LUTNet: Rethinking Inference in FPGA Soft Logic

Erwei Wang, James J. Davis, Peter Y. K. Cheung and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London, London, SW7 2AZ, United Kingdom
{erwei.wang13, james.davis, p.cheung, g.constantinides}@imperial.ac.uk

***Abstract*—Research has shown that deep neural networks contain significant redundancy, and that high classification accuracies can be achieved even when weights and activations are quantised down to binary values. Network binarisation on FPGAs greatly increases area efficiency by replacing resource-hungry multipliers with lightweight XNOR gates. However, an FPGA's fundamental building block, the K-LUT, is capable of implementing far more than an XNOR: it can perform any K-input Boolean operation. Inspired by this observation, we propose LUTNet, an end-to-end hardware-software framework for the construction of area-efficient FPGA-based neural network accelerators using the native LUTs as inference operators. We demonstrate that the exploitation of LUT flexibility allows for far heavier pruning than possible in prior works, resulting in significant area savings while achieving comparable accuracy. Against the state-of-the-art binarised neural network implementation, we achieve twice the area efficiency for several standard network models when inferencing popular datasets. We also demonstrate that even greater energy efficiency improvements are obtainable.**

I. Introduction and Motivation

During inference, the most common—and expensive—computational node in a deep neural network (DNN) performs a function of the form in (1), calculating a channel output y. Each weight w_n is a constant determined during training, $\boldsymbol{x}$ a vector of N channel inputs and f an activation function such as the widely used rectified linear unit. In the extreme case where $\boldsymbol{w} \in \{-1,1\}^N$—so-called binarised neural networks (BNNs)—the multiplications become cheap or free to implement. When time-multiplexed, multipliers become XNOR gates. When unrolled, they can be further simplified into buffers and inverters, all of which are usually subsumed into the downstream adder logic. Also beneficial for BNNs is the ability to use a population count (popcount) for the summation: an operation that consumes half the LUTs of the otherwise-throughput-optimal balanced adder tree [1].

$$y = f\left(\sum_{n=1}^{N} w_n x_n\right) \tag{1}$$

No matter how simple these multiplications become, however, all of the products still need to be summed. In modern networks, N commonly reaches numbers in the thousands [2], [3]. To tackle this, we propose the replacement of (1) with the specifically FPGA-inspired function (2), wherein the activation function is unchanged but each product is replaced with an *arbitrary* term-specific Boolean function $g_n : \{-1,1\}^K \rightarrow \{-1,1\}$. The input to this function is a vector $\tilde{\boldsymbol{x}}^{(n)}$ whose elements are any K components of the original input vector $\boldsymbol{x}$, *i.e.* $\tilde{\boldsymbol{x}}^{(n)} = \boldsymbol{S}_n \boldsymbol{x}$ for some binary selection matrix $\boldsymbol{S}_n \in \{0,1\}^{K \times N}$ with $\|\boldsymbol{S}_n\|_\infty = 1$. Since its inputs and outputs are binary, each g_n maps directly to a single K-LUT. BNNs are a special case of this function: they are recoverable for $K = 1$ and $\tilde{N} = N$, with $\boldsymbol{S}_n$ being the row vector with the n^{th} element equal to one and all others zero. An example of the resultant architectural transformation—excluding blocks for f, which are common to both approaches—is given in Fig. 1.

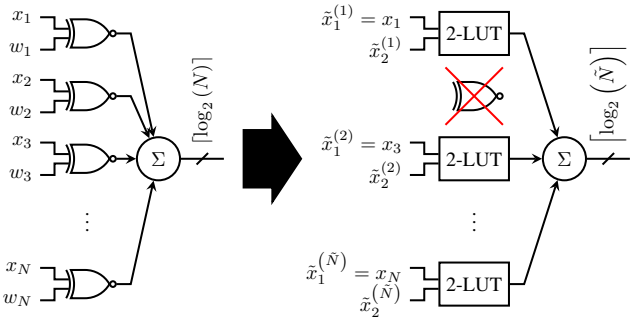

Fig. 1. BNN to LUTNet architectural transformation for a single channel, mirroring the replacement of (1) with (2). Activation function blocks are not shown, but follow the adders. N lookup tables (here, 2-LUTs) substitute N XNOR gates. $\tilde{N} \ll N$ is achieved via pre-substitution pruning, represented by the removal—*i.e.* lack of LUT substitution—of the second XNOR gate. LUT inputs $\tilde{x}_1^{(n)}$ $\forall n$ are connected to preserve the pruned BNN's structure. LUTNet's weights are encoded in its LUT masks, thus do not appear as inputs.

$$y = f\left(\sum_{n=1}^{\tilde{N}} g_n\left(\tilde{\boldsymbol{x}}^{(n)}\right)\right) \tag{2}$$

Notice that, while in (1) each element of $\boldsymbol{x}$ only participates in a single summation term, in (2) each can participate in many terms. The intuition here is that inputs can be arranged such that $\tilde{N} \ll N$ for comparable accuracy via network pruning, dramatically reducing the sizes of the required popcount trees. Our experiments demonstrate that this is indeed the case.

Our aim in proposing this inference node function is to play to the strengths of FPGA soft logic. While a LUT is capable of performing an arbitrary *nonlinear Boolean* function, traditional DNNs are based around *near-linear high-precision* functions: almost the exact opposite of the architecture's forte. Innovations such as BNNs have addressed one side of this

2576-2621/19/$31.00 ©2019 IEEE
DOI 10.1109/FCCM.2019.00014

weakness, by reducing precision [4]; we address both by also embracing the nonlinearity of the LUT.

In this paper, we make the following novel contributions:

- We introduce LUTNet, the first neural network architecture featuring K-LUTs as inference operators. Since each K-LUT is capable of performing an arbitrary Boolean operation on up to K binary inputs, LUTNet's logic density is much greater than that of BNNs.
- We propose a training regime resulting in the conversion of a BNN architecture from a dense array of simple XNOR gates into a sparse network of arbitrary K-input functions directly mappable onto K-LUTs.
- We empirically demonstrate the effects of LUTNet's increased logic density on area efficiency and accuracy. We also experimentally explore the associated energy and training efficiency impacts. Our results for 4-LUT-based inference operators reveal area compression of 2.08× and 1.90× for the CNV network classifying the CIFAR-10 dataset and AlexNet classifying ImageNet, respectively, against an unrolled and losslessly pruned implementation of ReBNet [5], the state-of-the-art BNN, with accuracy bounded within ±0.300 percentage points (pp).

II. Related Work

The authors of early BNN publications, such as BinaryConnect [6] and BinaryNet [7], proposed network training with binary weights and activations (channel inputs and outputs) used for forward propagation. High-precision formats—most commonly IEEE-754 single-precision floating point, used to approximate reals $\mathbb{R}$—are always used for backward propagation; this is essential in order for stochastic gradient descent to work well [6]. Tang *et al.* showed that training from scratch with binarised forward propagation is significantly slower than through the consistent use of high-precision data, however; learning rates some 100× lower are required than in the all-real case [8]. Furthermore, binary forward propagation results in the majority of real-valued weights being close to either -1 or 1, while a spread across $[-1, 1]$ is required to facilitate fine-grained pruning [9].

Use of fine-grained pruning effectively adds zero to the set of possible weight values, resulting in a ternary representation. Ternarisation has been shown by the authors of many works to deliver significantly higher accuracy than yielded through binarisation [10], [11], [12]. Pruning also promotes regularisation, reducing overfitting [13]. The latter is particularly relevant to this work since the use of K-LUTs as inference operators greatly increases potential network complexity.

In order to promote pruning, Han *et al.* proposed training with the l_2 sparsification regulariser in (3) [9]. During backward propagation, the value of Ω influences training loss, inducing weights carrying low significance to descend towards zero. λ, L and C are the regularisation factor, number of layers and number of channels per layer, respectively. $\hat{\boldsymbol{w}}^{(l,c)}$ denotes the real-valued weight vector of layer l's channel c.

$$\Omega = \lambda \sqrt{\sum_{l=1}^{L} \sum_{c=1}^{C} \left(\hat{\boldsymbol{w}}^{(l,c)} \right)^2} \tag{3}$$

Improving upon BinaryNet's data representation, Rastegari *et al.*'s BWN features layer-wise trainable scaling factors $\boldsymbol{\alpha}$ used in order to increase BNN expressiveness [14]. During training, each $\alpha_l \in \mathbb{R}$ assumes the mean value of layer l's weights. When inferencing, this is multiplied with the layer's popcount results, compensating for some of the information lost to binarisation and increasing accuracy.

Tang *et al.* [8] and the authors of ABC-Net [15] and ReBNet [5] demonstrated the alleviation of information loss from binarisation through the approximation of real-valued weights as linear combinations of multiple binary values. This is achieved via *residual binarisation*, a scheme in which each bit is the binarised residual error of its predecessor. Each bit b is associated with a trainable scaling factor $\gamma_b \in \mathbb{R}$, representing its relative importance. When quantising, each weight $\hat{w} \in \mathbb{R}$ is approximated as B binary weights $w_b = \text{sign}(\epsilon_b)$, as shown in (4), wherein ϵ_b is the b^{th} bit's residual error. During training, each γ_b is updated to minimise the total error. While accuracy was found to be positively correlated with B, diminishing returns were seen; little improvement was observed for $B > 2$.

$$\begin{aligned} \hat{w} &= \sum_{b=1}^{B} \gamma_b \; w_b \\ \epsilon_b &= \epsilon_{b-1} - \gamma_{b-1} \; \text{sign}(\epsilon_{b-1}) \end{aligned} \tag{4}$$

The aforementioned proposals are complementary to our approach, thus we embrace all of them. Through the use of high-precision training, fine-grained pruning, layer-wise scaling factors and residual binarisation, LUTNet achieves state-of-the-art accuracy. None of these lies at the heart of our proposal, however, and we do not consider their combination to represent novelty. Our novel use of K-LUTs allows us to reach such levels of performance significantly more cheaply than previously reported in the literature.

III. Network Construction and Training

LUTNet's initialisation comprises three successive stages: training, pruning and *"logic expansion"* (XNOR to K-LUT conversion), with each of the latter two including a retraining phase. These are shown enclosed within a dashed box in Fig. 2. All three phases were implemented with TensorFlow. While our training and pruning stages are fairly standard, logic expansion encompasses the key novelty of our approach.

A. Training

In order to both expedite learning and facilitate later pruning, our first step is to train the chosen network model using high-precision data during both forward and backward propagation. Layer-wise scaling factors α are learnt during this stage along with weights, and sparsification is induced through the use of the l_2 regulariser in (3) with $\lambda = 5 \times 10^{-7}$ as suggested by Tang *et al.* [8].

B. Pruning

Following high-precision training, fine-grained pruning is conducted through the application of threshold θ on each weight $\hat{w}$, as shown in (5). The higher the value of θ, the more weights are pruned away, exposing a continuum between area occupancy and accuracy.

$$\hat{w} \leftarrow \begin{cases} \hat{w} & \text{if } |\hat{w}| > \theta \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Once pruned, the network is binarised following the scheme shown in (4), after which it is retrained in order to recover some of the induced accuracy loss. Due to the diminishing returns previously found when applying residual binarisation [8], [15], [5], we used $B = 2$ (two-level binarisation) consistently.

C. Logic Expansion

At this point, we have obtained a residual-binarised ternary neural network with non-zero-weighted operators implemented as XNORs. It is from here that we depart from the standard BNN approach. Each XNOR gate is replaced with a K-LUT, whose first input $\tilde{x}_1^{(n)}$ is assigned to preserve the original connection, thereby retaining the pruned BNN's structure. The $K-1$ subsequent inputs to each LUT are then randomly selected from channel inputs within the same convolutional window as $\tilde{x}_1^{(n)}$, ensuring that the window shape remains unchanged. We additionally constrain their selection such that each channel input is not multiply connected to the same LUT.

The form of the inference function proposed in (2) is defined on the binary domain $\{-1,1\}^{\tilde{N}}$. In common with quantisation-inspired networks, such as BNNs, this causes difficulty for training algorithms designed to operate on real vectors $\mathbb{R}^{\tilde{N}}$, specifically in the backward propagation of derivatives. Our approach to this problem is to define an *interpolating extension* of the function $g_n : \{-1,1\}^K \to \{-1,1\}$, *i.e.* a function $\hat{g}_n : \mathbb{R}^K \to \mathbb{R}$ such that $\hat{g}_n\left(\tilde{\boldsymbol{x}}^{(n)}\right) = g_n\left(\tilde{\boldsymbol{x}}^{(n)}\right)$ for every $\tilde{\boldsymbol{x}}^{(n)}$ in the domain of g_n. There are many such functions. Of them, we prefer those that are as smooth as possible, allowing training optimisation methods to perform well, and that form a good interpolation in the sense that, if g_n remains constant when a Boolean input flips, so does $\hat{g}_n$. A natural choice for the extension is a Lagrange interpolating polynomial, leading to the form we use in (6).

$$\hat{g}_n\left(\tilde{\boldsymbol{x}}^{(n)}\right) = \sum_{\boldsymbol{d}\in\{-1,1\}^K} \left(\hat{c}_{\boldsymbol{d}} \prod_{k=1}^{K} \left(\tilde{x}_k^{(n)} - d_k \right) \right) \tag{6}$$

This expands as shown in (7) for $K \in \mathbb{N}_{>0}$, with each polynomial comprising 2^K trainable parameters $\hat{\boldsymbol{c}}$.

$$\hat{g}_n\left(\tilde{\boldsymbol{x}}^{(n)}\right) = \begin{cases} \hat{c}_{(-1)}\left(\tilde{x}_1^{(n)}+1\right) + \hat{c}_{(1)}\left(\tilde{x}_1^{(n)}-1\right) & \text{if } K=1 \\ \hat{c}_{(-1,-1)}\left(\tilde{x}_1^{(n)}+1\right)\left(\tilde{x}_2^{(n)}+1\right) & \\ \quad + \hat{c}_{(-1,1)}\left(\tilde{x}_1^{(n)}+1\right)\left(\tilde{x}_2^{(n)}-1\right) & \text{if } K=2 \\ \quad + \hat{c}_{(1,-1)}\left(\tilde{x}_1^{(n)}-1\right)\left(\tilde{x}_2^{(n)}+1\right) & \\ \quad + \hat{c}_{(1,1)}\left(\tilde{x}_1^{(n)}-1\right)\left(\tilde{x}_2^{(n)}-1\right) & \\ \ldots & \ldots \end{cases} \tag{7}$$

Since connections are effectively remade from an unpruned BNN (Section III-A), it makes sense to use those channel inputs' original weights as a starting point for retraining. For each LUT, this is done by solving (8) as shown in (9) for all $\hat{c}_{\boldsymbol{d}}$, wherein p represents the set of indices of reconnected channel inputs that were previously removed via pruning (Section III-B). This initialisation approach was motivated by the idea that the additional flexibility of the LUTs can be used to compensate for the pruned parts of the network.

$$\hat{g}_n\left(\tilde{\boldsymbol{x}}^{(n)}\right) = \hat{w}_1 \tilde{x}_1^{(n)} + \sum_{r\in p} \hat{w}_r \tilde{x}_r^{(n)} \tag{8}$$

$$\hat{c}_{\boldsymbol{d}} = \hat{w}_1 + \sum_{r\in p} d_r \hat{w}_r \tag{9}$$

Once all $\hat{g}_n$ are initialised, our second and final retraining phase is conducted, whereafter the binarised training parameters $c_{\boldsymbol{d}} = \text{sign}(\hat{c}_{\boldsymbol{d}})$ can be directly interpreted as the configuration mask of each K-LUT.

We elected to follow the network initialisation procedure detailed above rather than training with K-LUTs from scratch due to the exponential relationship between K and the number of trainable parameters $\hat{\boldsymbol{c}}$. Training these from the outset, particularly prior to network pruning, would cause both slow convergence and likely overfitting due to the large numbers of local minima in the search space. High-precision training followed by pruning not only ensures fast convergence, it also brings the starting point of K-LUT learning closer to global minima, reducing the likelihood of overfitting.

IV. Network Implementation

A representation of the overall LUTNet software training and hardware implementation flow is shown in Fig. 2. As input, the user provides the desired network model, training dataset, activation precision and the required pruning level to our TensorFlow-based training software, which performs training and pruning. Logic expansion is then performed on the chosen layers—also supplied as input—to construct the LUTNet architecture.

We chose to target Xilinx parts for this work, for which two parallel synthesis flows are required in order to convert the trained network into RTL. For ease of design and modification,

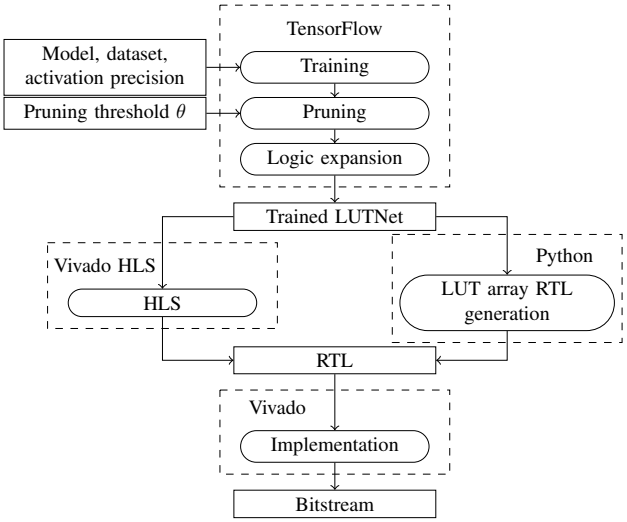

Fig. 2. LUTNet's fully automated training and FPGA implementation flow.

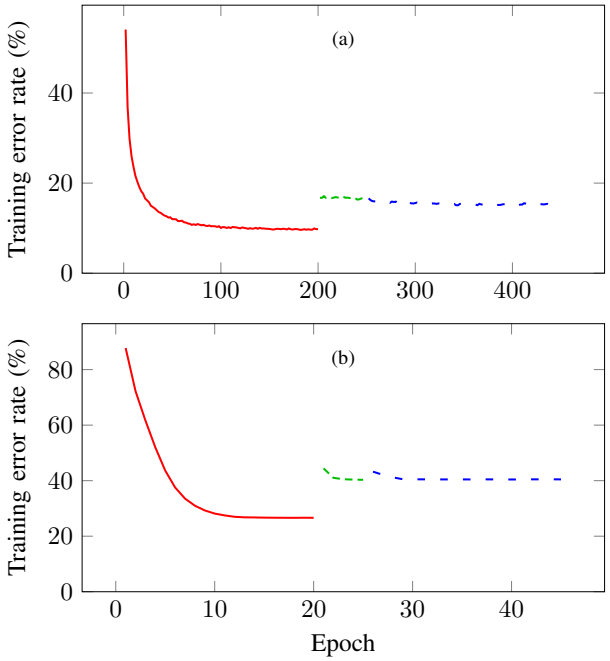

Fig. 3. Training losses for (a) the CNV network classifying the CIFAR-10 dataset and (b) AlexNet classifying ImageNet. Curves represent high-precision training (——), high-precision post-pruning retraining (- - - -) and post-logic expansion retraining with binarised forward propagation (- -).

all hardware apart from the inference K-LUTs is generated from C templates with Vivado HLS. LUT array generation is outsourced to a custom RTL generator written in Python, the output of which is combined with that from Vivado HLS after completion. Vivado is then used for implementation.

A separate LUT array generator is required because, as a general-purpose C-to-RTL synthesis tool, Vivado HLS compulsorily performs code transformations and optimisations for the synthesis of efficient RTL. Given that LUT configurations are already learnt during training, it is unnecessary—and extremely time-consuming—for such optimisation to be performed on this logic at the C level. Optimisation of RTL LUT arrays at the netlist level during synthesis with Vivado is a lot more efficient, typically taking a few hours—rather than days or weeks—to complete for large designs.

V. Evaluation

A. Benchmarks

For evaluation, we implemented end-to-end dataflow engines for the DNN models shown in Table I, using them to classify the listed datasets. Our primary baseline was the state-of-the-art BNN architecture, ReBNet [5]. All hardware implementations targetted the Xilinx Kintex UltraScale XCKU115 FPGA and met timing at 200 MHz.

In order to demonstrate the capabilities of specialised LUTs, we unrolled a subset of each network such that each node within that subset mapped to a distinct compute unit. We chose to unroll by layer, unrolling as many layers as the target device could accommodate and implementing those following the LUTNet approach. Those selected for unrolling are marked in bold in Table I. For fairness of comparison, BNN architectures (chiefly ReBNet) used as baselines had the same layers unrolled, and fine-grained pruning was performed identically to that carried out for LUTNet on those layers. The remaining layers were left time-multiplexed, with identical folding factors to those used for ReBNet's evaluation.

B. Training Particulars

For our simpler datasets (MNIST, SVHN and CIFAR-10), we performed the training, post-pruning retraining and post-logic expansion retraining described in Section III for 200, 50 and 200 epochs, respectively. For the more complex ImageNet dataset, these were performed for 20, 5 and 20 epochs instead. These periods were selected from our observations during training, the loss curves for which are shown in Fig. 3a, demonstrating saturation at or before these epochs. Non-LUTNet implementations were identically trained, but the logic expansion phase (Section III-C) was not performed. All training phases were executed in TensorFlow and accelerated using four Nvidia GTX 1080 Ti GPUs.

C. Area Efficiency

When evaluating our implementations, we were primarily interested in *logic density*, which we define as the number of LUTs required to construct a network able to achieve a particular test accuracy for a given dataset. The fewer LUTs needed to reach the same accuracy, the higher the density and thus the more efficient the implementation.

Fig. 4 shows the achieved whole-network area *vs* test accuracy points for ReBNet and LUTNet implementations, each pruned to various densities (proportion of remaining pre-pruning parameters) via the tuning of pruning threshold θ, for CNV classifying CIFAR-10. Each point marks the mean of five

TABLE I

NETWORK ARCHITECTURES FOR EVALUATED BENCHMARKS. $\text{CONV}_{x,\,y,\,z}$ DENOTES A CONVOLUTIONAL LAYER WITH x OUTPUTS, KERNEL SIZE $y \times y$ AND STRIDE z. FCONN_x IS A FULLY CONNECTED LAYER WITH x OUTPUTS. MAXPOOL_x IS AN $x \times x$ MAXIMUM-POOLING LAYER, AND BATCHNORM AND SOFTMAX ARE BATCH NORMALISATION AND NORMALISED EXPONENTIAL LAYERS, RESPECTIVELY. LAYERS IN BOLD WERE FULLY UNROLLED AND, FOR LUTNET, FEATURE K-LUT INFERENCE OPERATORS.

Dataset	Model	Network architecture
MNIST	LFC	FConn_{256}, BatchNorm, $\mathbf{FConn_{256}}$, BatchNorm, $\mathbf{FConn_{256}}$, BatchNorm, $\mathbf{FConn_{256}}$, BatchNorm, $\mathbf{FConn_{10}}$, BatchNorm, SoftMax
SVHN & CIFAR-10	CNV	$\text{Conv}_{64,\,3,\,1}$, BatchNorm, $\text{Conv}_{64,\,3,\,1}$, BatchNorm, MaxPool_2, $\text{Conv}_{128,\,3,\,1}$, BatchNorm, $\text{Conv}_{128,\,3,\,1}$, BatchNorm, MaxPool_2, $\text{Conv}_{256,\,3,\,1}$, BatchNorm, $\mathbf{Conv_{256,\,3,\,1}}$, BatchNorm, FConn_{512}, BatchNorm, FConn_{512}, BatchNorm, FConn_{10}, BatchNorm, SoftMax
ImageNet	AlexNet	$\text{Conv}_{96,\,11,\,4}$, BatchNorm, MaxPool_3, $\text{Conv}_{256,\,5,\,1}$, BatchNorm, MaxPool_3, $\text{Conv}_{384,\,3,\,1}$, BatchNorm, $\text{Conv}_{384,\,3,\,1}$, BatchNorm, $\mathbf{Conv_{256,\,3,\,1}}$, BatchNorm, MaxPool_3, FConn_{4096}, BatchNorm, FConn_{4096}, BatchNorm, FConn_{1000}, BatchNorm, SoftMax

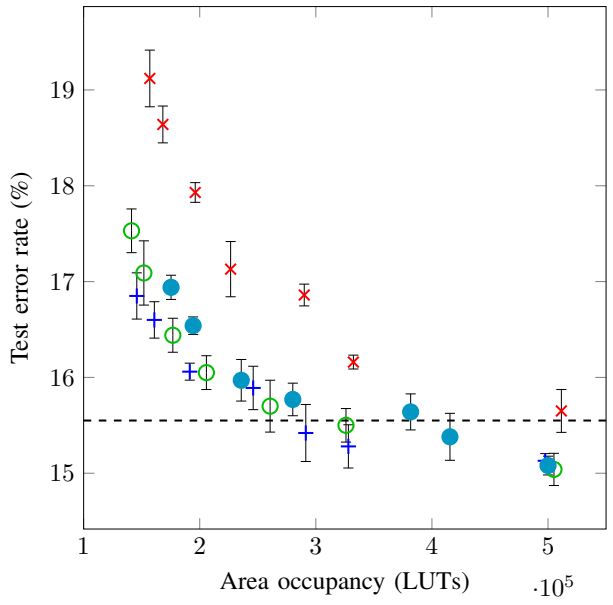

Fig. 4. Area-accuracy tradeoff for pruned ReBNet [5] (×), 2-LUTNet (○), 4-LUTNet (+) and 6-LUTNet (●) with the CNV network and CIFAR-10 dataset. Each point is representative of a distinct pruning threshold. The dashed line shows the baseline accuracy for unpruned ReBNet (660196 LUTs).

independent training runs, with an error bar indicating its 90% confidence interval. LUTNet implementations used 2-, 4- and 6-LUT inference operators. For reference, the mean test error rate of ReBNet without pruning—again averaged over five training runs—is also shown. From this data, one can clearly observe that while the error rate increases as more aggressive pruning is applied, LUTNet demonstrates greater robustness to that pruning than ReBNet through its increased logic density. That several LUTNet points achieve greater test accuracy than the unpruned baseline speaks to LUTNet's increased expressiveness. For example, despite having a significantly lower (2.27×) area requirement, our 91.1%-pruned 4-LUTNet implementation achieved an accuracy 0.590 pp above that of the ReBNet implementation without pruning.

It is interesting to note from Fig. 4 that 6-LUTNet implementations tended to achieve lower logic densities than those of 4- and sometimes even 2-LUTNet. To understand why this is the case, we must consider area and accuracy separately.

Fig. 5a shows the test accuracies of the same implementations—ReBNet, 2-, 4- and 6-LUTNet—for the same network and dataset—CNV and CIFAR-10—pruned to two densities: 4.02% and 11.3%. These densities were selected for comparison since they represent a wide spread over those found to achieve accuracies reasonably close (±2.00 pp) to ReBNet when unpruned. Of particular pertinence is the difference in accuracy spread between the two: those at 11.3% are much tighter than their 4.02% parallels. These diminishing accuracy returns when adding LUT inputs at higher densities point to complexity saturation.

Turning now to area, Fig. 5b shows the LUT requirements of the same implementations. While K-LUTNet designs for any K with equal density contain the same number of logical LUTs, this does not mean that they consume the same number of physical LUTs. The LUTs actually present in our target device are 6-LUTs, each capable of implementing either a single logical 6-LUT or two logical K-LUTs with at least five (for 5-LUTs), three (4-LUTs) or one (3-LUTs) shared inputs. 1- and 2-LUTs are not required to share any inputs; two of these can always be packed together. For 2- and 4-LUTNet, in which each inference operator uses fewer than five inputs, Vivado can often (for 4-LUTNet) or always (2-LUTNet) pack two logical K-LUTs into each physical 6-LUT, resulting in high logic density. Training-induced simplifications, *e.g.* inputs treated as don't-cares that are removed during synthesis, also lead to higher probabilities of additional packing when smaller logical LUTs are used. These optimisation phenomena are rarely seen for 6-LUTNet, hence its significantly higher area requirements at equal density.

When moving from 4- to 6-LUTs at the higher density, despite the >20% increase in physical LUTs, no accuracy benefit was obtained. In fact, 6-LUTNet's accuracy actually fell ∼0.1 pp below that of 4-LUTNet's as a result of overfitting. Due to this, as was shown in Fig. 4, 4-LUTNet almost always achieves a better area-accuracy tradeoff than 6-LUTNet.

As was noted in Section V-A, we also benchmarked LUTNet on other popular datasets and models: MNIST (on LFC), SVHN (on CNV) and ImageNet (on AlexNet). Fig. 6 shows

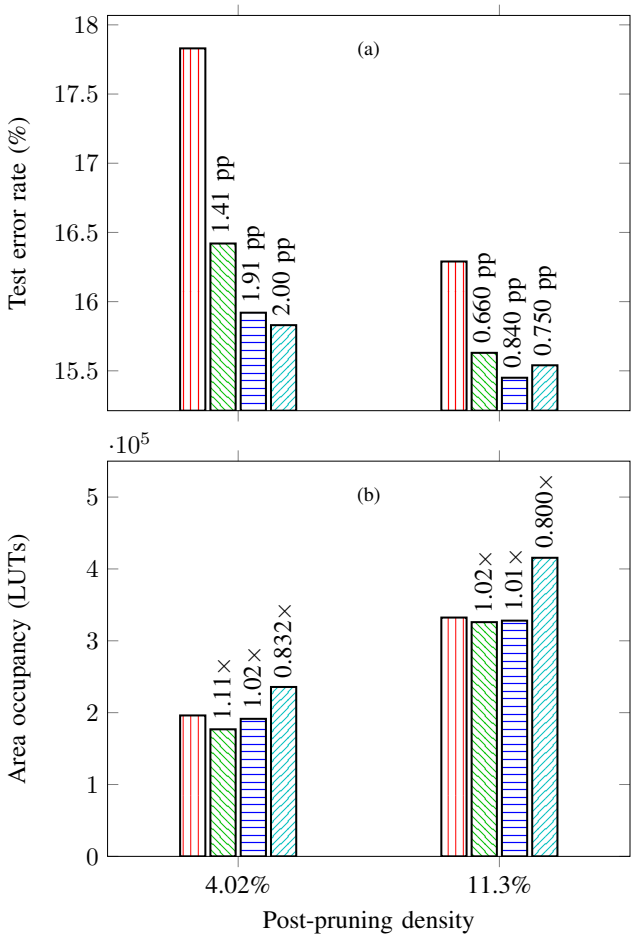

Fig. 5. (a) Accuracy and (b) area for ReBNet [5] (), 2-LUTNet (), 4-LUTNet () and 6-LUTNet () with CNV, pruned to two densities, classifying CIFAR-10. Annotations denote decreases *vs* ReBNet.

the LUT requirements of each of these model-dataset combinations when implemented using both the ReBNet and LUTNet inference architectures. The same layers for all pairs of implementations were fully unrolled and pruned, with the pruning threshold tuned to achieve an accuracy degradation no more than $\pm$0.300 pp *vs* ReBNet's without pruning.

For CNV and AlexNet, our use of arbitrary inference operators sees area reductions of around 2×. For the classification of SVHN, the CNV network used can be pruned more heavily than for CIFAR-10, hence the greater area saving for that dataset. For LFC classifying MNIST, however, more LUTs were consumed by LUTNet than its pruned ReBNet counterpart. While each of CNV's hidden layers has 2304 inputs per channel, LFC's channels each have only 256 inputs, presenting less opportunity for area reduction through popcount simplification. In this case, LUTNet's post-pruning LUT savings through popcount tree thinning were unable to make up for the inference operator LUT incursion.

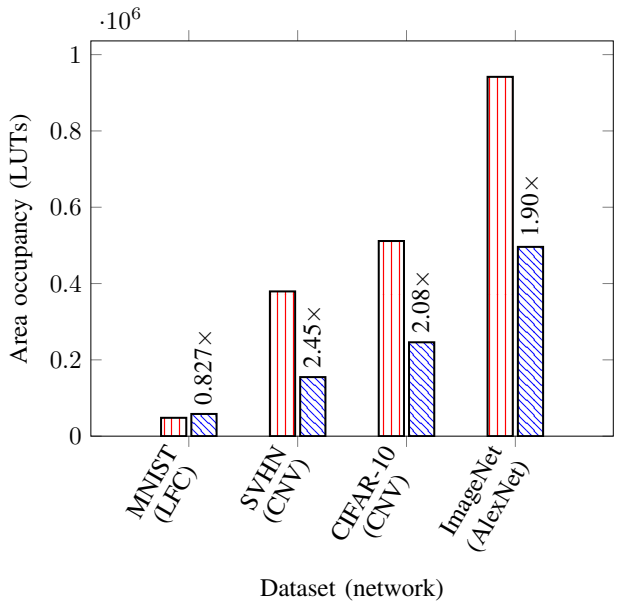

Fig. 6. Area occupancy for ReBNet [5] () and 4-LUTNet () with various models and datasets. Via pruning, each implementation's test accuracy was kept within $\pm$0.300 pp of that of the unpruned ReBNet baseline's. Annotations show the area decrease in each case.

D. Area Breakdown

As a crude method of verifying the source of LUTNet's area savings, we disabled design hierarchy optimisation in Vivado, preventing the synthesis engine from flattening across modules. By taking a slice of implementations shown in Fig. 4 at the unpruned ReBNet test error rate (84.5%) $\pm$0.300 pp, we obtained pruned ReBNet and 2-, 4-, 5-, 6- and 7-LUTNet implementations for CNV all of comparable CIFAR-10 test accuracy. Fig. 7 shows the LUT requirements for each of these, with area split into that required by popcount operators, inference operators and everything else. The overall height of each bar is the whole design's area occupancy with hierarchy optimisation *enabled*, but the height of each stacked bar is relative to the proportional area obtained with hierarchy optimisation *disabled*. We emphasise that these relative area data are not particularly meaningful, however this was the best we could do without significant manual tool intervention.

Generally, as more inputs are used per logical LUT, we can see that physical LUT requirements decrease, highlighting K-LUTNet's increasing logic density with K. Also shown in Fig. 7 is each implementation's post-pruning density. From the breakdowns, it can be seen that the number of physical LUTs required for popcount operators drops dramatically with density. More aggressive pruning reduces the number of branches in each popcount tree which, when unrolled, consume the majority of the target device's area.

As was pointed out in Section I, due to following a traditional BNN inference paradigm, ReBNet implementations—whether pruned or not—require zero LUTs for the realisation of their inference operators since, when unrolled, XNORs

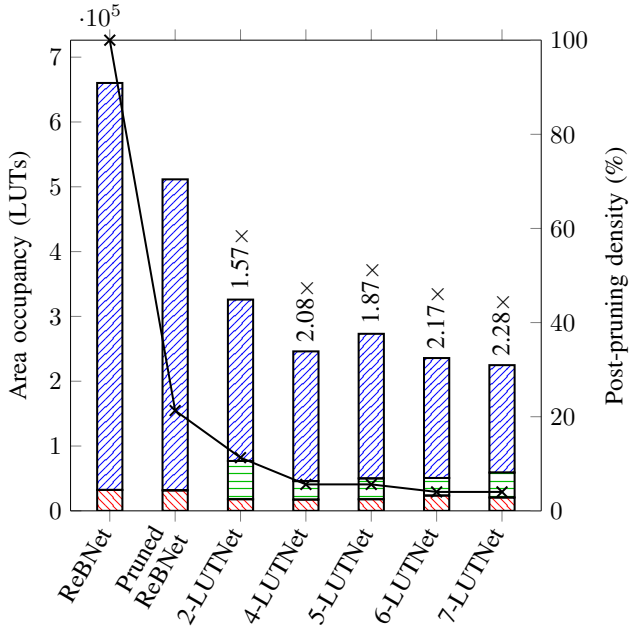

Fig. 7. LUT use breakdown, presented in terms of popcount operators (▨), inference operators (▭) and other layers (▧), for CNV implementations. Each implementation's test accuracy was within ±0.300 pp of that of the unpruned ReBNet baseline's [5]. Points (—×—) show post-pruning densities. Annotations show decreases *vs* ReBNet with pruning.

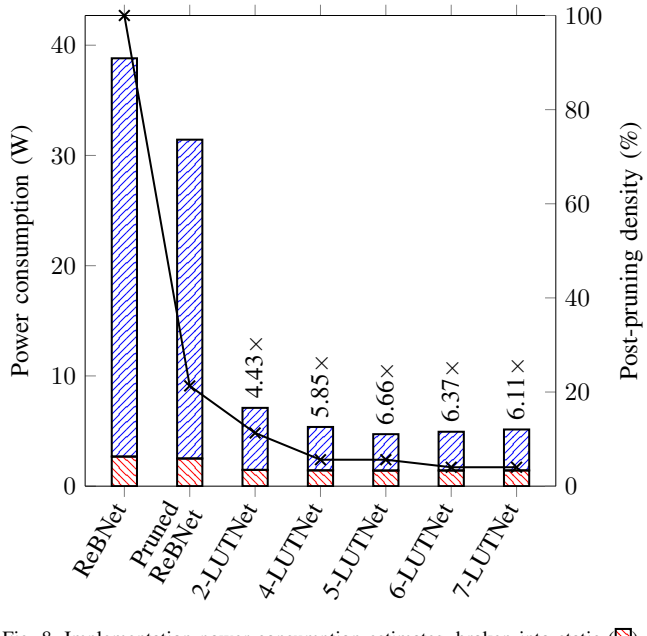

Fig. 8. Implementation power consumption estimates, broken into static (▧) and dynamic (▨) components, for the same CNV implementations used in Fig. 7. Points (—×—) show post-pruning densities. Annotations show decreases *vs* ReBNet with pruning.

become free-to-implement buffers and inverters. For LUTNet, this is not the case: physical LUTs are consumed by our logical K-LUTs. As shown in Fig. 7, however, this is more than outweighed by significant popcount area reduction. This confirms the statement made in Section I regarding $\tilde{N} \ll N$.

Between 2- and 6-LUTNet, we can observe a general trend of decreasing inference operator LUT requirements with density. Looking more closely, some more interesting features emerge. The jump in total area between 4- and 5-LUTNet can be attributed to two factors: lack of density reduction and decreased opportunity for LUT sharing. Here, the increased expressiveness of 5-LUTs was not significant enough to enable increased pruning while remaining within the required accuracy bound. On top of this, the logical-to-physical LUT packing effects discussed in Section V-C were marked, pushing both inference operator and total LUT requirements for 5-LUTNet above those for 4-LUTNet. Thereafter, although increasing numbers of physical LUTs were occupied by the 6- and 7-LUTNet implementations, decreases in density facilitated through increased network complexity caused more-than-compensatory popcount area reductions.

E. Energy Efficiency

We estimated LUTNet's energy efficiency using the Xilinx Power Analyzer (XPA) tool with default settings: vectorless mode (that not requiring specific input stimuli) and 12.5% primary input switching probability. The resultant power estimates, for the same implementations captured in Fig. 7, are shown in Fig. 8. All were obtained post-placement and -routing. Power consumption is equivalent to energy efficiency here since all implementations have identical throughput. While we acknowledge that vectorless power estimates are not particularly accurate—typically around ±10–20% from measured values [16]—they are sufficiently so for our purposes.

Since dynamic power consumption is directly related to area occupancy, Figs 7 and 8 show similar trends. Most of the fully unrolled networks' area consumption is attributable to popcount adder trees, whose carry chains are dominant with respect to switching activity. Popcount branch pruning shortens the chains, more than proportionately lowering their switching rates and thereby causing the large dynamic power drop. The reduction in static power between the ReBNet and LUTNet implementations can also be linked to area, although indirectly. Between Pruned ReBNet and 2-LUTNet there was a drop in estimated junction temperature from 60.1° to 31.3°, leading to reduced leakage current and therefore static power draw. Such temperature decreases are also useful since they limit ageing, thereby increasing device lifetime [17]. Overall, we can conclude that LUTNet's significant area reductions result in even greater energy efficiency improvements.

F. Training Efficiency

Each of LUTNet's inference K-LUTs consists of 2^K weights: $2\times$ more than that for $(K-1)$-LUTNet. Consequently, the number of training operations required per epoch increases exponentially with K. This does not necessarily translate to exponentially increasing training times over XNOR-based BNNs, however, since, as pointed out by Jouppi *et al.*, the majority of DNN training accelerators' speed is

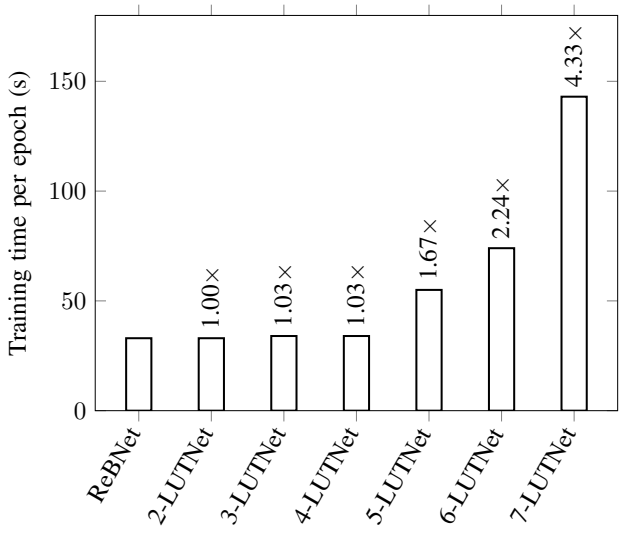

Fig. 9. Training efficiency of ReBNet [5] and LUTNet implementations for CNV classifying CIFAR-10. Annotations show increases over ReBNet.

bounded by memory bandwidth, not compute power [18]. This is evident from Fig. 9, which shows the per-epoch training times of ReBNet and 2-, 3-, 4-, 5-, 6- and 7-LUTNet implementations for CNV with CIFAR-10. Implementations from ReBNet to 4-LUTNet all have approximately the same training rate, despite the number of weights increasing by up to 16×. The training time did not increase because, for all of these implementations, progress was bottlenecked by high-precision activation transfer to and from GPU RAM. Increases of significance were seen for 5-LUTNet and beyond, for which the number of multiply-accumulate operations performed per activation transferred rose enough for the former to dominate.

VI. Limitations

While LUTNet implementations typically reach significantly higher logic density than XNOR-based BNNs, our proposal's greatest current limitation is that its inference K-LUTs cannot be time-multiplexed. Consequently, DNN hardware—in this paper, always complete layers—implemented following the LUTNet approach must be fully unrolled. While this may be acceptable in, for example, cloud deployments where throughput and energy efficiency are of paramount importance [19], it nevertheless limits the scalability of our proposal.

Time-multiplexing could be introduced in several ways. By adding a level of multiplexers prior to K-LUTs used as we propose herein, *i.e.* with hardened weights, each could be shared within or between channels or layers. Alternatively, K-LUT inputs could be sacrificed to allow some or all weights to be stored in RAM and updated at runtime, enabling up-to cycle-by-cycle switches in inference operator behaviour. While both of these proposals would result in lower throughput and logic density—and necessitate more complex and time-consuming training—the implementation of larger LUTNet-based networks on smaller devices would become feasible. We will explore these in our future work.

Fig. 4 shows that while our expansion to 2-LUTs results in significant logic density gains over XNORs, returns for movement to K-LUTs for $K > 2$ are diminishing. We suspect that this is due to our current restriction on the form of function g_n in (2), *i.e.* $\{-1,1\}^K \rightarrow \{-1,1\}$ rather than $\{-1,1\}^K \rightarrow \mathbb{N}$. This makes (9) insoluble when $\hat{c}_{\boldsymbol{d}}$ is restricted to binary values. We can overcome this, and potentially make even more efficient use of the underlying FPGA fabric, by learning the popcount circuitry along with XNOR substitutes, replacing the summation as well as $w_n x_n$ in (1).

While the introduction of nonlinearity significantly increases the expressiveness of each inference operator, the experiments reported in Section V-C revealed that 6-LUTNet showed signs of overfitting. In the future, we will explore methods of throttling expressiveness during training guided by losses, *e.g.* switching to higher or lower K when appropriate.

Finally, LUTNet's software does not currently skip zeroes during training. As networks increase in size, GPU RAM will be increasingly inefficiently used, resulting in unnecessarily long training times. A future revision will therefore incorporate sparse matrix multiplication, preventing the storage of and multiplication by zero-valued weights.

VII. Conclusion

In this paper, we introduced LUTNet: the first DNN architecture featuring K-LUTs as inference operators specifically designed to suit FPGA implementation. Our novel training approach results in the construction of K-LUT-based networks robust to high levels of pruning with little or no accuracy degradation, enabling the achievement of significantly higher area and energy efficiencies than that of traditional BNNs.

In our experiments with 4-LUT-based inference operators, FPGA implementations following our proposals achieved a mean area reduction of 1.81× *vs* the state-of-the-art BNN architecture with unrolling and pruning. These designs targetted a range of standard DNN models and datasets, required approximately the same training time and reached accuracies bounded within ±0.300 pp in all cases. Due to their efficient use of soft logic, LUTNet implementations can exhibit energy efficiencies up to 6.66× greater than reported by the authors of related prior works. Thanks to its parameter hardening, our architecture also requires no use of block RAM: a common bottleneck for FPGA-deployed DNNs.

The authors of existing works on low-precision DNN inference seem to have assumed that their forward-propagation functions must be good approximations of the linear dot product. With LUTNet, we argue for a tangential approach: through the embracement of nonlinearity, one can do more with less by unlocking the full potential of the K-LUT.

Acknowledgements

The authors are grateful for the support of the United Kingdom EPSRC (grant number EP/P010040/1), Imagination Technologies and the Royal Academy of Engineering.

Supporting data for this paper are available online at `https://doi.org/10.5281/zenodo.2616489`.

REFERENCES

[1] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2017.

[2] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Conference on Neural Information Processing Systems*, 2012.

[4] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going," *arXiv preprint arXiv:1901.06955*, 2019.

[5] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "ReBNet: Residual Binarized Neural Network," in *IEEE International Symposium on Field-programmable Custom Computing Machines*, 2018.

[6] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations," in *Conference on Neural Information Processing Systems*, 2015.

[7] M. Courbariaux and Y. Bengio, "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[8] W. Tang, G. Hua, and L. Wang, "How to Train a Compact Binary Neural Network with High Accuracy?" in *Association for the Advancement of Artificial Intelligence*, 2017.

[9] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning Both Weights and Connections for Efficient Neural Network," in *Conference on Neural Information Processing Systems*, 2015.

[10] F. Li and B. Liu, "Ternary Weight Networks," in *Conference on Neural Information Processing Systems*, 2016.

[11] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained Ternary Quantization," in *International Conference on Learning Representations*, 2017.

[12] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural Networks with Few Multiplications," in *International Conference on Learning Representations*, 2015.

[13] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning Structured Sparsity in Deep Neural Networks," in *Conference on Neural Information Processing Systems*, 2016.

[14] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *European Conference on Computer Vision*, 2016.

[15] X. Lin, C. Zhao, and W. Pan, "Towards Accurate Binary Convolutional Neural Network," in *Conference on Neural Information Processing Systems*, 2017.

[16] J. J. Davis, E. Hung, J. M. Levine, E. A. Stott, P. Y. K. Cheung, and G. A. Constantinides, "KAPow: High-accuracy, Low-overhead Online Per-module Power Estimation for FPGA Designs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 1, 2018.

[17] E. Stott, J. S. J. Wong, and P. Y. K. Cheung, "Degradation Analysis and Mitigation in FPGAs," in *International Conference on Field-programmable Logic and Applications*, 2010.

[18] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, and A. Borchers, "In-datacenter Performance Analysis of a Tensor Processing Unit," in *International Symposium on Computer Architecture*, 2017.

[19] R. Zhao, S. Liu, H.-C. Ng, E. Wang, J. J. Davis, X. Niu, X. Wang, H. Shi, G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Hardware Compilation of Deep Neural Networks: An Overview," in *International Conference on Application-specific Systems, Architectures and Processors*, 2018.