

Optimizing Bayesian Recurrent Neural Networks on an FPGA-based Accelerator

Martin Ferianc^{‡*}, Zhiqiang Que^{††}, Hongxiang Fan^{§†}, Wayne Luk[†], and Miguel Rodrigues^{*}

^{*}Department of Electronic and Electrical Engineering, University College London, London UK,

{martin.ferianc.19, m.rodrigues}@ucl.ac.uk

[†]Department of Computing, Imperial College London, London UK, {z.que, h.fan17, w.luk}@imperial.ac.uk

Abstract—Neural networks have demonstrated their outstanding performance in a wide range of tasks. Specifically recurrent architectures based on long-short term memory (LSTM) cells have manifested excellent capability to model time dependencies in real-world data. However, standard recurrent architectures cannot estimate their uncertainty which is essential for safety-critical applications such as in medicine. In contrast, Bayesian recurrent neural networks (RNNs) are able to provide uncertainty estimation with improved accuracy. Nonetheless, Bayesian RNNs are computationally and memory demanding, which limits their practicality despite their advantages. To address this issue, we propose an FPGA-based hardware design to accelerate Bayesian LSTM-based RNNs. To further improve the overall algorithmic-hardware performance, a co-design framework is proposed to explore the most fitting algorithmic-hardware configurations for Bayesian RNNs. We conduct extensive experiments on healthcare applications to demonstrate the improvement of our design and the effectiveness of our framework. Compared with GPU implementation, our FPGA-based design can achieve up to 10 times speedup with nearly 106 times higher energy efficiency. To the best of our knowledge, this is the first work targeting acceleration of Bayesian RNNs on FPGAs.

Index Terms—Recurrent neural networks, Bayesian inference, Field-programmable gate array, Hardware acceleration

I. INTRODUCTION

Recurrent neural networks (RNNs) have demonstrated their successes in various sequencing modelling tasks [1]. Among RNN variants [2], [3], Long Short-Term Memory (LSTM) has become the most wide-spread cell due to its ability in utilizing and remembering the past knowledge [3]. Although the regular LSTM-based RNNs show excellent capability in time-series modelling, they are not able to express their model-epistemic uncertainty and they may overfit on small datapools [4].

To enable uncertainty estimation, overfitting prevention and overall accuracy improvement, Bayesian LSTM-based RNNs have been proposed [4], which learn distributions over their weights instead of constant-pointwise values. Through repeated Monte Carlo (MC) sampling of the weights and corresponding multiple feedforward passes through the network, the Bayesian model is able to express its prediction along with both epistemic and aleatoric uncertainty [4]. Bayesian RNNs were applied in contrasting applications, for example: unemployment forecasting [5], fault detection [6], language modelling [7] or medicine [8].

[‡] Equal contribution. [§] Corresponding author.

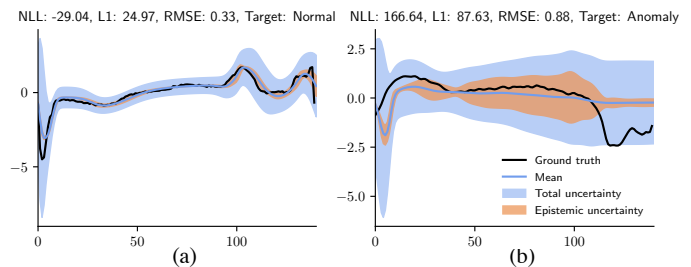


Fig. 1. Anomaly detection in a normal (a) and anomalous ECG case (b). The Bayesian model can perfectly fit the normal case, while not being able to replicate the anomalous case along with high uncertainty. The y-axis represents zero mean and unit variance centered voltage. The x-axis represents the timesteps with 140 timesteps in total. The fit is measured with respect to negative log-likelihood (NLL), L1 and root-mean-squared error (RMSE). Total uncertainty combines aleatoric and epistemic uncertainty. The predicted uncertainty, as a shaded area, is shown as ± 3 standard deviations.

The deployment of Bayesian RNNs is especially useful in medical applications where uncertainty estimation enables users to better understand and interpret the model's predictions. A demonstration of this is shown in Figure 1 where the Bayesian recurrent architecture is used to detect anomalies in an electrocardiogram (ECG) through its reconstruction. In an anomalous ECG on the right, the model is more uncertain in its prediction in comparison to the normal case. Therefore, a physician can be better guided in their investigations and diagnoses based on the modelled uncertainty, instead of looking only at the reconstructed mean or the quantitative metrics.

However, the benefits of Bayesian RNNs come with real-world execution burdens: the required MC sampling to obtain the prediction as well as the model uncertainty degrade their hardware performance, which limits their deployment in real-life applications. For instance, a typical three-layer Bayesian RNN with hidden size being 32 with 100 MC samples requires 10.46 seconds on an Intel Xeon CPU, which cannot meet the requirements of real-world applications, e.g. with respect to real-time ECG analysis [8] or fault detection [6].

Therefore, there is a demand for specific hardware accelerators for Bayesian RNNs. Nevertheless, there are several challenges while accelerating Bayesian RNNs:

- *Compute-intensive*: To make a prediction, Bayesian RNN might sequentially perform the feedforward pass through the whole network S times, which significantly increases

the amount of required computation.

- *Memory-intensive*: Sampling the weight distributions S times produces S different sets of weights, which multiplies the memory requirement by S times compared with that of pointwise non-Bayesian RNNs.
- *Resource-intensive*: As Bayesian RNN requires to implement both an RNN engine and random number generators, it demands more resources than a pointwise alternative.

In this work we introduce several strategies to target these challenges. The compute and memory demands are targeted by our proposed pipelining scheme and efficient random number generation that account for recurrence and data dependency of Bayesian RNNs. Moreover, the structure and portion of Bayesian layers in an RNN and the configuration of our hardware design present a trade-off between algorithmic and hardware performance. To provide efficient resource utilization, we introduce a framework for design space exploration (DSE) tailored to Bayesian RNNs and a configurable accelerator. To the best of our knowledge, this is the first field-programmable gate array (FPGA) based accelerator for Bayesian LSTM-based RNN architectures using Monte Carlo Dropout (MCD) [4]. In summary, our contributions include:

- A novel hardware architecture to accelerate Bayesian LSTM-based recurrent neural networks inferred through Monte Carlo Dropout, which achieves low latency and high energy efficiency (Section III).
- An automatic framework for exploring the algorithmic-hardware performance trade-off under users' requirements e.g. with respect to uncertainty estimation while targeting Bayesian recurrent architectures (Section IV).
- A comprehensive evaluation of algorithmic and hardware performance with respect to real-time ECG anomaly detection and classification with respect to different LSTM-based recurrent architectures (Section V).

II. PRELIMINARIES AND RELATED WORK

In this section we review recurrent neural networks, Bayesian inference and related hardware accelerators.

A. Recurrent Neural Networks

RNNs were demonstrated to achieve outstanding performance in a number of tasks where understanding time-related relationships was crucial [3], [9], [10]. In particular, LSTM [3] was proven to be effective in capturing long-term dependencies through recurrent processing and storing of useful information. Therefore, this work focuses on accelerating recurrent architectures built around LSTMs. LSTM operation can be described by the following equations:

$$\begin{aligned} \mathbf{i}_t &= \sigma(\mathbf{W}_x^i \mathbf{x}_t + \mathbf{W}_h^i \mathbf{h}_{t-1} + \mathbf{b}^i) & \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \\ \mathbf{f}_t &= \sigma(\mathbf{W}_x^f \mathbf{x}_t + \mathbf{W}_h^f \mathbf{h}_{t-1} + \mathbf{b}^f) & \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \\ \mathbf{g}_t &= \tanh(\mathbf{W}_x^g \mathbf{x}_t + \mathbf{W}_h^g \mathbf{h}_{t-1} + \mathbf{b}^g) \\ \mathbf{o}_t &= \sigma(\mathbf{W}_x^o \mathbf{x}_t + \mathbf{W}_h^o \mathbf{h}_{t-1} + \mathbf{b}^o) \end{aligned}$$

The σ , \tanh , \odot represent element-wise sigmoid, tanh and multiplication operations. $\mathbf{W} = \{\mathbf{W}_x^i, \mathbf{W}_x^f, \mathbf{W}_x^g, \mathbf{W}_x^o, \mathbf{W}_h^i, \mathbf{W}_h^f, \mathbf{W}_h^g, \mathbf{W}_h^o\}$ and $\mathbf{b} = \{\mathbf{b}^i, \mathbf{b}^f, \mathbf{b}^g, \mathbf{b}^o\}$ represent the learnable weights and biases. $\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}$ denote the input

$\mathbf{x}_t \in \mathbb{R}^I$ with I features, hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^H$ with H features and the cell state $\mathbf{c}_{t-1} \in \mathbb{R}^H$ with H features at the current time step t or $t-1$, the previous time step out of total time steps T . Note that, $\mathbf{h}_0, \mathbf{c}_0$ are initialized as zeroes. The intermediate outputs $\mathbf{i}_t, \mathbf{f}_t, \mathbf{g}_t, \mathbf{o}_t \in \mathbb{R}^H$ are the input, forget, modulation and output gates respectively. Note that we replicate the input \mathbf{x}_t and the hidden state \mathbf{h}_{t-1} such that:

$$\begin{aligned} \mathbf{x}_t^i, \mathbf{x}_t^f, \mathbf{x}_t^g, \mathbf{x}_t^o &= \mathbf{x}_t \\ \mathbf{h}_{t-1}^i, \mathbf{h}_{t-1}^f, \mathbf{h}_{t-1}^g, \mathbf{h}_{t-1}^o &= \mathbf{h}_{t-1} \end{aligned}$$

The decoupling of the input and the hidden state for each weight or gate is crucial for performing Bayesian inference [4].

B. Bayesian Inference

MCD in RNNs lays in casting dropout [11] as Bayesian inference with two major differences [4]. First, the dropout is enabled during training as well as evaluation. Second, the dropout mask $\mathbf{z} \sim \text{Bernoulli}(1-p)$; $\mathbf{z} = \{\mathbf{z}_x^i, \mathbf{z}_x^f, \mathbf{z}_x^g, \mathbf{z}_x^o \in \mathbb{R}^I; \mathbf{z}_h^i, \mathbf{z}_h^f, \mathbf{z}_h^g, \mathbf{z}_h^o \in \mathbb{R}^H\}$, with the same dimensionality as one time step of the input or the hidden state, is sampled only once for all time steps T and individually for all $\mathbf{x}_t^i, \mathbf{x}_t^f, \mathbf{x}_t^g, \mathbf{x}_t^o$ and $\mathbf{h}_{t-1}^i, \mathbf{h}_{t-1}^f, \mathbf{h}_{t-1}^g, \mathbf{h}_{t-1}^o$, such that for example $\mathbf{x}_t^i = \mathbf{x}_t^i \odot \mathbf{z}_x^i$ or $\mathbf{h}_{t-1}^i = \mathbf{h}_{t-1}^i \odot \mathbf{z}_h^i$. Probability $p \in [0, 1]$ of sampling 0 practically represents the trade-off between accuracy and calibration of the architecture. Dropout can be applied to only input, only the hidden states or both and it does not need to be applied to every cell in an architecture, which results in a partially Bayesian architecture [12]. From the hardware perspective, such architectures represent a trade-off between algorithmic and hardware performance [13]. The prediction in Bayesian architectures is obtained by running the same input through the RNN S times, each time with a different set of sampled masks \mathbf{z} for each layer i where MCD is applied. The collected outputs from the individual passes are then averaged to form a prediction. The S samples increase the compute and number of memory accesses linearly with complexity $\mathcal{O}(S)$. Bayesian RNNs have been used in time-series forecasting and classification [5], [8], where they demonstrated fine algorithmic performance.

C. Hardware Accelerators

Due to high computational, low-latency and reconfigurability demands, custom hardware accelerators for NNs represent a viable implementation platform. Especially FPGAs present an energy-efficient, configurable and high-performance hardware technology for accelerating NN architectures [14].

There has been ample work on FPGA-based implementations of persistent LSTMs whose weights are stored in on-chip memory [15]–[19]. For example, *FINN-L* [15] quantizes the RNN into 1-8 bits which surpasses a single-precision floating-point accuracy for a given dataset. Other studies on LSTM implementations store weights in the off-chip memory considering an FPGA, which was identified as a performance bottleneck [2], [20]–[22]. In addition, LSTM weights' reuse methods [21], [22] between various timestep were proposed to reduce the off-chip memory accesses to decrease the energy

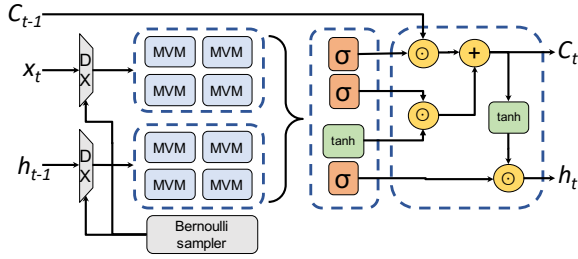


Fig. 2. Overview of the hardware implementation of the Bayesian LSTM.

cost and improve the overall system’s throughput. Some of the previous studies [1], [23]–[25] focused on weight pruning and model compression to reduce the size of weights to achieve favorable hardware performance. In [26], *BLINK* was proposed which utilized bit-sparse data representation for the LSTM runtime. It improved the energy efficiency of the LSTM by turning the multiplication into a bit shift operation without impairing its accuracy. However, none of these FPGA-based RNN designs target Bayesian RNNs.

At the same time, several hardware accelerators have been proposed to accelerate Bayesian NNs (BNNs) [13], [27]–[29]. However, these designs only focus on accelerating feedforward BNNs. Cai *et al.* [28] proposed a hardware design called *VIBNN* to accelerate BNNs consisting only of dense layers. Their accelerator consumes a large amount of resources while implementing Gaussian random number generators. Awano & Hashimoto [29] proposed *BYNQNet* to accelerate BNNs, which achieves 4.07 and 8.99 times higher throughput and energy efficiency than *VIBNN*. However, the design puts strict restrictions on the used nonlinear activation functions and thus limiting real-world applicability. By exploiting the activation sparsity in BNNs, [30] proposed a novel hardware architecture called *Fast-BCNN*. Nevertheless, the design can only be used to accelerate Bayesian convolutional NNs (BCNNs) with ReLU [31]. Fan *et al.* [13] proposed an FPGA-based accelerator for BCNNs inferred through MCD [4]. The design achieves nearly 10 times higher compute efficiency than *BYNQNet*. None of the previously mentioned accelerators for BNNs target RNNs, and thus, they do not consider the recurrence or inherent data dependency in RNNs.

In comparison to previous work, this paper focuses on accelerating Bayesian RNNs. To the best of our knowledge, this is the first work to accelerate Bayesian RNNs on an FPGA.

III. HARDWARE DESIGN

In this section we outline the proposed pipelined accelerator and an efficient random number generation for MCD-based BNNs and the target recurrent architectures.

A. Design Overview

This work adopts a streaming design [20], [32], [33] where all individual layers are mapped on-chip and different layers run in a pipelined fashion to achieve low latency. Unrolling the overall architecture in this way results in a more efficient

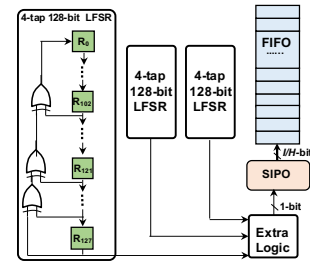


Fig. 3. Hardware architecture of the implemented Bernoulli sampler.

utilization of resources, with a 1-to-1 ratio of DSP blocks to compute units. Besides, this design adopts the initiation interval (II) balancing for multiple LSTM layers to achieve low latency and high hardware efficiency.

An overview of the proposed hardware design of a single LSTM layer is illustrated in Figure 2. The input and output data are transferred using DMA via an AXI bus. The input x_t and hidden state h_{t-1} are masked by the output of Bernoulli samplers and then fed to the LSTM gates. The masking along with the decomposition, as discussed in Section II-A, is performed by demultiplexor units (DX) that control which individual features get passed forward. There are four gates at the front of the LSTM layer, each containing a matrix-vector multiplication (MVM) unit. The element-wise operations and activation functions: sigmoid, tanh, addition and multiplication are performed on the output of the MVMs, previously factoring in the weights W_i and biases b_i of that given LSTM i . At the end of the layer, the current cell state c_t and hidden state h_t are produced. The h_t is required in the LSTM gates in the next time step iteration; it shows the existence of data dependencies which are not in forward-only NNs. The activation functions are implemented using BRAM-based lookup tables with a range of precomputed input values. The weights and biases are mapped on-chip automatically into registers when the design is synthesized. Hence, weight sampling is avoided along with additional memory traffic by introducing routing through DXs, enabling complete on-chip computation and elimination of the memory challenge in Section I. A similar design logic as presented here can be used for other recurrent units such as the gated recurrent unit [34].

B. Bernoulli Sampler and Design Pipelining

As MCD randomly sets inputs as zeros during runtime, it requires the hardware to generate random 1s and 0s. To achieve this goal, we design a Bernoulli sampler in hardware as illustrated in Figure 3. The 4-tap linear feedback shift register (LFSR) is the basic module in our Bernoulli sampler, which generates random binary values with a probability of $p=0.5$. To generate random binaries with user-defined probability, there are N_{lfsr} LFSRs followed by an extra logic block. For instance, to generate zeros with a probability $p=0.125$, it requires $N_{lfsr}=3$ with an extra three-input NAND gate as the extra logic. In this paper, to save the hardware resources, we set $N_{lfsr}=3$ and we set the dropout probability uniformly

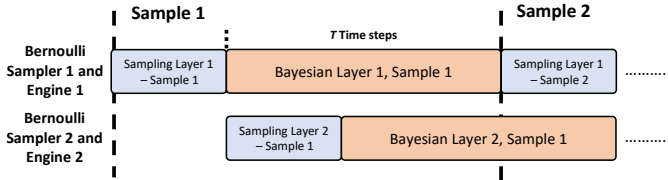


Fig. 4. Overlapping the computation with Bernoulli sampling.

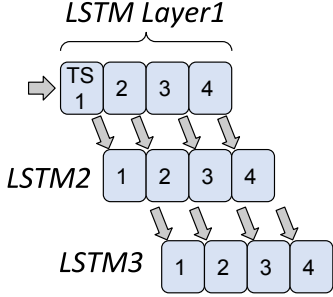


Fig. 5. Three cascaded LSTM layers with time step (TS) pipelining.

to $p=0.125$, as advised by [8], for both the inputs x as well as hidden states h . Since all the generated random binary values need to be outputted in parallel, a serial-in-parallel-out (SIPO) module is placed after LFSRs followed by a first-in-first-out (FIFO) module. If the given layer is not Bayesian, both DX and Bernoulli sampler are not needed.

To further improve the hardware performance, we propose to overlap the Bernoulli sampling with the computation of LSTMs, which is illustrated in Figure 4. As the Bernoulli sampling does not rely on the inputs, it can be performed before the start of all time steps T for a single LSTM. However, generating random binaries for all engines and inputs and hidden states will cost a large amount of on-chip memory. Therefore, all the Bernoulli samplers in our design only pre-sample random binaries required by a single input. This overlapping approach can hide the time cost of Bernoulli sampling into the computation, and at the same time, decrease the on-chip memory consumption. In addition to the sample-wise pipelining, we also introduce the pipelining over each time step to further increase parallelism, as shown in Figure 5. Note that while the illustrated example shows 3 cascaded LSTM layers with 4 time steps, the real design may involve more layers and more time steps. By leveraging the combination of sample-wise pipelining and time step pipelining, our design provides a fundamental solution to the Bayesian RNNs that demand repeated MC sampling, targeting the compute-intensive challenge mentioned in Section I.

C. Recurrent Autoencoder and Classifier

Figure 6 (a) demonstrates the hardware architecture of the recurrent autoencoder that is used for anomaly detection [9], [10] in our experiments. It consists of two parts: a pipelined encoder and a pipelined decoder, each balanced with NL LSTM

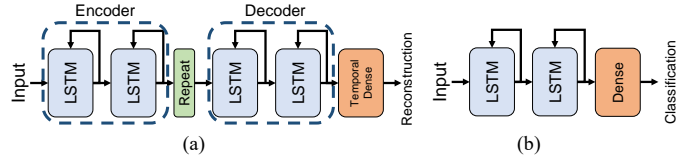


Fig. 6. Recurrent autoencoder (a) and classifier (b) architectures each with $NL=2$ LSTMs in their respective parts.

instances giving in total $2NL$ layers. Given the unrolling, the hardware resource consumption scales with the total layer count. Encoder processes the time-series input $x \in \mathbb{R}^{T \times I}$ into a bottleneck encoding containing only the last hidden state $h_T \in \mathbb{R}^{H/2}$ of the last LSTM in the encoder. The last hidden state in the decoder has a reduced dimensionality $\mathbb{R}^{H/2}$ in order to learn to convey only the most relevant information to the decoder [35]. The encoding is repeated T times which can be effectively achieved by caching it for exactly T time steps. The decoder transforms the repeated embedding time step by time step into output $h \in \mathbb{R}^{T \times H}$. h is then processed further by a temporal dense layer, where the same dense layer processes each output in the sequence to give the final reconstruction of the input, again in a pipelined fashion. A dense layer is simply implemented as a single MVM unit. The architecture aims to learn a useful embedding which captures the essence of the input signal and allows its efficient reconstruction. Based on the quality of the reconstruction the input is labelled as normal or anomalous.

The hardware model for classification can be built in a similar fashion, by considering only the encoder part of the architecture as shown in Figure 6 (b) with NL layers. The last hidden state $h_T \in \mathbb{R}^H$ of the encoder is not repeated but processed through a dense layer that reshapes it to the number of output classes and processes it through a softmax activation. Hence the classifier is fully pipelined. Given an input signal, the encoder captures variable-size input relationships into a consistent output embedding that is used for classification, given that the input is labelled. The hidden size H , number of layers NL , the portion of Bayesian LSTMs B or the hardware configuration can vary as we discuss in the optimization framework.

IV. OPTIMIZATION FRAMEWORK

In this section, we first present an overview of the proposed optimization framework. Then, the resource and latency models are introduced, which are used to accelerate the DSE.

A. Overview of Framework

An overview of the proposed framework is shown in Figure 7. Given user-defined priorities in terms of the target metric and the platform-specific hardware constraints, it is necessary to optimize both the algorithmic and hardware configuration of the RNN as well as the accelerator. Therefore, we propose an optimization framework to perform DSE under both user-defined algorithmic and hardware constraints. In our design,

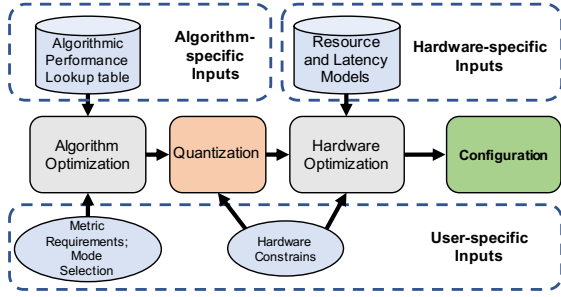


Fig. 7. Overview of the optimization framework.

the performance trade-off is decided by two categories of parameters: 1) Algorithmic architectural parameters, which include the overall network architecture \mathbf{A} : the hidden size H , the number of layers NL for encoder or decoder and the portion of Bayesian layers \mathbf{B} and 2) Hardware parameters \mathbf{R} : which consist of reuse factors R_x, R_h, R_d of processing engines. The objective of our framework is to optimize the latency and algorithmic metrics such as accuracy and quality of uncertainty prediction by exploring both \mathbf{A} and \mathbf{B} to target all three challenges mentioned in Section I.

At the start, the framework requires users to specify the hardware constraints, metric requirements and the focus mode. The main hardware constraint is the number of available DSPs on the target hardware platform. The optimization mode is selected to minimize or maximize the chosen objective through greedy optimization with respect to algorithmic and hardware configurations. At first, the algorithmic optimization is conducted with respect to a previously built lookup table consisting of algorithm-benchmarked architectures. Following algorithm optimization and potential re-training, the networks are quantized depending on hardware constraints. In this work we consider 16-bit fixed-point quantization. Next, the parameters \mathbf{R} of a hardware configuration are optimized with respect to a hardware model. The hardware model is used to estimate the resource consumption or latency given the available configurations. Based on the determined hardware parameters, the latency is estimated given a performance lookup table for various BNN configurations with different \mathbf{A} . At the end, the configurations which do not meet the minimal requirements are filtered resulting in a final configuration.

B. Resource Model

In this paper, we mainly consider the resource consumption in terms of DSPs as DSP_{design} , which represent the resource bottleneck, while being limited by the total available DSPs as DSP_{total} . The number of DSPs for a given LSTM layer DSP_i and the complete design using 16-bit representation, except c_{t-1}^i which is represented in 32-bit, is shown as:

$$DSP_i = \frac{4 \times I_i \times H_i}{R_x} + \frac{4 \times H_i^2}{R_h} + 4 \times H_i$$

$$DSP_{design} = \sum_{i=1}^L DSP_i + DSP_d \leq DSP_{total}$$

I_i, H_i and O represent the input, hidden state and output dimensionality for layer i . L is $2NL$ if considering autoencoder or NL if considering the classifier. The factor of 4 means there are 4 MVMs for input and 4 for the hidden state in a single LSTM layer. The $f_t \times c_{t-1}$ in the LSTM tail needs two Xilinx DSPs to construct one multiplier unit, thus the LSTM tail unit consumes $4 \times H_i$ DSPs. The DSP_d is the DSP consumption for the final dense layer which equals $\frac{H_L \times O \times T}{R_d}$ if considering autoencoder or $\frac{H_L \times O}{R_d}$ if considering the classifier. The R_x, R_h and R_d represent the reuse factors for the MVMs processing the input, hidden state or the final dense layer respectively. T is the time step or sequence length. In the design space exploration, additional 5% of the DSP_{total} was added since we found that the HLS tool often optimizes the DSP usage by replacing the multipliers using other simpler logic when possible.

The trade-off between latency, throughput and FPGA resource usage is determined by the parallelism of the calculation. This work adopts the reuse factor used in [32] to fine tune the parallelism, which is configured to set the number of times a multiplier is used in the computation of a module. With a reuse factor of $R, \frac{1}{R}$ fewer multipliers and computation are performed. With a higher reuse factor, the DSP resource usage can be reduced, however, the latency of processing MVMs will increase. The reuse factors should be carefully chosen so that the design can fit into the targeted FPGA chip while keeping the latency as small as possible.

C. Latency Model

The individual layer latency Lat_i and end-to-end latency Lat_{design} , which is dominated solely by the recurrent cells, are modeled as:

$$II = \max_{i=1, \dots, L} II_i \quad Lat_i = II \times T + (II_i - II)$$

$$Lat_{design} = II \times T + (II_i - II) \times NL$$

where II is the initiation interval of the single time step loop, T is the time sequence length, II_i is the iteration latency. The II is the number of clock cycles before a unit can accept new inputs and is generally the most critical performance metric in many systems [36]. In the proposed pipelined design, the processing of the cascaded LSTM layers can be overlapped. For example, the second layer $i=2$ does not need to wait for the whole sequence of hidden states to begin computation. Just a single hidden state from the former LSTM layer is sufficient for the computation in the next LSTM layer. Furthermore, since the II of a model is decided by the largest individual layer i , the II of the cascaded layers is set to be the same to achieve the best hardware resource efficiency. Thus, the total latency of a design with NL cascaded LSTM layers is given as Lat_{design} . It has to be noted that the decoder in the autoencoder can only be started after the encoder calculation is completed, since only the last time step hidden state is returned in the last layer of the encoder. Thus, the latency of an autoencoder with $2NL$ LSTM layers, NL for encoder and NL for decoder, is simply $Lat_{design} \times 2$.

As shown in the equations above, the Lat_{design} is dominated by the II when T is fixed. This work achieves the optimal II via identifying the proper reuse factors under the hardware resource limitations, as shown in Section IV-B to achieve the lowest II and end-to-end latency.

V. EXPERIMENTS

In this section we first review the general experimental setup followed by algorithmic DSE and hardware performance comparison with respect to different hardware platforms.

The experiments were performed on *ECG5000* dataset [37], that contains 5000 samples split into a training set of only 500 samples and a test set of 4500 samples. By default, the dataset has 4 classes: 1 normal and 3 anomalous. Each ECG has $T=140$ and it was preprocessed such that each sample was zero mean and unit variance centered. It is a challenging dataset mainly due to its small size and class imbalance, which can be associated with anomaly detection or classification tasks. For both tasks we trained various recurrent architectures with respect to 1000 epochs, batch size 64, gradient clipping set as 3.0 and weight decay set as 0.0001 to provide regularization.

Next, we present the algorithmic DSE that represents an algorithmic optimization and population of the lookup table in the proposed framework. It is followed by a hardware optimization and performance comparison. Arrows in Tables and Figures symbolize desired trends and bold values represent the best score.

A. Algorithmic Optimization

1) *Anomaly Detection*: For anomaly detection, we split the data into normal and anomalous samples. We appended anomalous cases from the train set to the test set and we trained the autoencoder from Section III-C only with respect to normal data to be able to recreate it. We measured the wellness of the fit with respect to root-mean squared error. Based on the fit, we analyzed the models with respect to receiver operating characteristic and the respective area under the curve (AUC), average precision (AP) and accuracy (ACC) at the cutoff point that maximizes true positive rate against false positive rate in detecting anomalies. We considered autoencoders with $A: H=\{8, 16, 24, 32\}$ and $NL=\{1, 2\}$ LSTMs in encoder and decoder with dropout B benchmarked at every position and combination.

The results with respect to the DSE in floating-point and $S=30$ are shown in Figure 8. It can be seen that the Pareto optimal architectures were at least partially Bayesian. The best model was with $H=16$, $NL=2$ layers in encoder or decoder and dropout applied both in the encoder and decoder $B=YNYN$. Y/N stands for MCD enabled or disabled respectively for that layer. The model achieved fine algorithmic performance with AUC, AP and ACC all approaching 1.

2) *Classification*: For classification we evaluated the models trained on all four classes with respect to ACC, macro AP and average recall (AR), since the dataset is severely unbalanced. Additionally, we considered uncertainty estimation qualities with respect to sequences of random Gaussian noise for which we measured the predictive entropy

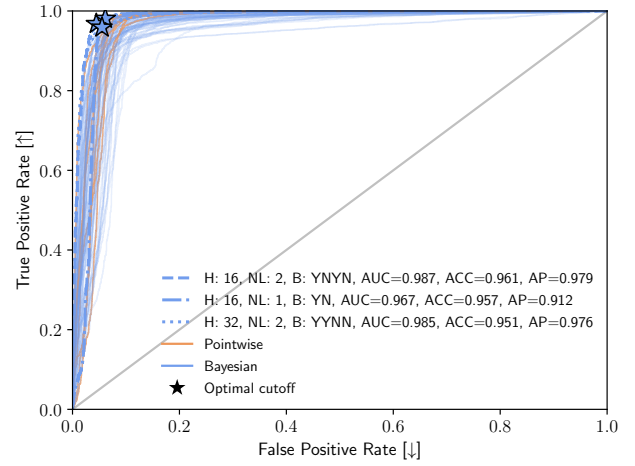


Fig. 8. Receiver operating characteristic on the ECG test set with respect to Bayesian and pointwise (without any Bayesian layers) autoencoders in anomaly detection. H is hidden size, NL is number of layers in encoder or decoder, B symbolizes Bayesian in the given layer enabled (Y) or disabled (N). AUC is area under the curve, ACC is accuracy and AP is average precision.

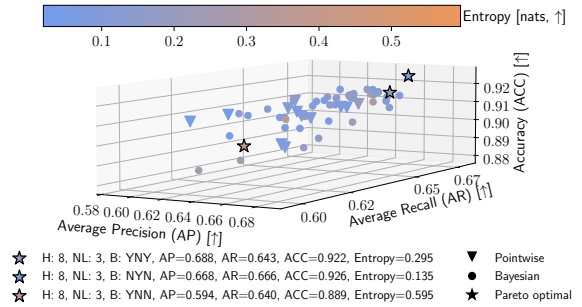


Fig. 9. Classification performance on the ECG test set with respect to Bayesian and pointwise (without any Bayesian layers) recurrent nets. H is hidden size, NL is number of layers, B symbolizes Bayesian in the given layer enabled (Y) or disabled (N).

in nats. We considered classifiers as per Section III-C with $A: H=\{8, 16, 32, 64\}$ and $NL=\{1, 2, 3\}$ LSTMs in the encoder with dropout B benchmarked at every position and combination. Figure 9 summarizes the performance of the considered architectures while running in floating-point and $S=30$. Similarly to anomaly detection, the best performing architectures were again at least partially Bayesian. In this case the optimal architecture was identified with $H=8$, $NL=3$ layers overall with dropout applied such that $B=YNY$. The model achieved high accuracy and precision.

3) *Sampling*: As discussed in Section II-B, the software performance of Bayesian architectures depends on the number of feedforward samples S that also affects the overall runtime. Figures 10 (a,b) demonstrate the relationship between the software metrics for both anomaly detection (a) and classification (b) with respect to the best architectures for each task. It can be seen that an S larger than 30 results in diminishing returns.

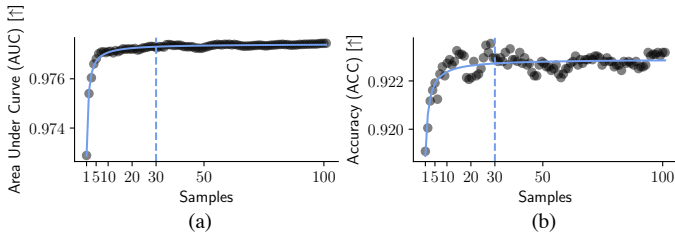


Fig. 10. Software performance change for anomaly detection (a) and classification (b) with increasing number of samples S from 1, 30 to 100 samples.

TABLE I
COMPARISON OF FLOATING-POINT AND QUANTIZED BEST MODEL FOR ANOMALY DETECTION.

Representation Precision	Accuracy [↑]	Average Precision [↑]	Area under Curve [↑]
Floating-point	0.95 ± 0.01	0.96 ± 0.02	0.98 ± 0.01
Fixed-point	0.95 ± 0.01	0.97 ± 0.01	0.98 ± 0.01

B. Quantization

The next step given the outlined framework in Section IV is quantization. In Tables I and II the performance of the best floating-point models for both anomaly detection and classification is compared with respect to the 16-bit fixed-point quantization when $S=30$. The results were collected with respect to retraining the best architectures three times to obtain the mean and the standard deviation for comparison. The results demonstrate that the chosen fixed-point quantization scheme and configuration preserves high accuracy and uncertainty estimation, seen in entropy, of both best models.

Next we discuss the hardware optimization and performance comparison with respect to different hardware platforms.

C. Performance Comparison with GPU and CPU

We implemented the proposed design from Section III on Xilinx ZC706, which consists of a XC7Z045 FPGA and a dual ARM Cortex-A9 processor. 1 GB DDR3 RAM is installed on the platform as the off-chip memory. The Xilinx Vivado HLS 2019.2 tool was used for synthesis. The FPGA power is reported by the Xilinx Vivado tool. The design frequency was 100MHz. The reuse factors were determined through the optimization framework and set as $R_x=16$ and $R_h=5$ when $H=16$ and $R_x=12$ and $R_h=1$ when $H=8$ given the FPGA. The R_d is set to R_x for autoencoder and is set to 1 for classifier to achieve low latency.

Table III shows the resource utilization for the designs of the optimal architectures for anomaly detection and classification on the FPGA. It can be seen that both Bayesian RNN models can fit the FPGA and almost all of the FPGA's DSPs or LUTs were utilized with 758 and 898 DSPs used for anomaly detection or classification architectures. At the same time, the estimated DSP consumption for these architectures with the model presented in Section IV-B were 754 and 915 re-

TABLE II
COMPARISON OF FLOATING-POINT AND QUANTIZED BEST MODEL FOR CLASSIFICATION.

Representation Precision	Accuracy [↑]	Average Precision [↑]	Average Recall [↑]	Entropy [nats,↑]
Floating-point	0.92 ± 0.0	0.68 ± 0.01	0.65 ± 0.01	0.36 ± 0.14
Fixed-point	0.92 ± 0.0	0.68 ± 0.01	0.65 ± 0.02	0.38 ± 0.11

spectively, demonstrating fine accuracy of the resource model, which is more than 98% accurate.

To demonstrate the advantage of our FPGA-based accelerator compared with CPU and GPU implementations, we evaluated the best RNN models found in anomaly detection and classification tasks on the FPGA, a TITAN X Pascal with 3,840 CUDA cores clocked at 1.4 GHz and an Intel Xeon E5-2680 v2 CPU with 8 CPU cores clocked at 2.4 GHz with respect to latency, power consumption and energy consumption per sample. We measured the power of the CPU using a power meter. The power of GPU was reported by an Nvidia toolkit. PyTorch 1.8 [38] is used for both CPU and GPU implementations. The random number generation for CPU and GPU implementations is performed via default PyTorch calls and default pseudo-random number generators for each platform. To optimize the hardware performance on each platform, we use TensorRT and CuDNN 8.11 libraries for the GPU implementation, and MKLDNN for the CPU implementation. The number of samples was set to be $S = 30$ as indicated by Section V-A. As GPUs always show advantages in multi-batch workload, we set the batch size to be 50 and 200 on all hardware platforms for a fair comparison. This batch size is realistic in our application, considering for example multiple patients. The results are presented in Table IV. Compared with GPU implementation, our FPGA-based design was nearly 2~8 times faster and consumed 20~26 times less power. In terms of the energy consumption, which was measured by energy per sample, our design was nearly 106 times more efficient than the GPU implementation. Comparing with the CPU implementation, our FPGA-based accelerator achieved approximately 100~400 times higher energy efficiency. FPGA implementations are faster and more efficient because they are unrolled on-chip with respect to our tailor-made design. The FPGA design

TABLE III
RESOURCE UTILIZATION FOR THE BEST ARCHITECTURES.

Task		LUT	FF	BRAM	DSP
	Available	219k	437k	545	900
Anomaly $H=16, NL=2, B=YNYN$	Used [↓]	207k	218k	149	758
	Utilized [%, ↓]	94	49	13	84
Classification $H=8, NL=3, B=YNY$	Used [↓]	62k	52k	64	898
	Utilized [%, ↓]	28	11	5	99.8

TABLE IV
HARDWARE COMPARISON BETWEEN FPGA, CPU AND GPU IMPLEMENTATIONS.

Task	Batch Size	Latency [ms, ↓]			Power [W, ↓]			Energy Consumption [J/Sample, ↓]		
		FPGA	CPU	GPU	FPGA	CPU	GPU	FPGA	CPU	GPU
Anomaly $H=16, NL=2,$ $B=YNYN$	50	41.31	4011	379.81	3.44	15	69	0.005	2.01	0.53
	200	165.24	5964	402.76				0.019	2.98	0.56
Classification $H=8, NL=3,$ $B=YNY$	50	25.23	3690	245.14	2.47	16	65	0.002	1.97	0.36
	200	100.92	4981	256.98				0.008	2.66	0.38

TABLE V
OPTIMIZATION FOR ANOMALY DETECTION.

Mode	$A : \{H, NL, B\}$	Latency [ms, ↓]			Accuracy [↑]	Average Precision [↑]	Area under Curve [↑]
		FPGA	CPU	GPU			
<i>Opt-Latency</i>	8, 1, NN	6.94	133.45	10.57	0.93	0.87	0.95
<i>Opt-Accuracy / Precision / Area under Curve</i>	16, 2, YNYN	165.24	5485	250.27	0.96	0.98	0.99

TABLE VI
OPTIMIZATION FOR CLASSIFICATION.

Mode	$A : \{H, NL, B\}$	Latency [ms, ↓]			Accuracy [↑]	Average Precision [↑]	Average Recall [↑]	Entropy [nats, ↑]
		FPGA	CPU	GPU				
<i>Opt-Latency</i>	8, 1, N	3.44	120.52	6.49	0.90	0.62	0.66	0.15
<i>Opt-Accuracy</i>	8, 3, NYN	100.92	4799	193.10	0.93	0.67	0.67	0.14
<i>Opt-Precision</i>	8, 3, YNY	100.92	4789	182.59	0.92	0.69	0.64	0.30
<i>Opt-Recall</i>	8, 2, YN	100.91	3176	123.59	0.91	0.64	0.67	0.20
<i>Opt-Entropy</i>	8, 3, YNN	100.92	4795	191.64	0.89	0.59	0.64	0.60

processes the input with batch size 1, since requests need to be processed as soon as they arrive.

Lastly, based on the latency model in Section IV-C, the estimated latencies of the two architectures with 50 batches were 42.25ms and 25.77ms respectively. Hence, the analytical prediction errors were only 2.26% and 2.13% respectively, confirming the accuracy of the latency model.

D. Optimization Framework Efficiency

To demonstrate the effectiveness of our framework on finding optimized designs under different user-defined priorities, we evaluated the proposed framework with respect to both anomaly detection and classification on the *ECG5000* dataset.

For the anomaly detection, since it is primarily a regression task, we set the optimization modes as *Opt-Latency*, *Opt-Accuracy*, *Opt-Precision* and *Opt-AUC*. If users wish to only optimize hardware performance, they would pick the configuration with the optimal latency. However, if users wish to obtain a model with minimized errors, they would maximize the accuracy. If users wish to maximize the true positive rate and minimize the false positive rate, or in general to obtain model that has high precision on a range of thresholds, they would pick the model with the highest AUC or AP respectively. The results are presented in Table V. Surprisingly,

we found that *Opt-Accuracy*, *Opt-Precision* and *Opt-AUC* generated the same model with $H=16$, $NL=2$ and MCD applied in the first and third layers. While the *Opt-Latency* simply traded-off the algorithmic performance for the smallest hidden size, $NL=1$ with no MCD using $S=1$ to achieve the lowest latency. As we can see from Table V, our FPGA-based design was still 1.4~33.2 times faster than both CPU and GPU implementations depending on different model architectures.

For the classification task, there can be up to five optimization modes, namely *Opt-Latency*, *Opt-Accuracy*, *Opt-Precision*, *Opt-Recall* and *Opt-Entropy*. In addition to the modes presented in the previous paragraph, if the users wish to minimize false negatives, e.g. diagnosing a normal condition in an anomalous ECG, they would pick the model with the highest recall. If the model has to support high uncertainty containing outlier ECG signal values, the user could pick the model with the highest entropy. Different optimization modes generated different model architectures as shown in Table VI. The highest accuracy we can achieve was 93%. Similarly, with the help of our framework, we achieved 0.68 AP, 0.67 AR and 0.60 nats entropy under different optimization modes with the speedup ranging from 1.2 to 1.9 compared to GPU implementations. Again, the *Opt-Latency* traded-off the algorithmic performance for the smallest hidden size, non-Bayesian archi-

texture with $NL=1$ and $S=1$ to target the improvement in the hardware performance. Note that, although the models with $NL=3$ or $NL=2$ LSTM layers in Table VI achieve similar latency as discussed in Section IV-C due to pipelining, their resource and energy consumption are different.

VI. CONCLUSION

This work proposes a novel high-performance FPGA-based design to accelerate Bayesian LSTM-based recurrent neural networks inferred through Monte Carlo Dropout. The presented design is sufficiently versatile to support a variety of network models with respect to different safety-critical tasks concerning real-time performance on analyzing electrocardiograms. In comparison to the GPU implementation, our FPGA-based design can achieve up to 10 times speedup with nearly 106 times higher energy efficiency. At the same time, this is the first work that is focused on accelerating Bayesian recurrent neural networks on an FPGA. Moreover, this work presents an end-to-end framework to automatically trade-off both algorithmic and hardware performance, given algorithmic requirements and hardware constraints. In future work we aim to explore co-design of custom recurrent cells and reconfigurable hardware accelerators, to obtain the most optimized configurations and hardware implementations. Additionally, we are interested in supporting a wide variety of dropout rates in hardware.

ACKNOWLEDGMENT

This work was supported in part by the United Kingdom EP-SRC under Grant EP/L016796/1, Grant EP/N031768/1, Grant EP/P010040/1, Grant EP/V028251/1 and Grant EP/S030069/1 and in part by the funds from Corerain, Maxeler, Intel, Xilinx and State key lab of Space-Ground Integrated Information Technology (SGIIT). Martin Ferienc was sponsored through a scholarship from the Institute of Communications and Connected Systems at UCL. We also thank the reviewers for insightful comments and suggestions.

REFERENCES

- [1] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 75–84.
- [2] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," *arXiv preprint arXiv:1511.05552*, 2015.
- [3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [4] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," *Advances in Neural Information Processing Systems*, vol. 29, pp. 1019–1027, 2016.
- [5] P. L. McDermott and C. K. Wikle, "Bayesian recurrent neural network models for forecasting and quantifying uncertainty in spatial-temporal data," *Entropy*, vol. 21, no. 2, p. 184, 2019.
- [6] W. Sun, A. R. Paiva, P. Xu, A. Sundaram, and R. D. Braatz, "Fault detection and identification using bayesian recurrent neural networks," *Computers & Chemical Engineering*, vol. 141, p. 106991, 2020.
- [7] M. Fortunato, C. Blundell, and O. Vinyals, "Bayesian recurrent neural networks," *arXiv preprint arXiv:1704.02798*, 2017.
- [8] J. van der Westhuizen and J. Lasenby, "Bayesian LSTMs in medicine," *arXiv preprint arXiv:1706.01242*, 2017.

- [9] N. Srivastava, E. Mansimov, and R. Salakhudinov, "Unsupervised learning of video representations using LSTMs," in *International Conference on Machine Learning*, PMLR, 2015, pp. 843–852.
- [10] B. Hou, J. Yang, P. Wang, and R. Yan, "LSTM-based auto-encoder model for ECG arrhythmias classification," *IEEE Transactions on Instrumentation and Measurement*, vol. 69, no. 4, pp. 1232–1240, 2019.
- [11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhudinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.
- [12] 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.
- [13] H. Fan, M. Ferienc, M. Rodrigues, H. Zhou, X. Niu, and W. Luk, *High-performance FPGA-based accelerator for Bayesian neural networks*, 2021. arXiv: 2105.09163.
- [14] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Computing and Applications*, vol. 32, no. 4, pp. 1109–1139, 2020.
- [15] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott, "FINN-L: library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs," in *28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2018.
- [16] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, *et al.*, "Why compete when you can work together: FPGA-ASIC integration for persistent RNNs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2019, pp. 199–207.
- [17] Z. Que, H. Nakahara, H. Fan, J. Meng, K. H. Tsoi, X. Niu, E. Nurvitadhi, and W. Luk, "A reconfigurable multithreaded accelerator for recurrent neural networks," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2020, pp. 20–28.
- [18] V. Rybalkin, C. Sudarshan, C. Weis, J. Lappas, N. Wehn, and L. Cheng, "Efficient Hardware Architectures for 1D-and MD-LSTM Networks," *Journal of Signal Processing Systems*, pp. 1–27, 2020.
- [19] V. Rybalkin and N. Wehn, "When Massive GPU Parallelism Ain't Enough: A Novel Hardware Architecture of 2D-LSTM Neural Network," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 111–121.
- [20] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, IEEE, 2017, pp. 629–634.
- [21] Z. Que, Y. Zhu, H. Fan, J. Meng, X. Niu, and W. Luk, "Mapping large LSTMs to FPGAs with weight reuse," *Journal of Signal Processing Systems*, vol. 92, no. 9, pp. 965–979, 2020.
- [22] N. Park, Y. Kim, D. Ahn, T. Kim, and J.-J. Kim, "Time-step interleaved weight reuse for LSTM neural network computing," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 13–18.
- [23] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2019, pp. 63–72.
- [24] R. Shi, J. Liu, K.-H. H. So, S. Wang, and Y. Liang, "E-LSTM: efficient inference of sparse LSTM on embedded heterogeneous system," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2019, pp. 1–6.
- [25] G. Nan, C. Wang, W. Liu, and F. Lombardi, "DC-LSTM: deep compressed LSTM with low bit-width and structured matrices," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2020, pp. 1–5.
- [26] Z. Chen, G. J. Blair, H. T. Blair, and J. Cong, "BLINK: bit-sparse LSTM inference kernel enabling efficient calcium trace extraction for neurofeedback devices," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 217–222.
- [27] X. Jia, J. Yang, R. Liu, X. Wang, S. D. Cotofana, and W. Zhao, "Efficient computation reduction in Bayesian neural networks through feature decomposition and memorization," *IEEE Transactions on Neu-*

- ral Networks and Learning Systems*, vol. 32, no. 4, pp. 1703–1712, 2020.
- [28] 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.
- [29] 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)*, IEEE, 2020, pp. 1402–1407.
- [30] Q. Wan and X. Fu, “Fast-BCNN: Massive neuron skipping in Bayesian convolutional neural networks,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 229–240.
- [31] A. F. Agarap, “Deep learning using rectified linear units (ReLU),” *arXiv preprint arXiv:1803.08375*, 2018.
- [32] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, *et al.*, “Fast inference of deep neural networks in FPGAs for particle physics,” *Journal of Instrumentation*, vol. 13, no. 07, P07027, 2018.
- [33] S. Tridgell, M. Kumm, M. Hardieck, D. Boland, D. Moss, P. Zipf, and P. H. Leong, “Unrolling ternary neural networks,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 4, pp. 1–23, 2019.
- [34] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.
- [35] D. P. Kingma and M. Welling, “Auto-encoding variational bayes,” *arXiv preprint arXiv:1312.6114*, 2013.
- [36] Xilinx, “SDSoC Profiling and Optimization Guide.”
- [37] A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley, “PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals,” *Circulation*, vol. 101, no. 23, e215–e220, 2000.
- [38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.