇f. If the sparsity condition is satisfied, PGUs will omit the computation of ∇S. Notice that one PGU will generate a single timestep ∇U for one neuron at a time. The number of PGUs can be configured to satisfy different throughput requirements. By default, SATA uses 128 PGUs.

D. Discussion on Sparsity Handling

In this subsection, we discuss the details of how we handle the sparsity inside PEs and PGUs. In general, we follow the



(a) PE design of ANN baseline (b) Architecture of ANN baseline

Fig. 7. Baseline architecture for ANN training. The PE and overall architecture design are based on Eyeriss, with additional hardware added to support backpropagation.

gating method used in [29] to omit the computation of MAC and memory read of filter scratch pad inside the PE when the input is zero. During the forward and weight update stage, we will directly use the spike input as the gating enable signal to disable the forward data path from switching and filter scratch pad from reading. During the backpropagation stage, similar gating logic will be applied, however, instead of directly using the input spike, we will use the bitmasks generated during the writing of the gradients to the input scratch pad. We have an extra 144-byte zero buffer to hold the bit masks.

In PGUs, we also apply a similar gating strategy as in PEs, however, this time we will check the ∇f sparsity condition as mentioned in section IV. During the writing of membrane potential U into the scratchpad, the binary masks are generated by monitoring ${U_t}^l$, such that, mask $= 1$ if $|{U_t}^l - U_{th}| < \beta/2$, else mask $= 0$. Once the bitmasks are generated and stored in the zero buffer, we then use the same gating logic as in PEs to omit the multiplication and read of ∇H if the ∇f sparsity condition is met.

E. Architecture of ANN baseline

To differentiate the training overhead of SNN (using BPTT algorithm) and ANN (standard backpropagation algorithm), we design a baseline architecture for standard backpropagation (BP)-based ANN training. The PE and architecture design is based on Eyeriss [16], an ANN inference accelerator that has the basic optimizations (reuse, zero-sliding, and memory hierarchy) that have been widely adopted in other ANN accelerator works [30]. In our baseline, we only attach necessary computation and memory components to the original design of Eyeriss to support BP-based training.

Inside the PE, we add a sign checker for carrying ReLU operation. The sign checker generates a bit-mask that is used during backpropagation to skip the unnecessary gradient computations (if the activation after ReLU is zero, we can skip the gradient calculation for that neuron during BP). Our baseline supports the same zero-sliding techniques as proposed in the original paper [16]. A 64KB global buffer is added to hold the gradients during BP, together with an 8KB buffer to hold the masks that were generated during forwarding propagation.

In the original Eyeriss paper [16], the *Row-Stationary* dataflow is utilized to exploit spatial reuse of ifmaps, filters, and psums. However, a recent work [14] has already shown that a rate-coded SNN is less energy efficient (up to $\sim 60\times$ more energy) when compared to the *Row-Stationary*-based Eyeriss. As a result, we force our ANN baseline to use a similar dataflow to the one of SATA, which is more SNN friendly. Note, that SATA's dataflow does not bring any redundant memory or computation operations to the ANN baseline, which ensures a fair comparison.

VI. Energy Simulation Model

In this section, we introduce our cost model for estimating the energy consumption of processing one single image based on SATA during SNN training. The total energy E_{total} is the sum of three components: computation energy, memory energy, and the control circuit energy (noted as E_c, E_m, and E_{ctrl}). We further divide the computation energy into three stages as discussed above: forward computation energy, backward computation energy, and weight update computation energy (E_c^{fwd}, E_c^{bwd}, and E_c^{wup}). For the memory energy, we also divide it into three stages (E_m^{fwd}, E_m^{bwd}, and E_m^{wup}). The formula for total energy is shown below:

$$\begin{aligned} E_{total} &= (E_c^{fwd} + E_c^{bwd} + E_c^{wup}) \\ &\quad + (E_m^{fwd} + E_m^{bwd} + E_m^{wup}) + E_{ctrl}. \end{aligned} \tag{8}$$

We further divide the sub-stage energies into groups of sub-operation energy that belong to a given stage. More specifically, we divide the computation energy of the forward stage into the energy of MAC and LIF operation, the backward stage into the energy of MAC and ∇U calculation, and the weight update stage into the energy of MAC operation. For each calculation operation type, the energy of all the units along the computing path will be taken into consideration (for example, the energy will be different for the MAC operation in the backward stage and the other two stages, due to the different computation path). We also divide the memory energy of all three stages into the energy of communicating with DRAM, global buffers, and scratch pads.

The general rule for calculating those sub-operation energies is $N \times E$, where N denotes the total number of the sub-stage operation that SATA requires to process one image and E denotes the energy consumption of a single operation. Furthermore, we use $N(sp)$ to indicate that N is the function of a given type of sparsity sp (for example, $N_{mac}^{fwd}(sp_S)$ is the total number of MAC operations during the forward propagation for SATA to process one image, and this number can be optimized by sparsity in S). We provide the energy cost estimation formula for all sub-stages as below:

$$\begin{aligned} E_c^{fwd} &= N_{mac}^{fwd}(sp_S) \times E_{mac}^{fwd} + N_{LIF} \times E_{LIF}, \\ E_c^{bwd} &= N_{mac}^{bwd}(sp_{\nabla U}) \times E_{mac}^{bwd} \\ &\quad + N_{\nabla U}(sp_{\nabla f}) \times E_{\nabla U}, \\ E_c^{wup} &= N_{mac}^{wup}(sp_S) \times E_{mac}^{wup}, \\ E_m^{fwd} &= N_{dram}^{fwd} \times E_{dram} + N_{glb}^{fwd} \times E_{glb} \\ &\quad + N_{spad}^{fwd}(sp_S) \times E_{spad}, \\ E_m^{bwd} &= N_{dram}^{bwd} \times E_{dram} + N_{glb}^{bwd} \times E_{glb} \\ &\quad + N_{spad}^{bwd}(sp_{\nabla f}) \times E_{spad}, \\ E_m^{wup} &= N_{dram}^{wup} \times E_{dram} + N_{glb}^{wup} \times E_{glb} \\ &\quad + N_{spad}^{wup}(sp_S) \times E_{spad}, \end{aligned} \tag{9}$$

where E_{mac}^{fwd}, E_{mac}^{bwd}, and E_{mac}^{wup} denote the different energy of MAC operation in different sub-stage. E_{LIF} denotes the energy of the LIF operation in the forward stage and $E_{\nabla U}$ denotes the energy of gradient calculation of ∇U. E_{dram}, E_{glb}, and E_{spad} denote the energy of a single time access to different memory units. The number of MAC operation in three stages are separately denoted as N_{mac}^{fwd}, N_{mac}^{bwd}, and N_{mac}^{wup}, which can be optimized by sparsity of S and ∇U. The number of LIF operations and calculation of ∇U (can be optimized by the sparsity of ∇f) are also denoted by the corresponding N notation. And the total number of data movement for three stages are denoted by the corresponding N notation with the stage name on the top and the memory component name on the bottom, where the number of scratchpad reading can be optimized by ∇f and S. Note that we consider the data access of filters during the backpropagation into the weight update stage.

In general, the number of computation operations is controlled by the network architecture of the SNN, while the number of memory movements will be determined by both the SNN network architecture and the hardware architecture and dataflow design. Table II provides the total number of computation and data movement operations used in Eqn. 9 on SATA for VGG5 as an example and a reference.

A. Energy model for considering the sparsity

In Eqn. 9, we define the total number of sparsity-related operations as a function of the sparsity. Then the user can define the abstraction level of the energy estimation results by setting the energy cost for a single operation E. For example, if one wants to test the theoretical maximum energy benefits that SATA can get from the sparsity, then E can be set without considering any sparsity handling overhead. If the user wants to include the energy overheads of the sparsity handling units, it can be easily done by including the energy overheads into E. In Table III, we give examples of the energy with and without sparsity handling units overheads. Then, the sparsity-aware energy with sparsity-handling overheads can be approximated by $N(sp) \times E(withoverhead) + N \times E(overhead)$, where $E(overhead)$ can be calculated by simply subtracting the energy of operation without overheads from the one with overheads.

B. Discussion on Model Choice and Estimation Method

In this section, we discuss our choice for the energy estimation model in Eqn. 8 and 9. The goal of our energy model is to make it flexible and simple enough for users to adjust the complexity and accuracy of the energy model. For instance, as we will show in the later experiment setup, we choose to neglect the E_{ctrl} in Eqn. 8 when we compare the training energy between SNNs and ANNs because the control energy would be approximately identical between SNNs and ANNs under the gradient-based training context. However, one can always apply the control energy to Eqn. 8 to make the energy value more accurate.

The estimation method used by our energy model is similar to the methodology proposed by [17] and is verified in [31]. Many prior works [31]–[35] also follow this method to estimate the energy cost. Based on the prior works, we attach SNN-specific parameters (e.g., T and $sp_{\nabla f}$) and consider SNN-specific operations (e.g., LIF and potential gradients update) to make the model work for SNNs. We can simply detach those efforts to make the model work for our ANN baseline.

TABLE II
THE DESCRIPTION OF SYMBOLS USED IN EQN. (9). THE TOTAL NUMBER OF EACH OPERATION IS CALCULATED FOR A SINGLE IMAGE DURING ONE FORWARD OR BACKWARD PROPAGATION ACROSS ALL TIMESTEPS. NOTED THAT WE DO NOT SHOW THE SPARSITY REDUCTION OF SCRATCHPAD ACCESSES IN THE TABLE FOR SIMPLICITY.

Parameters	**Description**
C	# of input feature map or filter channels
H	input feature map width/height
K	# of 3D filters or # of output feature maps
E	output feature map width/height
R	filter width/height
b	maximum bitwidth (8 in SATA)
T	# of timesteps (8 in SATA)
sp_S	spike sparsity (# of zero spikes / # of total spikes)
$sp_{\nabla f}$	firing gradient sparsity (# of invalid spike grads / # of total spike grads)
$sp_{\nabla U}$	potential gradient sparsity (# of zero potential grads / # of total potential grads)

# of Ops	**Description**
$N_{mac}^{fwd,wup}$	$T \times (1 - sp_S) \times \sum_{l=1}^{L}(C \times R^2 \times K \times E^2)_l$
$N_{LIF,\nabla U}$	$T \times \sum_{l=1}^{L}(K \times E^2)_l$
$N_{\nabla S}$	$T \times (1 - sp_{\nabla f}) \times \sum_{l=1}^{L}(K \times E^2)_l$
N_{mac}^{bwd}	$T \times (1 - sp_{\nabla U}) \times \sum_{l=1}^{L}(C \times R^2 \times K \times E^2)_l$
N_{dram}^{fwd}	$\sum_{l=1}^{L}(K \times C \times R^2)_l$ $+T \times \sum_{l=1}^{L}(K \times E^2 + 1/b \times C \times H^2)_l$
N_{glb}^{fwd}	$2 \times N_{dram}^{fwd}$
N_{spad}^{fwd}	$2 \times \sum_{l=1}^{L}(K \times C \times R^2 + T \times 1/b \times C \times H^2)_l$
N_{dram}^{bwd}	$T \times \sum_{l=1}^{L}(K \times E^2 + 1/b \times C \times H^2)_l$
N_{glb}^{bwd}	$(5 + 2(1 - sp_{\nabla f})) \times T \times \sum_{l=1}^{L}(K \times E^2)_l$ $+\sum_{l=1}^{L}(2T \times 1/b \times C \times H^2 + K \times C \times R^2)_l$
N_{spad}^{bwd}	$\sum_{l=1}^{L}(K \times C \times R^2 + T \times K \times E^2)$
N_{dram}^{wup}	$2 \times \sum_{l=1}^{L}(K \times C \times R^2)_l$
N_{glb}^{wup}	$2 \times (1 + T) \times \sum_{l=1}^{L}(K \times C \times R^2)_l$ $+T \times \sum_{l=1}^{L}(1/b \times C \times H^2 + K \times E^2)_l$
N_{spad}^{wup}	$N_{glb}^{wup} + 2 \times T \times \sum_{l=1}^{L}(K \times C \times R^2)$

VII. EXPERIMENT RESULTS

A. Experiment Setup

We use VGG5 [36] (configured as in Table V) as our baseline network architecture for comparing the training energy difference between ANNs and SNNs. We train the

TABLE III
ENERGY DIFFERENCE FOR A SINGLE OPERATION WITH AND WITHOUT OVERHEADS FOR SPARSITY HANDLING UNITS. THE ENERGY UNIT IS NORMALIZED IN TERMS OF THE ENERGY FOR A MAC OPERATION.

Operation	Without Overhead	With Overhead
E_{mac}^{fwd}	0.146	0.146
E_{mac}^{bwd}	1.003	1.120
$E_{\nabla U}$	0.952	1.078

TABLE IV
SYSTEM PARAMETERS FOR SATA AND EYERISS, WHICH ARE THE BASELINE FOR SNNS AND ANNS.

Parameter	SATA	Eyeriss
Technology	65 nm CMOS	65 nm CMOS
Precision	8 bits (W, U), 1 bit (S, M)	8 bits (W, Z)
GLB size	256 KB (U, ∇U) 144 KB (W) 32 KB (S)	32, 64 KB (Z, ∇Z) 144 KB (W) 8 KB (M)
Spad size	1.125 KB	1.125 KB
PE array size	128	128
PGU array size	128	-

VGG5 network on CIFAR10 with a learning rate of 0.001, a momentum of 0.9, and a weight decay factor of $1e^{-4}$. For SNN training, we further set the timestep as $T = 8$, leaking factor as $\alpha = 0.94$, firing threshold as $U_{th} = 0.75$, and the fire function width $\beta = 2.5$.

We use SATA-Sim [18] with the energy simulation model in VI to approximate the training energy of ANNs from the 8-bit version of our Eyeriss-based ANN baseline and SNN from SATA both with the computing units synthesized in Synopsys Design Compiler at 400MHz using 65nm CMOS technology and the memory units simulated in CACTI [37]. Since the main purpose of the energy results is for comparison, we assume perfect gating and no control overheads during the comparison (namely, assuming no leaking power for computation units when gated and setting E_{ctrl} in Eqn. (8) to 0 for both ANN and SNN). Unless otherwise stated, the hardware specifications are listed in Table IV. All the energy results denote the energy required to process one image and the unit of energy is normalized in terms of the energy for a MAC operation (e.g., 100 = energy of 100 MAC operations).

For performing energy analysis on sparse training, the inherent sparsity is collected for both SNN and ANN baseline during the training process. We collect the layerwise sparsity of activation (arising due to ReLU non-linearity which only passes non-negative values) and its gradient for ANNs and collect three categories of sparsity (namely, S, ∇f, and ∇U) for SNNs. All the SNN sparsity results are averaged across total timesteps, the number of images, and training epochs. The sparsity results are summarized in Table VI.

B. Training Energy: SNNs vs. ANNs

We first compare the training energy between SNNs and ANNs without considering any sparsity in Fig. 8. In our training scenario, SNN in total consumes 1.35× more energy than ANN. We further break up the energy comparison results into computation energy and memory energy. According to our comparison, SNN consumes 3.28× more total computation energy when compared to ANN and 1.28× more total memory movement energy compared to ANN.

TABLE V
NETWORK STRUCTURES FOR VGG5 AND VGG9. THE SYMBOLS C, MP, AND FC DENOTE CONVOLUTIONAL, MAX-POOLING, AND FULLY CONNECTED LAYERS, RESPECTIVELY. 64C3 REFERS TO A CONVOLUTIONAL LAYER WITH 64 CHANNELS AND 3×3 KERNELS.

Network	Structure	Dataset
VGG5	64C3-MP2-128C3-128C3-MP2-1024FC-10FC	MNIST CIFAR10 CIFAR100
VGG9	64C3-64C3-MP2-128C3-128C3-MP2-256C3-256C3-256C3-MP2-1024FC-10FC	CIFAR10

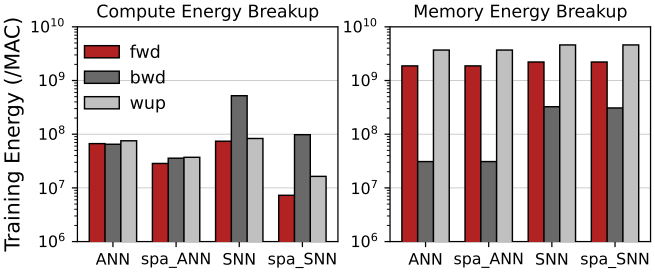


Fig. 8. Energy comparison between ANNs and SNNs, where spa-ANN and spa-SNN refer to sparse ANN and sparse SNN, respectively.

We then take sparsity into consideration. The sparsity results can be found in Table VI for SNNs and ANNs for CIFAR10 on VGG5. With inherent sparsity, the sparse SNN now consumes 1.27× more total energy compared to sparse ANN. Specifically, sparse SNN consumes 1.19× more total computation energy and 1.27× more total memory movement energy compared to sparse ANN. Compared to non-sparse SNN, SATA increases the computation energy efficiency of sparse SNN by 5.58× by utilizing the sparsity. In Fig. 8, we visualize the energy comparison results between ANNs and SNNs for both non-sparse and sparse training. We break up the energy results according to Eqn. (7) and (8). We further visualize the layerwise computation energy for the sparse SNN training in Fig. 9 and the break up of the total memory energy in Fig. 10.

We make the following key observations from the comparison results:

- We first identify that, in contrast to our impression that SNN is more energy-efficient than ANN, SNN training is more expensive (1.27× more even with sparsity) than ANN training. Separating the total training energy into computation and memory portions, we observe that though we can utilize the rich sparsity in SNNs to shrink the computation energy gap between SNNs and ANNs (3.28× to 1.19×), the total energy gap (1.27×) is still bounded by the memory energy gap (1.27×) between two types of networks.
- With the previous observation, we then identify that the memory communication energy is the bottleneck of the total energy consumption in SNN training. This is

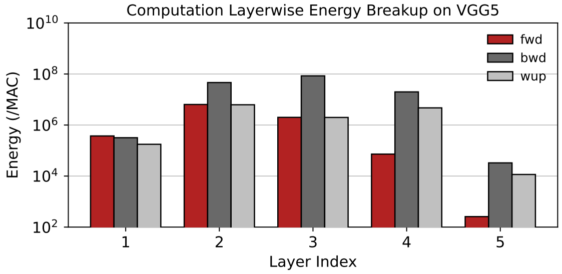


Fig. 9. Layerwise computation energy results on VGG5.

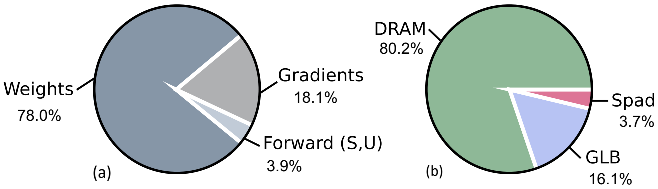


Fig. 10. Energy breakdown of the memory for VGG5 from the perspective of (a) algorithm memory components and (b) hardware memory components. In (a), the Gradients refer to the memory movement to calculate gradients (∇U, ∇H, and ∇W).

due to the expensive cost of accessing to GLBs and DRAMs, which together compose 96.3% of the total memory energy as shown in Fig. 10. While memory energy dominates the energy gap between SNNs and ANNs, sparsity hardly optimizes this energy inefficiency. S and ∇f sparsity can only reduce the memory reads from scratch pads inside PEs but can not optimize the cost of accessing DRAMs and GLBs, which are the most expensive operations in SNN training. Moreover, the access to DRAMs needs to be repeated multiple timesteps for reading and writing the necessary data (S and U, etc.) for BPTT. In our experiments, due to the small number of timesteps ($T = 8$), the memory access energy for ANNs and SNNs is mainly bounded by the DRAM access energy of filters (78% of the total memory energy as shown in Fig. 10), which is the same for both networks. We will show in the later section that larger timesteps will exponentially separate the memory access energy gap between ANN and SNN.

- We further break up the computation energy into three computation stages to identify the computation energy bottleneck for sparse SNN training. In fact, sparse SNN consumes only $0.26\times$ and $0.44\times$ of sparse ANN's computation energy on the forward and weight update stage. The major bottleneck for SNN's training computation is the backward stage where sparse SNN consumes $2.74\times$ more energy than sparse ANN. During the backward computation, SNNs require the same multi-bits MAC operation as ANNs but the operation needs to be repeated for multiple timesteps. This repetition of MAC operations is the source of computation energy inefficiency in SNN's backward computation.
- Though the memory energy bottleneck can not be easily fixed with sparsity, the bottleneck for computation energy (namely, backward stage) can be alleviated with sparsity in ∇U. The backward stage of the sparse SNN consumes $0.19\times$ reduced energy than that of the non-sparse SNN. By increasing the ∇U sparsity, the energy cost of the backward computation stage can be further reduced (refer to energy cost model in Eqn. (9)).

TABLE VI
ACCURACY AND LAYERWISE SPARSITY FOR VGG5. THE SPARSITY SHOWN IS THE AVERAGE SPARSITY PER IMAGE PER TIMESTEP. Z DENOTES THE RELU ACTIVATION OUTPUT FROM ANN.

Dataset	**Sparsity**	**Layerwise Results (%)**					
		inp	**cov1**	**cov2**	**cov3**	**lin4**	**lin5**
MNIST	S sparsity	68.83	93.03	91.98	98.06	92.91	-
Acc: 99%	∇f sparsity	-	22.87	38.19	55.11	32.75	46.30
(SNN)	∇U sparsity	-	94.14	85.02	94.65	67.93	57.01
CIFAR10	S sparsity	43.45	85.83	91.57	98.37	96.82	-
Acc: 75%	∇f sparsity	-	39.33	69.79	80.95	62.20	37.18
(SNN)	∇U sparsity	-	73.25	69.58	93.12	61.65	4.04
CIFAR10	Z sparsity	0.00	50.72	54.41	83.05	69.22	-
Acc: 82%	∇Z sparsity	-	75.67	3.51	75.51	1.07	41.71
(ANN)							
CIFAR100	S sparsity	47.16	86.17	89.58	98.47	94.32	-
Acc: 42%	∇f sparsity	-	35.20	66.65	86.66	55.58	54.19
(SNN)	∇U sparsity	-	70.10	65.09	95.00	54.66	10.58

Fortunately, SNNs not only are highly sparse in spikes but also inherently possess high sparsity in ∇U. We further make the ablation studies on the sparsity of SNNs and their relationship with SNN training energy in the following section.

C. Ablation Study on Sparsity and Training Energy

1) Sparsity and Datasets: We first study the effects of different datasets on SNN's sparsity. We train our VGG5 SNN model across three datasets: MNIST, CIFAR10, and CIFAR100, with the same configurations as in the previous section to generate sparsity results in the first 20 training epochs. For each epoch, each type of sparsity is calculated by averaging across images and timesteps. The results are illustrated in Fig. 11. We also provide layerwise sparsity results for three datasets in Table VI. Several points can be inferred:

- First, regardless of the choice of datasets, the spikes (S sparsity) are highly sparse ($> 94\%$) throughout the training, which can help SNN save its computation energy during the forward and weight update stages.
- Second, SNNs also possess a relatively high percentage of ∇U sparsity (on average 73% on CIFAR10 and 84% on MNIST), which can help SNNs reduce the computation energy for the backward stage.
- Furthermore, the sparsity of ∇f and sparsity of ∇U share similar increasing trends with the increasing number of training epochs. The sparsity-increasing effect is more significant on complex training data (CIFAR100) as compared to the simple one (MNIST).

2) Sparsity and SNN-unique Hyperparameters: We further study how the training hyperparameters that are unique to SNNs (namely, timestep T and firing width β in Fig. 2) affect the sparsity and the training energy. We train our VGG5 SNN model with different T and β to get different sparsity results, as shown in Fig. 12. Fig. 13 shows the corresponding energy results on sparse ANN and sparse SNN for a different choice of hyperparameters that result in different levels of sparsity.

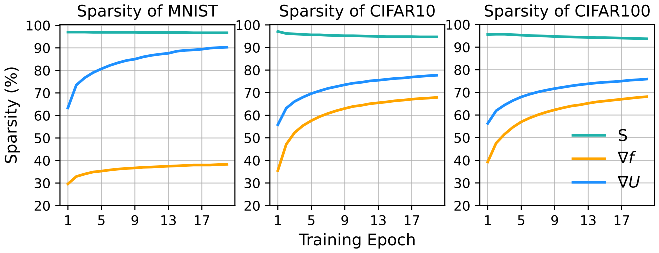


Fig. 11. Sparsity results across datasets.

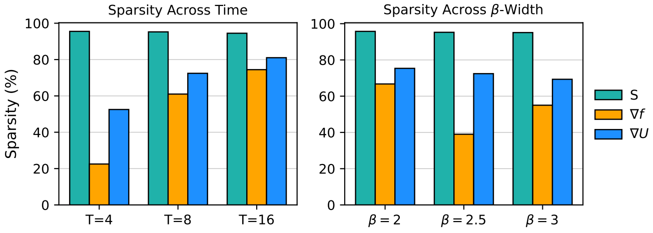


Fig. 12. Sparsity across timesteps and firing function width.

As shown in Fig. 13, changing firing width has almost no effect on the SNN training energy. Also, naively adjusting T does not result in a proportional change in computational energy. For example, while reducing T reduces the number of repeated computation operations, it also reduces the sparsity of ∇f and ∇U, and thus cancels out the saved energy from reduced computation operations. As we discussed in section 7.2, the memory communication energy is bounded by the movement of filters on our VGG5 example. Thus, we find that only the backward memory cost (which does not involve movements of filters) is proportional to the number of timesteps. We will have further discussions on the effects of the timestep in the next section.

3) Sparsity and Network Depth: Finally, we study the effects of network depth and sparsity. We further train a VGG9 network with the same training configurations as our previous VGG5 model on CIFAR10 and get the average layerwise sparsity results, as shown in Fig. 14. We observe that, while S sparsity gets more sparse in the deeper layers, the changing trend and average sparsity across layers are roughly the same for both networks. For ∇U sparsity, both networks also share a similar changing trend across layers. On average, VGG9 experiences less ∇U sparsity ($\sim 60\%$) across layers compared to VGG5 ($\sim 70\%$). We generate the layerwise computation energy with our energy estimation model and visualize the results in Fig. 15 for VGG9.

D. Discussion

1) SNN training algorithm: In this section, we further discuss some possible future directions for SNN algorithm design to make SNN training energy-efficient. One direction would be to optimize the total computation energy. As discussed in Section 7.2, the bottleneck for SNN training computation energy is backward computation. This bottleneck can be alleviated by introducing more ∇U sparsity during the training. While simply adjusting the training parameters can not effectively increase the ∇U sparsity, we provide a hypothetical analysis to show the tradeoff between the total computation energy and the ∇U sparsity in Fig. 16. We use the sparse ANN training energy as in section 7.2 and fix it. We take the ∇U layerwise sparsity of CIFAR10 on VGG5 SNN in Table VI as our baseline sparsity and gradually scale it up.We observe that by increasing the ∇U sparsity, SNN training will have less total computation energy overhead compared to ANN training. At 88% of baseline, the SNN breaks even with ANN.

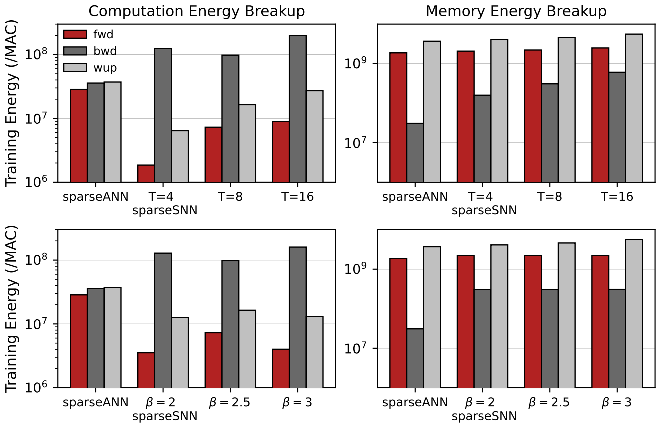


Fig. 13. Energy results across timesteps and firing function width.

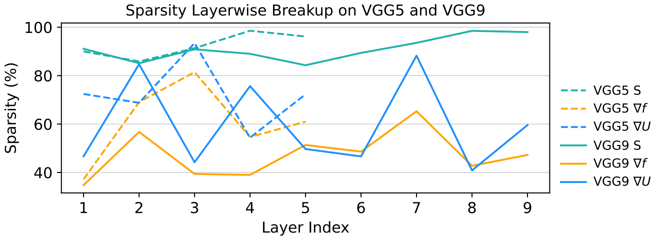


Fig. 14. Layerwise sparsity comparison between VGG5 and VGG9.

To optimize the total training energy of SNN, a large number of timesteps should be avoided. We make a similar hypothetical analysis as above on the relation between total timesteps T and training energy of SNNs in Fig. 17. We find that SNN's total training energy exponentially increases with the number of timesteps. This is because we need to repetitively access DRAMs for T times for getting membrane potential (U) and spike (S) for BPTT. This expensive memory operation will dominate the total energy when T gets large. Apart from the energy dominance, the training time of SNN will also increase as timesteps increase. Table VII shows how the training latency gap between SNNs and ANNs gets bigger when the timesteps increase.

TABLE VII
LATENCY COMPARISON OF ONE TRAINING EPOCH BETWEEN SNNS AND ANNS WITH VARYING TIMESTEPS OVER VGG5 ON NVIDIA V100 GPU.

Network	Latency of ANNs	Latency of SNNs	Latency Gap
VGG5	12.28 s	83.29 s ($T = 4$)	6.78×
VGG5	12.28 s	180.12 s ($T = 8$)	14.67×
VGG5	12.28 s	409.92 s ($T = 16$)	33.38×

Moreover, as we have shown in Fig. 10, energy for DRAM data movement of the weights becomes the bottleneck of the

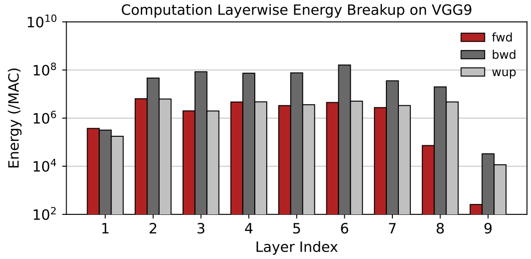


Fig. 15. Layerwise computation energy results on VGG9.

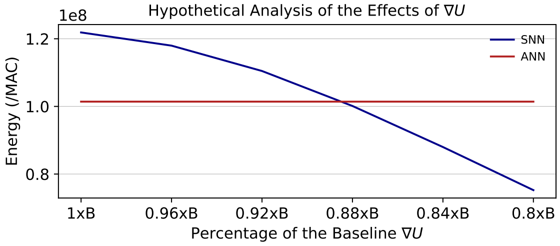


Fig. 16. Hypothetical analysis on how ∇U affects energy efficiency. B denotes the layerwise density i.e. $B = 1 - sp_{\nabla U}$. $sp_{\nabla U}$ denotes the layerwise sparsity due to ∇U.

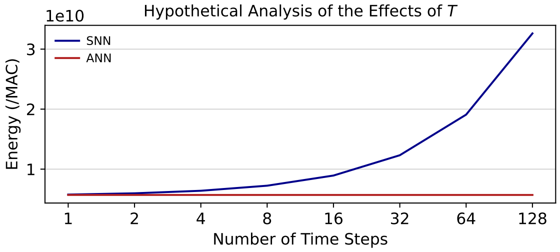


Fig. 17. Hypothetical analysis of total energy cost on total timesteps T.

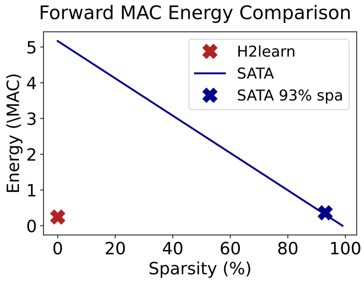


Fig. 18. Energy comparison between SATA and H2Learn for a forward convolution workload (3 × 3 kernel).

SNN training. One possible future direction is to train the SNNs with a sparsity constraint. The other possibility is to compress part of or even the whole model through methods like [38]–[40] and store the model on-chip as in [19].

2) SNN training accelerator & comparison with prior work: In this section, we discuss some considerations for the future design of SNN training accelerators based on the findings from this paper. From our energy comparison results, we find SNNs are less energy efficient than ANNs in a gradient-based training setup. SATA being a general purpose architecture targeted to perform fast energy estimation and comparison between different SNN structures, we do not pay much effort to the architectural level optimization for BPTT-based SNN training, except for the sparsity-aware PEs and PGUs. One future direction for the SNN training accelerator design would be optimizing the time-repetitive data movement for the BPTT-based method. For instance, the SNN-dedicated design proposed in H2Learn [15] indeed unveils some potential ways to alleviate the memory movement bottleneck for SNNs. We implement the LUT-based PE from the Forward Engine in H2Learn [15] with 65nm CMOS technology and use the same synthesis method as SATA. We compare the energy difference between SATA's PE and LUT PE on performing a convolution using a 3×3 kernel for one timestep. The energy difference is shown in Fig.18. Due to the LUT-based convolution that H2Learn utilizes, the energy result for the convolution in SNN's forward propagation does not suffer from the time-repetitive memory reading from scratchpads inside PEs. Also, the LUT-based convolution is sparsity-independent. Thus, SATA's general purpose PE consumes approximately $21.2\times$ more energy on a 3×3 convolution workload without considering sparsity. When considering the sparsity, SATA can only get the same energy efficiency as H2Learn with 93% sparsity (not possible for a 3×3 kernel that delivers information). Note, the above comparison is approximate, for example, the energy overheads of pre-calculating and loading the elements for LUTs are not considered. Indeed, considering those overheads would make the energy estimation and comparison between the training of different SNN structures complex. That's also one major motivation for having SATA, a general purpose architecture design for simple SNN training energy estimation and comparison.

VIII. Conclusions

We propose SATA, a sparsity-aware BPTT-based training accelerator for SNNs. The simple and highly re-configurable systolic-based design of SATA makes it easy to perform a training energy analysis on different SNN topologies. We further propose an energy estimation model based on SATA for energy estimation. Compared with not utilizing sparsity, sparsity-aware SATA increases its computation energy efficiency by $5.58\times$. The results also show that when running on Eyeriss-like systolic-based architecture, SNN training requires more energy compared to ANNs with and without considering sparsity. We make several observations and show how energy-efficiency trade-off with respect to different SNN-specific training parameters. Our results and estimation tool will hopefully guide future SNN algorithm works to design more energy-efficient and sparsity-aware training mechanisms, as well as future SNN training accelerator works to improve their architecture design to be more energy-efficient.

Acknowledgments

The research was funded in part by C-BRIC, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, the National Science Foundation (Grant#1947826), DARPA AI Exploration (ShELL), and, Technology Innovation Institute (Abu Dhabi).

REFERENCES

[1] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[2] K. Roy *et al.*, "Towards spike-based machine intelligence with neuromorphic computing," *Nature*, vol. 575, no. 7784, pp. 607–617, 2019.

[3] C. D. Schuman, S. R. Kulkarni, M. Parsa *et al.*, "Opportunities for neuromorphic computing algorithms and applications," *Nature Computational Science*, vol. 2, no. 1, pp. 10–19, 2022.

[4] D. V. Christensen *et al.*, "2022 roadmap on neuromorphic computing and engineering," *Neuromorphic Computing and Engineering*, 2022.

[5] Y. Wu *et al.*, "Spatio-temporal backpropagation for training high-performance spiking neural networks," *Frontiers in neuroscience*, vol. 12, p. 331, 2018.

[6] Y. Kim *et al.*, "Revisiting batch normalization for training low-latency deep spiking neural networks from scratch," *Frontiers in Neuroscience*, 2021.

[7] 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.

[8] P. U. Diehl *et al.*, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in computational neuroscience*, vol. 9, p. 99, 2015.

[9] C. Lee *et al.*, "Deep spiking convolutional neural network trained with unsupervised spike-timing-dependent plasticity," *IEEE Transactions on Cognitive and Developmental Systems*, vol. 11, no. 3, pp. 384–394, 2018.

[10] P. Panda, S. A. Aketi, and K. Roy, "Toward scalable, efficient, and accurate deep spiking neural networks with backward residual connections, stochastic softmax, and hybridization," *Frontiers in Neuroscience*, vol. 14, p. 653, 2020.

[11] Y. Li, S. Deng, X. Dong, R. Gong, and S. Gu, "A free lunch from ann: Towards efficient, accurate spiking neural networks calibration," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6316–6325.

[12] Y. Li, S. Deng, X. Dong, and S. Gu, "Converting artificial neural networks to spiking neural networks via parameter calibration," *arXiv preprint arXiv:2205.10121*, 2022.

[13] B. Wang *et al.*, "Shenjing: A low power reconfigurable neuromorphic accelerator with partial-sum and spike networks-on-chip," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 240–245.

[14] S. Narayanan *et al.*, "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.

[15] L. Liang *et al.*, "H2learn: High-efficiency learning accelerator for high-accuracy spiking neural networks," *arXiv preprint arXiv:2107.11746*, 2021.

[16] Y.-H. Chen *et al.*, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[17] T.-J. Yang, Y.-H. Chen, J. Emer, and V. Sze, "A method to estimate the energy consumption of deep neural networks," in *2017 51st asilomar conference on signals, systems, and computers*. IEEE, 2017, pp. 1916–1920.

[18] R. Yin. Sata-sim. [Online]. Available: https://github.com/RuokaiYin/SATA_Sim

[19] F. Akopyan *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.

[20] M. Davies *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *Ieee Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[21] A. Ankit *et al.*, "Resparc: A reconfigurable and energy-efficient architecture with memristive crossbars for deep spiking neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[22] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber, "Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 8, pp. 1943–1953, 2013.

[23] S. R. Kulkarni *et al.*, "Neuromorphic hardware accelerator for snn inference based on stt-ram crossbar arrays," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2019, pp. 438–441.

[24] C. Lee *et al.*, "Training deep spiking convolutional neural networks with stdp-based unsupervised pre-training followed by supervised fine-tuning," *Frontiers in neuroscience*, vol. 12, p. 435, 2018.

[25] S. Guo *et al.*, "A systolic snn inference accelerator and its co-optimized software framework," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, 2019, pp. 63–68.

[26] K. Ahmed *et al.*, "Probabilistic inference using stochastic spiking neural networks on a neurosynaptic processor," in *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2016, pp. 4286–4293.

[27] E. O. Neftci *et al.*, "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.

[28] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 609–622.

[29] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[30] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[31] Y. N. Wu *et al.*, "Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2019.

[32] W. Di *et al.*, "usystolic: Byte-crawling unary systolic array," in *The 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[33] H. Kwon *et al.*, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 754–768.

[34] 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.

[35] ——, "ugemm: Unary computing for gemm applications," *IEEE Micro*, vol. 41, no. 3, pp. 50–56, 2021.

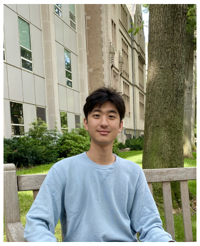
[36] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[37] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to understand large caches," *University of Utah and Hewlett Packard Laboratories, Tech. Rep*, vol. 147, 2009.

[38] Y.-H. Chen *et al.*, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[39] Y. Kim, Y. Li, H. Park, Y. Venkatesha, R. Yin, and P. Panda, "Lottery ticket hypothesis for spiking neural networks," *arXiv preprint arXiv:2207.01382*, 2022.

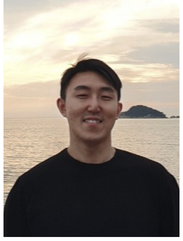
[40] C. Deng, Y. Sui, S. Liao, X. Qian, and B. Yuan, "Gospa: an energy-efficient high-performance globally optimized sparse convolutional neural network accelerator," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1110–1123.



Ruokai Yin is a Ph.D. student in the Department of Electrical Engineering at Yale University, advised by Prof. Priyadarshini Panda. His research interests lie in designing high-performance computer architectures for neural networks. Prior to joining Yale, he received his BS-Electrical Engineering degree from the University of Wisconsin-Madison, where he worked with Prof. Joshua San Miguel on computer architectures for stochastic computing.

Abhishek Moitra received his B.E. degree in Electrical Engineering from Birla Institute of Technology and Science Goa, India in 2019. Currently, he is pursuing his Ph.D. in the Intelligent Computing Lab at Yale. Previously, he worked as a research assistant at the Indian Institute of Science, Bangalore where he worked on designing FPGA-based hardware accelerators for Signal and Image processing applications. His research interests involve hardware-algorithm co-design with CMOS and emerging devices for efficient and robust Deep Learning.

Abhiroop Bhattacharjee received B.E. in Electrical and Electronics from Birla Institute of Technology and Science Pilani, India, in 2020. He joined Yale University, USA, in 2020 as a Ph.D. student in the Electrical Engineering department. Prior to joining Yale University, he worked as a guest researcher in the Chair for Processor Design, TU Dresden, Germany, in 2020, and as a research intern in the Institute of Materials in Electrical Engineering-I, RWTH Aachen University, Germany, in 2019. His research interests lie in the areas of adversarial security and process in-memory architectures for neuromorphic circuits.



Youngeun Kim is currently working toward a Ph.D. degree in Electrical Engineering at Yale University, New Haven, CT, USA. Prior to joining Yale, he worked as a full-time student intern at T-Brain, AI Center, SK telecom, South Korea. He received his B.S. degree in Electronic Engineering from Sogang University, South Korea, in 2018 and M.S. degree in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST), in 2020. His research interests include neuromorphic computing, computer vision, and deep learning.

Priyadarshini Panda is an assistant professor in the electrical engineering department at Yale University, USA. She received her B.E. degree in Electrical & Electronics and Master's degree in Physics from BITS, Pilani, India in 2013 and her PhD in Electrical & Computer Engineering from Purdue University, USA in 2019. She was the recipient of outstanding student award in Physics in 2013. From 2013-14, she worked at Intel, India as a design engineer and Nvidia, India as an intern. In 2017, she interned in Intel Labs, Oregon, USA where she developed large scale spiking neural network algorithms for benchmarking the Loihi chip. She is the recipient of the 2019 Amazon Research Award, 2022 Google Scholar Research Award, and 2022 DARPA Riser Award. She has published more than 60 publications in well-recognized venues including, Nature, Nature Communications, and IEEE among others. Her research interests include- neuromorphic computing, energy efficient deep learning, adversarial robustness, and hardware-centric design of robust neural systems.