High-Performance FPGA-based Accelerator for Bayesian Neural Networks

Hongxiang Fan*‡¶, Martin Ferianc†‡, Miguel Rodrigues†, Hongyu Zhou‖, Xinyu Niu§ and Wayne Luk*
*Department of Computing, Imperial College London, London UK, {*h.fan17, w.luk*}*@imperial.ac.uk*
†Department of Electronic and Electrical Engineering, University College London, London UK,
{*martin.ferianc.19, m.rodrigues*}*@ucl.ac.uk*
‖*hongyu.hyzhou@gmail.com*
§Corerain Technologies Ltd., Shenzhen China, *xinyu.niu@corerain.com*

***Abstract*—Neural networks (NNs) have demonstrated their potential in a wide range of applications such as image recognition, decision making or recommendation systems. However, standard NNs are unable to capture their model uncertainty which is crucial for many safety-critical applications including healthcare and autonomous vehicles. In comparison, Bayesian neural networks (BNNs) are able to express uncertainty in their prediction via a mathematical grounding. Nevertheless, BNNs have not been as widely used in industrial practice, mainly because of their expensive computational cost and limited hardware performance. This work proposes a novel FPGA-based hardware architecture to accelerate BNNs inferred through Monte Carlo Dropout. Compared with other state-of-the-art BNN accelerators, the proposed accelerator can achieve up to 4 times higher energy efficiency and 9 times better compute efficiency. Considering partial Bayesian inference, an automatic framework is proposed, which explores the trade-off between hardware and algorithmic performance. Extensive experiments are conducted to demonstrate that our proposed framework can effectively find the optimal points in the design space.**

I. Introduction

In recent years, neural networks (NNs) have demonstrated their outstanding performance in a variety of applications ranging from image classification [1] or segmentation [2] to human action recognition [3]. However, one of the main drawbacks in standard NNs is that they are not able to capture the model uncertainty which is crucial for many safety-critical applications such as healthcare [4] or autonomous vehicles [5]. In contrast to standard NNs, Bayesian neural networks (BNNs) [6], which adopt Bayesian inference to provide a principled uncertainty estimation, have become more popular in these applications.

BNNs [6] can describe complex stochastic patterns with well-calibrated confidence estimates. An example of this is shown in Figure 1, which demonstrates that the BNN is uncertain in its predictions when shown completely irrelevant input, in comparison to a standard NN, which is wrongfully overconfident. Hence with BNNs, we can treat special cases explicitly [7] and they have become relevant in applications where the notion of uncertainty is essential.

However, the advantage of BNNs comes with a burden: due to the high dimensionality of modern BNNs, it is intractable to analytically compute their predictive uncertainty. Instead, it is necessary to approximate the predictive distribution through Monte Carlo sampling that requires the users to perform repeated sampling of random numbers and then run the same input data through the BNN multiple times, which degradates the hardware performance. Several algorithmic approximation techniques and hardware architectures [8]–[10] have been proposed to improve the hardware performance of BNNs. Nevertheless, there are particular drawbacks in these approaches: *1)* The implementation needs of both an NN engine and a sampler makes the design resource and memory-demanding, and thus current accelerators can only support BNNs consisting solely of linear layers or binary operations, which does not reflect the need in the industrial or research communities in terms of the current state-of-the-art BNN architectures; *2)* To obtain the uncertainty prediction, these accelerators simply perform S forward passes through the whole network repeatedly without considering the actual algorithmic needs of BNNs, which makes them S times slower than standard NNs.

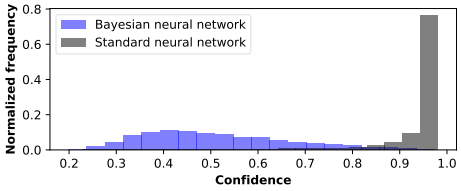


Fig. 1. Comparison of output confidence histograms for random noise input.

To address these challenges, we propose an FPGA-based design with the support for fine-grained parallelism to accelerate BNNs inferred through Monte Carlo Dropout (MCD) [7] with high performance. The proposed accelerator is versatile to support a variety of BNN architectures. To further improve the hardware performance, we consider partial BNNs to decrease the amount of computation required by BNNs. An automatic framework is proposed to explore the trade-off between hardware and algorithmic performance, which is able to find a suitable hardware configuration and algorithmic parameters given users' hardware constraints and algorithmic requirements. In summary, our contributions include:

- A novel hardware architecture with an intermediate-layer caching technique to accelerate Bayesian neural networks

‡ Equal contribution. ¶ Corresponding author.

inferred through Monte Carlo Dropout, which achieves high performance and resource efficiency (Section III).
- An exploration framework for hardware-algorithmic performance trade-off and uncertainty estimation provided by partial Bayesian neural network design (Section IV).
- A comprehensive evaluation of algorithmic and hardware performance on different datasets with respect to different state-of-the-art neural architectures (Section V).

II. Related Work

A. Field Programmable Gate Array-based Accelerators

Acceleration of standard NNs has enjoyed extensive research and industrial interests in the recent years [11]. Given the high computational demands of NNs, custom hardware accelerators are vital for boosting their performance. The high energy efficiency, computing capabilities and reconfigurability of FPGAs in particular make them a promising platform for acceleration of multiple different NN architectures [11]. Nevertheless, acceleration of BNNs specifically has not gained similar interests in the research community and there are only few works which approached this challenge [8]–[10].

In *VIBNN* [8], the authors developed an efficient FPGA-based accelerator for BNNs, however, they focused on BNNs consisting only of linear layers. Myojin *et al.* [9] propose a method for reducing the sampling time required for MCD [7] in edge computing by parallelising the calculation circuit using an FPGA. However, their method needs to binarise the BNN and they again focus only on linear layers. Awano & Hashimoto [10] propose a custom inference algorithm for BNNs consisting exclusively of linear layers - *BYNQNet* which employs quadratic nonlinear activation functions and hence the uncertainty propagation can be achieved using only polynomial operations. Although the design can achieve a high throughput, the restriction of the nonlinear activation functions limits generality for different application scenarios. In [5], the authors propose software-based intermediate-layer caching (IC), evaluated in last layer BNNs.

In comparison to these works, we focus on accelerating BNNs consisting of different layers with or without residual connections [12], including convolutions or pooling, that have been popular in the present-day networks [3], [12]. Additionally, our work wants to appeal to already wide-spread MCD without any additional software re-implementation effort.

B. Monte Carlo Dropout (MCD)

The concept MCD [7] lays in casting dropout [13] in NNs as Bayesian inference. Unlike the dropout used in standard NNs which is only enabled during training, MCD applies the dropout during both training and evaluation. MCD can be described as applying a random filter-wise mask $\boldsymbol{M}_i \in \mathbb{R}^{F_i}$ to the output feature maps $\boldsymbol{Y}_i$ of layer i with F_i filters. The mask $\boldsymbol{M}_i$ follows a Bernoulli distribution $p(\boldsymbol{M}_i|p_i)$ which generates binary random variables (0 or 1) with the probability $p_i \in [0, 1]$. p practically trades-off certainty, accuracy and calibration of the BNN. After MCD removes the output feature maps with zeros, the non-zero elements are then scaled by $\frac{1}{1-p_i}$. To get the final output $\boldsymbol{O}_i$ under MCD, the computation can be formulated as: $\boldsymbol{O}_i = \frac{1}{1-p_i}(\boldsymbol{Y}_i \odot \boldsymbol{M}_i)$ where $\odot$ represents a Hadamard product and $\boldsymbol{M}_i$ is generated by a Bernoulli sampler at runtime for different filters and layers. The uncertainty estimation and prediction is thus obtained by running the same input through the BNN S times, each time with different set of sampled masks $\boldsymbol{M}$ for each layer i where MCD is applied, and averaging the outputs. The works [2], [7] demonstrate that MCD can achieve a high quality of uncertainty estimation.

C. Partial Bayesian Inference

A full BNN should be trained with MCD applied after every layer [7]. However, the authors in [2], [14] have demonstrated theoretically and empirically that making a standard NN Bayesian in different parts of the NN, thus making it partially Bayesian, can improve uncertainty estimation and it can also improve accuracy. Assuming there is an N-layer NN, partial Bayesian inference applies MCD in the last $L; L \leq N$ layers and makes the first $N - L$ layers behave as a feature extractor for the Bayesian remainder. Partially applied dropout then represents a trade-off between hardware, algorithmic performance and uncertainty estimation [2]. In this paper, we exploit this trade-off by proposing a framework for exploring the positioning of MCD at different parts of the NN which results in a partial BNN.

III. Hardware Design

A. Design Overview

An overview of the proposed hardware design is illustrated in Figure 2. The computation of the BNN is performed layer-by-layer using the same hardware design. The intermediate outputs of each layer are transferred to off-chip memory to reduce the on-chip memory consumption, and they are loaded back to the input buffer for processing of the next layer. The weights of different layers are stored in off-chip memory and loaded to the weight buffer while processing the corresponding layer.

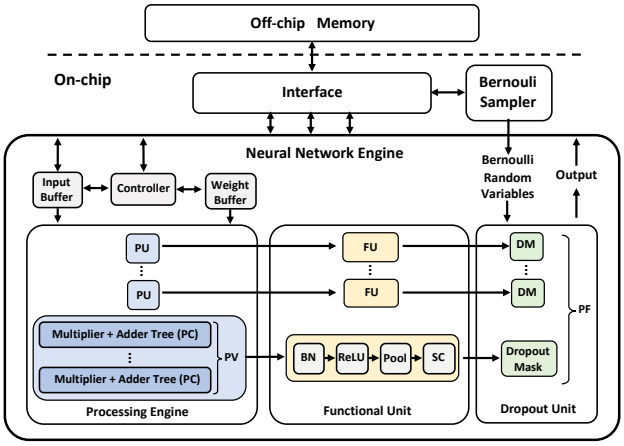


Fig. 2. Overview of the FPGA-based accelerator.

The main component in the proposed accelerator is a neural network engine (NNE), which is designed for running one layer at a time and general enough to run linear and convolutional layers with different kernel sizes. The NNE consists of a processing engine (PE), a functional unit (FU) and a dropout unit (DU). These sub-modules are queried in a pipeline manner to improve the hardware performance. The PE is designed to perform matrix multiplication, which supports three types of fine-grained parallelism: filter parallelism (PF), channel parallelism (PC) and vector parallelism (PV). In PE, there are PF processing units (PUs). Each PU contains PV multiplication-addition modules and each module contains PC multipliers followed by an adder tree for channel accumulation. After each PU, there is a chain of FU modules including batch normalization (*BN*) [15], Rectified linear unit (*ReLU*) activation, Pooling (*Pool*) and Shortcut (*SC*). The DU is placed at the end, which is a batch of multiplexers controller by the zeros and ones generated from the Bernoulli sampler.

B. Bernoulli Sampler

MCD is applied filter-wise, which means the number of Bernoulli random variables generated for each layer i is equal to the number of output filters. Therefore, we adopt the single-bit linear feedback shift register (LFSR) design to implement a Bernoulli sampler, which is illustrated in Figure 3.

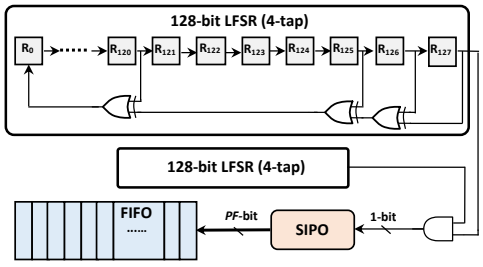


Fig. 3. Hardware architecture of the implemented Bernoulli sampler.

The LFSR is composed of a chain of shift registers formed as a loop. The maximum sequence length S_{max} of LFSR depends on the number of shift registers N_{reg} used in the loop: $S_{max} = 2^{N_{reg}} - 1$. The used LFSR design would take 1500 years to iterate through the whole sequence when clocked at 160MHz [16]. Since a single LFSR can only support Bernoulli sampling with 0.5 probability, the number of LFSRs depends on the required dropout rate. For instance, two LFSRs with an extra AND gate are required to implement Bernoulli sampler with $p = 0.25$. Also, as mentioned in Section III, the NNE only processes PF filters at a time, so we design a serial-in-parallel-out (SIPO) module, placed after LFSRs, to form a single Bernoulli bit of a PF-bit MCD mask. Since different filters are processed at different speeds, a first-in-first-out (FIFO) buffer is placed at the end of the Bernoulli sampler to cache generated Bernoulli random variables and pop out the mask when required. In case the overall processing is parallelised it is not necessary to use more than one sampler, however, the samples sampled during runtime for each instance need to be distinct.

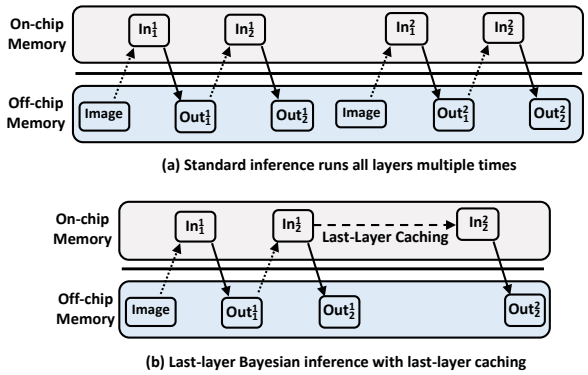


Fig. 4. An example of a two-layer neural network which computes two samples to obtain the prediction. The input and output data are denoted by In_j^i and Out_j^i where i means i^{th} iteration and j represents j^{th} layer.

C. Intermediate-layer Caching (IC)

To further improve the overall hardware performance, we propose a hardware implementation of IC technique [5] to decrease the required compute and the number of memory accesses. An example of using IC is illustrated in Figure 4, where the NN contains two layers and it only requires the user to apply the dropout mask and run the last layer S times when the partial Bayesian technique is applied. In IC, the input of the last layer is stored on chip until the sampling is finished. Assuming the NN requires to run the last L layers S times to obtain the prediction, the IC can reduce the compute by $(N - L) \times S$ times and the number of memory accesses by L times.

IV. Optimization Framework

A. Workflow of Framework

As mentioned in Section II-C, partially applying MCD represents a trade-off between latency, accuracy, confidence and uncertainty estimation. The trade-off is decided by three types of parameters: *1)* L which denotes the portion of Bayesian layers, *2)* S which represents the number of times needed to repetitively run the Bayesian parts and *3)* PC, PF, PV which represent hardware parallelism. In this paper, we propose a framework, shown in Figure 5, which automatically optimizes the configuration of the BNN with respect to parameters L, S and PC, PF, PV according to user requirements for the target hardware platform. In our hardware design space, we consider the domains for both PC and PF as $\{8, 16, 32, 64, 128\}$ and PV can be chosen from $\{1, 4, 8, 16\}$.

At the beginning, the framework requires users to specify the hardware constraints, optimization mode and the minimal requirement for each metric. The hardware constraints include the available DSPs and memory resources of the target hardware platform. The optimization mode is selected from optimal-latency, optimal-accuracy, optimal-uncertainty prediction and optimal-confidence to minimise or maximise

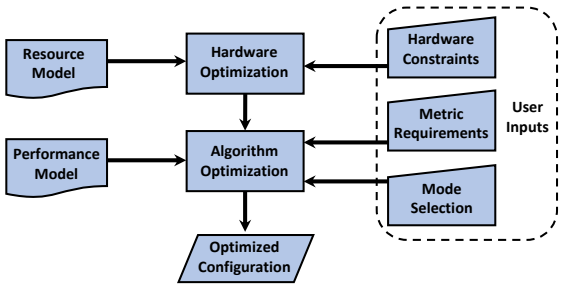


Fig. 5. Overview of the optimization framework.

the chosen objective through greedy optimisation with respect to software and hardware configurations. The first optimization is the hardware optimization, which determines the maximum parallelism level implementable on the target hardware in terms of PC, PF, PV. The resource model is used at this step to estimate the resource consumption given the available degrees of parallelism. During algorithmic optimization, based on the determined hardware parameters, we obtain the latency from the performance lookup table for various BNNs with different L and S. At the same time, the accuracy, the quality of uncertainty prediction and confidence of the BNN are evaluated in software. Then, the configurations which do not meet the minimal requirements are filtered. The final configurations are selected according to the optimization mode specified at the beginning.

B. Resource Model

As memory and DSPs are the limiting resources in FPGA-based NN accelerators [17], we mainly consider the memory and DSPs usage. The DSP usage depends on the multipliers used in the NNE. Due to 8-bit processing, we implement two multipliers using one DSP and thus the DSP consumption can be calculated as $DSP = \frac{PC \times PF \times PV}{2}$. The memory resources are mainly consumed by the weight buffer, input buffer in the NNE and the FIFO buffer in the Bernoulli sampler. As the width of the FIFO is PF, its memory consumption can be represented as $MEM_{FIFO} = D \times PF \times DW$, where D represents the depth of the FIFO used in the Bernoulli sampler and DW is the data width. As our design processes the NN layer-by-layer, the memory usage of input buffer is dominated by the layer with the maximal input size as $MEM_{in} = \max_{i=1,\dots,N}(C_i \times H_i \times W_i) \times DW$, where H_i, W_i and C_i are the height, width and number of input channels of the i^{th} layer respectively. Since the weight buffer only needs to cache PF filters, the memory consumption of the weight buffer can be formulated as $MEM_{weight} = \max_{i=1,\dots,N}(C_i \times K_i \times K_i) \times PF \times DW$, where K_i is the kernel size of the i^{th} layer.

V. Experiments

A. Experimental Setup

In this paper, Intel Arria 10 SX660 FPGA is set as our target hardware platform. 1GB DDR4 SDRAM is installed as off-chip memory. The PyTorch framework is used for the software implementation. We focus on image classification. We evaluate the networks on tuples $(\boldsymbol{x}, \boldsymbol{y})$, where the target $\boldsymbol{y}$ is an one-hot encoding of $k = 1, \dots, K$ classes. Given the image input $\boldsymbol{x}$, we approximate the predictive distribution $p(\boldsymbol{y}|\boldsymbol{x})$ over the target with respect to S samples as $\frac{1}{S}\sum_{s=1}^{S} p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{M})$; $\boldsymbol{M} \sim p(\boldsymbol{M}|p)$, where the $\boldsymbol{M}$ is the set of Bernoulli masks and S can be $S = \{3, 4, 5, 6, 7, 8, 9, 10, 20, 50, 100\}$. We consider $p = 0.25$ for all MCD instances.

For datasets, we consider classifying images of increasing difficulty: MNIST, SVHN and CIFAR-10, through which we control the complexity of the experiments. For MNIST we implement *LeNet-5* [18], *VGG-11* [19] for SVHN and *ResNet-18* [12] for CIFAR-10. We reduced the channel size of *VGG-11* and *ResNet-18* to fit them into memory. In terms of partial Bayesian inference, we explore adding dropout in the different parts of the NN, always following a convolutional, BN and ReLU layers, and optionally pooling. Similarly to the datasets, we explore state-of-the-art architectures of increasing complexity, whose core is widely used across practical applications. Their structural irregularities present challenges to the accelerator's design. We consider partial BNNs, such that $L = \{1, \frac{1}{3} \times N, \frac{1}{2} \times N, \frac{2}{3} \times N, N\}$. All experiments were repeated 5 times.

In addition to measuring the classification accuracy, we establish metrics for the evaluation of the predictive uncertainty and confidence. For the input that should rightfully confuse the net, we measure the quality of the uncertainty prediction with respect to random Gaussian noise with mean and variance of the training data with the average predictive entropy (aPE) over a dataset of size E as: $\text{aPE} = \frac{1}{E}\sum_{e=1}^{E} -\sum_{k=1}^{K} p(y_e^k|\boldsymbol{x}_e) \log p(y_e^k|\boldsymbol{x}_e)$. Additionally, we measure the confidence with which the net is making its predictions on the test data through the expected calibration error (ECE) [20]. ECE signals that a BNN is uncalibrated if it is making predictions whose confidence are not matching its accuracy. We calculate ECE with respect to 10 bins.

We implement our design using Verilog and Quartus 17 Prime Pro is used for synthesis and implementation. Based on the resource model and the available resources on our FPGA, PC, PF and PV are set to be 64, 64 and 1 respectively and the final design is clocked at 225 MHz. The resource usage of the proposed accelerator is presented in Table II. Since our accelerator is based on 8-bit precision, the 8-bit linear quantization [21] is applied on the trained models.

B. Hardware Performance Comparison

For each network, we measure the hardware performance on the FPGA, Intel Core i9-9900K CPU and NVIDIA RTX 2080 SUPER GPU, the batch size is 1 for all the hardware platforms for a fair comparison[¶]. For the FPGA implementation, we measure the latency with and without IC (Section III-C) to demonstrate its effect and the results are shown in Table III, the

[¶] Since PyTorch does not support 8-bit quantization on a GPU, the latency of GPU is estimated by dividing its floating-point performance by 4 times, which is the theoretically the lowest latency that the GPU can achieve.

TABLE I
THE RESULTANT CONFIGURATIONS OF BNNS UNDER DIFFERENT OPTIMIZATION MODES.

	Opt-Mode	$\{\boldsymbol{L}, \boldsymbol{S}\}$	Latency [ms] ↓			aPE [nats] ↑	ECE [%] ↓	Accuracy [%] ↑
			FPGA	CPU	GPU			
LetNet-5	*Opt-Latency*	1, 3	0.42	0.67	0.24	0.63 ± 0.09	0.25 ± 0.05	99.27 ± 0.04
	Opt-Accuracy	$\frac{2}{3} \times N$, 100	14.32	24.69	12.87	0.75 ± 0.15	0.13 ± 0.03	99.39 ± 0.05
	Opt-Uncertainty	N, 100	14.83	42.0	19.91	1.06 ± 0.19	0.17 ± 0.04	99.32 ± 0.04
	Opt-Confidence	N, 9	1.29	3.68	1.68	0.98 ± 0.18	0.1 ± 0.04	99.31 ± 0.03
VGG-11	*Opt-Latency*	1, 3	0.57	0.95	0.68	1.38 ± 0.28	2.8 ± 0.12	95.38 ± 0.1
	Opt-Accuracy	N, 100	57.32	186.24	88.93	1.97 ± 0.05	2.42 ± 0.19	96.49 ± 0.05
	Opt-Uncertainty	$\frac{2}{3} \times N$, 100	42.89	110.32	59.78	2.02 ± 0.11	0.41 ± 0.05	96.13 ± 0.1
	Opt-Confidence	$\frac{2}{3} \times N$, 100	42.89	110.32	59.78	2.02 ± 0.11	0.41 ± 0.05	96.13 ± 0.1
ResNet-18	*Opt-Latency*	1, 3	0.47	1.31	0.87	0.36 ± 0.26	4.85 ± 0.19	92.84 ± 0.16
	Opt-Accuracy	1, 8	0.50	2.03	1.17	0.38 ± 0.27	4.74 ± 0.14	92.91 ± 0.14
	Opt-Uncertainty	$\frac{1}{2} \times N$, 100	32.04	173.53	93.23	1.27 ± 0.27	2.74 ± 0.31	91.12 ± 0.2
	Opt-Confidence	$\frac{2}{3} \times N$, 3	1.20	7.66	3.93	1.05 ± 0.26	1.08 ± 0.06	89.99 ± 0.17

TABLE II
RESOURCE UTILIZATION OF THE ACCELERATOR ON THE FPGA.

Resources	ALMs	Registers	DSPs	M20K
Used	303,913	889,869	1,473	2,334
Total	427,200	1,708,800	1,518	2,713
Utilization	71%	52%	97%	86%

TABLE III
HARDWARE COMPARISON BETWEEN FPGA, CPU AND GPU.

	$\{\boldsymbol{L}, \boldsymbol{S}\}$	Latency [ms] ↓			
		FPGA		CPU	GPU
		w/ IC	w/o IC		
LetNet-5	1, 100	13.73	14.38	11.17	5.81
	$\frac{2}{3} \times N$, 50	7.16	7.20	12.02	6.07
VGG-11	1, 100	0.76	57.3	11.76	6.33
	$\frac{2}{3} \times N$, 50	21.52	28.67	55.94	30.09
ResNet-18	1, 100	1.22	44.97	13.96	7.05
	$\frac{2}{3} \times N$, 50	18.90	22.48	131.41	65.9

down and up arrows indicate the desired tendency for a given metric. While comparing FPGA implementations with and without IC on *VGG-11* and *ResNet-18*, it can be seen that the speed up brought by IC goes down when L increases and the S decreases. In comparison to CPU and GPU implementations, the BNNs on the FPGA with IC can achieve up to 15 times and 8 times speed up respectively. There are two reasons for the speedup: *1)* The adoption of IC technique together with MCD and partial Bayesian inference, which decreases the amount of memory accesses and computation; *2)* The support for fine-grained parallelism on the accelerator, which fully utilized the extensive concurrency exhibited in BNNs. On *LeNet-5*, since the execution time is mainly occupied by the last layer, IC does not bring too much improvement on FPGA compared with GPU and CPU. However, because the current state-of-the-art NNs only spend a small portion of the execution time in the last layer, our accelerator can still achieve speed up in most of NNs, and thus is practical enough for real-life applications.

We also compare our work with the other BNN accelerators [8], [10] in Table IV. Because the three-layer BNNs evaluated in [8] and [10] are unrealistic for real-life applications, we run a commonly-used ResNet-101 [12] on our design with MCD applied onto every layer, such that $L = N$. For a fair comparison, we evaluate all the accelerators in terms of throughput, compute and energy efficiency[¶]. As shown in Table IV, our accelerator can achieve 3 times to 4 times higher energy efficiency and 6 times 9 times better compute efficiency. Also, it is worth to mention that previous BNN accelerators only support linear layers, while our proposed accelerator is versatile enough to support a wide range operations including convolution, pooling or residual addition.

C. Effectiveness of Framework

As introduced in Section IV, our framework is designed to explore the trade-off between accuracy, latency, uncertainty estimation and confidence. This section investigates design space exploration with and without user constraints.

1) Exploration Without Constraints: To find the global optimal latency, accuracy, uncertainty and confidence points, we set four different optimization modes: *Opt-Latency*, *Opt-Accuracy*, *Opt-Uncertainty* and *Opt-Confidence*, for all BNNs without any constrains. The results are illustrated in Table I. The lowest latencies that our accelerator can achieve on these

[¶] The energy efficiency is quoted in in giga-operations per second per watt (GOP/s/W) and the total board power consumption is 45W.

TABLE IV
PERFORMANCE COMPARISON WITH OTHER BNN ACCELERATORS.

	VIBNN [8]	*BYNQNet* [10]	*Our work*
FPGA	Cyclone V 5CGTFD9E5F35C7	Zynq XC7Z020	Arria 10 SX660
Clock frequency [MHz]	212.95	200	225
Total number of DSPs	342	220	1473
Energy [W] ↓	6.11	2.76	45.00
Throughput [GOP/s] ↑	59.6	24.22	1590
Energy Eff. [GOP/s/W] ↑	9.75	8.77	33.3
Comp. Eff. [GOP/s/DSP] ↑	0.174	0.121	1.079

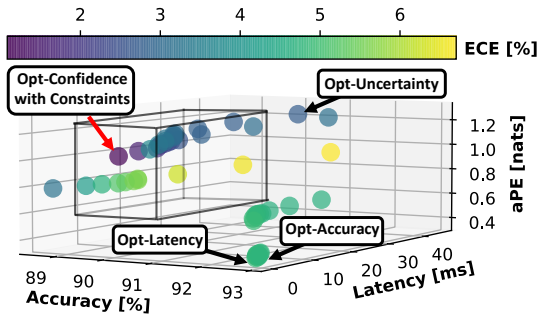


Fig. 6. Design space exploration with latency, accuracy and uncertainty constraints for *ResNet-18* on CIFAR-10.

three NNs are 0.42ms, 0.57ms and 0.47ms respectively. With the *Opt-Accuracy* mode enabled, these three NNs can achieve 99.39%, 96.49% and 92.91% accuracy respectively on their corresponding datasets. The framework also suggests different $\{L, S\}$ configurations to achieve the optimal aPE and ECE.

2) Exploration With Constraints: To demonstrate that our framework is able to find the optimal points when the user's requirements are given, we set latency, accuracy and uncertainty constraints for *ResNet-18* on CIFAR-10 dataset and select the *Opt-Confidence* mode for optimization. Figure 6 shows all the candidate points with respect to accuracy, latency, aPE and ECE. The global optimal points with respect to different metrics are highlighted by the black arrows. The feasible design space constructed by accuracy, latency and uncertainty constraints is represented by the black box. Within this feasible design space, our framework generates the point with the lowest ECE, which is marked by the red arrow. Therefore, the proposed framework is able to find the optimal points when users' constraints are given.

VI. CONCLUSION

This work proposes a high-performance FPGA-based design to accelerate Bayesian neural networks (BNNs) inferred through Monte Carlo Dropout. The accelerator is versatile enough to support a variety of Bayesian neural networks and it achieves up to 4 times higher energy efficiency and 9 times better compute efficiency than other state-of-the-art accelerators. Additionally, we presented a framework to automatically trade-off both hardware and algorithmic performance, given hardware constraints and algorithmic requirements. In future we aim to explore neural architecture search on BNN, and co-develop the hardware design for BNNs found.

REFERENCES

[1] M. Ferianc, H. Fan, and M. Rodrigues, "Vinnas: Variational inference-based neural network architecture search," *arXiv preprint arXiv:2007.06103*, 2020.

[2] 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.

[3] H. Fan, C. Luo, C. Zeng, M. Ferianc, Z. Que, S. Liu, X. Niu, and W. Luk, "F-E3D: FPGA-based acceleration of an efficient 3D convolutional neural network for human action recognition," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, vol. 2160, 2019, pp. 1–8.

[4] F. Liang, Q. Li, and L. Zhou, "Bayesian neural networks for selection of drug sensitive genes," *Journal of the American Statistical Association*, vol. 113, no. 523, pp. 955–972, 2018.

[5] T. Azevedo, R. de Jong, M. Mattina, and P. Maji, *Stochastic-yolo: Efficient probabilistic object detection under dataset shifts*, 2020. arXiv: 2009.02967.

[6] R. M. Neal, "Bayesian learning via stochastic dynamics," in *Advances in neural information processing systems*, 1993, pp. 475–482.

[7] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, 2016, pp. 1050–1059.

[8] 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.

[9] T. Myojin, S. Hashimoto, and N. Ishihama, "Detecting uncertain bnn outputs on fpga using monte carlo dropout sampling," in *International Conference on Artificial Neural Networks*, Springer, 2020, pp. 27–38.

[10] H. Awano and M. Hashimoto, "Bynqnet: Bayesian neural network with quadratic activations for sampling-free uncertainty estimation on fpga," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1402–1407. DOI: 10.23919/DATE48585.2020.9116302.

[11] S. Mittal, "A survey of fpga-based accelerators for convolutional neural networks," *Neural computing and applications*, pp. 1–31, 2020.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[13] 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.

[14] A. Kristiadi, M. Hein, and P. Hennig, "Being bayesian, even just a bit, fixes overconfidence in relu networks," *arXiv preprint arXiv:2002.10118*, 2020.

[15] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[16] R. Andraka and R. Phelps, "An FPGA based processor yields a real time high fidelity radar environment simulator," in *Military and Aerospace Applications of Programmable Devices and Technologies Conference*, 1998, pp. 220–224.

[17] S. Liu, H. Fan, X. Niu, H.-c. Ng, Y. Chu, and W. Luk, "Optimizing CNN-based segmentation with deeply customized convolutional and deconvolutional architectures on FPGA," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–22, 2018.

[18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

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

[20] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," *arXiv preprint arXiv:1706.04599*, 2017.

[21] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.