Shift-BNN: Highly-Efficient Probabilistic Bayesian Neural Network Training via Memory-Friendly Pattern Retrieving

Qiyu Wan
University of Houston
Houston, Texas, USA
qwan@uh.edu

Haojun Xia
University of Sydney
Sydney, Australia
xhjustc@gmail.com

Xingyao Zhang
University of Washington
Seattle, Washington, USA
xingyaoz@cs.washington.edu

Lening Wang
University of Houston
Houston, Texas, USA
lwang56@uh.edu

Shuaiwen Leon Song
University of Sydney
Sydney, USA
leonangel991@gmail.com

Xin Fu
University of Houston
Houston, Texas, USA
xfu8@central.uh.edu

ABSTRACT

Bayesian Neural Networks (*BNNs*) that possess a property of uncertainty estimation have been increasingly adopted in a wide range of safety-critical AI applications which demand reliable and robust decision making, e.g., self-driving, rescue robots, medical image diagnosis. The training procedure of a probabilistic BNN model involves training an ensemble of sampled DNN models, which induces orders of magnitude larger volume of data movement than training a single DNN model. In this paper, we reveal that the root cause for BNN training inefficiency originates from the massive off-chip data transfer by Gaussian Random Variables (*GRVs*). To tackle this challenge, we propose a novel design that eliminates all the off-chip data transfer by GRVs through the reversed shifting of Linear Feedback Shift Registers (*LFSRs*) without incurring any training accuracy loss. To efficiently support our LFSR reversion strategy at the hardware level, we explore the design space of the current DNN accelerators and identify the optimal computation mapping scheme to best accommodate our strategy. By leveraging this finding, we design and prototype the first highly efficient BNN training accelerator, named *Shift-BNN*, that is low-cost and scalable. Extensive evaluation on five representative BNN models demonstrates that Shift-BNN achieves an average of 4.9× (up to 10.8×) boost in energy efficiency and 1.6× (up to 2.8×) speedup over the baseline DNN training accelerator.

CCS CONCEPTS

• **Computer systems organization** → *Neural networks*; • **Hardware** → *Hardware accelerators.*

KEYWORDS

Bayesian neural networks accelerator, energy efficiency, random number generation

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480120

ACM Reference Format:
Qiyu Wan, Haojun Xia, Xingyao Zhang, Lening Wang, Shuaiwen Leon Song, and Xin Fu. 2021. Shift-BNN: Highly-Efficient Probabilistic Bayesian Neural Network Training via Memory-Friendly Pattern Retrieving. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480120

1 INTRODUCTION

Deep learning based AI technologies, such as deep convolutional neural networks (DNNs), have recently achieved tremendous success in numerous application domains, such as object detection, image classification, etc [7, 12, 22, 47, 50]. However, DNN models are known to be prone to over-fitting due to insufficient training data in the real world, which can lead to wrong predictions when the model is deployed in unfamiliar environments. With the increasing adaptation of safety-critical AI applications (e.g., healthcare and self-driving), wrong predictions can result in catastrophic incidents. For example, several accidents have been recently reported regarding poor safety-critical AI designs [40, 51], e.g., in 2020 an autopilot car crashed into a white truck because the sensor failed to distinguish the truck from the bright sky [51]. Therefore, enhancing the reliability and robustness of deep learning has become an urgent demand from AI practitioners.

As one of the most popular probabilistic machine learning tools, Bayesian Neural Networks (BNNs) have been increasingly employed in a wide range of real-world AI applications which require reliable and robust decision making such as self-driving, rescue robots, disease diagnosis, scene understanding, and so on [1, 32, 38, 57]. BNNs have also emerged as a promising solution in today's data center services for improving product experiences (e.g., Instagram and Youtube), infrastructure, and aiding cutting-edge research [20]. Different from the traditional DNNs which require massive training data, BNN models can more easily learn from small datasets and are more robust to over-fitting issues [6]. Furthermore, BNNs are capable of providing valuable uncertainty information for users to better interpret the situation without making over-confident decisions [2, 13, 24]. Generally, a BNN model can be viewed as a probabilistic model where each model parameter, i.e., weight, is a probability distribution. Training a BNN essentially calculates the probability distribution of weights, which requires integrating on infinite number of neural networks. This is often

intractable. To tackle this, recent efforts [6, 26, 46] leverage Gaussian distributions to approximate the target weight distributions via *weight sampling* to identify the mean and standard deviation of each weight.

Training DNN models on current hardware devices has long been considered as a slow and energy-consuming task [16, 25, 52]. Compared with the traditional DNN training, BNN training inefficiency is further exacerbated by the requirement of *training an ensemble of sampled DNN models* to ensure robustness. In consequence, we have observed that the total *data movement* during a BNN training procedure can be orders of magnitude larger than training one single DNN model. Moreover, as the existing DNN training optimization techniques [45, 54, 60, 64, 65] are oblivious to the unique sampling process of the probabilistic BNN models, they lack the capabilities to efficiently and effectively deal with the excessive data movement induced by the memory-intensive BNN training, resulting in poor energy efficiency and long training latency.

In this paper, we first conduct a comprehensive characterization of the state-of-the-art BNN training on current DNN accelerators and analyze its inefficiency. By carefully breaking down the memory activities in each BNN layer, we observe that the dominant factor that induces BNN training inefficiency is the massive data movement from Gaussian random variables (GRVs). These variables are generated during forward propagation for weight sampling and sent to off-chip memory for later reuse during backward propagation. They contribute the major portion of the total off-chip memory accesses for BNN training (e.g., up to 71%). To tackle this challenge, we propose a novel design that is capable of *eliminating all the off-chip memory accesses by GRVs without incurring any training accuracy loss.* Our design is based on a key observation that the software-level "forth-back" training procedure shares great similarity with the classic hardware-level reversed shifting of the Linear Feedback Shift Registers (LFSRs) which are used in modern BNNs to generate the GRVs [9]. By leveraging the reversible property of LFSR, we build a highly efficient memory-friendly design based on LFSR reversed shifting, which can accurately retrieve all the GRVs (i.e., bit patterns generated in forward propagation) locally during backpropagation without ever storing them during the forward propagation. Furthermore, to investigate the compatibility of our LFSR reversion strategy on real hardware, we qualitatively study the design possibilities by directly integrating our strategy to the existing DNN accelerators that adopt various computation mapping schemes, and eventually identify the optimal mapping to support our BNN training design. Based on this knowledge, we design and prototype the first highly efficient hardware accelerator for BNN training, named *Shift-BNN*, that takes advantage of drastically reduced data movement enabled by our LFSR reversion strategy. This study makes the following contributions:

- We characterize modern BNN training on the state-of-the-art DNN accelerators and reveal that the root cause for its training inefficiency originates from the massive data transfer induced by GRVs;
- We propose a novel design that eliminates all the off-chip data transfer related to GRVs through local LFSR reversed shifting without affecting the training accuracy;
- We present the potential hardware-level challenges when directly applying our design to BNN training and significantly mitigate these issues via a sophisticated and qualitative design space exploration;
- We design and prototype the first highly-efficient BNN training accelerator that is low-cost and scalable, well supported by a hybrid dataflow;
- Extensive evaluation on five representative BNN models demonstrates that Shift-BNN achieves an average of 4.9× (up to 10.8×) improvement in energy efficiency and 1.6× (up to 2.8×) speedup over the baseline accelerator. Shift-BNN also scales well to larger BNN model sample sizes.

2 BACKGROUND

2.1 Training BNNs with Variational Inference

A Bayesian neural network (BNN) can be viewed as a probabilistic model in which each model parameter,e.g., weight, is a probability distribution. One of the most popular method for training BNN models is known as *Variational Inference* [5, 29, 63] , which finds a probability distribution $q(\mathbf{w}|\theta) \in Q$ to approximate the target weight distribution (Q is a common distribution family). Searching for $q(\mathbf{w}|\theta)$ is an optimization problem that aims to minimize the loss function with respect to θ (Eq.1).

$$\mathcal{L}(\mathbf{w}, \theta) = \sum_{i=1}^{S} \log q(\mathbf{w^{(i)}}|\theta) - \log P(\mathbf{w^{(i)}}) - \log P(\mathbf{y}|\mathbf{x}, \mathbf{w^{(i)}}) \quad (1)$$

In Eq.1, $\mathbf{w^{(i)}}$ denotes the ith sample of weights drawn from the approximation distribution $q(\mathbf{w}|\theta)$. Typically, $q(\mathbf{w}|\theta)$ is assumed to be a Gaussian distribution $q(\mathbf{w}|(\mu, \sigma))$ where μ and σ are the mean and standard deviation of the Gaussian distribution, respectively. Each sample of weight $\mathbf{w^{(i)}}$ can be obtained by using $\mathbf{w^{(i)}} = \mu + \epsilon^{(i)} \circ \sigma$, where $\epsilon^{(i)}$ denotes the ith random variable drawn from unit Gaussian distribution $\mathcal{N}(0, \mathbf{I})$ and $\circ$ represent point-wise multiplication. $\log q(\mathbf{w^{(i)}}|\theta)$, $\log P(\mathbf{w^{(i)}})$ and $\log P(\mathbf{y}|\mathbf{x}, \mathbf{w^{(i)}})$ are defined as posterior P_s , prior P_r and log-likelihood, respectively. In summary, the model parameters μ and σ can be learned progressively by repeating the following steps (details are shown in Fig.1 (a)):

1. Generate S ϵ's from $\mathcal{N}(0, \mathbf{I})$ for each weight;
2. Obtain S samples for each weight via $\mathbf{w^{(i)}} = \mu + \epsilon^{(i)} \circ \sigma$;
3. Calculate the loss function $\mathcal{L}(\mathbf{w}, \theta)$, where $\theta = (\mu, \sigma)$;
4. Calculate the gradients with respect to μ and σ;
5. Update model parameters μ and σ.

2.2 Computation Flow of BNN Training

From an algorithmic perspective, Fig.1 (a) illustrates the computation flow of BNN training which consists of three main stages: Forward (FW), Backward (BW) and Gradient Calculation (GC).

Forward (FW) stage aims to calculate the loss of network function $\mathbf{f}$ given an input training example $\mathbf{x}$. For simplicity of discussion, we assume processing a minibatch with the size of 1. In each layer l, for one input training example, Gaussian random variables ϵ_l are sampled S times to obtain S samples of weights $w_l^1 \sim w_l^S$, denoted as process ①. These weights are convolved with their corresponding input samples, i.e., $D_l^1 \sim D_l^S$, producing S samples of the output, which are then treated as the input for the next layer. For the first layer, all weight samples are convolved with the input

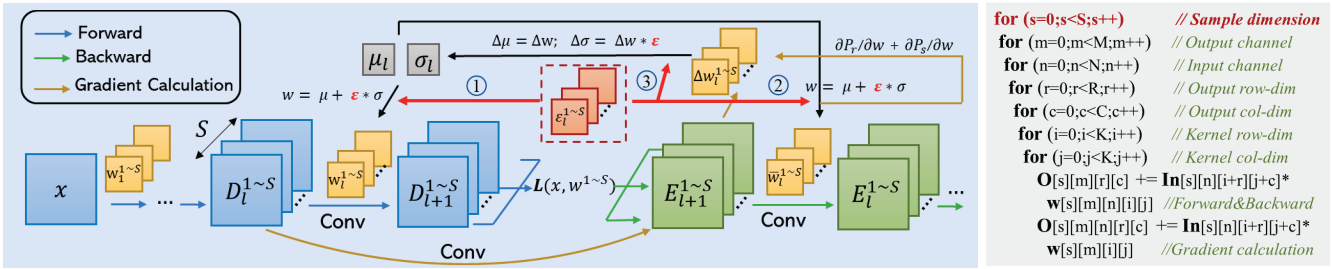


Figure 1: (a) Computation flow of BNN training. (b) 7-dimension for-loop for the convolutional layer in BNN.

x. The outputs of the last layer are compared with the groundtruth to obtain the loss (error).

Backward (BW) stage propagates the network errors from the last layer to the first layer. In each layer l, S samples of weight matrices are reconstructed using the original Gaussian random variables ϵ and model parameters (μ, σ), denoted as process ②. The reconstructed kernels are then rotated 180° and convolved with the corresponding samples of errors to obtain the errors of the previous layer, i.e., $E_l^1 \sim E_l^S$.

Gradient Calculation (GC) stage updates the model parameters μ and σ to minimize the training loss, which requires to calculate the gradients of the model parameters, $\Delta\mu$ and $\Delta\sigma$. The gradient of a sampled weight comes from prior P_r, posterior P_s and likelihood $P(\mathbf{y}|\mathbf{x}, \mathbf{w}^{(i)})$. The gradient of likelihood is generated by convolving the feature maps $D_l^1 \sim D_l^S$ with the errors $E_{l+1}^1 \sim E_{l+1}^S$. This part is the same as the normal DNN training. For the gradients of prior and posterior, they can be easily derived once the original weights are reconstructed because the computation for both prior and posterior requires no intermediate feature maps. Finally, the S samples of the gradients are summed up and then multiplied with a small coefficient to produce the weight updates Δw. Based on the sampling rule $w = \mu + \epsilon * \sigma$, Gaussian random variables ϵ are used to calculate the final updates $\Delta\sigma$. This step corresponds to step ③.

Fig.1 (b) illustrates the detailed computation within a single BNN's convolutional layer. The key feature here is a sample dimension that adds on top of normal DNNs' 6-dimension convolution. Note that different samples execute independently without any data exchange.

3 CHALLENGES OF BNN TRAINING

Traditional DNN training. DNN training has long been considered as a slow and energy harvesting task [16, 25, 52]. On the surface, the massive energy consumption and high latency mainly come from millions of Multiply-accumulate operations (MACs) and intensive data movement between memory and processing elements (PEs). As the unit energy cost (J/bit) of off-chip memory accesses is orders of magnitude higher than that of MACs [11, 15, 30], data movement usually poses greater challenges for energy-efficient DNN training [56]. Moreover, the ongoing development of low-precision training techniques [23, 27, 55] can potentially reduce the unit energy cost of MACs, but this could also result in a proportionally higher impact on the overall training's energy efficiency from the data movement.

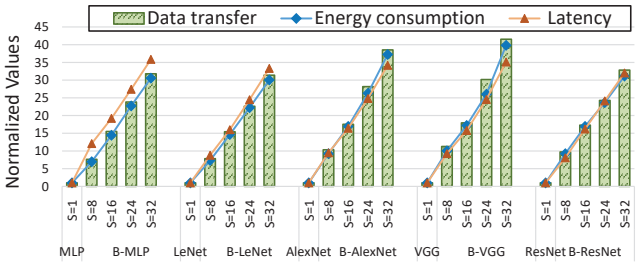


Figure 2: Comparison between five BNN models and their corresponding baseline DNN models.

Current BNN training. Compared to the traditional DNN training, BNN training inefficiency is further exacerbated by the requirement of training for *an ensemble of sampled DNN models*, shown in Fig. 1 (a). This is necessary because a sufficient number of training samples is essential for building a robust BNN model. But it could also incur an explosive amount of data movement during the training process.

To further quantify this, we investigate the impact of number of samples on the overall BNN training efficiency. We implemented five types of widely-adopted BNN models representing a broad range of domains, as well as their corresponding DNN models. Note that BNN models are typically built upon their matching DNN models, e.g., Bayesian AlexNet or *B-Alexnet* is based on AlexNet. For verification purposes, the training process is performed on a general Diannao-like DNN accelerator equipped with output stationary dataflow [10]. Detailed experimental setup can be found in Section 7.1. Three metrics are used for training evaluation, including data transfer, overall energy consumption, and training latency. The data transfer represents the amount of data that are read from and written to the off-chip memory. Due to the architectural heterogeneity of the five BNN models, each result is normalized to its corresponding baseline DNN model. Fig.2 shows that a BNN model with only 8 samples would drastically increase the off-chip data transfer by an average of 9.1× compared with its corresponding DNN model. This number grows to 35.3× as the number of BNN training samples scales up to 32. Specifically, for B-VGG model with 16 samples (s=16), training each input example for one iteration would require 22.6GB data transfer from/to off-chip memory, which is 17.9× increment over the original VGG model. Since the off-chip memory access is often considered a high-cost operation, a large amount of data transfer during BNN training could produce massive energy consumption and potentially lead to performance

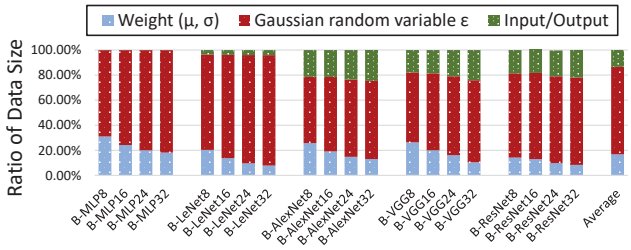


Figure 3: The ratio breakdown of the total off-chip data transfer across different BNN models.

degradation. For example, we observed that the overall energy consumption and training latency on 32 samples incur an average of 33.2× and 31.8× increment over those on the baseline DNN models, respectively.

Fig.3 shows the breakdown of the total off-chip data transfer when the accelerator evaluates every input training example during one training iteration. It can be observed that Gaussian random variables ϵ takes up the major portion of the total data transfer (i.e., 71% on average). Meanwhile, the weight parameters (μ, σ) and the input/output feature maps only contribute to 16% and 12% on average, respectively. There are several reasons behind such dominating presence of ϵ. First, as a unique variable introduced by BNN execution, ϵ must be stored and reused in two different stages. As shown in Fig.1 (a) ①, during the forward stage, S samples of ϵ are generated from the local random number generators for each pair of (μ, σ) to obtain S samples of weights. After that, ϵs have to be stored into the off-chip memory due to its large data volume and reside there until the later weight reconstruction during the backward stage (②) and the gradient of σ computation during the gradient calculation (GC) stage (③). Note that recent memory-centric approaches such as vDNN [45], Echo [65] and SuperNeurons [54] reduce the memory accesses through smart recomputation in backpropagation via selected small intermediate data from forward propagation. However, since ϵs are a large amount of independent random numbers that cannot be recomputed, these works cannot help reduce intensive memory accesses in BNN training. Second, the size of ϵ is much larger than the weight parameters (μ, σ) and the intermediate feature maps/errors. Since one pair of weight parameters (μ, σ) requires S samples of ϵ for weight sampling, the total size of ϵ can be $S/2$ times of the weight parameters. And for the current BNN models, the size of weights (i.e., half of (μ, σ)) is still much larger than the size of feature maps. For instance, on average the size of weights is 122× of the size of feature maps/errors across five BNN models. Therefore, although input/output feature maps also consist of S samples, the total transferred intermediate data size is still much less than that from ϵ.

In summary, the long reuse distance of a large amount of Gaussian random variables ϵ across different training stages is the key problem that causes a huge amount of off-chip memory accesses (the transferred amount of ϵs grows linearly with the sample size). This further leads to massive energy consumption and potential performance degradation during BNN training. Besides the existing DNN accelerator, such a challenge is also observed on conventional CPU/GPU platforms as the cross-stage memory access of ϵ is inevitable in the BNN training algorithm. Therefore, a special solution is needed.

4 KEY DESIGN INSIGHTS OF SHIFT-BNN

To overcome these challenges brought by the excessive data movement for the Gaussian random variables ϵs (or GRVs), we propose a novel design that is able to *eliminate all the memory accesses related to ϵ without training accuracy loss*. We made a key observation that the nature of software-level"forth-back" training procedure shares similarity with the classic hardware-level reversed shifting of Linear Feedback Shift Register (LFSR) which is used in BNNs to generate the Gaussian random variables ϵ [9]. Specifically, we can potentially retrieve all the ϵs locally during the Backward stage through *shifting the LFSRs backward*, instead of storing them during the Forward stage. In the following subsections, we will first introduce the principles of LFSR function, and then illustrate how to use LFSR reversed shifting to retrieve Gaussian random variables ϵs. Finally, we showcase a detailed example to demonstrate the feasibility of our strategy while also exposing some potential hardware-level issues when directly applying it to BNN training.

4.1 Generating GRVs via LFSR Shifting

According to the Central Limit Theorem [8], a binomial distribution $\mathbf{B}(n, p)$ can approximate a Gaussian distribution $\mathbf{N}(np, np(1-p))$ if n is large enough. Here n represents the total number of independent trials and p denotes the possibility of success for each trial. For instance, assume if there are n individual bits that have the equal possibility of being 0 or 1, the total number of "1s" in these n bits will follow the binomial distribution $\mathbf{B}(n, 0.5)$, and further approximate the Gaussian distribution as $\mathbf{N}(0.5n, 0.25n)$ when n is large enough. Based on this insight, previous efforts [3, 9, 14, 31] have proposed efficient Gaussian Random Number Generator (GRNG) by implementing an n-bit LFSR for uniformly distributed random bits generation and an adder tree for counting the number of "1s". The structure of an 8-bit Fibonacci LFSR is illustrated in Fig. 4(a). In each cycle, values in the tap registers, i.e., R_4, R_5, R_6 and R_8, are combined using three XOR gates and produce one bit to update the value in the head register R_1 (highlighted in blue). Meanwhile, the rest of the values shift to the neighbour register from left to right and the value in the tail register R_8 is dropped (highlighted in red). Through this procedure, the LFSR creates a random bit sequence named "pattern" upon every shifting. For each pattern, the number of "1"s are counted by the adder tree to form a Gaussian random variable (GRV).

4.2 Retrieving ϵ via Pattern Reproduction

Assume we employ one LFSR to generate ϵs for sampling all the weights during BNN training. At the Forward stage, ϵs are generated sequentially to sample from the first weight of the first layer to the last weight of the last layer, during which the LFSR continuously shifts from its initial pattern #1 to the latest pattern #N. At the Backward stage, we notice that the generated ϵs are requested in a reversed order, i.e., from the latest pattern #N to the initial pattern #1 of the LFSR, due to the two key features of the training process. At the layer-level, back-propagation executes from the last layer to the first layer, thus the ϵs generated in the last layer in the Forward stage are needed first. At the kernel-level, constructing the kernels that were rotated 180° during back-propagation is equivalent to sampling the previous weights reversely (shown in Fig. 5 (a)). The

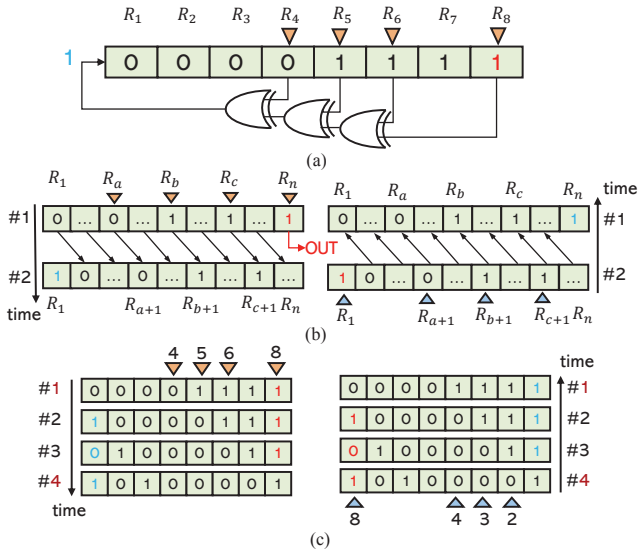


Figure 4: (a) An 8-bit Fibonacci LFSR. (b) Illustration of reproducing the previous patterns by shifting the LFSR reversely. (c) Demonstration of shifting the 8-bit LFSR reversely to obtain the previous patterns in 4 cycles. Note that #N refers to the pattern number.

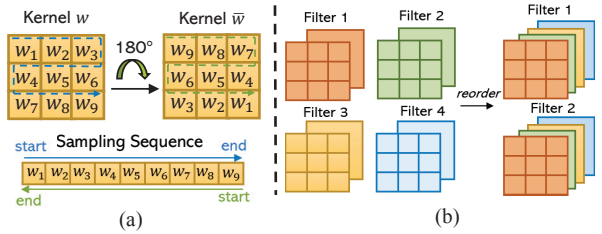


Figure 5: (a) Kernel rotation and its relation with reversed sampling sequence. (b) Kernel reorganization.

aforementioned insights motivate us to reproduce the previous LFSR patterns also in a reversed order so that all the previous ϵs can be *retrieved locally by LFSRs* instead of storing/fetching them during Forward/Backward stage.

Key design insight. This comes from our finding that reproducing previous LFSR patterns can be simply accomplished by shifting the current LFSR pattern in an opposite direction, combined with three XOR operations on certain registers within an LFSR, as illustrated in Fig. 4 (b). Assume a n-bit LFSR with taps **t**=(a,b,c,n) is shifting right to generate the latest pattern #2 from its initial pattern #1. The value in the head register $R_1^{'}$ of pattern #2 is generated by XORing the tail tap R_n with other taps R_c, R_b, R_a in an order:

$$R_1^{'} = ((R_n \oplus R_c) \oplus R_b) \oplus R_a \tag{2}$$

where $\oplus$ denotes XOR operation. Meanwhile, the value in the tail register R_n is dropped from the LFSR. In order to reproduce pattern #1 from #2, the values in $R_1, R_2 ... R_{n-1}$ of pattern #1 can be obtained by left shifting pattern #2. Now the key question is how to reproduce the value in R_n of pattern #1 since it has been dropped previously. Interestingly, for the XOR operation, one can prove that $A = C \oplus B$ if $A \oplus B = C$. Thus we rewrite Eq.2 in a reversed order:

$$R_n = ((R_1^{'} \oplus R_a) \oplus R_b) \oplus R_c \tag{3}$$

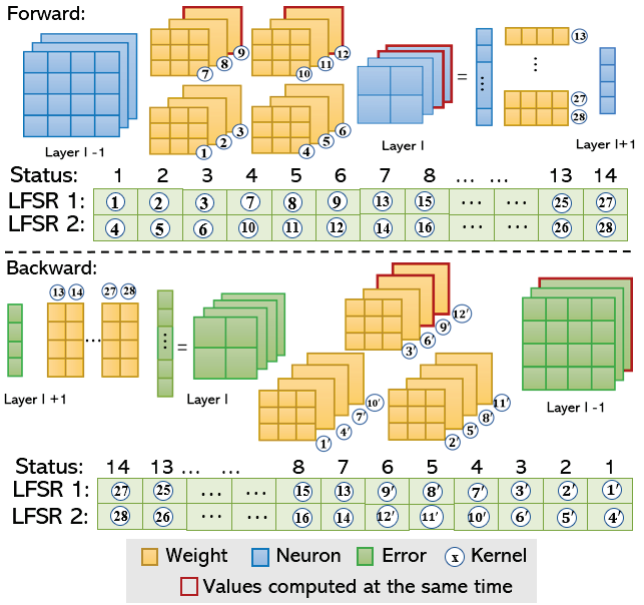


Figure 6: An example of directly applying LFSR reversion strategy to BNN training.

where $R_1^{'}$ is the head register of pattern #2, and R_a, R_b, R_c in pattern #1 are actually $R_{a+1}, R_{b+1}, R_{c+1}$ in pattern #2. Therefore, we can simply set $R_1, R_{a+1}, R_{b+1}, R_{c+1}$ as tap registers of pattern #2 *for the retrieval of R_n in pattern #1*, as shown in the right part of Fig. 4(b). Furthermore, since the LFSR in pattern #2 shifts reversely, the tail register R_n of pattern #2 should be updated by XORing $R_1^{'}, R_{a+1}, R_{b+1}, R_{c+1}$ of pattern #2 orderly. In this fashion, this interesting feature can always be leveraged to retrieve the value in R_n through Eq.3. As can be seen, pattern #1 is successfully retrieved from pattern #2 via very simple logic operations. Fig. 4 (c) provides an example of reversing an 8-bit LFSR to retrieve the previous patterns.

4.3 Potential Issues of Directly Applying LFSR Reversion to BNN Training

Fig. 6 depicts the details of applying our LFSR reversion strategy in a two-layer (convolution + fully-connected (FC)) BNN training. For simplicity of discussion, we assume two LFSRs are deployed for GRN generation. During forward stage, for the convolutional layer, the LFSRs shift from status 1 to 6. Each status contains 9 sequential patterns to generate GRVs for a 3×3 kernel (each pattern per weight). For the FC layer, the LFSRs continue shifting from status 7 to 14. Each status contains 4 sequential patterns for a 1×4 weight vector. During the Backward stage, by shifting the LFSRs reversely, all the previous status are retrieved in a reversed sequence that satisfies the weight fetching request by backpropagation. Note that for convolutional layers, the flipped (180° rotated) kernels $\bar{x}$ can be constructed by the reversed order of x according to Fig. 5(a). And for the FC layers, since the internal weight order of each weight column (e.g.,1×4 matrix) is not altered, the original weight matrices can all be retrieved via LFSR reversion. However, as shown in Fig.5 (b), since the kernels are reorganized across the input channel (N) dimension and output channel (M) dimension during the Backward

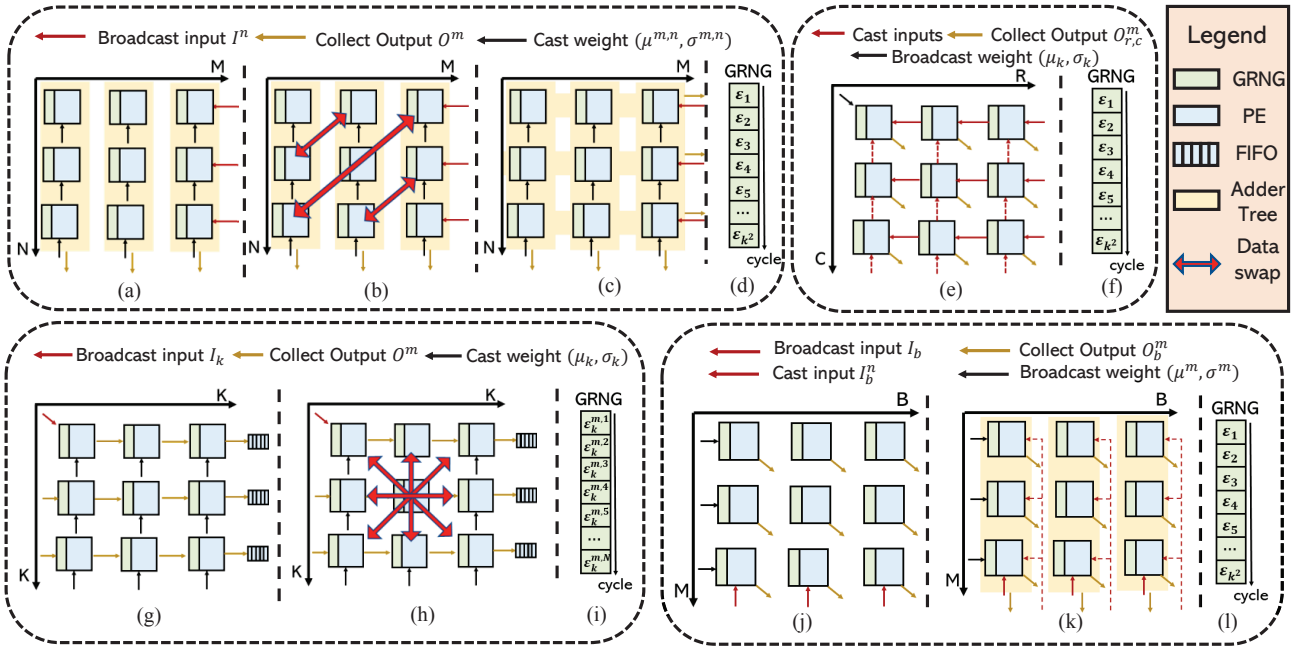


Figure 7: (a)~(d): Basic MN-mapping, modified MN-mapping-v1, modified MN-mapping-v2 and GRNG of MN-mapping. (e)~(f): Basic RC-mapping, and GRNG of RC-mapping. (g)~(i): Basic K-mapping, modified K-mapping-v1 and GRNG of K-mapping. (j)~(l): Basic BM-mapping, modified BM-mapping-v1 and GRNG of BM-mapping.

stage, the computation flow could become inconsistent with that in the Forward stage. For example, at status 6 during the Forward stage in Fig.6, the partial sums calculated by kernel ⑨ and ⑫ are *accumulated separately* for the last two output channels (i.e., the blue blocks highlighted by red at layer l). When applying our LFSR reversion, kernel ⑨′ and ⑫′ will be constructed at status 6 during Backward stage. At this time, instead of being accumulated separately, the partial sums calculated by kernel ⑨′ and ⑫′ are *added together* for one single output channel (i.e., the green block highlighted by red at layer $l-1$). Although our LFSR reversed shifting can still retrieve all the ϵs, such computation inconsistency between the Forward and Backward stages may pose significant design inefficiency for training accelerator design. Furthermore, this factor complicates the design choice selection due to the unclear impact our LFSR reversion strategy may pose on accelerators that adopt different computation mapping schemes. Thus, it is important to first understand the accelerator design space for our shift-BNN.

5 DESIGN SPACE EXPLORATION

As discussed in Section 2.2 (also see Fig. 1 (b)), processing a typical DNN layer during any training stage can be decomposed into a six-dimension for-loop execution. Instead of executing each dimension sequentially, the state-of-the-art DNN accelerators usually select several dimensions and compute them simultaneously, during which MACs along a certain dimension are mapped onto a group of Processing Elements (PEs) that operate in parallel. Choosing different mapping dimensions creates a significant divergence in design efficiency. Generally, there have been three major types of computation mapping strategies for DNN *inference*: kernel (K-dimension) mapping, e.g., systolic array [21], input channel and output channel (MN-dimension) mapping, e.g., Diannao [10], NVDLA [41], and output feature mapping (RC-dimension) mapping, e.g., Shidiannao [19]. Since DNN *training* could also perform mini-batch processing, a batch and output channel (BM-dimension) mapping method [60] is also under consideration. To efficiently apply our design insights into BNN training, we comprehensively study the impact of our LFSR reversed shifting strategy on the four types of state-of-the-art computation mappings to explore the design space for BNN training accelerator. Specifically, we qualitatively discuss the design possibility by using each mapping,and finally select the optimal mapping to support our proposed Shift-BNN design. In the following analysis, we apply superscript m and n to denote the index of output and input channel, and subscript k, (r, c) and b to denote the weight location inside a kernel, the neuron/error location on an output feature map and the index of a training example in a mini-batch, respectively.

MN-dimension mapping. Fig. 7 (a) illustrates a basic architecture for MN-dimension mapping. The x-axis of the 2-D PE array (we assume the size is 3×3 for simplicity) represents M-dimension mapping and the y-axis represents N-dimension mapping. As BNN training demands weight sampling, a GRNG is attached to each PE to generate ϵs for weight parameters (μ, σ), which will be the common case among all four types of mapping methods. In each cycle, an input neuron I^n from a certain input channel n broadcasts horizontally to a row of PEs, where each PE calculates the partial sums for a certain output channel m. These partial sums are collected vertically by an adder tree (denoted by the yellow bar) and summed up until a output neuron is generated. In this scheme, a PE located at coordinate (m, n) will require a $K \times K$ kernel from input channel n and output channel m to produce the partial sum of an output neuron. Therefore, during FW, the LFSR in each

GRNG generates $\{\epsilon_1, \epsilon_2, ..., \epsilon_{K^2}\}$ sequentially to produce a sampled kernel $\{w_1, w_2, ..., w_{K^2}\}$, as shown in Fig. 7 (d). With the proposed LFSR reversion strategy, the flipped kernel can be reconstructed by shifting the LFSR reversely during BW. However, also during this stage, as the kernels are also reorganized in the MN-dimension, the partial sums generated in PE rows should be summed up instead of being accumulated separately (Sec.4.3). This results in the inconsistent computation patterns between FW and BW. To address this inconsistency in a uniform architecture design, one possible solution is to swap the Gaussian random variables, i.e., ϵs, between PE (m, n) and PE (n, m) and then load the corresponding weight parameters and input neurons during the BW stage, as shown in Fig. 7 (b). Nevertheless, such design requires extra interconnections between PEs, leading to $O(n^2)$ wiring overhead for a $n \times n$ PE array, which hinders design scalability. Moreover, there must be an equal number of PEs in a row and a column due to the swapping mechanism, which further limits the design flexibility. Fig. 7 (c) shows an alternative design that avoids the data communication between PEs. In this design, during BW the partial sums generated by a PE row are summed up to an output neuron with duplicated adder trees. The partial sums generated by a PE column are accumulated separately by directly sending each of them to the output buffer. However, this method still requires an n-input adder tree for each row of PEs, which incurs extra resource and energy overheads.

RC-dimension mapping. Fig. 7 (e) shows the basic output feature map (RC) dimension mapping strategy, where neurons on a $R \times C$ output feature map are mapped to a $R \times C$ 2-D PE array and computed simultaneously. In each cycle, one weight from a $K \times K$ kernel is broadcast to all PEs while a group of new input neurons are fed to the rightmost (or bottom) PEs. The partial sums stay in the PE and are accumulated to generate the output neurons as the input neurons flow from right to left (or bottom to up) through the PE array. Since the weight is fetched sequentially from a $K \times K$ kernel, the GRNG also produces $\{\epsilon_1, \epsilon_2, ..., \epsilon_{K^2}\}$ during FW. Thus, the flipped kernels can be reconstructed by shifting LFSR reversely during BW. Furthermore, since RC-dimension mapping is irrelevant with M- or N-dimension parallelism, it will not suffer from the ϵ swapping issue from MN-mapping. Nevertheless, kernel reorganization still has a slight impact on RC-mapping. During the FW stage, since the kernels are fetched along the N-dimension first and then M-dimension, the partial sum of an output neuron is accumulated inside the PE continuously until the output neuron is generated. However, during the BW stage, the kernels are fetched along the M-dimension first and then N-dimension; so the partial sum of an output neuron is sent to the output buffer and waits to be read and accumulated in the PE intermittently. Therefore, two types of control modes are required in RC-mapping.

K-dimension mapping. Fig. 7 (g) shows the basic kernel (K) dimension mapping method, where a $K \times K$ kernel is mapped to a $K \times K$ 2-D PE array and stays until all the computation related to that kernel is completed. In each cycle, an input neuron is broadcast to all the PEs and multiplied with $K \times K$ weights inside a kernel. The partial sums are propagated and accumulated through the PEs to generate the output neurons. Under this scheme, during FW the PE array requires the kernel from the next input channel when the computation of the current kernel is finished. Hence, the GRNG generates ϵs for weights along the N-dimension sequentially from the first to the last input channel, i.e., $\{\epsilon_k^{m,1}, \epsilon_k^{m,2}, ..., \epsilon_k^{m,N}\}$, as shown in Fig. 7 (i). During BW, reverse shifting LFSR can retrieve the original kernels from the last to the first input channel. However, K-dimension mapping can not reorder the weights to construct the flipped kernels required by the BW stage as the weights inside a kernel are sampled simultaneously. In fact, due to the kernel flipping, the ϵ generated by a certain PE during FW is required by another PE during BW. Fig. 7 (h) illustrates a solution for K-dimension mapping: adding datapaths between PEs for ϵ swapping. However, similar to the MN-dimension-v1 (as shown in Fig.7 (b)), this design causes $O(n^2)$ wiring overhead for a $n \times n$ PE array. Moreover, due to the kernel reorganization, K-mapping also requires two types of control modes for different accumulation manners.

BM-dimension mapping. Fig. 7 (j) illustrates the basic batch and output channel (BM) dimension mapping strategy, where the horizontally distributed PEs are processing different training examples and the vertically distributed PEs are calculating neurons in different output channels separately. In each cycle, a pair of weight parameters (μ^m, σ^m) from a certain output channel m is broadcast to an entire row of PEs while an input neuron I_b from a certain training example b is broadcast to an entire column. The output neurons can be collected in each PE. As the weights inside a certain kernel are requested sequentially (shown in Fig. 7 (l)), LFSR reversion can help reconstruct the flipped kernels. However, due to the kernel reorganization, the reconstructed kernels in a column of PEs should be used for N-dimension computation instead of M-dimension computation. Specifically, at the BW stage, the partial sums generated by PE columns should be summed up instead of being accumulated separately. To address this issue, an additional n-input adder tree is required for each PE column. Meanwhile, different input neurons I_b^n from input channel n are sent to each PE column, resulting in two different input buffer designs (Fig. 7 (k)). Therefore, this architecture not only incurs large hardware overhead but also leads to high design complexity.

In conclusion, the RC-dimension mapping strategy (Fig. 7 (e)) only incurs modest design overhead compared to the other three mapping methods when applying our LFSR reversion strategy, which makes it an ideal fundamental computation mapping for designing our Shift-BNN architecture.

6 SHIFT-BNN ARCHITECTURE DESIGN

6.1 Architecture Overview

Figure 8 illustrates the overall architecture of our proposed Shift-BNN training accelerator, which comprises of a 3D PE array distributed to 16 Sample Processing Units (SPUs), a weight parameter buffer (WPB), and a central controller. Each SPU consists of an input/output neuron buffer (NBin/NBout), 16 slices of GRNG and function units, a 4 × 4 PE tile, a 4 ×4 array of shift units, and a crossbar. Following the aforementioned LFSR reversion technique and the computation mapping consideration, our accelerator presents the following features: (1) *a hybrid dataflow* that adopts RC-dimension on 2D PE tiles and sample-level parallelism across SPUs, both of which exploit significant opportunities for data reuse; (2) *an efficient GRNG design* which can generate Gaussian random variables

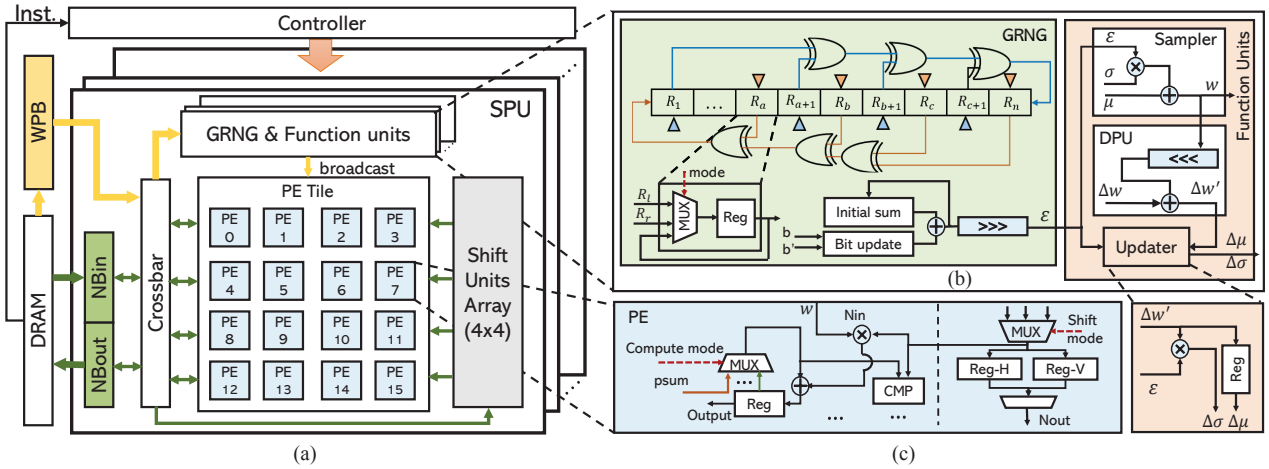


Figure 8: (a) Overview of our proposed Shift-BNN training accelerator. (b) The microarchitecture of GRNG and function units. (c) PE implementation for RC-mapping computation flow.

ϵs sequentially during FW stage and reproduce the previous ϵs reversely during BW stage; (3) *function units* design that satisfies necessary mathematical operations, i.e., weight sampling, derivative calculation of prior and posterior, and weight updating during the BNN training; (4) *light implementation* of RC-dimension mapping architecture by using a PE tile, an array of shift units and a crossbar.

6.2 SPUs and Dataflow

Since the weight parameters (μ, σ) are shared among sampled models, it is natural to process a batch of sampled models in parallel to increase the data reuse of weight parameters. Our design leverages such opportunities by allocating the workloads of training each sampled model to an individual SPU, which operates independently and in parallel with other SPUs. Each SPU is further equipped with the RC-dimension mapping scheme that maximizes the data reuse of input neurons on a 2D feature map. We describe the main features of an SPU as follows.

PE tile, shift unit and crossbar. All convolution operations are performed in the 2D PE tile during all three stages of BNN training (i.e., FW, BW and GC). For simplicity of discussion, we use the FW stage as an example to illustrate the datapath design and the computation flow. Fig. 8 (a) shows the datapath for a convolutional layer, in which a sampled weight from the GRNG & function units is broadcast to all the PEs and multiplies with the input neuron, which will shift to the left (or up) neighbour PE in the next cycle (Fig.7(e)-(f)). To support this type of dataflow, a dedicated PE design is implemented upon a typical inference accelerator [19] that adopts RC-dimension mapping, shown in Fig.8 (c). The right part of the PE is a shift unit. It determines which input neuron (Nin) should be received by the PE and which neuron that is stored in Reg-H/Reg-V should be sent (Nout) to the other PEs. The selected input neuron and the broadcast weight will then enter into the computation unit, which is depicted at the left part of the PE and performs basic MAC operations, ReLU functions and max pooling operations to produce the output neurons. Importantly, due to the kernel reorganization and ϵ reproducing technique at the BW stage (Sec.5), our PE design supports *two types of accumulation modes*. (1) During the FW stage, since the kernels are fetched along the N-dimension first and then M-dimension, the partial sum is repeatedly fetched back to the PE, depicted by the green arrow in Fig. 8 (c). (2) During BW stage, the kernels are fetched along the M-dimension first and then N-dimension, thus the partial sum (named psum in the figure) is fetched from NBout and then gets accumulated in the PE intermittently, depicted by the orange arrow in Fig. 8 (c). Our PE design switches between these two accumulation modes for FW and BW stages. Furthermore, to satisfy the complex data requests from the PE tile, a crossbar is inserted between WPB, NBin, NBout and PE tile to select the appropriate data read from the buffer. Additionally, instead of using a column buffer in [19], we employ a light-weight 4×4 shift units array which stores the candidate input neurons that the PE tile will need in the next four cycles. The array is organized in the same way as the PE tile spatially and each shift unit is actually the same as the right part of the PE for simple data shifting operations.

Efficient GRNG design. A SPU contains 4×4 GRNGs, which corresponds to the 4×4 PE tile. For a convolutional layer, since one weight is shared by every PE, only one GRNG needs to be enabled to generate one ϵ at a time. While for a FC layer, PEs require different sampled weights from the GRNG & function units thus all GRNGs are enabled to provide ϵs to sample weights for their corresponding PE. Fig. 8 (b) left illustrates the microarchitecture of a single GRNG which consists of a 256-bit LFSR and an ϵ generator. The GRNG features two properties. Firstly, it possesses three operating modes. (1) The forward mode for FW stage, during which the LFSR shifts from left to right. Each register (except R_1) of LFSR receives the values from the left neighbour register (named R_l) while R_1 gets updated by the orange taps. (2) The backward mode for BW stage, during which the GRNG switches to the reverse mode and shifts from right to left. Each register (except R_n) of LFSR receives the values from the right neighbour register (named R_r) while R_n gets updated by the blue taps. (3) The idle mode, during which registers in the LFSR receive their own values and will not be updated. Secondly, since counting the number of "1s" (or the sum) of a LFSR pattern with an adder tree may cause large overhead [9], the proposed ϵ generator uses a more efficient way to generate ϵs based on the LFSR patterns.

Specifically, we store the sum of the bits in the LFSR's initial seed in a register and track the difference between the old value (b) and the updated value ($b^{'}$) at R_1 or R_n depending on the operating mode. The difference, i.e., bit update, will be added to the initial sum to form the current sum of LFSR which are then used to update the register of the initial sum.

Function units. The function units consist of a sampler, a derivative processing unit (DPU), and a weight parameter updater. As a whole, the function units receive the (μ, σ) and ϵ from the crossbar and the GRNG respectively, and accomplish two tasks: weight sampling and final gradient calculation of the weight parameters. During both FW and BW stages, the weight sampling is performed in a sampler that applies the weight parameters (μ, σ) to the Gaussian random number ϵ using a multiplier and an adder. The produced weight is sent to the PE tile and the DPU. During the BW stage, the DPU and the updater are both activated. The DPU uses the received reconstructed weight to calculate the derivatives of the sum of the prior and posterior with respect to the weight, Δw_p. By decomposing the prior and posterior terms into a log form, Δw_p can be approximated as $\frac{w}{\sigma_c^2}$. Since σ_c is a constant value of prior distribution and is usually chosen as 0.5, we thus calculate the Δw_p by left shifting w 2 bits. The Δw_p is then added to the gradient of likelihood computed in the GC stage to obtain the final gradient $\Delta w^{'}$. Lastly, in order to update the weight parameters, the updater calculates the gradients of (μ, σ) using $\Delta w^{'}$ and ϵ, which corresponds to the process ③ in Fig.1 (a). The produced $(\Delta\mu, \Delta\sigma)$ will be further averaged across different SPUs and then used to update the weight parameters.

Buffer design. To support the dataflow of RC-mapping in an SPU, we follow a similar design principle in [19] to organize the data in the neuron buffer,i.e., NBin/NBout. NBin/NBout comprises multiple banks. Each bank provides the neurons requested by a PE row through the crossbar. For the weight parameter buffer (or *WPB*), we split it into two sub-buffers that store μ and σ separately. Each sub-buffer is also designed to consist of multiple banks and each entry of the bank stores the weights for a PE row. For a convolutional layer, one weight parameter is selected by the crossbar at each cycle while for an FC layer the entire entry read from the bank is sent to a PE row. Note that although the convolution operands (e.g., weight, neuron, error, and gradient) vary across the three BNN training stages, our uniform design of data organization in WPB, NBin and NBout is beneficial for the buffer function swapping. For example, during the BW stage, the error feature maps of layer $l+1$ stored in NBout can serve as the weights for the gradient calculation of layer l by temporarily treating the NBout as WPB.

7 EVALUATION

7.1 Experimental Methodology

BNN models and training datasets. We evaluate Shift-BNN by training on five representative BNN models. Among them, B-MLP [9] (fully-connected BNN with 3 hidden layer) is trained with MNIST [18]. B-LeNet (built on LeNet[37]) is trained with CIFAR-10 [34]. These two networks are mostly adopted to handle small but safety-critical tasks. B-AlexNet (built on AlexNet[35]), B-VGG (built on VGG16 [48]) and B-ResNet (built on ResNet-18 [28]) are trained with ImageNet datasets [17], which are used to deal with more complex tasks in the unfamiliar environments. For generality, the BNN models are trained with various number of samples, e.g., 8, 16, 32, 64, and 128 (if needed) samples.

Comparison cases. To demonstrate the effectiveness of Shift-BNN, we compare it with three training accelerators: Firstly, since Shift-BNN adopts RC-mapping as the fundamental design strategy, we compare it with the *RC-accelerator* that adopts RC-mapping strategy but without LFSR reversion technique. Secondly, since MN-mapping is commonly used in existing DNN training accelerators [39, 64], we employ an *MN-accelerator* that adopts MN-mapping strategy without LFSR reversion technique as the baseline accelerator for generality, which is also used for our preliminary investigation in Sec.3. Thirdly, to verify the analysis about design alternatives (see Sec.5), we further test the effectiveness of our LFSR reversion strategy on MN-accelerator by comparing with an *MN-Shift-accelerator* that adopts both MN-mapping strategy and LFSR reversion technique. To overcome the challenges caused by our LFSR reversion to the MN-mapping scheme, we follow the design principle in Fig. 7 (c). For fair comparison, all accelerators employ 16 4×4 PE tile and are allocated with on-chip buffer of the same size. The 16 PE tiles process 16 sampled models simultaneously for the same extent of weight parameter reuse. We evaluate the energy efficiency (performance/power) of Shift-BNN and compare with the modern GPU, i.e., Nvidia Telsa P100. We use Pytorch [43] to implement and train the BNNs from scratch, and the training hyperparameters (e.g., batch size, epochs. etc) are kept the same as in other comparison cases. The execution latency and energy consumption are extracted from the GPU runtime information obtained by Nvidia Profiler [42].

Experimental Setup. All accelerator designs are implemented in Verilog RTL and synthesized on a Xilinx Virtex-7 VC709 FPGA evaluation board. For off-chip memory access, the accelerators communicate with two sets of DDR3 DRAM that provide sufficient data transfer rate to the PE tiles via a Memory Interface Generator [58]. The execution time results are obtained from the post-synthesis design and the energy consumption is further evaluated with Xilinx Power Estimator (XPE) [59]. The data precision for all architectures is set to 16-bit and the operating frequency is set as 200MHz.

Training quality. Figure 9 compares the training curve of B-LeNet when using the vanilla BNN training algorithm on Pytorch (baseline) and Shift-BNN. The training hyperparameters and data type are kept the same in baseline and Shift-BNN. It can be seen that Shift-BNN does not affect the overall training iterations to convergence and the final accuracy. Similar behavior is observed on the other networks. This is because our LFSR reversion strategy fundamentally does not modify the training algorithm and simply manages to accurately retrieve all the ϵs during the entire training process. Hence, we only evaluate and validate the training quality results on Shift-BNN when using different bit length. Table 1 shows how different bit lengths affect the validation accuracy of five BNN models. The accuracy results are obtained after the same training epochs for a certain network. As can be seen, training with 16-bit precision only brings an average 0.31% accuracy drop compared with single-precision training. This negligible loss may be due to the error tolerance nature of the sampling process during BNN training. While Shift-BNN can employ 32-bit floating point

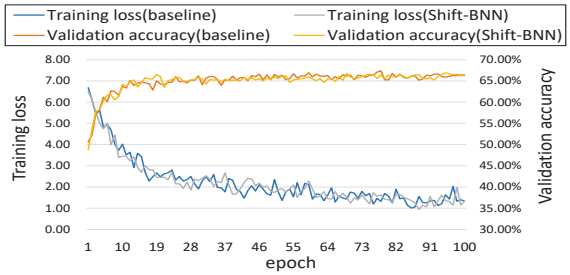


Figure 9: Validation accuracy and training loss over training time for Shift-BNN and vanilla BNN training algorithm on B-LeNet trained with CIFAR-10.

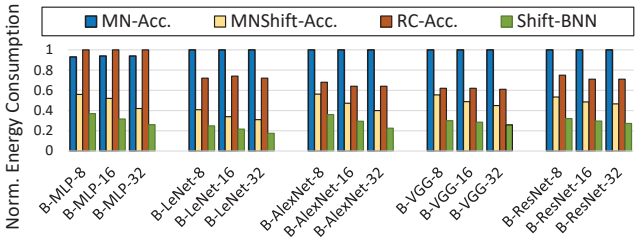


Figure 10: Energy consumption comparison between Shift-BNN and other designs.

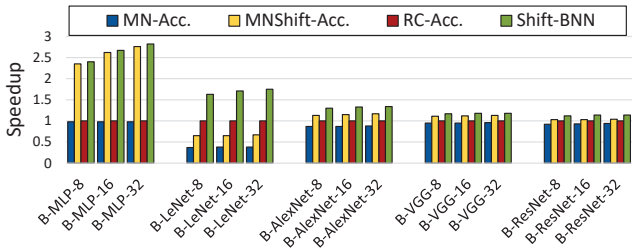


Figure 11: The speedup of Shift-BNN compared with other accelerators.

arithmetic to achieve lossless training, the lower precision training is more attractive as lower precision computation potentially consumes much less energy.

Table 1: Data type vs validation accuracy.

Network	B-MLP	B-LeNet	B-AlexNet	B-VGG	B-ResNet
Dataset	MNIST	CIFAR-10	ImageNet	ImageNet	ImageNet
Val-acc(8b)	95.67%	62.80%	NaN[1]	45.50%	NaN
Val-acc(16b)	98.05%	65.62%	59.95%	67.52%	68.12%
Val-acc(32b)	98.11%	65.81%	60.10%	67.76%	69.03%

[1]The network hardly converges due to the low precision 8-bit BNN training.

7.2 Evaluation Results

Effectiveness on energy and performance. Fig. 10 illustrates the energy consumption of Shift-BNN compared against other accelerators. As it shows, the Shift-BNN accelerator achieves an averagely 62% (up to 76%), 70% (up to 82%), and 39% (up to 44%) energy consumption reduction compared with RC-accelerator (RC-Acc), MN-accelerator (MN-Acc), and MN-Shift-accelerator (MNShift-Acc), respectively. The outstanding energy reduction of Shift-BNN is from the elimination of ϵ's DRAM accesses by using our LFSR reversion strategy. The MNShift-Acc reduces the energy consumption by 53%

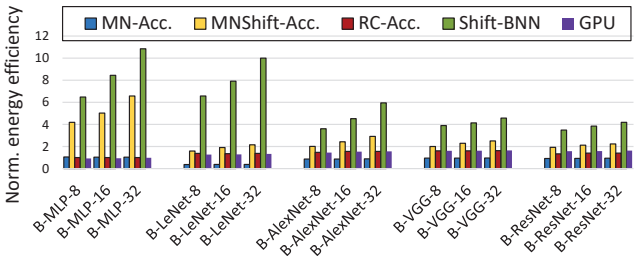
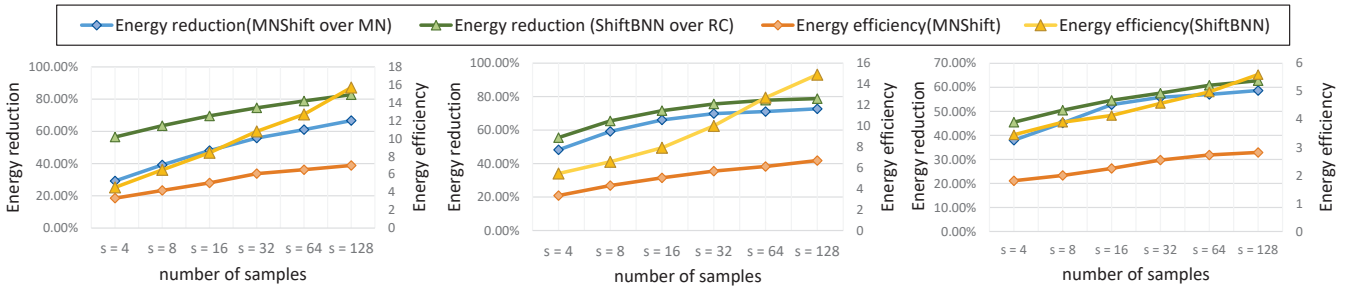


Figure 12: Energy efficiency comparison between Shift-BNN and other designs.

averagely compared with MN-Acc which is less than that of Shift-BNN over RC-Acc (i.e., 62% reduction). This implies that our LFSR reversion technique is also effective on MN-accelerator but reaps less energy saving than applying on RC-accelerator. As discussed in Section 6, this is caused by the large design overhead, e.g., duplicated adder trees, etc, when applying LFSR reversion strategy to MN-mapping scheme. We further observe that Shift-BNN achieves 68% and 70% energy consumption reduction over RC-Acc when evaluating on B-MLP and B-LeNet models, respectively. These number are larger than that of other BNN models. This is because ϵ takes a larger portion in the total off-chip data transfer, and off-chip memory access consumes a larger portion of total training energy consumption for B-MLP and B-LeNet.

Since Shift-BNN mainly targets on reducing the data transfer during training, it is interesting to see if the data transfer reduction can be converted to performance improvement. Fig. 11 shows the speedup of Shift-BNN over other accelerators. From the figure, we observe that Shift-BNN accelerator achieves an average 1.6× (up to 2.8×) speedup over RC-Acc. We found that the reduced execution time mainly comes from the removal of all memory accesses of ϵ in FC layers. As we know, the memory access of ϵ and the computation in a certain layer can be done in parallel by using double-buffering. Thus, in the computation-dominated convolutional layers, removing the memory access of ϵ may not reduce the latency. However, in the parameter-dominated FC layers, the memory access (including storing in FW and fetching in BW) time of S *samples* of ϵ significantly exceeds the computation time since the number of MACs in FC layers are much smaller than that of convolutional layers. For example, the memory access time of ϵ is 8 × over computation time in the 1st layer of B-MLP-8. Accordingly, there is an obvious variance in the performance improvement across different BNN models. For instance, for the fully-connected B-MLP models, the Shift-BNN gains the maximum 2.6× speedup on average, while for the convolution dominated B-VGG and B-ResNet models, Shift-BNN achieves an averagely 1.18× performance improvement.

Fig. 12 shows the energy efficiency of Shift-BNN accelerator compared with other designs. The energy efficiency is defined as throughput per watt (GOPS/Watt). It is shown that Shift-BNN boosts the energy efficiency by 4.9× (up to 10.8×), 10.3× (up to 26.1×) and 2.5× (up to 4.6×) averagely compared with RC-Acc, MN-Acc and MNShift-Acc, respectively. The highest energy efficiency achieved by Shift-BNN is observed on B-MLP-32 model, which enjoys significant reductions on both energy consumption and latency. Furthermore, Shift-BNN also yields averagely 4.7× energy efficiency compared with Telsa P100. We observe that the GPU outperforms the baseline when training deeper BNNs with larger



(a) B-MLP (b) B-LeNet (c) B-VGG

Figure 13: The energy reduction of Shift-BNN (MNShift-Acc) over RC-Acc (MN-Acc) and energy efficiency of Shift-BNN and MNShift-Acc when training with different sample size.

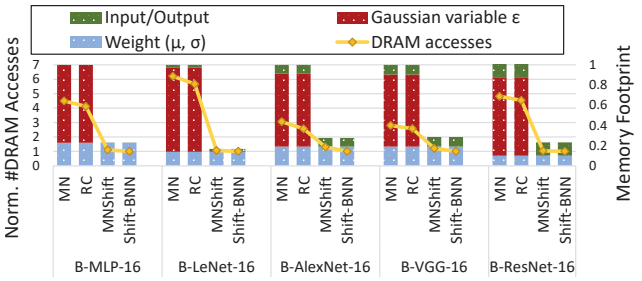


Figure 14: The effectiveness of our LFSR reversion strategy on reducing DRAM accesses and memory footprint.

sample size because of its highly parallel computing and sufficient memory bandwidth. However, it is still beaten by the variants of Shift-BNN that are equipped with our techniques, e.g., even MNShift outperforms GPU by 1.9× energy efficiency. This is because the off-chip memory access of a large amount of GRVs can not be avoided when training BNNs on GPUs either.

Reduction of DRAM accesses and memory footprint. Fig. 14 shows the number of DRAM accesses and memory footprint breakdown of four accelerators when training on BNN models with 16 samples. For the DRAM accesses, we observe that the MN-Acc and RC-Acc always require much more DRAM accesses than Shift-BNN and MNShift-Acc in different BNN models. For example, the number of DRAM accesses in MN-Acc (RC-Acc) are 5.7× (5.8×) larger than that in MNShift-Acc (Shift-BNN) in the ϵ-dominated B-LeNet-16 model. Even in the wider and deeper models (e.g., B-VGG-16 and B-ResNet-16) where the weight parameters and intermediate feature maps occupy a considerate portion of total data transfer, Shift-BNN still gains averagely 2.6× reduction on DRAM accesses. The significant reduction of DRAM access is the major source of Shift-BNN's high energy efficiency. As various lower-precision training techniques [4, 23, 62], e.g., 8-bit integer training, have been proposed recently, the cost of MACs could become much less. Thus the memory saving techniques of Shift-BNN could have more benefits once these techniques are extended to BNN models. Furthermore, as the figure shows, both Shift-BNN and MNShift-Acc reduce averagely 76.1% memory footprint during training compared with accelerators without LFSR reversion technique. From the figure, we can observe that the memory footprint taken by Gaussian variable ϵ is completely eliminated by MNShift-Acc and Shift-BNN.

Scalability to larger sample size. In some high-risk applications, one may need a more robust BNN model to make decisions, thus requires training BNNs with a larger sample size to strictly approximate the loss function in Eq.1. We evaluate three BNN models including B-MLP, B-LeNet and B-VGG by training them with different number of samples and report the corresponding energy consumption reduction and energy efficiency under different hardware designs. As can be seen, for all three models, the energy reduction achieved by both MNShift-Acc and Shift-BNN increases as the sample size becomes larger. For example, the energy savings increase from 55.5% to 78.8% as the sample size grows from 4 to 128 in B-LeNet. The outstanding scalability of our LFSR reversion technique is because of the increasing ratio of ϵ in the total off-chip memory accesses when we use more samples. We observe the similar increase in energy efficiency for MNShift-Acc and Shift-BNN as the training sample increases. Lastly, compared with MNShift-Acc, Shift-BNN achieves higher energy efficiency with various sample sizes.

Table 2: Resource usage of Shift-BNN components.

Resource	PE tile	Shift array	Function units	GRNGs	NBin /NBout
LUT	966	222	785	2277	0
FF	469	464	399	4224	0
DSP	16	0	32	0	0
BRAM	0	0	0	0	48
P_{avg} (W)	0.076	0.016	0.008	0.005	0.112

Resource usage and power. Table 2 lists the resource usage and average power of different hardware modules in one SPU. As can be seen, the shift units array and function units consume less LUT and FF resources compared with the PE tile. Although the function units requires more DSPs to implement function units due to the sampling, derivative calculation and updating processes, their average power dissipation is much smaller than that of PEs since only 1 of 16 function units is activated during convolutional layers. The similar effect can be observed on GRNGs whose average power is only 0.005W, albeit occupying more LUT and FF resources than others.

8 RELATED WORKS

Accelerators for BNNs. There is an increasing demand for designing specific BNN accelerators recently. VIBNN [9] optimizes the hardware design of GRNGs and proposes an FPGA-based implementation for BNN inference. FastBCNN [53] targets on accelerating the BNN inference via neuron skipping technique. [61] proposes a BNN inference accelerator by leveraging post-CMOS technology. Different from the above efforts, our work proposes a highly efficient BNN accelerator that focuses on optimizing the training procedure.

DNN training optimization has been extensively studied [39, 44, 49, 60, 64]. For example, eager pruning [64] and Procrustes [60] exploit the weight sparsity during the training stage by leveraging aggressive pruning algorithms and develop customized hardware to improve the performance. Procrustes also employs LFSR-based GRNGs but in purpose of enabling weight initialization and decay. Since our work reveals the key challenge in BNN training and mainly focuses the reducing the data transfer of ϵ which is irrelevant to sparsity, the above works are orthogonal to ours.

Reducing DRAM energy consumption Many works focus on addressing costly DRAM accesses during the DNN inference or training process. EDEN [33] leverages approximating DRAM technique to reduce the energy and latency while strictly meets the target accuracy. Shapeshifter [36] explores the opportunities in shortening the transferred data width during DNN inference. These works are orthogonal to ours since we explore the unique feature of BNN training and eliminate intensive data transfer without accuracy loss from a different perspective.

9 CONCLUSION

In this paper, we reveal that the massive data movement of GRVs is the key bottleneck that induces the BNN training inefficiency. We propose an innovative method that eliminates all the off-chip memory accesses related to the GRVs without affecting the training accuracy. We further explore the hardware design space and propose a low-cost and scalable BNN accelerator to conduct highly efficient BNN training. Our experimental results show that our design achieves averagely 4.9× (up to 10.8×) boost in energy efficiency and 1.6× (up to 2.8×) speedup compared with the baseline accelerator.

ACKNOWLEDGMENTS

This research is partially supported by NSF grants CCF-2130688, CCF-1900904, CNS-2107057, University of Sydney faculty startup funding, and Australia Research Council (ARC) Discovery Project DP210101984.

REFERENCES

[1] Alexander Amini, Ava Soleimany, Sertac Karaman, and Daniela Rus. 2018. Spatial uncertainty sampling for end-to-end control. *arXiv preprint arXiv:1805.04829* (2018).
[2] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565* (2016).
[3] R Andraka and R Phelps. 1998. An FPGA based processor yields a real time high fidelity radar environment simulator. In *Military and Aerospace Applications of Programmable Devices and Technologies Conference*. 220–224.
[4] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. 2018. Scalable methods for 8-bit training of neural networks. *arXiv preprint arXiv:1805.11046* (2018).
[5] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. 2017. Variational inference: A review for statisticians. *Journal of the American statistical Association* 112, 518 (2017), 859–877.
[6] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. 2015. Weight uncertainty in neural network. In *International Conference on Machine Learning*. PMLR, 1613–1622.
[7] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).
[8] Gunnar A Brosamler. 1988. An almost everywhere central limit theorem. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 104. Cambridge University Press, 561–574.
[9] Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. 2018. Vibnn: Hardware acceleration of bayesian neural networks. *ACM SIGPLAN Notices* 53, 2 (2018), 476–488.
[10] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.
[11] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 367–379.
[12] Guilhem Chéron, Ivan Laptev, and Cordelia Schmid. 2015. P-cnn: Pose-based cnn features for action recognition. In *Proceedings of the IEEE international conference on computer vision*. 3218–3226.
[13] Sai Hung Cheung, Todd A Oliver, Ernesto E Prudencio, Serge Prudhomme, and Robert D Moser. 2011. Bayesian uncertainty analysis with applications to turbulence modeling. *Reliability Engineering & System Safety* 96, 9 (2011), 1137–1149.
[14] C Condo and WJ Gross. 2015. Pseudo-random Gaussian distribution through optimised LFSR permutations. *Electronics Letters* 51, 25 (2015), 2098–2100.
[15] Bill Dally. 2011. Power, programmability, and granularity: The challenges of exascale computing. In *2011 IEEE International Test Conference*. IEEE Computer Society, 12–12.
[16] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).
[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.
[18] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.
[19] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 92–104.
[20] Facebook. 2021. Baysian Optimization Research. Retrieved June 8, 2021 from https://research.fb.com/programs/research-awards/proposals/sample-efficient-sequential-bayesian-decision-making-rfp/
[21] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. 2009. Cnp: An fpga-based processor for convolutional networks. In *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 32–37.
[22] Ammarah Farooq, SyedMuhammad Anwar, Muhammad Awais, and Saad Rehman. 2017. A deep CNN based multi-class classification of Alzheimer's disease using MRI. In *2017 IEEE International Conference on Imaging systems and techniques (IST)*. IEEE, 1–6.
[23] Yonggan Fu, Haoran You, Yang Zhao, Yue Wang, Chaojian Li, Kailash Gopalakrishnan, Zhangyang Wang, and Yingyan Lin. 2020. Fractrain: Fractionally squeezing bit savings both temporally and spatially for efficient dnn training. *arXiv preprint arXiv:2012.13113* (2020).
[24] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*. PMLR, 1050–1059.
[25] Abhinav Goel, Caleb Tung, Yung-Hsiang Lu, and George K Thiruvathukal. 2020. A survey of methods for low-power deep learning and computer vision. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. IEEE, 1–6.
[26] Alex Graves. 2011. Practical variational inference for neural networks. In *Advances in neural information processing systems*. Citeseer, 2348–2356.
[27] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International conference on machine learning*. PMLR, 1737–1746.
[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[29] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *Journal of Machine Learning Research* 14, 5 (2013).
[30] Mark Horowitz. 2014. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 10–14.
[31] Minsu Kang. 2010. FPGA implementation of Gaussian-distributed pseudo-random number generator. In *6th International Conference on Digital Content, Multimedia Technology and its Applications*. IEEE, 11–13.
[32] Alex Kendall, Vijay Badrinarayanan, and Roberto Cipolla. 2015. Bayesian segnet: Model uncertainty in deep convolutional encoder-decoder architectures for scene understanding. *arXiv preprint arXiv:1511.02680* (2015).
[33] Skanda Koppula, Lois Orosa, A Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. 2019. EDEN: enabling energy-efficient, high-performance deep neural network inference using approximate DRAM. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on*

Microarchitecture. 166–181.
[34] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.
[36] Alberto Delmás Lascorz, Sayeh Sharify, Isak Edo, Dylan Malone Stuart, Omar Mohamed Awad, Patrick Judd, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Zissis Poulos, et al. 2019. Shapeshifter: Enabling fine-grain data width adaptation in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 28–41.
[37] Yann LeCun et al. 2015. LeNet-5, convolutional neural networks. URL: http://yann.lecun.com/exdb/lenet.
[38] Christian Leibig, Vaneeda Allken, Murat Seçkin Ayhan, Philipp Berens, and Siegfried Wahl. 2017. Leveraging uncertainty information from deep neural networks for disease detection. *Scientific reports* 7, 1 (2017), 1–14.
[39] Mostafa Mahmoud, Isak Edo, Ali Hadi Zadeh, Omar Mohamed Awad, Gennady Pekhimenko, Jorge Albericio, and Andreas Moshovos. 2020. Tensordash: Exploiting sparsity to accelerate deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 781–795.
[40] NHTSA. 2017. *Tesla Crash Preliminary Evaluation Report.* Technical Report. U.S. Department of Transportation, National Highway Traffic Safety Administration.
[41] Nvidia. 2021. Nvidia Deep Learning Accelerator. Retrieved June 8, 2021 from http://nvdla.org/
[42] Nvidia. 2021. Nvidia Visual Profiler. Retrieved June 8, 2021 from https://docs.nvidia.com/cuda/profiler-users-guide/index.html
[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).
[44] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. 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, 58–70.
[45] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.
[46] Kumar Shridhar, Felix Laumann, and Marcus Liwicki. 2019. A comprehensive guide to bayesian convolutional neural network with variational inference. *arXiv preprint arXiv:1901.02731* (2019).
[47] Karen Simonyan and Andrew Zisserman. 2014. Two-stream convolutional networks for action recognition in videos. *arXiv preprint arXiv:1406.2199* (2014).
[48] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[49] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2019. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 56–68.
[50] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. 2013. Deep neural networks for object detection. (2013).
[51] Tesla. 2020. Tesla car crash report 2020. Retrieved June 8, 2021 from https://arstechnica.com/cars/2019/05/feds-autopilot-was-active-during-deadly-march-tesla-crash/
[52] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, et al. 2017. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 13–26.
[53] Qiyu Wan and Xin Fu. 2020. Fast-BCNN: Massive Neuron Skipping in Bayesian Convolutional Neural Networks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 229–240.
[54] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53.
[55] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training deep neural networks with 8-bit floating point numbers. *arXiv preprint arXiv:1812.08011* (2018).
[56] Yue Wang, Ziyu Jiang, Xiaohan Chen, Pengfei Xu, Yang Zhao, Yingyan Lin, and Zhangyang Wang. 2019. E2-train: Training state-of-the-art cnns with over 80% energy savings. *arXiv preprint arXiv:1910.13349* (2019).
[57] Markus Wulfmeier. 2018. On machine learning and structure for mobile robots. *arXiv preprint arXiv:1806.06003* (2018).
[58] Xilinx. 2019. Xilinx Memory Interface Generator. Retrieved June 8, 2021 from https://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf
[59] Xilinx. 2020. Xilinx Power Estimator. Retrieved June 8, 2021 from https://www.xilinx.com/products/technology/power/xpe.html
[60] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2020. Procrustes: a dataflow and accelerator for sparse deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 711–724.
[61] Kezhou Yang, Akul Malhotra, Sen Lu, and Abhronil Sengupta. 2020. All-spin Bayesian neural networks. *IEEE Transactions on Electron Devices* 67, 3 (2020), 1340–1347.
[62] Yukuan Yang, Lei Deng, Shuang Wu, Tianyi Yan, Yuan Xie, and Guoqi Li. 2020. Training high-performance and large-scale deep neural networks with full 8-bit integers. *Neural Networks* 125 (2020), 70–82.
[63] Cheng Zhang, Judith Bütepage, Hedvig Kjellström, and Stephan Mandt. 2018. Advances in variational inference. *IEEE transactions on pattern analysis and machine intelligence* 41, 8 (2018), 2008–2026.
[64] Jiaqi Zhang, Xiangru Chen, Mingcong Song, and Tao Li. 2019. Eager pruning: algorithm and architecture support for fast training of deep neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 292–303.
[65] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1089–1102.