Fast-BCNN: Massive Neuron Skipping in Bayesian Convolutional Neural Networks

1st Qiyu Wan
ECOMS Lab, ECE Department
University of Houston
Houston, USA
qwan@uh.edu

2nd Xin Fu
ECOMS Lab, ECE Department
University of Houston
Houston, USA
xfu8@central.uh.edu

Abstract*—Bayesian Convolutional Neural Networks (BCNNs) have emerged as a robust form of Convolutional Neural Networks (CNNs) with the capability of uncertainty estimation. A BCNN model is implemented by adding a dropout layer after each convolutional layer in the original CNN. By executing the stochastic inferences many times, BCNNs are able to provide an output distribution that reflects the uncertainty of the final prediction. Repeated inferences in this process lead to much longer execution time, which makes it challenging to apply Bayesian technique to CNNs in real-world applications. In this study, we propose Fast-BCNN, an FPGA-based hardware accelerator design that intelligently skips the redundant computations for two types of neurons during repeated BCNN inferences. Firstly, within a sample inference, we aim to skip the dropped neurons that predetermined by dropout masks. Secondly, by leveraging the information from the first inference and dropout masks, we predict the zero neurons and skip all their corresponding computations during the following sample inferences. Particularly, an optimization algorithm is employed to guarantee the accuracy of zero neuron prediction while achieving the maximal computation reduction. To support our neuron skipping strategy at hardware level, we explore an efficient parallelism for CNN convolution to gracefully skip the corresponding computations for both types of neurons, we then propose a novel PE architecture that accommodates the parallel operation of convolution and prediction with negligible overhead. Experimental results demonstrate that our Fast-BCNN achieves 2.1∼8.2× speedup and 44%∼84% energy reduction over the baseline CNN accelerator.

***Index Terms*—Bayesian deep learning, uncertainty, Neural networks accelerator, Neuron skipping**

I. Introduction

As one of the most popular branches of machine learning family, Convolutional Neural Networks (CNNs) have been adopted as a powerful tool for solving tricky problems in various areas, such as action recognition [1], human disease detection [2], autonomous driving [3], etc. However, benefits are often accompanied by risks. In some cases, insufficient labeled training data may cause CNN models to be overfitted, which leads to an overconfident decision. For example, when an overfitted CNN model is deployed into a new environment and receives an unfamiliar input, it may give a false judgement to the situation and incur a disaster. Recently, a fatality incident was reported in an autonomous driving system when the sensors captured a confusing image of unfamiliar road conditions [4].

To address this issue, Bayesian modeling has become an appealing solution since it provides a mathematically grounded framework to quantify uncertainties for model's final prediction. As a key example, Bayesian Convolutional Neural Networks (BCNNs) are one of the most successful Bayesian models that are employed across a wide range of domains, such as disease detection [5], self-driving [6] and scene understanding [7]. Compared with conventional CNNs, BCNNs possesses a valuable property of uncertainty estimation, which is a critical information for avoiding overconfident decisions. For instance, Kendall et al. have used BCNN models for visual scene understanding and measuring the uncertainties during the decision making [7]. In their experiments, around 80% prediction mistakes were avoided when a low tolerance of uncertainty is applied. Also, in the safety-demanded medical diagnosis system, Leibig et al. have leveraged BCNN models to assist the diagnostic procedure by successfully handling the uncertain decisions for further disease inspection [5]. Now, BCNN has become the mainstream approach to provide uncertainty information to assist reliable decision making in various tasks.

Generally, a BCNN model is implemented by adding a dropout layer [8] after each convolutional layer in original CNNs [9]. Inside the dropout layer of BCNNs, neurons are randomly dropped before being taken as inputs of the next layer. Different from conventional CNNs, during the inference phase of BCNNs, for the same input, every single inference execution would generate a unique output due to the random dropout pattern. By repeatedly performing these stochastic inferences that share the same input (we define this as sample inference), BCNNs are capable of providing a probability distribution of the final output. It is well known that a massive amount of computations and memory accesses are conducted during the inference process of modern CNNs. For a BCNN model, tens to hundreds of the CNN-like sample inferences are required to obtain the final distribution of output and uncertainty information. In consequence, the total computation load for a single BCNN inference task is significantly large, which hinders the deployment of BCNNs for real-world applications that generally have emergent requirements on fast and power-efficient executions. Although the existing CNN accelerators and high-end platforms (e.g. GPUs) can still be utilized to accelerate the BCNN inference, the distinct features in BCNNs inference, such as Monte-Carlo dropout [9], induce a large amount of redundant computations on top of CNN inference. Moreover, a complete BCNN inference involves repeated stochastic forward passes for a single input, while the normal CNNs conduct inference only once for one input. Unfortunately, CNN accelerators and GPUs are completely oblivious of the opportunities buried in stochastic processes. Therefore, the unique nature of BCNNs calls for a more customized hardware design to amortize the huge computational cost.

In this study, we first unveil two types of unique neurons, both of which can be dynamically predicted to be zero during the BCNN inferences. Firstly, inside a single BCNN inference, we observe that *dropped neurons* (i.e., neurons dropped during the dropout layer) can be predetermined as long as its corresponding dropout bit (i.e., 0 or 1) is generated in advance. Secondly, we explore the relation between multiple BCNN sample inferences and make a key observation of *unaffected neurons*: most (above 90% on average) zero neurons will remain zero across sample inferences without being affected by the dropout technique.

Since both types of neurons are pre-known as zero at runtime, the corresponding convolution operations for them can be entirely skipped. Based on this insight, we then propose an efficient FPGA-based hardware design to support our neuron skipping strategy. Note that different from the existing sparse CNN accelerators, e.g. *SCNN* [10], *Cnvlutin* [11], our method eliminates all the multiplications contributing to zero-valued output neurons while *Cnvlutin/SCNN* is unaware of the value of output neurons and only removes multiplications where one of the input operands is zero. Thus our design provides significant extra computation savings. The main contributions of this study are summarized as follows:

- We characterize two types of unique neurons existing in BCNN inference, i.e. *dropped neurons and unaffected neurons*. The corresponding computation for both of them can be potentially skipped at runtime without any accuracy loss.
- We propose a prediction method to predetermine the locations of both types of neurons by simply analyzing the dropout mask, which only contains 0 or 1. Moreover, an optimization algorithm is developed to ensure the maximized computation savings while maintaining the robustness of the BCNN model.
- We propose Fast-BCNN, an FPGA-based accelerator equipped with a flexible parallelism scheme that maximizes the benefits of dynamic neuron skipping strategy during BCNN inference.
- In Fast-BCNN, we design a group of unique Processing Elements (PEs) to accommodate the parallel operation of convolution and prediction in a well-organized way. Moreover, hardware modules for prediction operation is re-designed to restrict the overhead within an extremely low range.
- Experimental results show that our Fast-BCNN achieves 2.1$\sim$8.2$\times$ speedup and 44%$\sim$84% energy reduction over the baseline CNN accelerator without skipping strategy. Our design also achieves 1.9$\times$ speedup and 34% energy reduction compared with the *Cnvlutin*-style accelerator.

II. Background on BCNN

A. BCNNs with Bernoulli Approximation

In contrast to deterministic models in conventional CNNs, BCNNs present a probabilistic interpretation of NN models by placing probability distributions over models' weights, i.e. *each weight in the model is treated as a distribution instead of a fixed value*. As a result, BCNNs can provide a more robust prediction than CNNs. To train a BCNN, we are interested in finding the posterior distribution of model's weights $\mathbf{w}$ given observed data $\mathscr{D}$. By using the Bayes formulation, we have:

$$P(\mathbf{w}|\mathscr{D}) = \frac{P(\mathscr{D})|\mathbf{w})P(\mathbf{w})}{P(\mathscr{D})} \tag{1}$$

where the denominator $P(\mathscr{D})$ is called the *evidence*, which can be calculated as $P(\mathscr{D}) = \int P(\mathscr{D}|\mathbf{w})P(\mathbf{w})d\mathbf{w}$. However, the integral process is not tractable. This appeals to a more efficient method to obtain the posterior distribution. *variational inference* [12] [13] was proposed to approximate the posterior distribution by using the common distributions. To be specific, they targeted on finding the optimal approximating distribution $q(\mathbf{w}|\theta)$ by minimizing the Kullback-Leibler(KL) divergence with true posterior [14].

Since CNNs usually accommodate a large amount of model parameters, choosing a reasonable approximating distribution $q(\mathbf{w}|\theta)$ is important. For example, the use of Gaussian approximating distribution $q(\mathbf{w}|\mu,\sigma)$ will double the model's parameters, leading to expensive computational cost. To address this issue, [9] developed a new form of variational inference using Bernoulli approximating distributions, which acquires no additional parameters. The approximating distribution $q(\mathbf{w}_i)$ for every $K \times K$ dimension layer i is defined as:

$$q(\mathbf{w}_i) = \mathbf{M}_i \odot \mathrm{diag}([z_i^j]_{j=1}^{K_i}) \tag{2}$$

$$z_i^j \sim \mathrm{Bernoulli}(p_i) \text{ for } i = 1,...,L, j = 1,...,K_{i-1} \tag{3}$$

where z_i^j denotes Bernoulli distributed random variable with probability p_i, and $\mathbf{M}_i$ are model parameters to be trained. [15] proved that minimising KL-divergence term has the effect of minimising the cross entropy loss objective function. Based on this, it was further demonstrated that Bernoulli variational inference and dropout training result in the same model parameters that best explain the observed data. This is because sampling from $q(\mathbf{w}_i)$ can be interpreted as performing dropout training, e.g. the random variable $z_i^j = 0$ corresponds to the jth neuron in layer $i-1$ being dropped out. As a result of this dropout interpretation, *BCNN can be implemented by adding a dropout layer after every convolutional layer of a normal CNN.*

B. Monte-Carlo Dropout

Previously, dropout was applied only after fully connected (FC) layers in normal CNNs [8]. By reformulating convolution operations into inner-product operations, applying dropout after convolution layers results in random neuron dropping. As shown in Fig. 1, a BCNN convolutional layer comprises of two

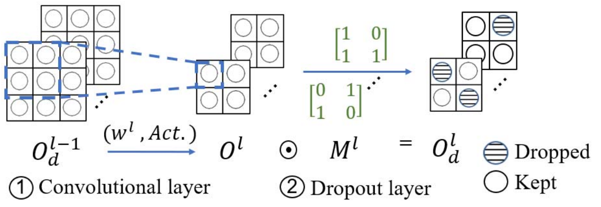

Fig. 1. Computation flow of BCNN convolutional layer.

layers, where the outputs generated by the normal convolutional layer are masked by 0/1 matrice in the dropout layer. From the computational perspective, the output of a BCNN layer l can be expressed as: $O_d^l = Act(w^l \cdot O_d^{l-1}) \odot M^l$, where O_d^{l-1} is the output of last layer, M^l is dropout masks filled with Bernoulli random variables 0 and 1, and $\odot$ represents the element-wise multiplication.

In this study, we mainly focus on the inference process of BCNN. The inference process of a trained BCNN model is referred as Monte-Carlo (MC) dropout, which is executed as follows: Given an input $\mathbf{x}$ and the network function $\mathbf{f}^{q(w)}(\mathbf{x})$, we carry out the inference process for T times with the stochastic dropout. Therefore, the hidden neurons are randomly dropped during the sample inferences. We obtain a set of output predictions $\{\mathbf{y_1}, \mathbf{y_2}, ...\mathbf{y}_T\}$. The final prediction of the network, $\bar{\mathbf{y}}$, is calculated by averaging all the outputs:

$$\bar{\mathbf{y}} = \frac{1}{T}\sum_{i=1}^{T}\mathbf{y_t} \tag{4}$$

III. Challenges and Opportunities in BCNN Inference

Although the computation pattern in BCNNs is still convolutions, the current CNN accelerators fail to fully utilize the distinct features in BCNNs. Firstly, dropout layers have not been well studied in current CNN accelerators because they are only applied after fully connected layers in CNN models, which is trivial during a whole CNN inference. In contrast, dropout layers are concatenated to every convolutional layer in BCNN models, which results in a large amount of neurons being randomly dropped. However, prior works fail to consider such invalid dropped neurons, instead, these neurons are set to zero *only after being calculated*, incurring massive redundant computations. Secondly, since most of the current designs focus on one-time inference, the problem of overwhelmed computation loads brought by MC-dropout is not well addressed. Similarly, this unique feature of BCNN makes the high-end platforms (e.g. GPUs) inefficient as well. During a complete BCNN inference with 50 samples, we observe an 51.0(50.6)× slowdown and 59(55.4)× energy consumption compared with a CNN inference, when tested on NVIDIA Tesla P100 GPU (CNN accelerators).

In this section, we investigate the MC-dropout to explore the unique opportunities in BCNNs. Specifically, we characterize two types of invalid neurons whose corresponding computations can be dynamically skipped during the inference.

Dropped neuron For a certain BCNN convolutional layer, output neurons are randomly dropped due to dropout masks.

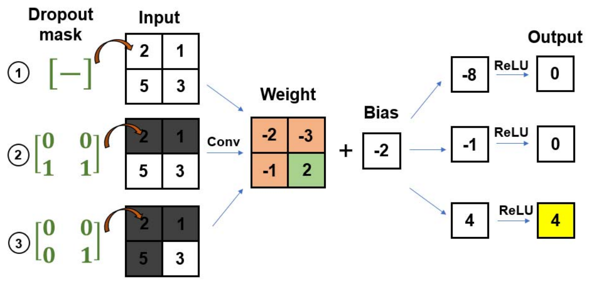

Fig. 2. The impact of dropout masks on the value of output neurons.

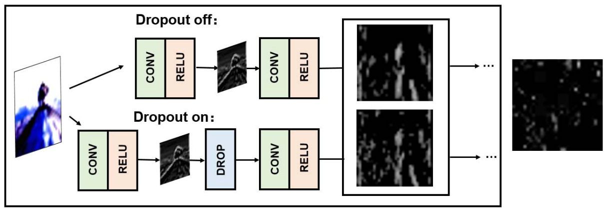

Fig. 3. Comparison of intermediate feature maps in the 2nd layer of Bayesian-VGG16 between non-dropout & dropout BCNN inference. The rightmost black-white image shows the affected neurons in white color.

As the corresponding dropout bit can be generated before the computation for a certain neuron starts, it takes no effort to predetermine the dropped neurons and directly set them to 0 instead of spending extra time to calculate them.

Unaffected neuron During the BCNN inference, output neurons from the last layer are multiplied by the dropout mask before they are taken as input for the current layer. As can be seen, dropout masks play an indispensable role to diversify the outputs in different sample inference. Interestingly, we observe that dropout masks may have a weak influence on the zero-valued output neurons, as illustrated in Fig. 2. Example ① ∼ ③ present three scenarios where the only differences are in dropout masks and output values. In example ①, none of the inputs are dropped and a negative output is observed. In example ②, there are two input activations whose corresponding weights are negative (we define such input as *nw-input*) being dropped in the same layer. Since input activations are always non-negative, dropping these two inputs means losing two negative products during the summation for the output, which makes the value of output neuron ”less negative”, i.e. *from* -8 *to* -1. However, since any negative-valued output will be restricted to zero by function of rectified linear unit (ReLU), in case ② the dropout mask takes no effect on changing the output. A turn around is eventually achieved in an extreme example ③ where three nw-inputs are dropped at the same time.

Fig. 3 magnifies two intermediate feature maps during the inference of Bayesian-VGG16 networks with and without dropout turning on. It can be observed that a majority of zero-valued neurons (in this paper, zero neuron and zero-valued neuron are used alternatively) in the non-dropout inference are still zeroes in the dropout inference. These unchanged zero neurons are defined as *unaffected neurons*, which correspond to the shared black area in Fig. 3. Meanwhile, a tiny portion

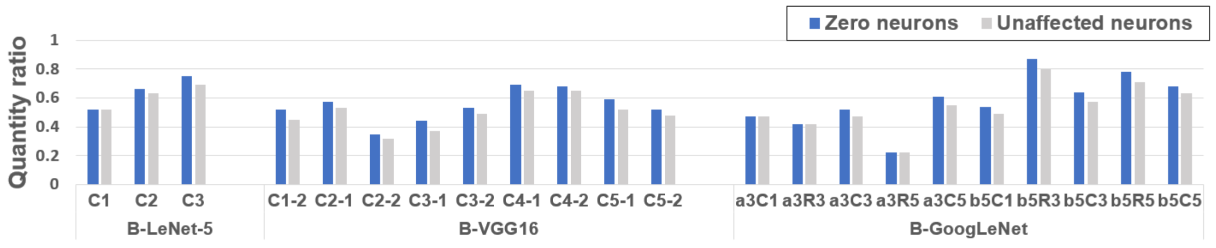

Fig. 4. Characterization of neurons of typical layers in three BCNN models. For GoogLeNet, a3C1 represents the 1×1 convolutional layer in Inception_3a while b5R3 represents the 3×3 reduce layer in inception_5b.

of zero neurons in the non-dropout inference become non-zero neurons in the dropout inference, which are defined as *affected neurons*. By further profiling the two intermediate feature maps, we present the ratio of affected neurons in the rightmost picture of Fig. 3, in which the affected neurons are represented by white color. As can be seen, the affected neurons account for a very small percentage.

To further demonstrate the general existence of unaffected and affected neurons, we transform three CNN models into BCNN models, which include Lenet-5 [16], VGG16 [17], and GoogLeNet [18]. To reduce the impact of randomness, the ratio of unaffected neurons is averaged by running the inferences 50 times with dropout turning on. The ratio of zero neurons is obtained by non-dropout inference.

Fig. 4 shows that on average, the unaffected neurons occupy 61.3%, 49.5% area of intermediate feature maps in B-LeNet-5, B-VGG16, respectively. Also, an averaging of 64% unaffected neurons are observed in Inception_5b for B-GoogLeNet. Furthermore, across different layers, over 90% of zero neurons belong to unaffected neurons. Even with such high ratio of unaffected neurons, MC-dropout is still capable of providing adequate variance for outputs since the value of non-zero neurons are changed by dropout masks.

To exploit the numerous unaffected neurons repeatedly appeared in each sample inference, we consider a complete MC-dropout process comprised of T samples. During the non-dropout inference, the positions of zero neurons can be recorded. In the following sample inferences, if we manage to find the unaffected neurons beforehand, the corresponding computations for these neurons can be entirely skipped. Thus, our skipping strategy differs from existing sparse CNN accelerators (e.g.*Cnvlutin*, *SCNN*) and brings extra computation savings. This is because *Cnvlutin/SCNN* cannot predetermine whether the output neuron is zero or not, they only remove the ineffectual multiplications where the *input neuron* or weight is zero. Our method manages to predict the invalid *output neurons* and eliminates all the multiplications, including non-zero ones, that contribute to these neurons.

Considering the skipping strategy for both dropped neurons and unaffected neurons, since an unaffected neuron may also be dropped during inference, the overall percentage of skipped neurons can not be directly calculated by adding the ratio of dropped and unaffected neurons without concerning their overlapped ratio. Commonly, dropped neurons account for 20% ∼ 50% of all neurons according to the drop rate while unaffected neurons account for above 50% in average. Accordingly, around 50% neurons can be skipped if they are determined as unaffected neurons, while 20% to 50% of the rest neurons are skipped as dropped neurons. As a result, the overall skip rate can be estimated at a range from 60% to 75%.

IV. Fast-BCNN

In this section, we propose Fast-BCNN which skips the computations for both dropped and unaffected neurons. We first propose an efficient method to predict the unaffected neurons early before its convolution operation starts, we then explore an efficient parallelism scheme to well support our skipping mechanism.

A. Unaffected Neuron Prediction

1) Prediction principle: The zero output is converted into positive output by dropout masks only in a small number of extreme cases, where a dominant amount of nw-inputs are dropped (See Fig. 2 example ③). As we know, the output value is summed up by a set of products of inputs and weights. The value of the output tends to become positive when a dominant number of negative products are dropped, e.g. in example ③, three greyed inputs indicate three negative products are lost, resulting in a positive output. This nature inspires us to leverage the number of dropped nw-inputs as a criterion to identify unaffected neurons. To be specific, if the number of dropped nw-inputs for a certain output neuron is small enough (i.e., smaller than a threshold), this neuron is identified as unaffected with high confidence and its corresponding computation can be skipped. Otherwise, this neuron will be calculated normally. Given a zero neuron *ZN* in the non-dropout inference, the number of dropped nw-inputs N_d and threshold α, we have:

$$\mathrm{ZN} \in \begin{cases} \text{Unaffected neurons}, & \text{if } N_d < \alpha \\ \text{Affected neurons}, & \text{Otherwise} \end{cases} \tag{5}$$

To conduct the prediction task, the number of dropped nw-inputs for each output neuron should be counted before the computation of current layer starts. This is feasible since the dropout masks for inputs of current layer are already generated when processing the previous layer. Hence, the prediction for the current layer can be performed in parallel with convolution operations of the previous layer. To assure the prediction results are ready for the current layer, a simple strategy is allocating more computing resources for counting process. Fortunately, the counting procedure can be implemented via low-overhead logics in hardware (details can be found in Section V-B2).

2) *Optimization of thresholds:* According to our prediction criterion, a higher threshold α will predict more ZNs as unaffected, which may incorrectly identify affected neurons as unaffected, resulting in the accuracy loss of BCNN model. In contrary, a smaller threshold α may incorrectly predict the unaffected neurons as affected, leading to unnecessary computations. Therefore, the threshold α should be picked appropriately. We use confidence level p_{cf}, which is the percentage of correctly predicted neurons over all neurons in the feature map, to guide the selection of the threshold. The confidence level can be flexibly chosen according to the user's demand on the model accuracy and computation reduction, which is discussed in Section VI-B2. At first, the threshold α is initialized with a proper value (usually large), which may induce many incorrect predictions. Then α is decreased step by step until the ratio of correctly predicted neurons meet the required confidence level. By doing this, it allows the number of unaffected neurons in a maximized fashion while the confidence level is guaranteed.[1] Moreover, we notice that the distributed pattern of negative weights and positive weights varies across different kernels in different layers, thus it is necessary to acquire the threshold α for all *kernels* in a BCNN model. Note that the thresholds are model(kernel)-dependent rather than input-dependent, thus the optimized thresholds are robust to uncertainty environments where the distribution of inputs may have high variance. Moreover, the optimization process only needs to be performed once during the offline stage when the model has to be retrained.

In Algorithm 1, we assume there are L convolutional layers in the network and M_l kernels in a certain layer. We denote $O_{(t,l,m)}$ as the **m**-th output feature maps in convolutional layer **l** from inference **t**.

Preparation In line 1-3, the intermediate activations are registered into the Activation[L] and locations of the zero neurons are recorded by **GetIndex**. In line 4-5, each kernel is profiled and the indice of negative weights and positive weights are stored.

Optimization In the optimization stage, the threshold values are optimized iteratively across all kernels. In line 9-10, threshold α is initialized with proper value Th and added to the threshold set at the beginning. In line 12-15, for each α of kernel $K_{(l,m)}$, T sample inferences are conducted normally by function **Inference**. In each inference we only aim to obtain the output $O_{(t,l,m)}$, Mask$_{(t,l,m)}$ in the current layer l. Then, the function **PrecitionInference** executes the forward pass in the prediction mode. It skips the computation of neurons that predicted as unaffected, based on dropout masks from the previous layer, threshold α, and the static indices generated in preparation stage. The prediction mode will be employed from the first layer to the current layer. Similarly to function **Inference**, the output feature map $O^*_{(t,l,m)}$ of the current layer l is extracted. In line 16, the function **EvaluatePredict** aims to measure the ratio of correctly predicted neurons by comparing between the true output feature map $O_{(t,l,m)}$ and predictive feature map $O^*_{(t,l,m)}$. Threshold α keeps decreasing one step (i.e. Δs) at each iteration, until it finally satisfies the required confidence level p_{cf}.

[1]The same optimization goal can be achieved if we initialize α with small value and update it increasingly. Here we simply choose one strategy.

Algorithm 1 Finding the optimal threshold values for each kernel.

```
Input:
    A BCNN model f. An optimization dataset $\mathscr{D}$.
    Initialization value of threshold $Th$.
    Adjusting step $\Delta$s
    Required confidence level $p_{cf}$.
Output:
    Threshold values $\alpha$
    % Obtain the location of zero neurons for L convolutional
    layers from the first sample inference.
 1: Initial Activation[L] $\leftarrow$ 0
 2: $y_0$, Activation[L] = Inference(f, $\mathscr{D}$, dropout = off)
 3: location[L] $\leftarrow$ GetIndex($a_i$ == 0), for $a_i$ in Activation[L]
    % Obtain the location of postive and negative weights in
    a kernel $K_{(l,m)}$.
 4: Idx_n[L][M$_l$] $\leftarrow$ GetIndex($w \leq 0$), for $w$ in $K_{(l,m)}$
 5: Idx_p[L][M$_l$] $\leftarrow$ GetIndex($w > 0$), for $w$ in $K_{(l,m)}$
 6: Threshold[L][M] $\leftarrow$ 0
    % Fine-tune the thresholds iteratively for each kernel.
 7: for l in range(1, L) do
 8:   for m in range(0, M$_l$) do
 9:     Initial $\alpha_{(l,m)}$ = $Th$
10:     Threshold.append ($\alpha_{(l,m)}$)
11:     while 1 do
12:       for t in range(1,T) do
13:         $O_{(t,l,m)}$, Mask$_{(t,l,m)}$ = Inference(f, $\mathscr{D}$, dropout =
            on)
14:         $O^*_{(t,l,m)}$ = PredictInference(f, $\mathscr{D}$, dropout = on,
            Mask$_{(t,l-1,m)}$), location[L], Idx_n$_{(l,m)}$,
            Idx_p$_{(l,m)}$, Threshold)
15:       end for
16:       if (EvaluatePredict($O_{(l,m)}$, $O^*_{(l,m)}$) $\geq p_{\mathrm{cf}}$) then
17:         Break;
18:       else
19:         $\alpha_{(l,m)} \leftarrow \alpha_{(l,m)}$ - $\Delta$s
20:       end if
21:     end while
22:   end for
23: end for
```

B. Skip-Friendly Parallelism

To acquire a high throughput, prior CNN accelerators adopted various parallelism scheme, which is actually the most important factor to differentiate their works [10], [19], [20]. In this section, we discuss the problems and limitations of existing parallelism schemes when applying skipping technique on them. Furthermore, we propose an adapted parallelism scheme to maximize the computation reductions by fully leveraging the benefits of neuron skipping strategy.

Modern CNN accelerators adopt parallel computing by unrolling convolution loops and map them to the duplicated hardware modules. Basically, there are six unrolling factors $< T_m, T_n, T_r, T_c, T_i, T_j >$ that match up with six convolutional dimensions: M (number of output channels), N (number of input channels), R (number of rows in output feature maps), C (number of columns in output feature maps), K (or I/J, number of rows/columns in kernels). Current parallelism strategies can be categorized into three types [20]: Synapse Parallelism [21], Neuron Parallelism [22] and Feature map Parallism [23].

Synapse Parallelism Synapse Parallelism adopts $< T_i, T_j >$ parallelism strategy. $T_i \times T_j$ synapses of one kernel are multiplied with their corresponding input activations at the same time. This parallelism is usually operating in a systolic array, where $T_i \times T_j$ synapses stay in the processing elements (PEs) and one input activation is fed into the array at each cycle. After the pipeline of the systolic array is filled, one output neuron is generated at each clock cycle. Our neuron skipping technique cannot employ this architecture due to the fixed dataflow in Synapse Parallelism. To be specific, it cannot abandon all the corresponding computations associated with a invalid neuron as the input activations involved in the computations could affect other neurons.

Neuron Parallelism Neuron Parallelism adopts $< T_r, T_c >$ parallelism strategy. $T_r \times T_c$ neurons are mapped to $T_r \times T_c$ PEs and stay stationary until the accumulating is finished, while input activations are flowing through continuously. In each cycle, one weight is broadcasting to all PEs, and $T_r \times T_c$ neurons are completed in every $K \times K \times N$ cycles. As shown in Fig. 5, since $T_r \times T_c$ neurons are computed in parallel in this architecture, to employ the skipping method, PEs who meet the invalid neurons need to calculate for the neuron in the next channel and fetch the weights in the next kernel. Thus, every PE is working for different output channels irregularly and fetching different weights from the on-chip buffer, which results in a read-conflict to the global weight buffer.

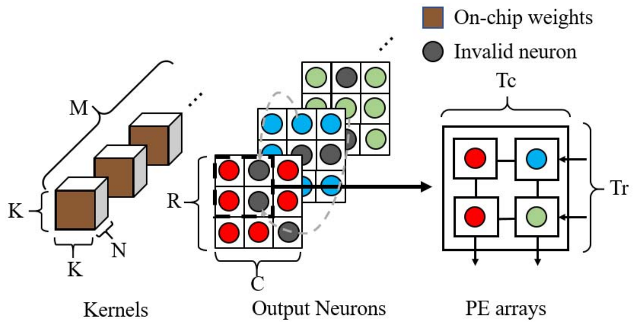

Fig. 5. Skipping in Neuron Parallelism.

To address this issue, each PE should be allocated an independent weight buffer that can hold at least a group of kernels (brown-colored in Fig. 5) from different channels. In consequence, the on-chip storage for weights is forced to become large. Assume there is a single global weight buffer containing $K \times K \times M$ weights in the original architecture, each PE needs to buffer these weights as well. Thus, the percentage of increased on-chip storage would be:

$$\text{Buffer}_{\text{np\_overhead}} = \frac{K^2 M T_r T_c - K^2 M}{K^2 M} = T_r T_c - 1. \quad (6)$$

Obviously, the on-chip weight buffer is duplicated $T_r \times T_c$ times.

Feature map Parallelism Feature map Parallelism adopts $< T_m, T_n >$ parallelism strategy. In this architecture, there is an array of T_m PEs operating in parallel. In each cycle, T_n input activations are broadcasting to all PEs. In each PE, T_n weights are multiplied with T_n input activations at a time, and the results are summed up to generate one neuron. Thus, it takes $K \times K \times \frac{N}{T_n}$ cycles to complete T_m neurons. As shown in Fig. 6, since T_m neurons are computed in parallel, if any neuron is dropped, the corresponding PE should skip to the neuron in the next position in the same channel, otherwise it will remain idle until other PEs finish. To avoid the idleness, the corresponding input activations for the next kept neuron has to be loaded into the PEs. In this case, every PE is working ahead or behind each other and fetching different input activations according to the dropout pattern in their responsible channels, which leads to a read-conflict to the global input buffer.

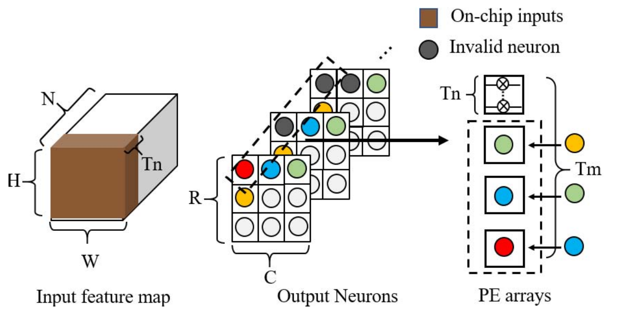

Fig. 6. Skipping in Feature map Parallelism.

To meet the read request from all PEs and keep them busy, a naive way is to duplicate the global input buffer and allocate them to each PE. Compared with the strategy applied in Neuron Parallelism, a significant observation is that the input buffer is only duplicated T_m times instead of $T_m \times T_n$ times. Assuming the size of input feature map is (H, W), the size of global input buffer in previous architecture is usually designed to be $T_n \times W \times H \times Data\_Width$ considering the requirement of on-chip area. As shown in Fig. 6, the brown-colored cube represents the input data stored on chip in the original global buffer. When skipping mechanism is applied, this buffer should be duplicated for each PE. The percentage of increased on-chip storage can be derived as:

$$\text{Buffer}_{\text{fp\_overhead}} = \frac{WHT_nT_m - WHT_n}{WHT_n} = T_m - 1. \quad (7)$$

Comparing Eq.(6) and Eq.(7), the extra overhead of Feature map Parallelism brought by neuron skipping strategy is much less than Neuron Parallelism, given the same amount of computing resources, i.e. $T_r \times T_c$ and $T_m \times T_n$. Thus, we adapt feature map parallelism and allocate an independent input buffer for each PE to avoid the read-conflict.

V. Implementation of Fast-BCNN

A. Overview

We propose a highly customized hardware implementation of Fast-BCNN, which accommodates dynamic neuron prediction and skip-friendly parallelism in a coordinated fashion. As shown in Fig. 7 (a), Fast-BCNN accelerator consists of a group of PEs, a central predictor and a controller. The detailed design are presented in the following subsection.

B. PE Architecture

Fig. 7 (b) depicts three hardware modules (i.e., convolution unit, Bernoulli random number generator (BRNG), and prediction unit) that cooperate with each other to function as a single PE. The convolution unit is the main computing part that executes basic convolution operations in the skip-friendly parallelism manner. There are two types of skipping signals—dropout bit and prediction bit—coming from BRNG array and central predictor, respectively. The prediction unit, which works in parallel with convolution unit, counts the number of dropped nw-inputs for each neuron in the next layer based on the dropout bits received from BRNGs. The results are streamed into central predictor to generate the final prediction bits. To better illustrate the dataflow in our PE, for a certain layer l, we define the input channel N_l, kernel size K_l, output channel M_l and output size (R_l, C_l).

1) ***Convolution unit:*** The main components for basic convolution operation are an array of T_n multipliers, an adder tree, an accumulator and an input/output/weight buffer. To support the neuron skipping strategy, it is also equipped with a skip engine affiliated with two FIFOs and a multiplexer.

Datapath During the first inference without dropout applied, i.e. *pre-inference*, the skip engine generates a coordinate (r, c) = (0, 0) of the output neuron which increments by 1 periodically. The controller fetches the corresponding inputs and weights according to the current coordinate. At each cycle, T_n inputs and T_n weights are fetched and sent into the multiplier array. One input is multiplied with its corresponding weight and generating T_n products. These products are summed up together by the adder tree and the partial sum gets accumulated in the accelerator. It takes $K \times K \times \lceil N_l/T_n \rceil$ cycles to produce one neuron. The outputs go through the pooling unit if necessary and reside in the output buffer. When the values in the output buffer need to be written to DRAM, it will be judged as being 0 or not, if yes, a one-bit representative (0/1), which indicates whether the output is zero, will be generated and also sent to off-chip. During the following inferences, an exclusive shortcut (dotted line in Fig. 7) of computation is built for the execution of the first layer. The results of the first layer in the pre-inference is loaded to the input buffer and multiplied with dropout bits directly. Compared with re-calculating all the output neurons, this significantly saves the execution cycles for the first layer. From the second layer, the skip engine will receive both dropout bit and prediction bit, based on which it determines the new coordinate of neuron that needs to be computed.

I/O buffer As the input buffer is duplicated T_m times across the T_m PEs, the size of the buffer should be carefully considered. As discussed in Section IV-B, the input buffer only holds part of the input feature maps. To provide sufficient bandwidth, the data block at each entry is configured to be $(T_n \times Data\_Width)$ bits, which is mapped to the input data in T_n channels. For the output buffer, it stores the outputs for only one sample. The output data are transferred to DRAM once the buffer is full. It is employed with the same memory mapping strategy as input buffer for better data alignment.

Skip engine Fig. 8 (a) shows the microarchitecture of skip engine. An OR gate receives dropout bit and prediction bit once the controller sends a ready signal, and generates a "1" signal when a neuron is dropped (dropout bit = 1) or being predicted as unaffected (prediction bit = 1). The output will trigger the current neuron being dropped. Specifically, it goes to MUX to select the zero value as output, disables the ALUs and enables the write operation of output buffer. Moreover, it increases the index in the counter by 1, which points to the neuron in the next position. For the counter, it contains two cascaded sub-counters that generate the coordinates r and c, which is controlled by the parameters of the layer configuration.

2) ***Prediction unit:*** The essential function of prediction unit is to calculate the number of dropped negative products for each output neuron in the *next* layer. since the cost of storing indice of negative weights in every kernel is not trivial, the information of kernels is compressed as indicator bits that refers to the polarity of weight in the corresponding location. In other words, all negative weights are represented as "1" while the positive ones are represented as "0". For the dropout mask, dropout bit "1" means the neuron shall be dropped. As shown in Fig. 9 (a), the dropout bits in one convolution window are convolved with indicator bits to generate the total counts of dropped negative products for one neuron. To minimize the hardware utilization, the binary convolution is implemented by just adopting the simplest computation units, i.e. AND gates and counters, and mini-buffers that store only binary values.

Dataflow The main function of prediction unit is achieved by T_m' counting lanes and each of them is a concatenation of an AND gate and a counter. Before the counting process starts, the dropout bits pass through mask pooling unit if pooling is required for their corresponding neurons. During mask pooling, a "1" dropout bit is generated only when all dropout bits are "1", since the zero neurons are all filtered out if there is a non-zero value during max pooling.

The entire computation flow is similar to the $< T_m, T_n >$ parallelism adopted in the convolution unit. Fig. 9 (b) illustrates an example with 4 counting lanes. One dropout bit $D_{(r+i,c+j)}$ is broadcasting to all counting lanes at each cycle when the corresponding indicator bits from four kernels $I_{(i,j)}^{(1\sim4)}$ are requested by four counting lanes. Each pair in the counting lane is responsible to count for one neuron in a certain channel. For each counting lane, when both dropout bit and indicator bit are "1", the counter will be triggered to increment by 1. The output from all counting lanes are collected by data packaging center, where all the data are organized into one data block

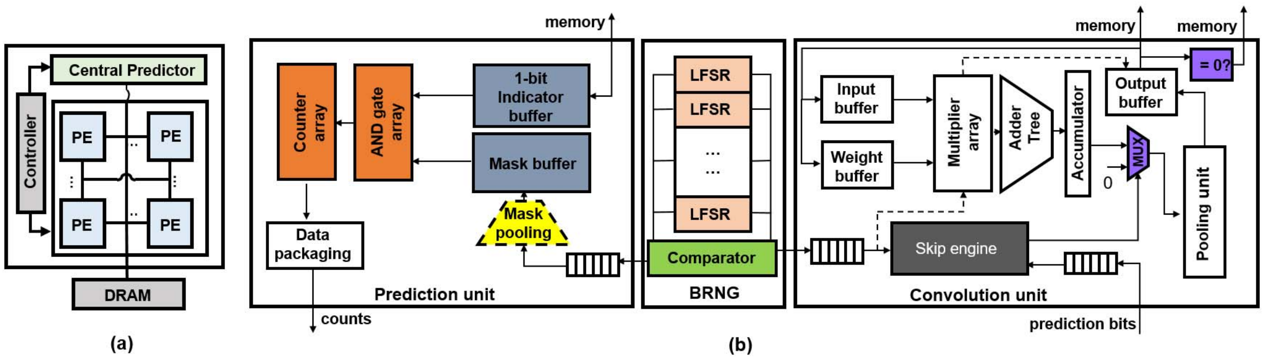

Fig. 7. (a) An overview of Fast-BCNN architecture. (b) Hardware block diagram of PE microarchitecture.

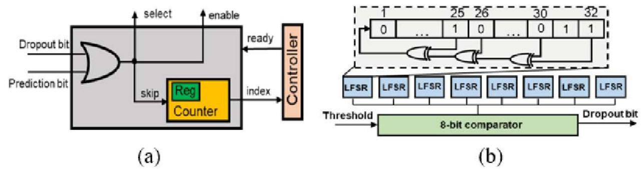

Fig. 8. (a) A low-level description of skip engine. (b) LFSR-based Bernoulli random generator.

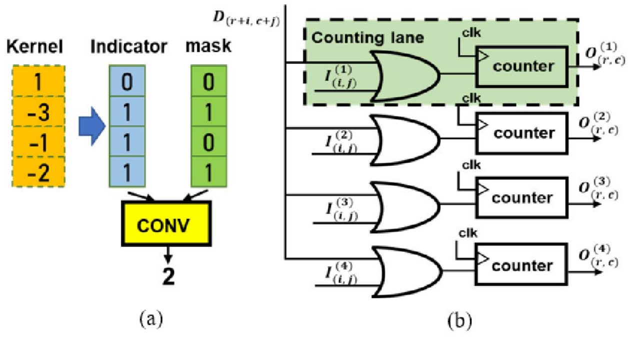

Fig. 9. (a) Illustration of counting the number of dropped nw-inputs in a convolution window. (b) An example of dataflow in AND gate array and counter array.

for later summation in the central predictor.

Synchronization with convolution unit The computation in prediction unit starts when dropout bits for the current channel are all streamed into mask buffer. To prepare the prediction bits for convolution unit timely, prediction procedure shall be finished before the next layer starts. With such requirement we rationally analyze the number of counting lanes to ensure a decent synchronization. In a single PE, assume there are T_m' counting lanes in the prediction unit, to complete the counting process for R×C dropout bits, the total clock cycles are $K'K'[M'/T_m']R'C'$. It shall be lower than the total execution time for neurons in the current channel to avoid the idleness of convolution unit. Hence we have:

$$(K')^2[M'/T_m']R'C' \le K^2[N/T_n]RC(1-skip\_rate) \qquad (8)$$

Among the above parameters, the kernel size K varies little between consecutive layers for most networks while the number of channels M sometimes increases in the deeper layers [17]. In contrary, the size of feature map R×C may shrink in deeper layers. For the skip rate, it can be estimated by profiling the number of dropped and unaffected neurons beforehand. With these analyses, the number of counting lanes can be simplified as:

$$T_m' \ge (M'R'C'/NRC(1-skip\_rate))T_n = \delta T_n \qquad (9)$$

In most cases, the range of factor δ is 4∼8. As the resource consumption of an AND gate and a counter is register level, compared with the array of T_n 32-bit multipliers and adders, the overhead of δT_n counting lanes is negligible.

Mini-buffers The indicator buffer receives a portion of 1-bit indicators from the external memory at a time. During the counting process, T_m' indicator bits are requested at each cycle. To fully utilize the memory bandwidth, the indicator buffer is configured to hold T_m' indicator bits per entry. Considering the buffer size, the overhead of indicator buffer is about $1/bit\_width$ of weight buffer in the convolution unit. For mask buffer, it receives one dropout bit from BRNG at each cyle and stores at most $R_l \times C_l$ bits.

*3) **Bernoulli random number generator**:* To generate the dropout bits, we implement a Bernoulli random number generator (BRNG) based on 32-bit linear feedback shift register (LFSR) [24], [25]. As shown in Fig. 8 (b), in each LFSR, a single bit in the register shifts to the right neighbour register at each cycle. For the bit at the head of the LFSR, it receives a combination results from the bits at the taps (25, 26, 30, 32). At each cycle the leftmost bit of a LFSR is read and treated as a uniformly distributed random number [26]. Bits from eight LFSRs are combined as a 8-bit uniform random number. This value is compared with a threshold $t = 2^8 \times drop\_rate$, if the value is smaller than t, the dropout bit is set to 1, otherwise set to 0. We test the functionality of the BRNG and compare it with the software solution in Section VI.

C. Central Predictor

Note that outputs from the prediction unit is a partial result since each PE only counts M/T_m input channels for neurons in the next layer. To acquire the final counting value for each neuron, an adder tree is employed in the central predictor to calculate the summation of all counting values from T_m PEs. Then the output is compared with the thresholds from Algorithm 1. According to the prediction criterion, the prediction bit is generated based on the comparison result. Since only zero neurons can be predicted as unaffected, the prediction bit further pass through AND gates along with zero indexer generated by first inference. To minimize the hardware resource usage, only 10-bit adders are used to implement the adder tree since the counting values are all integers and occupy no more than 10 bits. We discuss the extra energy overhead and resource usage of central predictor in Section VI.

VI. Evaluation

A. Methodology

BCNN models and datasets We evaluate three typical and commonly used BCNN models from the current BCNN applications, including B-LeNet-5, B-VGG16 and B-GoogLeNet. We select B-LeNet-5 trained with MNIST dataset since it is the first successfully tested BCNN model [9]. We choose B-VGG16 and B-GoogLeNet trained with CIFAR-100 as both networks have been maturely employed in many applications. Our selected networks are representative since other customized BCNN models are mostly built on top of them with minor revisions that do not affect the essential features. During both training and testing, the dropout rate is set as 0.3 for all networks. The sensitivity analysis to dropout rate is shown in Section VI-B2.

Design space Parallelism parameters $< T_m, T_n >$ determine the basic structure of our accelerator, including the number of PEs, the number of multipliers, on-chip storage, etc. Particularly, according to Eq. 9, the number of counting lanes T_m' is designed depending on the value of T_n. Table I presents four types of hardware configurations by varying T_m (i.e. *the number of PEs*), while the total number of MACs is fixed at 256. Among these candidates, we aim to choose the best hardware configurations for Fast-BCNN by comparing their improvement in terms of speed and energy. To demonstrate the effectiveness of our proposed skipping strategy, we build up a baseline accelerator without the neuron skipping strategy. It adopts the same state-of-the-art parallelism scheme and parameters (i.e. $< T_m, T_n >$) as Fast-BCNN (T_m = 64). We also compare our Fast-BCNN with *Cnvlutin* [11], a typical sparse accelerator that can skip zero-valued neurons. For fair comparison, we scale down the original design of *Cnvlutin* into 8x8 sub-units, each with 4 synapse lanes. Since pruning technique has not been applied in BCNNs, another type of sparse accelerators leveraging both weight and input sparsity (e.g., *SCNN* [10], *Cambricon-X* [27]) are not taken into our comparison.

Environment setup All hardware modules of Fast-BCNN are implemented in Verilog and synthesized on an Xilinx Virtex-7 VC709 FPGA evaluation board. All the designs are operating at frequency of 100MHz. For off-chip memory access, a Memory Interface Generator [28] is adopted to communicate with external DDR3 memory equipped on board. The performance and accuracy results are obtained from the post-synthesis design. The energy consumption is further measured with Xilinx Power Estimator (XPE) [29]. Multipliers and adders in convolution unit are operating in 32-bit floating point mode to maintain the computational accuracy. During the offline stage, the confidence level p_{cf} is set as 68% to obtain the thresholds since it is an optimal value to guarantee a considerable benefits for both speedup and accuracy (sensitivity analysis about confidence level can be found in Section VI-B2). Note that our design is also user friendly, users can set the target for accuracy to obtain the maximum performance and energy improvement. The experiment results are evaluated from a complete BCNN inference, during which the sample inference is performed 50 times. Additionally, in order to obtain the location of zero neurons, a non-dropout pre-inference is conducted in Fast-BCNN at the beginning. In fact, the total execution time for Fast-BCNN includes the execution time of 51 sample inferences. The total execution time of all the investigated designs are averaged by 50 and then normalized for analysis.

TABLE I
Hardware parameters for Fast-BCNN designs.

Type	Total MACs	T_m	T_n	T_m'
Baseline	256	64	4	-
Fast-BCNN8 ($T_m = 8$)	256	8	32	128
Fast-BCNN16 ($T_m = 16$)	256	16	16	64
Fast-BCNN32 ($T_m = 32$)	256	32	8	32
Fast-BCNN64 ($T_m = 64$)	256	64	4	16

B. Experimental Results

1) Performance and energy efficiency: Fig. 10 shows the execution cycles, energy consumption, and accuracy obtained by four types of Fast-BCNN designs compared with the baseline accelerator for three evaluated networks. All results are normalized to the baseline. For B-LeNet-5, all types of Fast-BCNN designs achieve at least 86% cycle reduction, which can be transferred to 7× speedup over baseline. This is because the first convolutional layer takes most of cycles due to its large size of feature map. The breakdown of layer execution cycles shows the execution of first layer enjoys an averaged 8.2× boost on performance since all outputs of the first layer can be reused in the sample inferences. Fast-BCNN32 and Fast-BCNN16 outperform other two accelerators by yielding nearly 90% reduction in execution cycles.

For B-LeNet-5, four types of Fast-BCNN accelerators give an averaged 84% energy reduction over the baseline due to the extremely small execution time. To analyze the energy overhead of prediction strategy, the energy consumption is decomposed into three parts: convolution unit, prediction unit and central predictor. The latter two components consume 12% and 15% overall energy averaging across four types of designs. Specifically in Fast-BCNN64, only 8% and 5% total energy are consumed in prediction unit and central predictor, respectively.

For B-VGG-16, Fast-BCNN64 exhibits a 59% cycle reduction, i.e. 2.4× speedup, which slightly outperforms other accelerators. This is due to the faster execution of the first layer in Fast-BCNN64. From the cycle breakdown for each layer, we show that this advantage diminishes slowly as deeper layers with larger computation loads are executed. Four types of Fast-BCNN give an energy reduction ranging from 41% - 50%, among which Fast-BCNN64 restricts the energy overhead of prediction strategy within the lowest level and achieves the largest energy reduction, i.e. 50% at the same time.

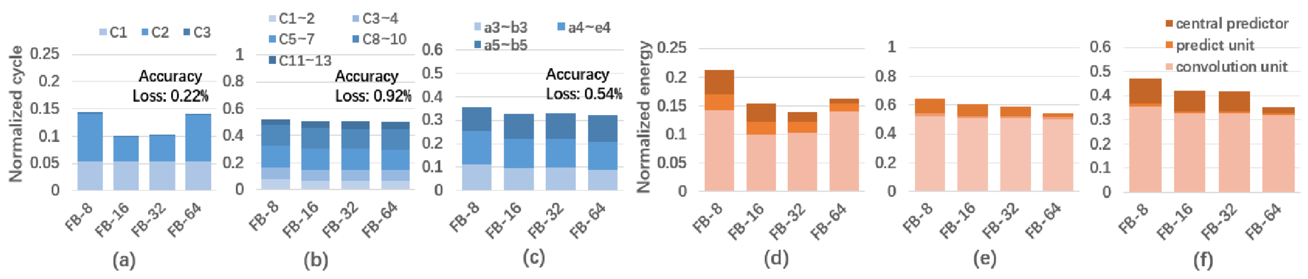

Fig. 10. The performance & energy improvement and accuracy loss over baseline accelerator within design space. (a)&(d)—B-LeNet-5, (b)&(e)— B-VGG16, (c)&(f)—B-GoogLeNet, FB-64 is abbreviate for Fast-BCNN64, etc.

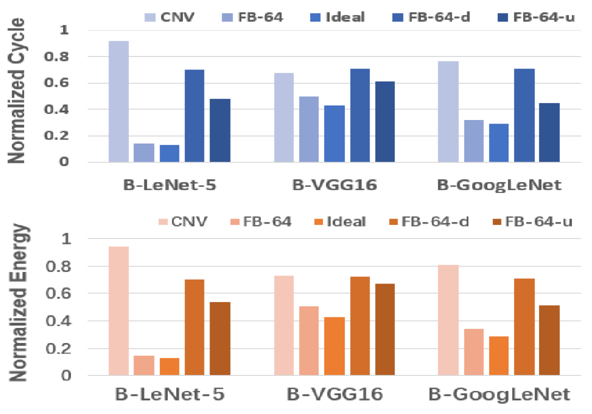

Fig. 11. Comparisons with *Cnvlutin*, ideal case and two modes of Fast-BCNN64. FB-64-d (FB-64-u) is abbreviate for Fast-BCNN64-d (Fast-BCNN64-u).

For B-GoogLeNet. the execution time is divided into three groups of consecutive layers: Inception_a3 to b3, Inception_a4 to e4 and Inception_a5 to b5, which are almost evenly accelerated. In terms of performance, Fast-BCNN64 outperforms others by achieving a 69% cycle reduction, i.e. 3.1× speedup. Additionally, compared with an averaging 59% energy reduction across all Fast-BCNN designs, Fast-BCNN64 provides the highest 65% energy reduction.

In our proposed design, we deploy an array of 64 PEs operating in parallel, within each there are 4 multipliers and 3 adders for convolution unit and 16 counting lanes for prediction unit. The overall resource utilization results are listed in Table II, where LUT and FF are computing resources while BRAM represents on-chip storage. As it shows, prediction units & central predictor only incur less than 1% overhead of LUT and FF. For prediction unit, the overhead of BRAMs is larger (i.e. 4%) than the overhead of LUT and FF. This is because implementing a 1KB mask buffer in prediction unit requires a minimum BRAM of 18KB on our board, which far exceeds what we actually need.

Based on the above discussions, Fast-BCNN64 achieves significant performance and energy improvement over baseline architecture and outperforms other types of Fast-BCNN designs. We thus compare Fast-BCNN64 with *Cnvlutin* and the ideal case where all the computation savings are transferred to the speedup and energy reduction, as shown in Fig. 11. Furthermore, to investigate the effectiveness of Fast-BCNN on skipping dropped and unaffected neurons separately, the

TABLE II
COMPARISON OF RESOURCE USAGE AMONG TOTAL CONVOLUTION UNITS, PREDICTION UNITS & CENTRAL PREDICTOR.

Resource	Convolution units	Prediction units	Central predictor
LUT	276736/433K (64%)	1024/433K (<1%)	10246/433K(1%)
FF	359360/866K (41%)	1024/866K(<1%)	10246/866K(1%)
BRAM	512/1470 (35%)	64/1470(4%)	2/1470(<1%)

operation of Fast-BCNN64 is separated into two modes: Fast-BCNN64-d that only skips dropped neurons and Fast-BCNN64-u that only skips unaffected neuron. Fig. 11 also shows the performance and energy results for these two designs. All results are normalized to the baseline case. Note that the accuracy loss brought by unaffected neuron skipping, i.e. Fast-BCNN64 and Fast-BCNN64-u, is already shown in Fig. 10 (a)∼(c), and not included in Fig. 11.

Compared with *Cnvlutin*, Fast-BCNN64 achieves additional 38% (1.9x) cycle reduction (speedup) and 34% energy reduction on average. Such improvement comes from the elimination of all multiplications (with both zero and non-zero inputs) contributing to invalid output neurons while *Cnvlutin* can only skip multiplications with zero inputs. Thus, even only skipping unaffected neurons (FB-64-u) consistently saves more execution time than *Cnvlutin*. Furthermore, *Cnvlutin* achieves little performance and energy improvement for B-LeNet-5 as it does not skip neurons in the 1st layer. *Cnvlutin* is also oblivious of dropped neurons, some of which are non-zero neurons before being dropped.

Fig. 11 further shows there is a 11.3%(15.3%) performance (energy) gap between Fast-BCNN64 and the ideal case. This is mainly caused by synchronization issues across different PEs. Since each PE is responsible for one output channel, the channel with more invalid neurons will become idle. In most cases, the idle time is small, for example, the performance gap is around 7% between Fast-BCNN64 and the ideal case for B-LeNet-5. For B-VGG16, the performance gap is relatively larger as PE idleness accounts for 15% of the overall execution time. Finally, as Fig. 11 shows, the sum of the cycle (energy) reduction of Fast-BCNN64-d and Fast-BCNN64-u is slightly larger than that of Fast-BCNN64. For example, in B-GoogLeNet, the sum of the cycle reductions of Fast-BCNN64-d (i.e. 29%) and Fast-BCNN64-u (i.e. 45%) is larger than that of Fast-BCNN64 (i.e. 68%). This is because the total number of skipped neurons is smaller than the sum of the dropped and unaffected neurons

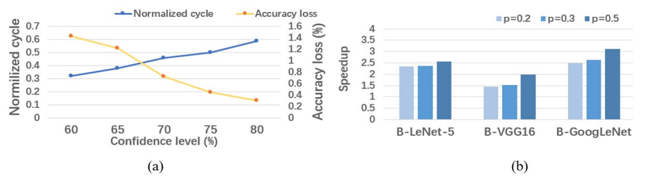

Fig. 12. (a) Accuracy loss and execution cycle vs confidence level $p_{cf}(\%)$ for predicting unaffected neurons. (b) Speedup vs drop rate.

TABLE III
REAL DROP RATE OBTAINED FROM 2000 AND 4000 DROPOUT BITS.

	LFSR-based BRNG		Software	
Drop rate	2000 cycles	4000 cycles	2000 samples	4000 samples
p = 0.5	0.5010	0.5008	0.5095	0.5038
p = 0.2	0.1997	0.1998	0.1950	0.1990
p = 0.1	0.1025	0.1010	0.0940	0.0980

due to the overlapping between these two types of neurons.

2) Sensitivity analysis: **Sensitivity to confidence level p_{cf}** The confidence level p_{cf} in Algorithm 1 directly affects the flexibility of prediction criterion for unaffected neurons, which further determines how many zero neurons can be identified as unaffected. Fig.12 (a) shows the capability of confidence level to control the accuracy loss and speedup. In our test on Fast-BCNN64 for B-VGG16, with 60% confidence level the quality loss of final output is only around 1.4% while the performance benefits from a 63% cycle reduction. As we decrease the acceleration to a 42% cycle reduction, the accuracy loss can be restricted within 0.3% by setting p_{cf} at 80%. The negligible accuracy loss is probably because each final output is obtained by averaging the outputs from a number of BCNN inferences, during which the errors are mitigated. Furthermore, an optimal balance between speed and accuracy is observed by setting p_{cf} at 68%.

Sensitivity to dropout rate To measure the impact of drop rate on overall performance of our accelerator, we retrain the three networks with different drop rates p (i.e. 0.2 and 0.5), and rerun the inferences on both baseline accelerator and Fast-BCNN ($T_m = 64$). Fig.12 (b) indicates a degradation of speedup when drop rate decreases, especially in B-VGG16 and B-GoogLeNet. However, even with low drop rate, e.g. $p = 0.2$, Fast-BCNN still exhibits an averaged 3.5× speedup over baseline accelerator, which proves the effectiveness of unaffected neuron skipping strategy. We also observe a non-linear (slower) increase in speedup for each network. This is reasonable for the fact that speedup will not fully benefit from the increase of drop rate because more unaffected neurons may also become dropped neurons.

3) Performance of BRNGs: To test the functionality of our hardware implementation of BRNGs, we run the BRNG for 2000 cycles and 4000 cycles separately and calculate the real drop rate from the generated dropout bits. For comparison, we also obtain the real drop rate from 2000 and 4000 dropout bits generated by a software approach. The results in Table III suggest that our LFSR-based BRNG design achieves a better approximation to the theoretical drop rate.

VII. RELATED WORK

Bayesian deep learning A variety of Bayesian techniques has been proposed to address the uncertainty problem inherent in deep learning models. With the nature of simplicity and computational efficiency, the dropout interpretation of Bayesian approach [15] has become a popular method in a wide range of applications [30]–[32]. A majority of these applications focuses on deploying Bayesian approach on CNN models to enhance robustness of prediction in uncertain environment. Our work provides an architecture support to accelerate MC-dropout inference process in BCNN models.

BNN accelerator VIBNN [33] is the first work that proposes a hardware realization and acceleration for BNNs. They observe the frequent Gaussian random generation in BNN inference as the key challenge and target on designing high performance Gaussian random number generators. In contrast, our design is interested in optimizing the inner computation flow in BCNN inference via massive neuron skipping, since the cost of BRNG implementation is trivial. Additionally, we focus on accelerating CNN inference that is more complex and computationally expensive, while in VIBNN the accelerated neural network is a feed-forward neural network with only two hidden layers.

Skipping strategy in CNN accelerators Although there have been abundant research efforts that dive into skipping opportunities in CNNs, the unique MC-dropout technique in BCNNs enables us to adopt a new skipping mechanism that different from prior works. SCNN [10] designs a novel dataflow that dynamically skips the zero inputs/weights as well as maintaining the maximized data reuse. Cnvlutin [11] targets on removing ineffectual multiplications where the input activation is zero. These studies both focus on skipping the input activations, while in Fast-BCNN we aim to directly skip the output neurons based on the pre-generated dropout masks. Another skipping-based CNN accelerator is SnaPEA [34]. They manage to predict the zero neurons by examining the value of neurons at runtime and perform ReLU function early, thus, skipping the following computations for the current neuron. Fast-BCNN, however, is featured by skipping the entire computation for invalid neurons.

VIII. CONCLUSION

In this paper, we propose Fast-BCNN, an FPGA-based hardware accelerator for BCNN inferences. We characterize two types of unique neurons existing in BCNN inference that can be potentially skipped via dropout mask-based prediction. The abundant skipping opportunities motivate us to adopt a customized parallelism that perfectly serves neuron skipping mechanism. Furthermore, a novel PE design is proposed to support the dynamic skipping and predicting strategy with negligible overheads. Experimental results show that Fast-BCNN provides a 2.1∼8.2× speedup and 44%∼84% energy reduction for different BCNN models compared with the baseline accelerator.

ACKNOWLEDGMENT

This research is partially supported by National Science Foundation grants CCF-1900904, CCF-1619243, CCF1537085 (CAREER).

References

[1] G. Chéron, I. Laptev, and C. Schmid, "P-cnn: Pose-based cnn features for action recognition," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 3218–3226.

[2] A. Farooq, S. Anwar, M. Awais, and S. Rehman, "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, 2017, pp. 1–6.

[3] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[4] NHTSA, "Tesla crash preliminary evaluation report," U.S. Department of Transportation, National Highway Traffic Safety Administration, Tech. Rep., Jan 2017.

[5] C. Leibig, V. Allken, M. S. Ayhan, P. Berens, and S. Wahl, "Leveraging uncertainty information from deep neural networks for disease detection," *Scientific reports*, vol. 7, no. 1, pp. 1–14, 2017.

[6] A. Amini, A. Soleimany, S. Karaman, and D. Rus, "Spatial uncertainty sampling for end-to-end control," *arXiv preprint arXiv:1805.04829*, 2018.

[7] A. Kendall, V. Badrinarayanan, and R. Cipolla, "Bayesian segnet: Model uncertainty in deep convolutional encoder-decoder architectures for scene understanding," *arXiv preprint arXiv:1511.02680*, 2015.

[8] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[9] Y. Gal and Z. Ghahramani, "Bayesian convolutional neural networks with bernoulli approximate variational inference," *arXiv preprint arXiv:1506.02158*, 2015.

[10] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[11] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.

[12] G. Hinton and D. Van Camp, "Keeping neural networks simple by minimizing the description length of the weights," in *in Proc. of the 6th Ann. ACM Conf. on Computational Learning Theory*. Citeseer, 1993.

[13] D. Barber and C. M. Bishop, "Ensemble learning in bayesian neural networks," *Nato ASI Series F Computer and Systems Sciences*, vol. 168, pp. 215–238, 1998.

[14] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[15] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *International Conference on Machine Learning (ICML)*, 2016, pp. 1050–1059.

[16] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," 2015, URL: http://yann.lecun.com/exdb/lenet.

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

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[19] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[20] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 553–564.

[21] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 32–37.

[22] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[23] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[24] M. Murase, "Linear feedback shift register," Feb. 18 1992, US Patent 5,090,035.

[25] P. Bhaskar and P. Gawande, "A survey on implementation of random number generator in fpga," *International Journal of Science and Research (IJSR)*, vol. 4, no. 3, pp. 1590–1592, 2015.

[26] B. Unal, A. Akoglu, F. Ghaffari, and B. Vasić, "Hardware implementation and performance analysis of resource efficient probabilistic hard decision ldpc decoders," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 3074–3084, 2018.

[27] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[28] "Xilinx memory interface generator," https://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf Accessed April 10, 2019.

[29] "Xilinx power estimator," https://www.xilinx.com/products/technology/power/xpe.html Accessed April 10, 2019.

[30] A. Kendall and Y. Gal, "What uncertainties do we need in bayesian deep learning for computer vision?" in *Advances in neural information processing systems*, 2017, pp. 5574–5584.

[31] V. Peretroukhin, L. Clement, and J. Kelly, "Reducing drift in visual odometry by inferring sun direction using a bayesian convolutional neural network," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 2035–2042.

[32] K. Lee, K. Saigol, and E. A. Theodorou, "Safe end-to-end imitation learning for model predictive control," *arXiv preprint arXiv:1803.10231*, 2018.

[33] R. Cai, A. Ren, N. Liu, C. Ding, L. Wang, X. Qian, M. Pedram, and Y. Wang, "Vibnn: Hardware acceleration of bayesian neural networks," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 476–488, 2018.

[34] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 662–673.