

VIBNN: Hardware Acceleration of Bayesian Neural Networks

Ruizhe Cai*

Ao Ren*

Ning Liu

Caiwen Ding

Department of Electrical Engineering
and Computer Science, Syracuse
University
Syracuse, New York
{rcai100, aren, nliu03, cading}@syr.edu

Luhao Wang

Xuehai Qian

Massoud Pedram

Department of Electrical Engineering,
University of Southern California
Los Angeles, California
{luhaowan, xuehai.qian, pedram}@usc.
edu

Yanzhi Wang

Department of Electrical Engineering
and Computer Science, Syracuse
University
Syracuse, New York
ywang393@syr.edu

ABSTRACT

Bayesian Neural Networks (BNNs) have been proposed to address the problem of model uncertainty in training and inference. By introducing weights associated with conditioned probability distributions, BNNs are capable of resolving the overfitting issue commonly seen in conventional neural networks and allow for small-data training, through the variational inference process. Frequent usage of Gaussian random variables in this process requires a properly optimized Gaussian Random Number Generator (GRNG). The high hardware cost of conventional GRNG makes the hardware implementation of BNNs challenging.

In this paper, we propose VIBNN, an FPGA-based hardware accelerator design for variational inference on BNNs. We explore the design space for massive amount of Gaussian variable sampling tasks in BNNs. Specifically, we introduce two high performance Gaussian (pseudo) random number generators: 1) the RAM-based Linear Feedback Gaussian Random Number Generator (RLF-GRNG), which is inspired by the properties of binomial distribution and linear feedback logics; and 2) the Bayesian Neural Network-oriented Wallace Gaussian Random Number Generator. To achieve high scalability and efficient memory access, we propose a deep pipelined accelerator architecture with fast execution and good hardware utilization. Experimental results demonstrate that the proposed VIBNN implementations on an FPGA can achieve throughput of 321,543.4 Images/s and energy efficiency upto 52,694.8 Images/J while maintaining similar accuracy as its software counterpart.

KEYWORDS

Bayesian Neural Network, Neural Network, FPGA

ACM Reference Format:

Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. 2018. VIBNN: Hardware Acceleration

* Ruizhe Cai and Ao Ren contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-4911-6/18/03...\$15.00

<https://doi.org/10.1145/3173162.3173212>

of Bayesian Neural Networks. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3173162.3173212>

1 INTRODUCTION

As a key branch of machine learning and artificial intelligence techniques, Artificial Neural Networks (ANNs) have been introduced to create machines that can learn and inference [22]. Many different types and models of ANNs have been developed for a variety of applications and higher performance, including *Convolutional Neural Networks* (CNNs), *Multi-Layer Perceptron Networks* (MLPs), *Recurrent Neural Networks* (RNNs), etc. [44]. With the development and broad applications of deep learning algorithms, neural networks have recently achieved tremendous success in various fields, such as image classification, object recognition, natural language processing, autonomous driving, cancer detection, etc. [1, 14, 45, 47].

With the success of deep learning, a rising amount of recent works studied the highly parallel computing paradigm and the hardware implementations of neural networks[2, 12, 13, 24, 29, 40, 42, 46]. These hardware approaches typically accelerate the inference process of neural networks and have shown promising performances in terms of speed, energy efficiency, and accuracy, making this approach ideal for embedded and IoT systems. Despite the significant progress of neural network acceleration , it is well known that conventional neural networks are prone to the *overfitting issue* — situations where the model fail to generalize well from the training data to the test data [20]. The fundamental reason is that traditional neural network models fail to provide estimates with uncertainty information [9]. This missing characteristic is crucial to avoid making over-confident decisions, especially for many supervised learning applications with missing or noisy training data.

To solve this issue, the *ensemble model* has been introduced [17, 21] to combine the results from multiple neural network models, so that the degraded generalization performance can be avoided. As a key example, *Bayesian Neural Network* (BNNs) are capable of forming ensemble models while maintaining limited memory space overhead [32, 58]. In addition, unlike conventional neural networks that rely on huge amount of data for training, BNNs can easily learn from small datasets, with the ability to offer uncertainty estimates and the robustness to mitigate over-fitting issues[20]. Moreover, the overall accuracy can be improved as well.

Specifically, BNNs apply Bayesian inference to provide the principled uncertainty estimates. In contrast to traditional neural networks whose weights are fixed values, each weight in BNNs is a *random number* following a *posterior probability distribution*, which is conditioned on a *prior probability* and its *observed data*. Unfortunately, the exact Bayesian inference in general is an intractable problem and obtaining closed-form solutions requires either the assumption of special families of models[11] or the availability of probability distributions [21]. Therefore, an *approximation method of Bayesian inference* is generally used to ensure low computational complexity and high degree of generality [8, 48]. Among various approximation techniques, *variational approximation*, or *variational inference*, tends to be faster and easier to implement. Besides, the variational inference method offers better scalability with large models, especially for large-scale neural networks in deep learning applications[26], compared with other commonly adopted Bayesian inference methods such as Markov Chain Monte Carlo (MCMC) [4]. In addition to faster computation, the variational inference method can efficiently represent weights (probability distributions) with limited amount of parameters. The Bayes-by-Backprop algorithm proposed by Bluendell *et al.*[9], for instance, only doubles the parameters compared to ANNs while achieving an infinitely large ensemble of models.

Our focused BNNs in this work belong to the category of *feed-forward neural networks (FNNs)* that have achieved great successes in many important fields, such as the HIGGS challenge, the Merck Molecular Activity challenge, and the Tox21 Data challenge [30]. Despite the tremendous attentions on CNNs and RNNs, the accelerators for FNN models are imperative as well, which is noted in the recent Google paper [29] and the very recently invented SeLU technique [30]. With the recent shift in various fields towards the deployment of BNNs [9, 19, 50], hardware acceleration for BNNs becomes critical and has not been well considered in prior works. However, hardware realizations of BNNs pose a fundamental challenge compared to the traditional ANNs: the frequent operations on *probability distributions* requires additional logic circuits designed and optimized for (Gaussian) random number generation.

In this paper, we propose *VIBNN*, an FPGA-based hardware accelerator design for variational inference on BNNs. In VIBNN, posterior distributions of network weights are approximated as Gaussian distributions associated with variational parameters (trainable mean values and variances). As a common practice, network training is performed on CPU/GPU clusters before the parameters for weights distributions. We explore the design space for massive amount of Gaussian variable sampling tasks in BNNs. Specifically, we introduce two high performance Gaussian (psuedo) random number generators: 1) the RAM-based Linear Feedback Gaussian Random Number Generator (RLF-GRNG), which is inspired by the properties of binomial distribution and linear feedback logics; and 2) the Bayesian Neural Network-oriented Wallace Gaussian Random Number Generator. To achieve high scalability and efficient memory access, we propose a deep pipelined accelerator architecture with fast execution and good hardware utilization. It is important to note that BNN is a mathematical model, instead of a specific type of neural network structure. Therefore, the design principles

of VIBNN are orthogonal to the optimization techniques on convolutional layers in previous works [16, 27, 36], and can be applied to CNNs and RNNs as well.

Experimental results suggest that the proposed VIBNN can achieve similar accuracy as its software counterpart (BNNs) with very high energy efficiency of 52694.9 images/J thanks to the proposed GRNG structure.

2 BNNS USING VARIATIONAL INFERENCE

2.1 Bayesian Model, Variational Inference, and Gaussian Approximation

For a general Bayesian model, the *latent variables* are \mathbf{w} and *observed data points* are \mathcal{D} . From the Bayes rule, the *posterior probability* can be calculated as:

$$P(\mathbf{w}|\mathcal{D}) = \frac{P(\mathcal{D}|\mathbf{w})P(\mathbf{w})}{P(\mathcal{D})} \quad (1)$$

where $P(\mathbf{w})$ is called the *prior probability* that indicates the probability of latent variables \mathbf{w} before any data observations. $P(\mathcal{D}|\mathbf{w})$ is called the *likelihood*, which is the probability of the data \mathcal{D} based on the latent variable observations \mathbf{w} . The denominator $P(\mathcal{D})$ is calculated as the integral of sum over all possible latent variables, i.e., $P(\mathcal{D}) = \int P(\mathcal{D}|\mathbf{w})P(\mathbf{w})d\mathbf{w}$.

For most of applications of interests, this integral process is intractable, therefore effective approaches are needed to *approximately* estimate/evaluate the posterior probability. *Variational inference*[28, 52] is a machine learning method for approximating the *posterior probability* densities in Bayesian inference models, with a higher convergence rate (compared to MCMC method) and scalability to large problems. As shown in [21], the variational inference method posits a family of probability distributions $q(\mathbf{w}; \theta)$ with *variation parameters* θ to approximate the posterior distribution $p(\mathbf{w}|\mathcal{D})$.

For simplicity, one can assume that the variational posterior distribution is a Gaussian distribution, then the variational parameters (vector) θ can be specified as $\theta = (\mu, \rho)$, where μ is the vector of mean values and ρ is used to further produce the non-negative vector of standard deviations, i.e., $\sigma = \ln(1 + \exp(\rho))$. Therefore, a sample of \mathbf{w} can be obtained by shifting and scaling *unit Gaussian variables*:

$$\mathbf{w} = \mu + \epsilon \circ \ln(1 + \exp(\rho)) \quad (2)$$

where $\epsilon \sim \mathcal{N}(0, \mathbf{I})$ is a vector of independent unit Gaussian variables, and \circ denotes the element-wise multiplication operation. Therefore, *generating unit Gaussian variables is the key step in generating samples of \mathbf{w}* .

2.2 Gaussian Variational Inference with a BNN

Bayesian Neural Networks (BNNs) are first suggested in the 1990s and studied extensively since then [33, 39]. Compared to conventional neural networks with fixed value weights representing deterministic models, BNNs offer ensembles of models by *incorporating probability distributions over models' weights*, as shown in Figure 1. As a result, BNNs achieve higher overall accuracy and are more robust to over-fitting. In addition, unlike conventional neural networks that require large data sets to achieve good accuracy, BNNs

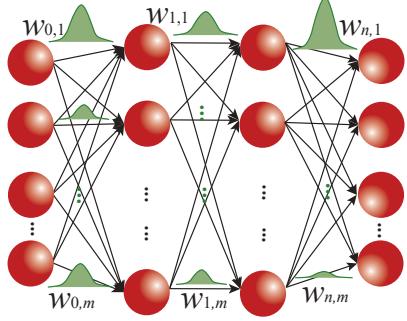


Figure 1: Illustration of model structure of BNNs.

can easily learn from small data with much improved convergence rate.

This paper considers Gaussian variational inference for BNNs. The goal is to perform inference using a BNN with learned posterior distribution $p(\mathbf{w}|\mathcal{D})$ of network weights. To fully utilize the posterior distribution, the network output should be derived by *averaging over the outputs produced according to the posterior distribution*. In other words, the output of network, \mathbf{y} , is calculated as:

$$\mathbf{y} = \mathbb{E}_{p(\mathbf{w}|\mathcal{D})}[g(\mathbf{x}_0, \mathbf{w})] \quad (3)$$

where $g(\mathbf{x}_0, \mathbf{w})$ is the network function and \mathbf{x}_0 is the network input. With the variational inference method, the network output can be represented as:

$$\mathbf{y} \approx \mathbb{E}_{q(\mathbf{w}; \theta)}[g(\mathbf{x}_0, \mathbf{w})] \quad (4)$$

Using a Monte Carlo Sampling, the output can be further approximated as:

$$\mathbf{y} \approx \frac{1}{N} \sum_{i=1}^N g(\mathbf{x}_0, \mathbf{w}^i) \quad (5)$$

where N is the total sample count and \mathbf{w}^i is the i -th Monte Carlo sample drawn from the variational posterior distribution $q(\mathbf{w}; \theta)$, which can be obtained using equation (2). Then the output is estimated as:

$$\mathbf{y} \approx \frac{1}{N} \sum_{i=1}^N g(\mathbf{x}_0, \mu + \sigma \circ \epsilon^i) \quad (6)$$

where ϵ^i is the i -th Monte Carlo sample of unit Gaussian variables. As shown in Figure 1, each weight requires sampling a unit Gaussian random variable. Then a BNN can perform inference using the same method as a normal ANN.

Since our target implementation platforms are low-power, embedded systems based on FPGAs, the network is trained offline, as a widely adopted practice in hardware deep learning research [2, 12, 18, 41, 42, 56], using high performance computing platforms such as GPUs. Afterward, the trained variational parameters (vectors) μ and σ are migrated to the memory of the target FPGA platform.

2.3 Gaussian Random Number Generators (GRNGs)

As introduced before, weights and biases in the BNNs of interests are drawn from Gaussian distributions $\mathcal{N}(\mu, \sigma^2)$. Therefore, it is crucial to rapidly generate high-quality Gaussian random numbers.

The methods of generating *Gaussian Random Numbers* (GRNs) can be classified into four categories: 1) the *cumulative density function* (CDF) inversion method that obtains GRNs simply by inverting the CDFs, such as [7, 37]; 2) the *transformation method* that obtains GRNs through a series of operations on uniform distributions, according to the *Central Limit Theorem* (CLT), such as [38, 49]; 3) the *rejection method* that generates GRNs based on the transformation method, but with one additional rejection step that conditionally rejects some of the transformed numbers, such as the Ziggurat algorithm [35]; 4) the *recursion method* that generates GRNs by combining previously generated GRNs in a linear manner, such as the Wallace method [54].

For hardware implementations, not all algorithms that are successful on software are appropriate. This is because of the restrictions such as hardware resources, power/energy constraints, and implementation complexity. Therefore, the appropriate selection of hardware-based GRNGs is typically application-specific and needs to be carefully investigated. In this paper, we believe the *CLT-based methods* and the *Wallace method* to be the most appropriate choices for hardware neural network acceleration. The main consideration is the lower computation overhead, which facilitates the efficient hardware implementation of BNNs.

One of the major advantages of the CLT-based GRNGs is that uniform random numbers can be easily generated with *linear-feedback shift registers* (LFSRs). However, the generation quality is affected by various factors such as the number of stages in LFSRs, the bit-width, etc. [34]. Being another method with low computation overhead, Wallace algorithm [34] guarantees the correctness by the fact that a linear combination of Gaussian random numbers is still a Gaussian random number [54]. Nevertheless, Wallace algorithm has two main drawbacks: First, an initial pool of Gaussian random numbers is needed, thereby adding requirements of memory storage; Second, the generated random numbers have correlations to some degree. We propose novel hardware implementation of the CLT-based and Wallace method, to sufficiently exploit the advantages of these GRNG methods but at the same time mitigate the drawbacks. Moreover, a series of optimizations are performed to further reduce the hardware cost and meanwhile guarantee the high quality of GRNs generated.

3 VIBNN ARCHITECTURE OVERVIEW

In this section, we present the overview of our VIBNN architecture. As shown in Fig. 2, the architecture of VIBNN accelerator consists of the external memory, on-chip memories, a computation block, a weight generator and a global controller. The external memory initially stores the input features to the input layer and weight parameters (μ, σ). These inputs and weights are then loaded into on-chip memories for online inference. There are two types of on-chip memories: one stores the input features, intermediate results and the inference results; the other stores the weight parameters.

As discussed in Section 2.1, the weights that actually participate in the variational inference process are calculated by equation (2). Hence, a *weight generator* that comprises a GRNG and a *weight updater* is needed. The GRNG generates the random numbers ϵ 's, and the weight updater is responsible for implementing the weight updating equation. The computation block comprises groups of

processing elements (PEs), each consisting of a number of PEs. To improve memory-access efficiency, the *memory distributor* in computation block collects the outputs of PEs and performs necessary operations before writing back to on-chip memories. The PEs work in a time-multiplexed manner to perform all computations in the whole neural network. To correctly implement the overall functionality of VIBNN, the operations of all components are controlled by a *global controller*. Overall, the VIBNN architecture has three key benefits: scalability, portability, and memory-access efficiency.

4 BNN-ORIENTED GRNGS

In this section, we explore the design of GRNGs suitable for BNN hardware implementation. Specifically, we propose two parallel GRNG designs tailored for BNNs: 1) *RAM-based Linear Feedback GRNG (RLF-GRNG)*, which is based on a CLT-based method inspired by the *binomial distribution*. This method is capable of approximating a Gaussian distribution [10] when the sample size is large enough. Importantly, it can be effectively implemented *using RAM with compact additional logics* for indexing and computing. In addition, the control module can be shared among parallel GRNGs. 2) *BNN-Oriented Wallace GRNG*, which is based on Wallace method. The proposed design substantially mitigated the two main drawbacks, i.e., a large initial pool requiring large memory block and multi-loop transformations that result in long latency, making the Wallace algorithm suitable for hardware BNNs.

4.1 RLF-GRNG Design

4.1.1 The Binomial Distribution Approximation Method. A binomial distribution, denoted as $X \sim B(n, p)$, is a discrete probability distribution of the number of successes in a sequence of n independent experiments, each with a boolean valued outcome with the probability of success p . The probability of getting exact k successes in n trials is given by the following probability mass function:

$$f(k; n, p) = Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k} \quad (7)$$

for $k = 0, 1, 2, \dots, n$, where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

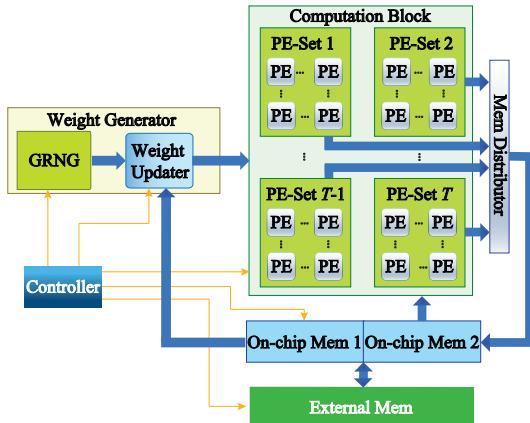


Figure 2: Overall architecture of VIBNN

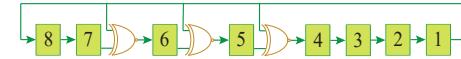
If n is large enough, $B(n, p)$ can reasonably approximate a normal distribution: $N(np, np(1-p))$. More specifically, n is considered to be large enough [53] if:

$$n > 9 \cdot \frac{1-p}{p} \text{ and } n > 9 \cdot \frac{p}{1-p} \quad (8)$$

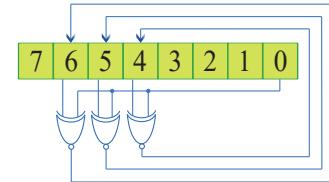
As a simple example, the summation of n individual bits should follow a binomial distribution $B(n, 0.5)$ if each bit has equal chances of being 0 or 1. According to equation (8), if n is greater than 18, the aforementioned distribution can approximate a normal distribution $N(0.5n, 0.25n)$. We adopt this method in the design of a GRNG. For example, an implementation can use a 128-bit *Linear Feedback Shift Register* (LFSR) for random bits generation and a *Parallel Counter* (PC) to convert the number of 1's in the LFSR to a binary number[3]. As a simple method to generate uniformly distributed (pseudo) random numbers, an LFSR [15] implements a well-chosen *linear feedback function*, allowing it to produce all binary combinations in a random order.

When the depth of LFSR (i.e., number of registers) is large enough, e.g., 128, it costs 3000 years to cycle through when clocked at 80MHz[3]. We believe that the degree of randomness is high enough when the number of registers is large. The LFSR implements the linear feedback function by shifting and updating the values in the *taps* using XOR gates. An 8-bit LFSR is shown in Figure 3 (a). The taps for the 8-bit linear feedback function are 4, 5, and 6. The LFSR uses fixed head location (register 1) and shifts its contents in each cycle. Registers at tap locations are updated with the XOR result of their left neighbor and the head. For each tap t :

$$R(t) \leftarrow R(t+1) \text{ XOR } R(1) \quad (9)$$



a: 8-bit LFSR (Register 1 is the head, register 4, 5, 6 are taps)



b: Equivalent 8-bit RLF Logic

Figure 3: Examples of LFSR and RAM based Linear Feedback.

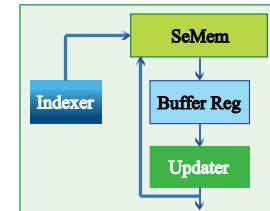


Figure 4: Block diagram of the RLF logic

where $R(t)$ denotes the value of t -th register in the LFSR, and $R(1)$ is the value of the head.

By using a PC, the output of LFSR can be converted to a binary number representing the number of 1's in LFSR. The PC can be implemented using adders in a tree structure. However, the hardware cost for larger inputs can be huge as a 127-input PC requires 120 full adders. This motivates the following RAM-based implementation.

4.1.2 RAM-Based Linear Feedback Function. Despite conceptually straightforward, the binomial distribution approximation method using LFSR and PC is not suitable for parallel implementation due to its high usage of registers and adders. The LFSR requires huge register resources. In addition, even though only taps are updated in each iteration (the numbers of taps are always 3 for 4-bit to 2048-bit LFSR [55]), the PC needs to accumulate all bits of LFSR. Such large PC not only requires huge amount of hardware resources, but also leads to extra computation latency.

To overcome the challenge, we propose RLF, a compact *RAM-based Linear Feedback function* implementation, which incurs lower cost but achieves the same functionality as LFSR. The design requires a much smaller PC that only calculates *the summation of tap values*.

Figure 3 (b) illustrates the operations using an 8-bit RAM-based Linear Feedback (RLF) logic. RLF logic stores the seed in fixed location while using *a self-accumulating indexer*, which changes in each cycle, to track the head and taps to produce a pseudo shift operation. The updated results are fed back to the same locations. Compared to equation (9), for each tap t in the linear feedback function, the corresponding operations on the RLF logic is:

$$x(h + t) \leftarrow x(h + t) \text{ XOR } x(h) \quad (10)$$

where h is the current head location, and $x(i)$ is the value stored in the i -th entry of the vector (seed memory).

As shown in Figure 4, the proposed RLF logic consists of four components: the seed memory (SeMem), the indexer, the updater and the buffer register. The SeMem stores the seeds using several blocks of 2-port RAM. The length of the SeMem is the size of the seeds, and the word width of the SeMem is the number of parallel RLF logics. The buffer register caches the values of taps and the head. The updater performs XOR based operations. The indexer stores the locations of the taps and the head, i.e. h and $h + t$ for all taps in equation (10), and increments them every cycle. This organization allows: 1) efficient hardware utilization for parallel operations, because only one indexer is needed regardless of the number of RLF logics running in parallel; 2) very small PC needed to calculate the seed summation, because only the values of the taps, instead of all seeds, are outputted.

Based on the basic structure of the proposed RLF logic, we perform optimizations in two aspects: 1) the quality of random numbers generated from the seeds, and 2) seed storage scheme. To explain the ideas, we consider a 255-bit RLF logic for 8-bit GRNG (each GRN uses 8-bit representation). The taps for the 255-bit linear feedback function are 250, 252, and 253. According to equation (9), they need to be updated according to the following operations:

$$x(h + 250) \leftarrow x(h + 250) \text{ XOR } x(h) \quad (11a)$$

$$x(h + 252) \leftarrow x(h + 252) \text{ XOR } x(h) \quad (11b)$$

$$x(h + 253) \leftarrow x(h + 253) \text{ XOR } x(h) \quad (11c)$$

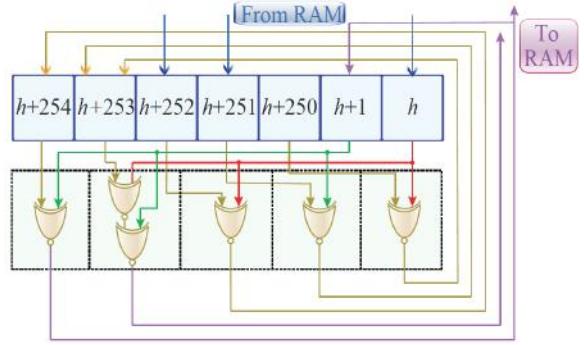


Figure 5: Buffer register and updater for 8-bit RLF-GRNG

The index i is always less than or equal to 255, i.e., $i \leftarrow i - 255$ if $i > 255$. As the number of taps of the 255-bit linear feedback function is 3, the absolute difference between the output summations in two consecutive cycles cannot exceed three. This could affect the quality of random numbers produced, consequently undermining the performance of the BNN. To address this issue, the implemented linear feedback is modified to increase the number of taps by *combining two consecutive updates to create operations involving more bits*, which are:

$$x(h + 250) \leftarrow x(h + 250) \text{ XOR } x(h) \quad (12a)$$

$$x(h + 251) \leftarrow x(h + 251) \text{ XOR } x(h + 1) \quad (12b)$$

$$x(h + 252) \leftarrow x(h + 252) \text{ XOR } x(h) \quad (12c)$$

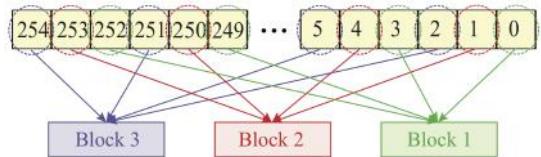
$$x(h + 253) \leftarrow (x(h + 253) \text{ XOR } x(h)) \text{ XOR } x(h + 1) \quad (12d)$$

$$x(h + 254) \leftarrow x(h + 254) \text{ XOR } x(h + 1) \quad (12e)$$

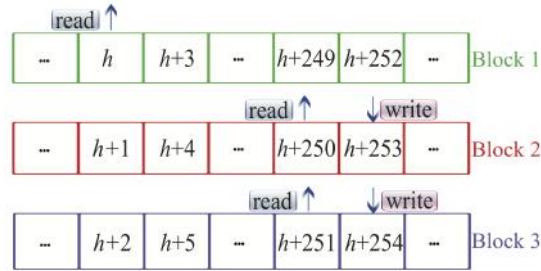
By combining two consecutive updates into a single one, the maximum absolute difference between the outputs in two cycles is increased to five. Accordingly, the index increment per cycle is increased from one to two. The taps for the updated linear feedback function is from 250 to 254. Therefore, each update needs to read seven entries (the head, and second head, and all taps) and write five entries (the taps). The read/write bandwidth can be greatly reduced by optimizing the buffer register.

As illustrated in Figure 5, the buffer register and the updater are designed to minimize RAM access bandwidth. As the combined linear feedback function has five taps and two heads locations, the buffer register is 7-bit. Since the index is increased by 2 in each iteration/cycle, the left 5 bits, which represent the taps, all shift left by 2 bits when being updated; while the right 2 bits are read from the RAM. The updated values of the left 2 bits are then written back to the RAM. Given this circulant operation, the updated value of the leftmost bit ($h + 254$) is the bit next to the head in the following iteration (as $\text{mod}(h + 254 + 2, 255) = \text{mod}(h + 1, 255)$). Therefore, aided by the buffer register and 2-port RAM, each iteration only requires 3 entries to read and 2 entries to write:
Read: $x(h)$, $x(h + 250)$ and $x(h + 251)$.
Write: $x(h + 253)$ and $x(h + 254)$.

To allow the above operations performed in one cycle on 2-port RAM, we propose a *3-block RAM storing scheme* for the 255-bit seeds. As shown in Figure 6, the 255-bit seeds are separated evenly



a: Seed storing scheme



b: RAM operations

Figure 6: A 3-block RAM storing scheme for the 255-bit seed

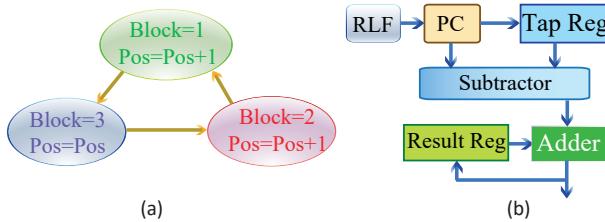


Figure 7: (a) FSM for the indexer (b) Data Flow Graph of the RAM-based Linear Feedback GRNG (RLF-GRNG)

into three blocks based on the modulo operation results over three of their locations. Therefore, the five r/w operations on RAM can be performed in one cycle on three 2-port RAM blocks. The indexer, which stores and updates the locations for each tap and the head, can be implemented as a simple state machine shown in Figure 7 to track the block number (Block) and the relative position (Pos).

4.1.3 Overall RLF-GRNG Design. The data flow graph of the proposed RAM-based Linear Feedback GRNG (RLF-GRNG) is shown in Figure 7. The RLF logic, as discussed before, contains a SeMem for seeds storing, a buffer register for caching taps, an LF updater for updating taps, and an indexer to track tap locations. The RLF logic outputs a stream of updated taps to the PC, which accumulates the output stream. The previous tap summation is stored in the *tap register*. Their difference can be obtained from the subtractor. Finally, the difference is accumulated to the previous random number which is stored in the *result register* to produce a new output. Please note that, the initial value stored in the result register is the summation of the initial values stored in the SeMem. Therefore, this value can be *pre-calculated and stored in memory*.

Figure 8 shows the block diagram of the parallel RLF-GRNG, designed for the efficient implementation of BNNs. The Initialization ROM stores the initial summation results of the seeds for the

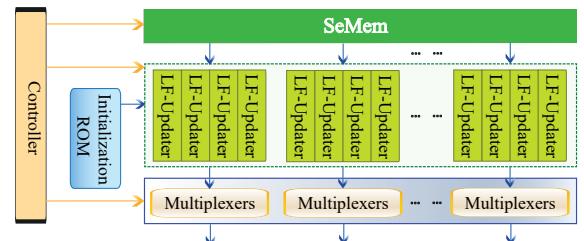


Figure 8: Hardware block diagram of the 2-stage RLF-GRNG

GRNG. The SeMem stores all seeds for random number generations. To produce m n -bit random numbers in parallel, the size of the SeMem is $2^n - 1$ words, where each word is m -bit. Each bit of the word read from the RAM is propagated to an LF-updater for tap update and random number calculation. The updated taps are collected from all LF-updaters and formed into one word to be written back to the SeMem. The results generated by every four LF-updaters are selected sequentially to four outputs, with different orders through its multiplexer for enhanced randomness. All select signals are shared and generated by the controller. The controller also produces indices and memory access signals for the SeMem, as well as command signals for all LF-updaters. Finally, a set of individual Gaussian random numbers are produced in parallel.

4.2 BNNs-Oriented Wallace GRNGs

4.2.1 The Wallace Method. The Wallace method relies on the property that the linear combinations of Gaussian random numbers are still Gaussian distributed. The linear combinations are achieved through multiplying a vector of Gaussian random numbers x by a Hadamard matrix H : $x' = H \times x$. Below is a typical Hadamard matrix:

$$H = \begin{bmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

Thus, the transformed random numbers are:

$$\begin{aligned} x'[1] &= t - x[1]; & x'[2] &= t - x[2]; \\ x'[3] &= x[3] - t; & x'[4] &= x[4] - t; \end{aligned} \quad (13)$$

where $x'[1] - x'[4]$ are newly generated (pseudo) random numbers, $x[1] - x[4]$ are original random numbers fetched from the pool, $t = \frac{1}{2}(x[1] + x[2] + x[3] + x[4])$. This approach does not require multiplication operations. In Wallace algorithm, the original elements in x are randomly chosen from the pool, and after multiple loops of transformations, the newly generated random numbers are written back to random positions in the pool to replace the original elements. In this way, the size of the pool keeps constant. However, the exact implementation in hardware costs another random number generator to produce the random positions, as well as extra delay to perform the multi-loop transformations. Besides, the initial pool should be large enough to ensure the randomness of the generated random numbers and the stability of (μ, σ) . In Section 4.2.2, we propose a *sharing and shifting scheme* that is proven to be capable of overcoming the downsides of the algorithm.

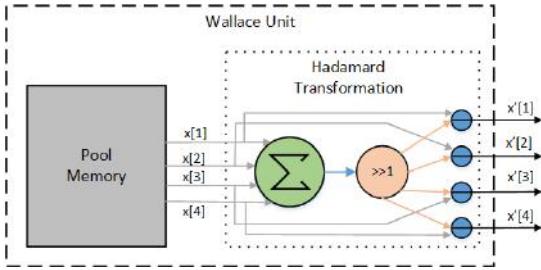


Figure 9: Architecture of the Wallace Unit

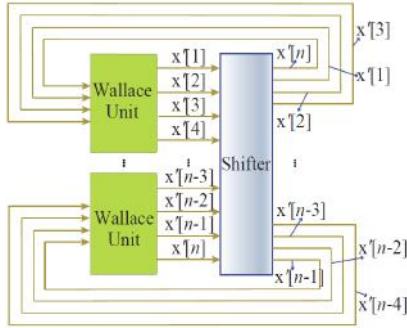


Figure 10: Architecture of the BNNWallace-GRNG

4.2.2 Hardware Design and Optimization of the Wallace Method.
 Figure 9 illustrates the hardware design for performing one Hadamard transformation. The Pool Memory stores the pre-generated Gaussian random numbers, and four random numbers $x[1] - x[4]$ are fetched from the pool in every clock cycle. Then a summation is performed on the four random numbers, which is followed by a one-bit right-shifting to achieve the operation of dividing by 2. The output of the shifter is t mentioned in equation 13, then t will participate in four parallel subtractions to generate new random numbers $x'[1] - x'[4]$. This module is named the *Wallace Unit*, and it is the basic component in our proposed *BNN-oriented Wallace GRNG (BNNWallace GRNG)*. Even though a larger Hadamard matrix can be composed of small Hadamard matrices as $H' = \begin{pmatrix} H & H \\ H & -H \end{pmatrix}$, using the larger one implies that each new random number $x'[i]$ is generated by the computations involving all original random numbers. For instance, if we perform a Hadamard transformation using a 8×8 matrix, $x'[1] = \sum_{i=1}^8 x[i] - (x[1] + x[5])$. Therefore, more hardware resources and longer clock cycle are needed. Based on the above analysis, we implement the Wallace Unit using the 4×4 Hadamard matrix.

As introduced before, large initial pool and multi-loop transforms are required to guarantee that the generated random numbers are highly random, less correlated, and stably distributed. These requirements or downsides of Wallace method are substantially magnified when it is realized on hardware due to the limited resources. We introduce a *sharing and shifting-based optimization* to overcome these downsides. The key insight is: to make small pools of Wallace Units work as a whole, so the memory requirement on each pool is roughly divided by N if there are N Wallace Units. This is achieved by shifting the generated random numbers by one number

before they are written back to the pool, and Figure 10 illustrates the scheme. Due to the shifting, the written back random numbers are partially generated by other Wallace Units. This strategy can increase the randomness and decrease the correlations among the random numbers. Besides, by shifting, the generated random numbers flow through all Wallace Units, so all small pools constitute a large pool. As a result, the stability of (μ, σ) is guaranteed. The experimental results in Section 6.1 demonstrate the effectiveness of BNNWallace-GRNG.

5 FPGA-BASED IMPLEMENTATION OF VIBNN

To implement VIBNN architecture on resource-limited FPGAs, we explore a series of optimization strategies at three levels: *the arithmetic unit-level, the PE-level, and the system-level*, in addition to the optimized GRNGs.

5.1 PE Design

In VIBNN, a PE works as a neuron with architecture illustrated in Figure 11-(a). The Multiplication and Accumulation (MAC) unit with architecture shown in Figure 11-(b) calculates the dot-product of its input features and weights through multipliers and an adder tree. Typically, a neuron in FNNs has hundreds of inputs, but it is infeasible for a PE due to the limited resources on FPGAs. A large number of neurons per PE will decrease the number of PEs that can be implemented on a FPGA, leading to a rigid system implementation and limited optimization space. Therefore, we choose to implement a PE with a reasonable number of inputs and use it in a time-multiplexed manner. Inside a PE, we use an accumulator to accumulate the partial dot-products. After some iterations, the accumulated value will be sent to an adder to add a bias up. Next, the biased dot-product is fed into the *Rectifier Linear Unit (ReLU)* so that the activation result is obtained.

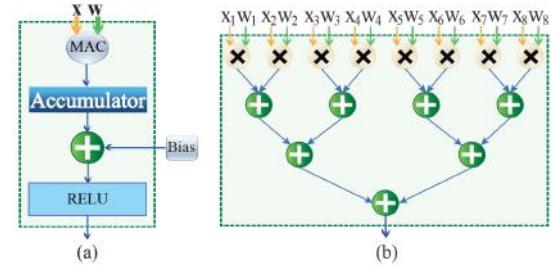


Figure 11: (a) PE Architecture (b) MAC Architecture

5.2 Bit-Length Optimization

Since the arithmetic units are implemented in look-up tables (LUTs) on FPGAs, there is no much optimization space on the structures and implementation principles of those units. The most straightforward and feasible method is to optimize the bit-length of operands, e.g., the input and output size of the arithmetic units. Despite the previous works such as [23, 25, 51] leveraging this technique, it is not trivial to investigate and apply this technique in VIBNN, because the introduced random noises may lead to a different conclusion. We select 16-bit as the starting point, which is widely adopted in

hardware neural network implementations, then a binary search is used to figure out the smallest required bit-length that can guarantee acceptable accuracy. Optimization results are reported in Section 6.3.

5.3 Weight Generator

As shown in Figure 12, the Weight Generator consists of a GRNG, a Weight Updater, and an on-chip Weight Parameter Memory. The Weight Updater receives random numbers sampled by GRNG, then generates a weight sample according to the corresponding variational parameters read from the memory. It applies the variational parameters (μ and σ) to the Gaussian numbers using multipliers and adders.

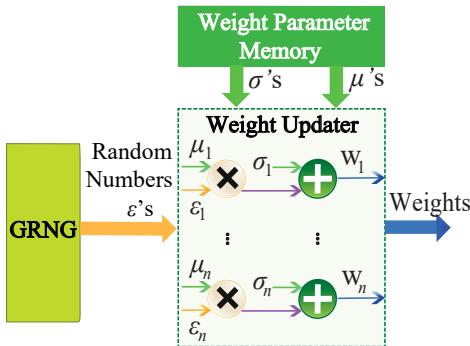


Figure 12: Architecture of the Weight Generator

5.4 Joint Optimization of PE Size/Number and Memory Access

The throughput of a neural network accelerator is mainly determined by computation and communication [57]. In FNNs, the computations can be reduced by applying optimizations that reduce links between neurons, such as weight pruning [25]. In our work, we focus on a different aspect. Specifically, from the computation aspect, the throughput can be improved by increasing computation parallelism, in the form of increasing PE size or increasing the number of PEs. From the communication aspect, the throughput improvements come from reducing the memory traffic. The key problem is that, the computation parallelism and memory traffic are not completely independent factors, so we need to jointly consider the parallelism and memory traffic to obtain the best throughput performance.

On-chip memories are divided into two categories according to different usage purposes: 1) IFMem (input feature memory): the memory that holds input features as well as activation outputs; and 2) WPMem (weight parameters memory), the memory that holds weight parameters (μ, σ)'. The optimization techniques for reducing memory accesses to IFMems and WPMsgs need to be considered separately.

5.4.1 Reducing IFMem Accesses. First, our hardware realized BNNs are aimed at FNN models and one PE corresponds to one neuron, any two PEs will have the same input features in a certain cycle. Therefore, the word size of the IFMem should be $B \times N$ to

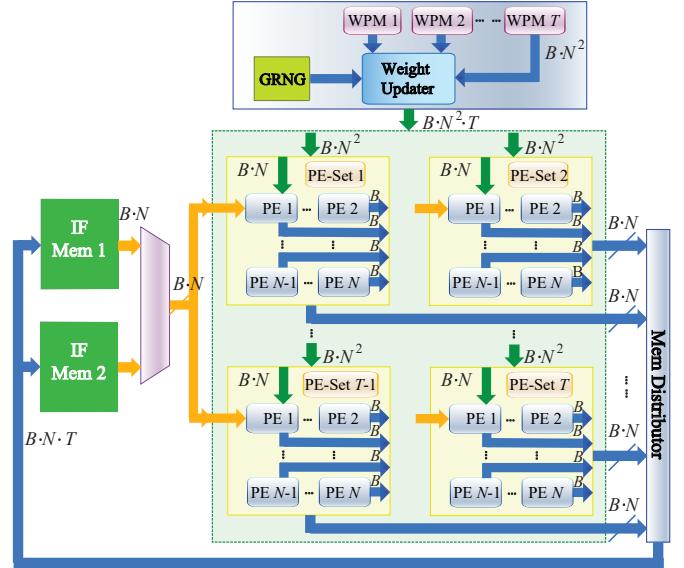


Figure 13: Joint Optimization Architecture

avoid extra delays: all required input features can be sent to PEs at each access to IFMem. As illustrated in Figure 13, in our design, the whole network is constructed on the basis of PE-sets and each set has $S = N$ PEs. Thus the outputs of a PE set correspond to a word to be written back to the IFMem, and all the outputs from different PE sets are buffered in the Memory Distributor. To illustrate how the model works and its advantages, we define three additional parameters: 1) T : the number of PE sets; 2) $MinIn$: the minimum input size of a neuron in the target neural network to be realized; 3) $MaxWS$: the maximum allowable word size of an on-chip memory. Their relations are derived as:

$$T \times S < ceil\left(\frac{MinIn}{N}\right) \quad (14a)$$

$$B \times N \leq MaxWS \quad (14b)$$

$$S = N \quad (14c)$$

$$M = T \times S \quad (14d)$$

where function $ceil(x)$ takes the ceiling of variable x , B is the bit-length of an operand, N is the input size of a PE, and M is the total number of PEs.

The advantages of the design are three-fold: 1) scalability: we can easily change the PE size N and PE sets number T to adopt to neural networks with various size, according to equation (14a); 2) portability: we can change the PE size N , PE sets number T , and even bit-length B to deploy the design on various FPGA platforms, according to equations (14a) and (14b); 3) access efficiency, equations (14a)-(14d) guarantee that all the required inputs of PEs can be read at one access to the IFMem, and the buffered activation results will be written back to the IFMem before the next set of results coming into the Memory Distributor.

Besides, we use two IFMems alternatively to avoid any latent read&write conflicts. For instance, at a certain layer, if IFMem1 is used to access input features, the activation results that are also

input features for next layer will be written into IFMem2. Then at next layer, we switch the roles of IFMem1 and IFMem2.

5.4.2 Reducing WPMem Accesses. Since each input among the whole network has an individual weight, to read all weight parameters required at a certain clock cycle, the word size should be $B \times N \times S \times T$. Here, we only consider μ 's for simplicity, and the WPMems for storing parameters σ 's are just duplicated. However, the maximum allowable word size $MaxWS$ restricts N , S , and T to dramatically small values and the computation parallelism is hence restricted. Therefore, the optimization schemes for reducing IFMem accesses are not appropriate for WPMems. In VIBNN, we propose to use multiple distributed WPMems so that each WPMem corresponds to a PE set for maintaining a structural architecture and the ease of control. In this way, the bandwidth for a WPMem is $B \times N \times S$. Therefore, equations (14a)-(14d) need to be rewritten as (15).

$$T \times S < ceil\left(\frac{MinIn}{N}\right) \quad (15a)$$

$$B \times N \times S \leq MaxWS \quad (15b)$$

$$S = N \quad (15c)$$

$$M = T \times S \quad (15d)$$

where function $ceil(x)$ takes the ceiling of variable x , T is the number of PE sets, S is the number of PEs in a set, B is the bit-length of an operand, N is the input size of a PE, and M is the total number of PEs.

5.5 Deep Pipelining

As shown in Figure 14, the proposed design implements a two-tier pipeline structure. The first tier is placed between weight generator and PE, which holds the sampled weights. The second tier is inserted inside the weight updater and the PE to further reduce system clock period. We insert DFFs between the GRNG and the weight updater. Each PE unit is optimized as a three-stage pipeline. The first stage calculates all multiplications; the next stage accumulates all internal products from previous stage. The ReLU is performed at the final stage.

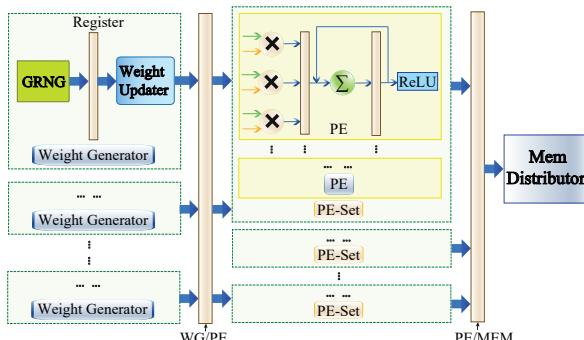


Figure 14: Pipeline structure of the proposed BNN

Table 1: Stability errors to $(\mu, \sigma) = (0, 1)$ of Various Wallace Designs

GRNG Designs	μ Errors	σ Errors
Software 256 Pool Size	0.0012	0.3050
Software 1024 Pool Size	0.0010	0.0850
Software 4096 Pool Size	0.0004	0.0145
Hardware Wallace NSS	0.0013	0.4660
BNNWallace-GRNG	0.0006	0.0038
RLF-GRNG	0.0006	0.0074

6 EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we present the experimental results regarding following respects: (i) the effectiveness and hardware cost of the proposed RLF-GRNG and BNNWallace-GRNG; (ii) the training characteristics of both FNNs and BNNs are compared when they are trained with small data; (iii) the bit-length optimization; (iv) the hardware resource utilization of the proposed VIBNNs; (v) the accuracy comparison between FNNs (software), BNNs (software), and our VIBNN (hardware) when they are applied on image classification task using MNIST dataset [31], the Parkinson Speech Dataset[43], the Diabetics Retinopathy Debrecen Dataset[5], the Thoracic Surgery Dataset[59] and the TOX21 Dataset[6].

6.1 Performance of Proposed GRNGs

The performance and hardware cost of the proposed GRNGs are tested and presented in this section. And also, their advantages and disadvantages are listed and compared according to the experimental results.

First, we implement an RLF-GRNG with 255-bit SeMem and a BNNWallace-GRNG with 8 Wallace Units, in which each has a pool of 256 initial random numbers. Then their performances in terms of randomness and stability are tested and compared with software-based Wallace method with various initial pool sizes. In addition, to illustrate the effectiveness of the sharing and shifting scheme of the proposed BNNWallace-GRNG, a hardware realized Wallace Method with neither sharing and shifting scheme nor multi-loop transformations (Wallace-NSS) is implemented and compared.

The results regarding the stability is shown in Table 1. The distribution of the initial pool is sampled from the standard normal distribution, and the μ Errors and σ Errors are the absolute errors between the generated distributions and the standard normal distribution. From the table, it can be observed that the size of the initial pool significantly affects the stability of the generated distributions. Results suggest that the proposed BNNWallace-GRNG and RLG-GRNG are comparable with the software method with the pool size 4096. Furthermore, both methods can generate multiple random numbers in parallel, which is a big improvement compared to the software method that can only generate 4 random numbers at one time. The memory savings of BNNWallace-GRNG is 2X, which can be further improved by sharing more Wallace Units because the pool size of each Wallace Unit can be further reduced when there are more units participating in the sharing and shifting.

In addition to the stability test, we also perform the randomness test on RLF-GRNG and various Wallace methods, which are shown in Figure 15. Each method is used to generate 100,000 random

Table 2: Hardware Utilization and Performance Comparison between RLF-GRNG and Wallace-based GRNG for 64 Parallel Gaussian Random Number Generation Task

Type	RLF-GRNG	BNNWallace-GRNG
Total ALMs	831/113,560 (< 1%)	401/113,560 (< 1%)
Total Registers	1780	1166
Total Block Memory Bits	16,384/12,492,800 (< 1%)	1,048,576/12,492,800 (8%)
Total RAM Blocks	3/1220 (< 1%)	103/1220 (8%)
Power(mW)	528.69	560.25
Clock Frequency (MHz)	212.95	117.63

Table 3: RLF-GRNG and BNNWallace-GRNG Comparison

	RLF-GRNG	BNNWallace-GRNG
Advantages	low memory usage high frequency high power efficiency	Adjustable distribution high scalability low ALM and register usage
Disadvantages	low scalability	high latency

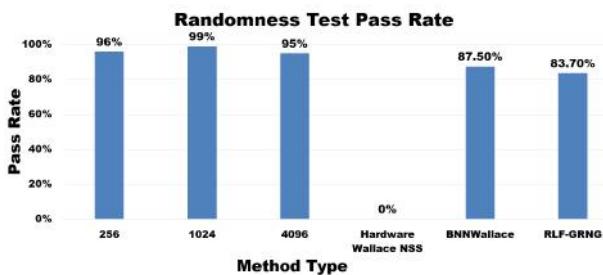


Figure 15: Randomness tests pass rate of software-based Wallace method and BNNWallace with various pool sizes

numbers and tested using the *runtest* function in Matlab. The same test is repeated for 1000 times and the rates of passed tests are collected. It can be concluded that the pool size has insignificant impact on the randomness of the newly generated random numbers. However, which is worth noting is that our proposed BNNWallace-GRNG is comparable with the software-based methods, while the hardware version that has neither sharing and shifting scheme nor multi-loop transforms fails to pass any randomness test. This result verifies the effectiveness of our proposed GRNGs again.

Table 2 presents hardware performance comparison for RLF-GRNG and BNNWallace-GRNG. Both yield highly efficient hardware usage for 64 parallel Gaussian random number generating task. However, the RLF-GRNG utilizes less memory while the BNNWallace-GRNG requires less ALM resources. Given its shallow PC to accumulate taps and less complex computations, the RLF-GRNG is capable to operate at a higher frequency.

As presented in Table 3, both RLF-GRNG and BNNWallace-GRNG have certain limitations. The RLF-GRNG offers relatively limited generality as different size requires individual optimization. In addition, the fact that the RAM width required for RLF-GRNG is exponential to the bit length restricts its application to high bit-length Gaussian variable sampling. However, for 8-bit applications, RLF-GRNG is a highly memory efficient and fast solution. The RLF-GRNG is much more power efficient than the BNNWallace-GRNG as its power consumption is the same as the BNNWallace-GRNG while

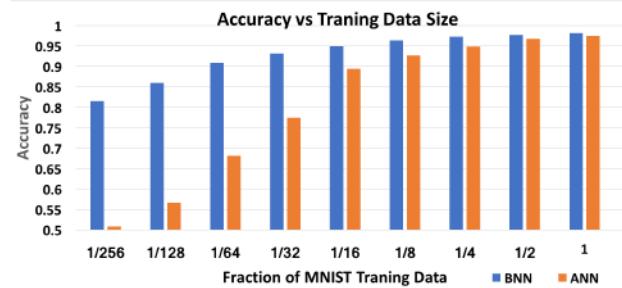


Figure 16: Accuracy comparison between FNN and BNN with a fraction of training data

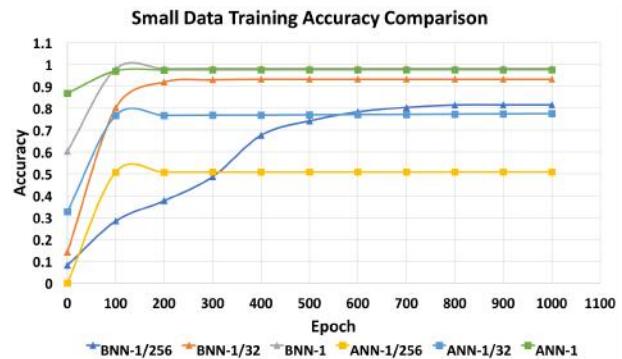


Figure 17: Training accuracy comparison between FNN and BNN with a fraction of training data

operating at a much higher frequency. For parallel random number generation, RLF-GRNG performs better in terms of hardware resource utilization given the limited size of SeMem to implement one extra output.

6.2 Small Data

To demonstrate the performance on small data, we test both BNN and FNN on the MNIST dataset[31]. The MNIST dataset contains 60K images for training and 10K images for testing. Each image is a 28×28 pixel gray-scale bitmap of hand written digit. The task is to classify the images into ten classes (from "0" to "9"). The BNN implemented has 784 inputs, 10 outputs, and 2 hidden layers both with 200 neurons. All layers are fully connected ($784 - 200 - 200 - 10$). We randomly choose a fraction of the training data to train a BNN and an FNN with same structure and test their performance. Figure 16 shows accuracy comparison between FNN and BNN when training with part of the data from 1/256 to entire training data set, and suggest that BNN performances much better as training data size shrinks. Detailed converge performance is shown in Figure 17.

6.3 Bit-Length Optimization Results

According to our experiments on software, the BNNs can achieve 98.1% test accuracy on image classification task using MNIST dataset, so we set 97.5% as the threshold accuracy in selecting a proper bit-length. Figure 18 shows the test accuracy when the bit-length is

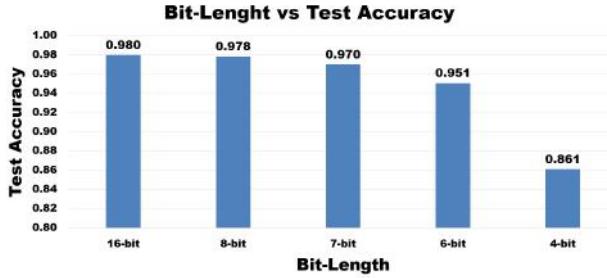


Figure 18: Bit-Length vs Test Accuracy

Table 4: Summary of FPGA resources utilization

Type	RLF-based Network	BNNWallace-based Network
Total ALMs	98,006/113,560 (86.3%)	91,126/113,560 (80.2%)
Total DSPs	342/342 (100%)	342/342 (100%)
Total Registers	88,720	78,800
Total Block Memory Bits	4,572,928/12,492,800 (36.6%)	4,880,128/12,492,800 (39.1%)

Table 5: Performance comparison on the MNIST dataset

Configuration	Throughput (Images/s)	Energy (Images/J)
Intel i7-6700k	10478.1	115.1
Nvidia GTX1070	27988.1	186.6
RLF-based FPGA Implementation	321543.4	52694.8
BNNWallace-based FPGA Implementation	321543.4	37722.1

Table 6: Accuracy Comparison on the MNIST Dataset

Model	Testing Accuracy
FNN+Dropout (Software)	97.50%
BNN (Software)	98.10%
VIBNN (Hardware)	97.81%

set to different values, and we can observe that 8-bit is the smallest bit-length that can reach the accuracy threshold.

6.4 Hardware Resources Usage

We implement the proposed design of VIBNN on an Altera Cyclone V FPGA (Model Number 5CGTFD9E5F35C7). We opt for a 16 PE-set parallel design, each has eight 8-input PEs. The resource utilization results are shown in Table 4. It can be observed that both the two VIBNNs have high throughput and energy efficiency, and the RLF-based VIBNN provides even higher energy efficiency, while the BNNWallace-based VIBNN has higher design flexibility as explained in Section 6.1.

To demonstrate the efficiency of the proposed design, we compared the proposed implementation against software implementations on GPU(Nvidia GTX 1070) and CPU(Intel i7-6700k) on the MNIST dataset. As shown in Table 5, the proposed design is 283× more energy efficient compared with an Nvidia GTX 1070 and 458× more energy efficient compared with an Intel i7-6700k processor.

6.5 Accuracy: Classification Tasks

In this work, we test the classification performance of the BNNs with several classification tasks on both software and hardware

Table 7: Accuracy comparison on classification tasks

Dataset	FNN (Software)	BNN (Software)	VIBNN (Hardware)
Parkinson Speech Dataset (Modified)	60.28%	95.68%	95.33%
Parkinson Speech Dataset (Original)	85.71%	95.23%	94.67%
Diabetics Retinopathy Debrecen Dataset	70.56%	75.76%	75.21%
Thoracic Surgery Dataset	76.69%	82.98%	82.54%
TOX21:NR.AhR	91.10%	90.42%	90.11%
TOX21:SR.ARE	83.41%	83.24%	83.01%
TOX21:SR.ATAD5	93.36%	94.05%	93.67%
TOX21:SR.MMP	89.69%	88.76%	88.43%
TOX21:SR.P53	91.88%	93.33%	92.87%

platforms, comparing with the performance of FNNs. The first task is the image classification on MNIST dataset. As discussed in the previous subsection, the network we use has two hidden layers of neurons apart from the 784 (28×28) inputs and 10 outputs. Compared with the 8-bit fixed point representation used in hardware, the BNN implementation in software uses 32-bit floating point for all operations. Results of accuracy testing are reported in Table 6. The BNN implemented in software has higher classification accuracy than that of FNNs with dropout applied. The proposed implementation of BNN on FPGA degrades only 0.29% accuracy compared to its software model.

To further evaluate accuracy performance, we test on several disease diagnosis datasets including the Parkinson Speech Dataset[43], the Diabetics Retinopathy Debrecen Dataset[5], Thoracic Surgery Dataset[59] and part of the TOX21 dataset[6] in which multiple features of chemicals are given to detect toxic compound. For the parkinson speech dataset, we relocate part of the training data for testing to create a small data training scenario(shown as Parkinson Speech Dataset(modified) in Table 7). As shown in Table 7, BNNs can achieve similar or better accuracy compared to FNNs, especially for small datasets. The accuracy of VIBNN degrades very little compared to the software implementation of BNN.

7 CONCLUSION

In this paper, we propose VIBNN, an FPGA-based hardware accelerator design for variational inference on BNNs. We explore the design space for massive amount of Gaussian variable sampling tasks in BNNs. Specifically, we introduce two high performance parallel Gaussian random number generators: 1) the RAM-based Linear Feedback Gaussian Random Number Generator (RLF-GRNG), which is inspired by the properties of binomial distribution and linear feedback logics; and 2) the Bayesian Neural Network-oriented Wallace Gaussian Random Number Generator. To achieve high scalability and efficient memory access, we propose a deep pipelined accelerator architecture with fast execution and good hardware utilization. The proposed VIBNN achieves high throughput of 321,543.4 Images/s and high energy efficiency up to 52,694.8 Images/J. Experimental results suggest that the proposed VIBNN can achieve similar accuracy performance as software implemented BNNs.

ACKNOWLEDGMENTS

This paper is in part supported by National Science Foundation CNS-1704662, CCF-1717754, CCF-1750656, and Defense Advanced Research Projects Agency (DARPA) MTO seedling project.

REFERENCES

- [1] Pulkit Agrawal, Ross Girshick, and Jitendra Malik. 2014. Analyzing the performance of multilayer neural networks for object recognition. In *European Conference on Computer Vision*. Springer, 329–344.
- [2] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, and Gi-Joon Nam. 2015. 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* 34, 10 (2015), 1537–1557.
- [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] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. 2003. An introduction to MCMC for machine learning. *Machine learning* 50, 1-2 (2003), 5–43.
- [5] Bálint Antal and András Hajdu. 2014. An Ensemble-based System for Automatic Screening of Diabetic Retinopathy. *Know-Based Syst.* 60 (April 2014), 20–27. <https://doi.org/10.1016/j.knosys.2013.12.023>
- [6] Matias S Attene-Ramos, Nicole Miller, Ruili Huang, Sam Michael, Misha Itkin, Robert J Kavlock, Christopher P Austin, Paul Shim, Anton Simeonov, and Raymond R Tice. 2013. The Tox21 robotic platform for the assessment of environmental chemicals—from vision to reality. *Drug discovery today* 18, 15 (2013), 716–723.
- [7] JD Beasley and SG Springer. 1985. The percentage points of the normal distribution. *Algorithm AS* 111 (1985).
- [8] David M Blei, Thomas L Griffiths, and Michael I Jordan. 2010. The nested chinese restaurant process and bayesian nonparametric inference of topic hierarchies. *Journal of the ACM (JACM)* 57, 2 (2010), 7.
- [9] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. 2015. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424* (2015).
- [10] George EP Box, William Gordon Hunter, and J Stuart Hunter. 1978. *Statistics for experimenters: an introduction to design, data analysis, and model building*. Vol. 1. JSTOR.
- [11] Michael Braun and Jon McAuliffe. 2010. Variational inference for large-scale models of discrete choice. *J. Amer. Statist. Assoc.* 105, 489 (2010), 324–335.
- [12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tian-shi Chen, Zhiwei Xu, and Ninghui Sun. 2014. Dadiaobao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.
- [13] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.
- [14] Dan C Cireşan, Alessandro Giusti, Luca M Gambardella, and Jürgen Schmidhuber. 2013. Mitosis detection in breast cancer histology images with deep neural networks. In *International Conference on Medical Image Computing and Computer-assisted Intervention*. Springer, 411–418.
- [15] R. David. 1980. Testing by Feedback Shift Register. *IEEE Trans. Comput.* 29, 7 (July 1980), 668–673. <https://doi.org/10.1109/TC.1980.1675641>
- [16] Misha Denil, Babak Shakibi, Laurent Dinh, and Nando de Freitas. 2013. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*. 2148–2156.
- [17] Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*. Springer, 1–15.
- [18] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 109–116.
- [19] Meire Fortunato, Charles Blundell, and Oriol Vinyals. 2017. Bayesian Recurrent Networks. *arXiv preprint arXiv:1704.02798* (2017).
- [20] Yarin Gal and Zoubin Ghahramani. 2015. Bayesian convolutional neural networks with Bernoulli approximate variational inference. *arXiv preprint arXiv:1506.02158* (2015).
- [21] Zoubin Ghahramani and Matthew J Beal. 2001. Propagation algorithms for variational Bayesian learning. In *Advances in neural information processing systems*. 507–513.
- [22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>.
- [23] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168* (2016).
- [24] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 243–254.
- [25] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
- [26] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. 2016. VIME: Variational Information Maximizing Exploration. In *Advances In Neural Information Processing Systems*. 1109–1117.
- [27] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. 2014. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866* (2014).
- [28] Michael I Jordan, Zoubin Ghahramani, Tommi S Jaakkola, and Lawrence K Saul. 1999. An introduction to variational methods for graphical models. *Machine learning* 37, 2 (1999), 183–233.
- [29] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, and Al Borchers. 2017. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760* (2017).
- [30] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-Normalizing Neural Networks. *arXiv preprint arXiv:1706.02515* (2017).
- [31] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 2010. MNIST handwritten digit database. AT&T Labs [Online]. Available: <http://yann.lecun.com/exdb/mnist> 2 (2010).
- [32] Yong Liu and Xin Yao. 1999. Ensemble learning via negative correlation. *Neural networks* 12, 10 (1999), 1399–1404.
- [33] David JC MacKay. 1992. *Bayesian methods for adaptive models*. Ph.D. Dissertation. California Institute of Technology.
- [34] Jamshaid Sarwar Malik and Ahmed Hemani. 2016. Gaussian Random Number Generation: A Survey on Hardware Architectures. *ACM Computing Surveys (CSUR)* 49, 3 (2016), 53.
- [35] George Marsaglia and Wai Wan Tsang. 1984. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM Journal on scientific and statistical computing* 5, 2 (1984), 349–359.
- [36] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851* (2013).
- [37] Mervin E Muller. 1958. An inverse method for the generation of random normal deviates on large-scale computers. *Mathematical tables and other aids to computation* 12, 63 (1958), 167–174.
- [38] Mervin E Muller. 1959. A comparison of methods for generating normal deviates on digital computers. *Journal of the ACM (JACM)* 6, 3 (1959), 376–383.
- [39] Radford M Neal. 2012. *Bayesian learning for neural networks*. Vol. 118. Springer Science & Business Media.
- [40] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, and Suchit Subhaschandra. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *FPGA*. 5–14.
- [41] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. 2015. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper* 2, 11 (2015).
- [42] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. 2017. Sc-denn: Highly-scalable deep convolutional neural network using stochastic computing. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 405–418.
- [43] B. E. Sakar, M. E. Isenkul, C. O. Sakar, A. Sertbas, F. Gurgen, S. Delil, H. Apaydin, and O. Kursun. 2013. Collection and Analysis of a Parkinson Speech Dataset With Multiple Types of Sound Recordings. *IEEE Journal of Biomedical and Health Informatics* 17, 4 (July 2013), 828–834. <https://doi.org/10.1109/JBHI.2013.2245674>
- [44] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.
- [45] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
- [46] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 16–25.
- [47] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.
- [48] Yee Whye Teh and Michael I Jordan. 2010. Hierarchical Bayesian nonparametric models with applications. *Bayesian nonparametrics* 1 (2010).
- [49] Daniel Teichroew. 1953. *Distribution sampling with high speed computers*. Ph.D. Dissertation. North Carolina State College.
- [50] Jonathan L Ticknor. 2013. A Bayesian regularized artificial neural network for stock market forecasting. *Expert Systems with Applications* 40, 14 (2013), 5501–5506.
- [51] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. 2017. Accelerating Deep Convolutional Networks using low-precision and sparsity. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE, 2861–2865.
- [52] Martin J Wainwright and Michael I Jordan. 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*

- 1, 1-2 (2008), 1–305.
- [53] Helen M Walker. 1985. De Moivre on the law of normal probability. *Smith, David Eugene. A Source Book in Mathematics*. Dover. ISBN 0-486-64690-4 (1985).
- [54] Chris S Wallace. 1996. Fast pseudorandom generators for normal and exponential variates. *ACM Transactions on Mathematical Software (TOMS)* 22, 1 (1996), 119–127.
- [55] Roy Ward and Tim Molteno. 2007. Table of linear feedback shift registers. *Datasheet, Department of Physics, University of Otago* (2007).
- [56] Wei Wen, Chunpeng Wu, Yandan Wang, Kent Nixon, Qing Wu, Mark Barnell, Hai Li, and Yiran Chen. 2016. A new learning method for inference accuracy, core occupation, and performance co-optimization on TrueNorth chip. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.
- [57] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.
- [58] Cha Zhang and Yunqian Ma. 2012. *Ensemble machine learning: methods and applications*. Springer.
- [59] Maciej Zikeba, Jakub M Tomczak, Marek Lubicz, and Jerzy 'Swikatek. 2013. Boosted SVM for extracting rules from imbalanced data in application to prediction of the post-operative life expectancy in the lung cancer patients. *Applied Soft Computing* (2013). [https://doi.org/\[WebLink\]](https://doi.org/[WebLink])