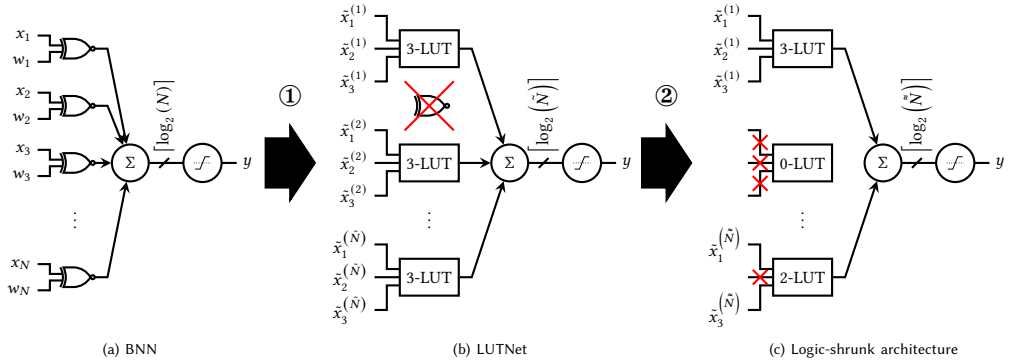
Logic Shrinkage: Learned Connectivity Sparsification for LUT-Based Neural Networks

ERWEI WANG, AMD, United Kingdom
MARIE AUFFRET, Imperial College London, United Kingdom
GEORGIOS-ILIAS STAVROU, Imperial College London, United Kingdom
PETER Y. K. CHEUNG, Imperial College London, United Kingdom
GEORGE A. CONSTANTINIDES, Imperial College London, United Kingdom
MOHAMED S. ABDELFATTAH, Cornell University, United States
JAMES J. DAVIS, Imperial College London, United Kingdom



(a) BNN (b) LUTNet (c) Logic-shrunk architecture

Fig. 1. Summary of our proposed training regime, demonstrating the structural transformation of a single DNN channel from a BNN (a) to a logic-shrunk architecture (c). In ①, the BNN is sparsified and logic-expanded, producing a LUTNet architecture (b) with $\tilde{N} \ll N$ K-LUTs (3-LUTs in this example) replacing N XNORs. This is then logic-shrunk in ②. LUT input pruning sees each K-LUT n replaced with a K'_n-LUT, where $K'_n \leq K$. When $K'_n = 0$, LUT n can be removed entirely. This results in $\tilde{\tilde{N}} \leq \tilde{N}$.

FPGA-specific DNN architectures using the native LUTs as independently trainable inference operators have been shown to achieve favorable area-accuracy and energy-accuracy tradeoffs. The first work in this area, LUTNet, exhibited state-of-the-art performance for standard DNN benchmarks. In this article, we propose the learned optimization of such LUT-based topologies, resulting in higher-efficiency designs than via the direct use of off-the-shelf, hand-designed networks. Existing implementations of this class of architecture require the manual specification of the number of inputs per LUT, K. Choosing appropriate K *a priori* is challenging,

Authors' addresses: Erwei Wang, AMD, Cambridge, United Kingdom, erwei.wang@amd.com; Marie Auffret, Imperial College London, London, United Kingdom, marie.auffret21@imperial.ac.uk; Georgios-Ilias Stavrou, Imperial College London, London, United Kingdom, georgios-ilias.stavrou18@imperial.ac.uk; Peter Y. K. Cheung, Imperial College London, London, United Kingdom, p.cheung@imperial.ac.uk; George A. Constantinides, Imperial College London, London, United Kingdom, g.constantinides@imperial.ac.uk; Mohamed S. Abdelfattah, Cornell University, New York, NY, United States, mohamed@cornell.edu; James J. Davis, Imperial College London, London, United Kingdom, james.davis@imperial.ac.uk.

1936-7406/2023/1-ART1 $15.00
https://doi.org/10.1145/3583075

and doing so at even high granularity, *e.g.* per layer, is a time-consuming and error-prone process that leaves FPGAs' spatial flexibility underexploited. Furthermore, prior works see LUT inputs connected randomly, which does not guarantee a good choice of network topology. To address these issues, we propose *logic shrinkage*, a fine-grained netlist pruning methodology enabling K to be automatically learned for every LUT in a neural network targeted for FPGA inference. By removing LUT inputs determined to be of low importance, our method increases the efficiency of the resultant accelerators. Our GPU-friendly solution to LUT input removal is capable of processing large topologies during their training with negligible slowdown. With logic shrinkage, we better the area and energy efficiency of the best-performing LUTNet implementation of the CNV network classifying CIFAR-10 by 1.54× and 1.31×, respectively, while matching its accuracy. This implementation also reaches 2.71× the area efficiency of an equally accurate, heavily pruned BNN. On ImageNet with the Bi-Real Net architecture, employment of logic shrinkage results in a post-synthesis area reduction of 2.67× *vs* LUTNet, allowing for implementation that was previously impossible on today's largest FPGAs. We validate the benefits of logic shrinkage in the context of real application deployment by implementing a face mask detection DNN using BNN, LUTNet and logic-shrunk layers. Our results show that logic shrinkage results in area gains versus LUTNet (up to 1.20×) and equally pruned BNNs (up to 1.08×), along with accuracy improvements.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Logic synthesis**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: LUT-based neural networks, binary neural networks, pruning, neural architecture search.

ACM Reference Format:
Erwei Wang, Marie Auffret, Georgios-Ilias Stavrou, Peter Y. K. Cheung, George A. Constantinides, Mohamed S. Abdelfattah, and James J. Davis. 2023. Logic Shrinkage: Learned Connectivity Sparsification for LUT-Based Neural Networks. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2023), 26 pages. https://doi.org/10.1145/3583075

1 INTRODUCTION

Deep neural network (DNN) inference is particularly well suited to custom hardware acceleration due to the application's inherent parallelism. In order to exploit this in the quest for ever-greater performance within given area and power budgets, researchers and industrial practitioners alike are increasingly turning to low-precision data types [3, 26, 32]. Binary neural networks (BNNs), in which weights and activations assume one of just two values, see this concept taken to the extreme. Figure 1a shows a generic BNN implementation of the quantized linear dot product operation central to DNN inference, wherein XNOR gates perform multiplication. Here, output $y = \phi(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{w})$, with inputs $\boldsymbol{x} \in \{-1, 1\}^N$, weights $\boldsymbol{w} \in \{-1, 1\}^N$ and activation function $\phi : \mathbb{N}_{\geq 0} \rightarrow \{-1, 1\}$. Such structures are compact and eminently parallelizable. When deployed on field-programmable gate arrays (FPGAs), these architectures can fully exploit FPGA's gate-level flexibility, and achieve superior energy efficiencies over GPUs, which require regularities in data or compute patterns for efficient SIMD or SIMT execution [15, 28, 32]. However, their simplicity tends to lead to underuse of the rich compute and routing resources that the target device provides.

We previously posited that more complex networks—netlists of small lookup tables (LUTs)—would ideally suit FPGA implementation due to their architectural similarity to the target fabric [29]. In that work, LUTNet, a BNN is first sparsified before its remaining XNORs are replaced with trainable $K : 1$ Boolean operators: a process we termed *logic expansion*. Each of these, directly implementable as a K-LUT, has $K\times$ more inputs than its XNOR predecessor, enabling recovery of the accuracy lost due to pruning. Formally, LUT n takes $\tilde{x}_i^{(n)} \sim \boldsymbol{x},\ i \in \{1, \cdots, K\}$ as input. The weights are hardened within the LUT masks and so no longer appear externally. The result of this transformation is a fast and efficient task-specific inference accelerator. This is exemplified in Figure 1b, in which $\tilde{N}$ K-LUTs (here, 3-LUTs) have been substituted for N XNOR gates. Since

$\tilde{N} \ll N$, compaction of the adder tree more than compensates for the marginal area penalty attributable to the K-LUTs. With LUTNet, we reported area efficiency improvements of around 2×, with identical inference throughput, over ReBNet [7]—the state-of-the-art BNN at the time—for problems of widely varying scale. More recent tools, including NullaNet [20] and LogicNets [27], also generate small LUTs as core components, but LUTNet remains unique in directly exposing a netlist's LUTs as differentiable functions trainable via stochastic gradient descent.

In such a LUT-based network, fixed K will inevitably be suboptimal. For example, while it may be the case that 6-LUTs map particularly well to a given device, $K = 6$ may be too many (or too few) inputs for a given node to suit the training data. We therefore propose that the size of each LUT be *learned* during training. Starting from a netlist of K-LUTs, we achieve this by removing input connections determined to be unimportant, resulting in a new netlist in which $K'_n \leq K \; \forall n \in \left\{1, \cdots, \tilde{N}\right\}$. Where $K'_n = 0$, LUT n can be removed entirely. We exemplify this process in Figure 1c, in which the total number of LUTs $\tilde{\tilde{N}} \leq \tilde{N}$, tending to further reduce area. The heterogeneity of the resultant netlists plays to the strengths of FPGA synthesis tools, which are adept at the low-level optimization of small Boolean functions. We find that networks constructed in this way are superior to their homogeneous counterparts, requiring fewer device resources to reach a target accuracy.

We take inspiration from the field of neural architecture search (NAS), in which a sparse and efficient topology is typically found by cutting away parts of a dense network [24]. While our end goal is similar, the netlist-level NAS we propose presents unique challenges. In particular, unlike standard topologies with a single weight per node, each node in a network of K-LUTs has K inputs sharing 2^K trainable parameters. Severance of one LUT input requires the manipulation of all 2^K entries within the respective truth table. Given that modern DNNs contain hundreds of thousands or even millions of nodes, naïve operation on all of these would quickly become intractable. We thus present a vectorized implementation of our input pruning proposal ideally suited to GPU acceleration.

In this article, we present *logic shrinkage*: the automated search for, and construction of, DNN inference topologies featuring learned netlist sparsity. We make the following novel contributions.

- We propose a method for the evaluation of input connection salience within a netlist of LUTs used for DNN inference.
- We cast LUT input removal as a matrix-vector operation, enabling us to take advantage of GPUs for its realization.
- We present a TensorFlow-based implementation of logic shrinkage, in which DNNs composed of LUTs of fixed size are automatically transformed into sparser, heterogeneous networks more efficiently mappable onto FPGAs.
- We empirically explore the effects of logic shrinkage on area efficiency and accuracy via comparison with LUTNet [29], our state-of-the-art FPGA-specific DNN inference topology, across a broad range of standard network models and datasets. We also experimentally determine logic shrinkage's impact on energy and training efficiency. Against LUTNet with fixed $K = 4$, ordinarily the best-performing choice of constant K, we achieve area compression of 1.54× and an energy saving of 1.31× for the CNV network [28] classifying the CIFAR-10 dataset [10] while reaching comparable accuracy. Finally, we report positive results at scale, with our logic-shrunk Bi-Real Net [17] design classifying ImageNet [4] demanding 2.67× lower post-synthesis area than LUTNet.
- We validate the benefits of logic shrinkage through evaluation of a real-world machine learning application, face mask detection, including complete system verification and deployment on a PYNQ development board. Here, we better LUTNet's area requirements by up to 1.20×

while simultaneously improving accuracy. Our implementations of this application represent the first deployments of the ReBNet, LUTNet and logic-shrunk architectures on real devices.
- We provide an open-source release[1] of our work for the community to use and build upon.

A preliminary version of this work appeared in the proceedings of the 30th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2022) [31]. In that paper, we evaluated our work with standard DNN models and datasets, and verified the correctness of our designs in simulation. In Section 6 of this article, we go further by implementing a topical machine learning application for hardware verification and evaluation. Rather than stopping at simulation, we built complete systems using three DNN inference architectures—ReBNet, LUTNet and logic-shrunk—that run on real devices. None of these architectures had seen deployment to date. Along with the rest of our code, all of the designs present are freely available in our open-source repository.

2 RELATED WORK

2.1 FPGA-Tailored DNN Architectures

LUT-based DNN inference accelerators have been shown to achieve remarkable performance when deployed on FPGAs. NullaNet [20] and LogicNets [27] were conceived with small-scale classification tasks in mind, for which they reached latency in the tens of nanoseconds and throughput in the hundreds of millions of samples per second. Going beyond FPGA-tailored network design, our previously proposed LUTNet topologies can be trained via stochastic gradient descent [29]. LUTNet's trainable netlists are compatible with common machine learning optimization strategies such as pruning, thereby affording opportunities for increased performance and efficiency. Furthermore, the LUTNet approach suits tasks spanning a broad range of scales, including ImageNet classification.

LUTNet netlists tend to be large due to the one-to-one mapping between DNN nodes and LUTs. Consequently, in typical deployments, only a subset of network layers are logic-expanded: the remainder are kept as standard BNN structures. We have also proposed a time-multiplexed version of the LUTNet architecture, which negates the need for each LUT to be specific to a single node by reintroducing runtime-variable weights [30]. This increases LUTNet's scalability, but also reduces its potential area and energy efficiency gains over BNNs due to the lost freedom in LUT specialization.

We use LUTNet netlists as a starting point for logic shrinkage, and demonstrate that the resultant designs are more area and energy efficient. Our automated design flow maintains the deployment flexibility, scalability and ease of use of LUTNet's. To evaluate the potential of logic shrinkage in the most generic setting, we assume the use of hardened weights, in line with vanilla LUTNet. Our approach could be applied to time-multiplexed architectures, however we would similarly expect lower gains from doing so.

2.2 Activation Pruning

Activations within a DNN commonly contribute to its output to varying degrees. Activation pruning exploits this by assigning compute only to those with high relative importance (or salience), leading to increased efficiency. While crude attempts to establish salience, such as taking the mean of activations across a training dataset, were reportedly unsuccessful [19], use of the partial derivative of a cost function with respect to the activations has been shown to work well [14, 19]. Such a partial derivative—an activation gradient—quantifies the impact of a perturbation of that activation on the output of the network. It is intuitive to therefore prioritize activations with low gradient

[1] https://github.com/awai54st/Logic-Shrinkage

magnitude for pruning. Molchanov *et al.* [19] and Lee *et al.* [14] both took this approach, reporting state-of-the-art results with and without retraining, respectively.

For the aforementioned works, which targeted standard DNN topologies, it was assumed that the gradients of activations within a layer are independent. This assumption does not hold in the context of LUT input pruning; input interdependence exists due to the configuration bits of each K-LUT since these are shared between each of its K inputs. We introduce a pruning strategy that solves this problem, formulate it such that it is ideally suited to GPU acceleration and use it to generate area-efficient netlists.

2.3 Neural Architecture Search

Neural architecture search (NAS) automates the process of DNN design. In many NAS works, candidate functions are placed in parallel to form "supernets", after the training of which only those found to be of highest salience are retained. The granularity of functions that compose supernets varies. In DARTS, selections are made between small convolutional layers, with around 10 of these available to choose from in each instance [16]. Candidate function outputs are scaled by trainable scaling factors before they are accumulated, making the search space continuous and therefore differentiable. The scaling factors capture function salience, and these are used post-training to determine the makeup of the final network. Works including DARTS have been shown to produce high-performance architectures orders of magnitude more quickly than their non-differentiable counterparts, including those using reinforcement learning [35] and evolution [23]. The authors of AtomNAS proposed finer-grained search, decomposing convolutions into combinations of "atomic blocks" and greatly increasing the number of possible output architectures *vs* DARTS [18]. This richness in flexibility resulted in the production of state-of-the-art ImageNet classifiers.

We propose a network topology search approach analogous to prior works on NAS. We start with an overprovisioned K-LUT-based architecture—a supernet—and selectively remove its redundancy at ultra-fine granularity via LUT input pruning.

3 BACKGROUND: LOGIC EXPANSION

To enable post-logic expansion retraining for LUTNet, we defined an *interpolating extension* to the complete set of $K:1$ Boolean operations as our training function [29]:

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

Real-valued parameters $\hat{\boldsymbol{c}}$ are trainable with stochastic gradient descent and, when binarized for use during inference, represent LUT masks, $\boldsymbol{c}$. (1) expands as

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

Table 1. An example 2-LUT truth table with real-valued entries, in this case representing an AND gate. Δ_{x_i} captures the change in LUT output with respect to a change in input x_i.

x_2 \ x_1	-1	1	$\lvert\Delta_{x_1}\rvert$
-1	-0.90	-0.01	**0.89**
1	-0.85	0.05	**0.90**
$\lvert\Delta_{x_2}\rvert$	**0.05**	**0.06**	

for $K \in \mathbb{N}_{>0}$, with each polynomial comprising 2^K trainable parameters. We use a logic-expanded, retrained network as the starting point for logic shrinkage.

4 MECHANICS OF LOGIC SHRINKAGE

4.1 LUT Input Salience

The activation gradient-based salience criteria commonly used with standard neural networks are not directly applicable to netlist pruning due to the interdependence of LUT inputs. However, their fundamental concept—gauging an activation's importance by the impact on the network's outputs with respect to a change in that activation—remains relevant, thus we adopt it for the purpose of establishing LUT input salience.

Consider a K-binary-input LUT with truth table entries encoded as $\{0, 1\} \rightarrow \{-1, 1\}$. Each entry represents the output with respect to a unique combination of inputs; changing one or more input values will alter the selection of LUT entry used as output. We define a particular LUT input's salience to be the sum of such changes across all combinations of the remaining inputs. If the flipping of a given input never leads to a change in LUT output, that input can clearly be removed without having any impact on the functionality of the network. Such an input therefore has zero salience. If toggling an input *sometimes*—but rarely—results in output change, we consider that input to be of low salience, while the opposite holds for an input whose toggling often causes the LUT's output to change.

LUTNet-style Lagrangian interpolation, which we introduced to make LUTs differentiable [29], presents us with an opportunity to more precisely quantify LUT input salience. Since LUT entries in this scenario are real-valued, output changes are typically less coarse than when operating in the binary domain.

To exemplify our approach, Table 1 contains possible real-valued LUT entries $\hat{\boldsymbol{c}}$ of a 2-LUT, where x_1 and x_2 are its inputs. The LUT's entries will be binarized prior to synthesis; once this is done, this LUT will function as an AND gate.

In Table 1, the salience of input x_1, s_1, is defined as the total disturbance to the LUT output across both $x_2 = 1$ and $x_2 = -1$ when x_1 experiences a change in sign, *i.e.* the sum of column $\lvert\Delta_{x_1}\rvert$. Similarly, the salience of x_2, s_2, is defined as the sum of row $\lvert\Delta_{x_2}\rvert$. In general, we define the salience of K-LUT input i as

$$s_i = \sum_{\boldsymbol{d}_1 \in \{-1,1\}^{i-1}} \sum_{\boldsymbol{d}_2 \in \{-1,1\}^{K-i}} \left| \hat{c}_{(\boldsymbol{d}_1, 1, \boldsymbol{d}_2)} - \hat{c}_{(\boldsymbol{d}_1, -1, \boldsymbol{d}_2)} \right|. \tag{3}$$

From Table 1, since $s_1 > s_2$, we can conclude that toggles of input x_1 lead to greater impact on the LUT output than toggles of x_2. x_2 is therefore less important than x_1 and so should be prioritized for disconnection. Once the less-salient inputs of a network's LUTs have been identified, we can turn to their removal.

We experimented with other candidate salience criteria—including weight gradient- [14] and Taylor expansion-based [19] methods—before settling on the aforedescribed approach. While these

were shown to work well for conventional DNN node pruning, we did not observe positive results in their use for LUT input removal.

4.2 Pruning

In a similar vein to the establishment of salience, LUT input pruning also requires a nonstandard approach. With a conventional neural network node, an activation can be removed by setting its corresponding weight to zero. The removal of an input from a K-LUT, on the other hand, requires the manipulation of all of the 2^K $\hat{\boldsymbol{c}}$ parameters that define the contents of the LUT.

Here we demonstrate the process of removing LUT inputs using the 2-LUT in (2) as an example. In order to remove input $\tilde{x}_1^{(n)}$, the LUT mask $\hat{\boldsymbol{c}}$ should be transformed into $\hat{\boldsymbol{c}}'$ such that $\hat{c}'_{(-1,-1)} = \hat{c}'_{(1,-1)}$ and $\hat{c}'_{(-1,1)} = \hat{c}'_{(1,1)}$. Countless functions can be used to achieve this. Of them, we chose the computationally cheapest: assignment using the means of their pre-shrinkage values. The removal of input $\tilde{x}_1^{(n)}$ is thus achieved by performing

$$\begin{aligned}
\hat{c}'_{(-1,-1)} &= 1/2\left(\hat{c}_{(-1,-1)} + \hat{c}_{(1,-1)}\right)\\
\hat{c}'_{(1,-1)} &= 1/2\left(\hat{c}_{(-1,-1)} + \hat{c}_{(1,-1)}\right)\\
\hat{c}'_{(-1,1)} &= 1/2\left(\hat{c}_{(-1,1)} + \hat{c}_{(1,1)}\right)\\
\hat{c}'_{(1,1)} &= 1/2\left(\hat{c}_{(-1,1)} + \hat{c}_{(1,1)}\right)
\end{aligned} \tag{4}$$

Similarly, the removal of LUT input $\tilde{x}_2^{(n)}$ is achieved as

$$\begin{aligned}
\hat{c}'_{(-1,-1)} &= 1/2\left(\hat{c}_{(-1,-1)} + \hat{c}_{(-1,1)}\right)\\
\hat{c}'_{(1,-1)} &= 1/2\left(\hat{c}_{(1,-1)} + \hat{c}_{(1,1)}\right)\\
\hat{c}'_{(-1,1)} &= 1/2\left(\hat{c}_{(-1,-1)} + \hat{c}_{(-1,1)}\right)\\
\hat{c}'_{(1,1)} &= 1/2\left(\hat{c}_{(1,-1)} + \hat{c}_{(1,1)}\right)
\end{aligned} \tag{5}$$

Referring back to the 2-LUT example in Table 1, removal of less-salient input x_2 requires the application of (5) to the LUT mask, $\hat{\boldsymbol{c}}$. This results in new parameters $\hat{c}'_{(-1,-1)} = \hat{c}'_{(-1,1)} = -0.88$ and $\hat{c}'_{(1,-1)} = \hat{c}'_{(1,1)} = 0.02$. Once $\hat{\boldsymbol{c}}'$ is binarized, the 2-LUT performs the single-input function $y = x_1$, *i.e.* it is transformed into a wire.

4.3 Pruning at Scale

While pruning when $K = 2$, as exemplified in (4) and (5), is straightforward, the complexity of these operations increases exponentially with K. Logic shrinkage of one K-LUT involves the transformation of 2^K parameters, and the assignments are unique for each of the $\sum_{i=1}^{K}\binom{K}{i} = 2^K - 1$ possible LUT input combinations. This complexity further scales with the number of LUTs being trained. To ensure scalability, the implementation of our pruning method must therefore take advantage of the high-performance linear algebraic capabilities of modern GPUs and DNN training frameworks.

We implement functions such as (4) and (5) as matrix-vector multiplications $\hat{\boldsymbol{c}}' = \boldsymbol{U}\hat{\boldsymbol{c}}$ with a transformation matrix $\boldsymbol{U} \in \mathbb{R}^{2^K \times 2^K}$.

Continuing with those examples, the removal of input $\tilde{x}_1^{(n)}$ in (4) and of input $\tilde{x}_2^{(n)}$ in (5) are performed as

$$\begin{pmatrix} \hat{c}'_{(-1,-1)} \\ \hat{c}'_{(1,-1)} \\ \hat{c}'_{(-1,1)} \\ \hat{c}'_{(1,1)} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} \hat{c}_{(-1,-1)} \\ \hat{c}_{(1,-1)} \\ \hat{c}_{(-1,1)} \\ \hat{c}_{(1,1)} \end{pmatrix},$$

$$\textit{i.e.} \quad \boldsymbol{U}_1 = \frac{1}{2} \boldsymbol{I}^{2\times 2} \otimes \mathbf{1}^{2\times 2}$$

and

$$\begin{pmatrix} \hat{c}'_{(-1,-1)} \\ \hat{c}'_{(1,-1)} \\ \hat{c}'_{(-1,1)} \\ \hat{c}'_{(1,1)} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{c}_{(-1,-1)} \\ \hat{c}_{(1,-1)} \\ \hat{c}_{(-1,1)} \\ \hat{c}_{(1,1)} \end{pmatrix},$$

$$\textit{i.e.} \quad \boldsymbol{U}_2 = \frac{1}{2} \mathbf{1}^{2\times 2} \otimes \boldsymbol{I}^{2\times 2},$$

respectively, where $\otimes$ is the Kronecker product and use of $\boldsymbol{U}_i$ causes the removal of LUT input i.

The removal of a single input can be conceptualized as the merging of LUT parameter pairs followed by the forking of their means back to their original locations. This is achieved by $\mathbf{1}^{2\times 2}$ in the aforementioned examples. The Kronecker product with the identity matrix permutes the merges and forks as required. In general,

$$\boldsymbol{U}_i = \frac{1}{2} \boldsymbol{I}^{2^{K-i}\times 2^{K-i}} \otimes \mathbf{1}^{2\times 2} \otimes \boldsymbol{I}^{2^{i-1}\times 2^{i-1}}. \tag{6}$$

Where removal of multiple LUT inputs is desired, $\boldsymbol{U}_i$ for each input i can simply be multiplied together to form a single transformation matrix, $\boldsymbol{U}$, before application.

The construction of $\boldsymbol{U}$, although computationally expensive, is a one-time process that we have found to never exceed 10 s. During retraining, logic shrinkage is implemented as one instance of matrix-vector multiplication, which is ideally suited to GPU acceleration.

Although a post-shrinkage LUT mask $\hat{\boldsymbol{c}}'$ will always represent a simpler function, dependent on fewer inputs, than its predecessor $\hat{\boldsymbol{c}}$, $\hat{\boldsymbol{c}}'$ will retain 2^K parameters. While this means that a post-shrinkage netlist will contain redundancy, a benefit of this is that such a netlist will remain compatible with the existing LUTNet implementation flow. Our experiments revealed that Vivado effectively recognizes and removes this redundancy during synthesis with no noticeable overhead. Representation of sparse input connections in a dense format, as we propose, also simplifies our training software.

4.4 Iterative Pruning

The authors of many network pruning works, including Han *et al.* [9] and See *et al.* [25], proposed pruning across multiple iterations, with each including a post-pruning retraining phase. In keeping with this approach, we separate our LUT input pruning process into multiple iterations, each greedier than the last, with retraining following each. In early experiments, we confirmed that this approach outperforms one-shot pruning, and found that $T = 3$ iterations with $P = 20$ retraining epochs following each performed favorably. As exemplified in Figure 3, this setup results in training stability being reached quickly in each iteration.

Algorithm 1 details the iterative logic shrinkage training process. In each of the T total iterations, salience scores of all LUT inputs in the subset of the network subject to logic shrinkage are evaluated

Algorithm 1 Logic shrinkage retraining process.

Inputs:
$K \in \mathbb{N}$, ▷ # pre-shrinkage inputs per LUT
$\tilde{N} \in \mathbb{N}$, ▷ # pre-shrinkage LUTs
$\delta \in [0, 1]$, ▷ Target sparsity
$T \in \mathbb{N}$, ▷ # shrinkage iterations
$P \in \mathbb{N}$, ▷ # epochs per shrinkage iteration
$\hat{C} \in \mathbb{R}^{2^K \times \tilde{N}}$ ▷ Pre-shrinkage LUT masks
Output:
$\hat{C}' \in \mathbb{R}^{2^K \times \tilde{N}}$ ▷ Post-shrinkage LUT masks

procedure LOGICSHRINK
  $\hat{C}' \leftarrow \hat{C}$
  for $t \leftarrow \{1, \cdots, T\}$ **do**
    $\boldsymbol{S} \leftarrow \text{getsalience}\left(\hat{C}'\right)$ ▷ Per (3); $\boldsymbol{S} \in \mathbb{R}_{\geq 0}^{\tilde{N} \times K}$
    $\boldsymbol{r} \leftarrow \text{getRankOrder}(\text{vec}(\boldsymbol{S}))$
    $\delta_t \leftarrow \delta \times {}^{t}/_{T}$
    $\boldsymbol{M} \leftarrow \text{vec}^{-1}_{\tilde{N} \times K}\left(\mathbf{1}_{\boldsymbol{r} < \delta_t \tilde{N} K}\right)$ ▷ $\boldsymbol{M} \in \{0, 1\}^{\tilde{N} \times K}$
    $\boldsymbol{V} \leftarrow \mathbf{0}^{\tilde{N} \times 2^K \times 2^K}$
    for $n \leftarrow \left\{1, \cdots, \tilde{N}\right\}$ **do**
      $\boldsymbol{V}_n \leftarrow \boldsymbol{I}^{2^K \times 2^K}$
      for $i \leftarrow \{1, \cdots, K\}$ **do**
        if $m_{ni} = 1$ **then**
          $\boldsymbol{V}_n \leftarrow \boldsymbol{V}_n \boldsymbol{U}_i$ ▷ Per (6); $\boldsymbol{V}_n \in \mathbb{R}^{2^K \times 2^K}$
    $\hat{C}' \leftarrow \text{retrain}\left(\hat{C}', \boldsymbol{V}, \text{epochs} = P\right)$
  return $\hat{C}'$

using (3) and then ranked. The input sparsity for iteration t, δ_t, increases with t until the target sparsity δ has been reached. Binary mask $\boldsymbol{M}$ indicates the low-salience LUT inputs to be pruned. Finally, logic shrinkage transformation matrices $\boldsymbol{U}$ are constructed based on $\boldsymbol{M}$, and the network is retrained with input connections sparsified for P epochs. When retraining, we consistently apply all $\boldsymbol{U}$s formed in order to ensure that inputs previously severed by logic shrinkage remain so from then on. The topology of the portion of the network not subject to logic shrinkage is preserved throughout this process, but its parameters remain trainable.

5 EVALUATION

5.1 Implementation

For ease of development and evaluation, we engineered logic shrinkage as a bolt-on addition to the existing LUTNet training and hardware implementation flow [29]. A high-level view of the augmented flow, with the logic shrinkage stage annotated in red, can be found in Figure 2. Now, in addition to the network model, training dataset, input precision and node pruning level that LUTNet takes as input, the user provides their desired LUT input pruning level as well. The back-end FPGA implementation steps remain unchanged.

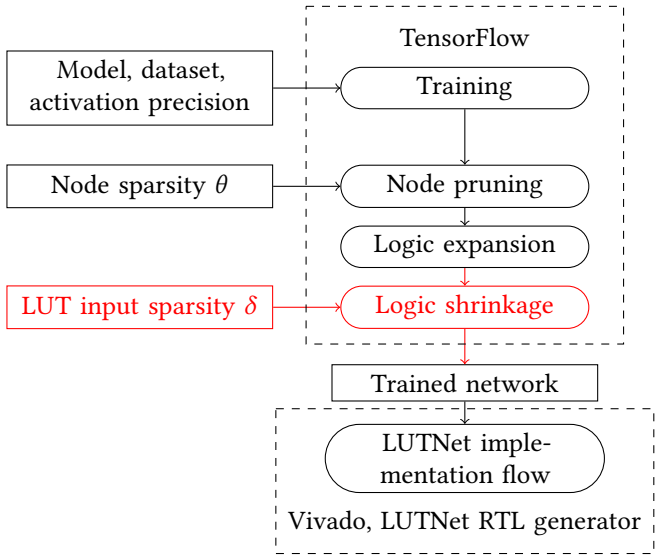


Fig. 2. Incorporation of logic shrinkage within LUTNet's fully automated training and FPGA implementation flow.

Table 2. 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. $\text{MaxPool}_{x,\,y}$ is an $x \times x$ maximum-pooling layer with stride y, and BatchNorm and SoftMax are batch normalization and normalized exponential layers, respectively. $\text{ResBlk}_{x,\,y,\,z}$ denotes a residual block with two $\text{Conv}_{x,\,y,\,z}$ layers, each followed by a BatchNorm. Layers in bold were unrolled and targeted for logic expansion (and shrinkage). For ImageNet, the residual block in bold had its first convolutional layer unrolled and targeted.

Dataset	Model	Network architecture
MNIST [13]	LFC [28]	FConn_{256}, BatchNorm, $\mathbf{\text{FConn}_{256}}$, BatchNorm, $\mathbf{\text{FConn}_{256}}$, BatchNorm, $\mathbf{\text{FConn}_{256}}$, BatchNorm, $\mathbf{\text{FConn}_{10}}$, BatchNorm, SoftMax
SVHN [21] & CIFAR-10 [10]	CNV [28]	$\text{Conv}_{64,\,3,\,1}$, BatchNorm, $\text{Conv}_{64,\,3,\,1}$, BatchNorm, $\text{MaxPool}_{2,\,2}$, $\text{Conv}_{128,\,3,\,1}$, BatchNorm, $\text{Conv}_{128,\,3,\,1}$, BatchNorm, $\text{MaxPool}_{2,\,2}$, $\text{Conv}_{256,\,3,\,1}$, BatchNorm, $\mathbf{\text{Conv}_{256,\,3,\,1}}$, BatchNorm, FConn_{512}, BatchNorm, FConn_{512}, BatchNorm, FConn_{10}, BatchNorm, SoftMax
ImageNet [4]	Bi-Real-18 [17]	$\text{Conv}_{64,\,7,\,2}$, BatchNorm, $\text{MaxPool}_{3,\,2}$, $\text{ResBlk}_{64,\,3,\,1}$, $\text{ResBlk}_{64,\,3,\,1}$, $\text{ResBlk}_{128,\,3,\,2}$, $\text{ResBlk}_{128,\,3,\,2}$, $\text{ResBlk}_{256,\,3,\,2}$, $\mathbf{\text{ResBlk}_{256,\,3,\,2}}$, $\text{ResBlk}_{512,\,3,\,2}$, $\text{ResBlk}_{512,\,3,\,2}$, FConn_{1000}, SoftMax

In common with LUTNet, employment of logic shrinkage necessitates no FPGA knowledge. Parameterized Keras layers and C++ templates are provided for training and implementation, respectively, enabling low-effort construction of dataflow DNN engines.

5.2 Benchmarks

We evaluated our approach using the DNN model and dataset combinations detailed in Table 2. Hardware implementations for all datasets other than ImageNet targeted the AMD Kintex UltraScale XCKU115. For ImageNet, we targeted the largest FPGA available to us: the Virtex UltraScale+ XCVU9P. All implementations met timing at 200 MHz. Our primary comparison point was LUTNet,

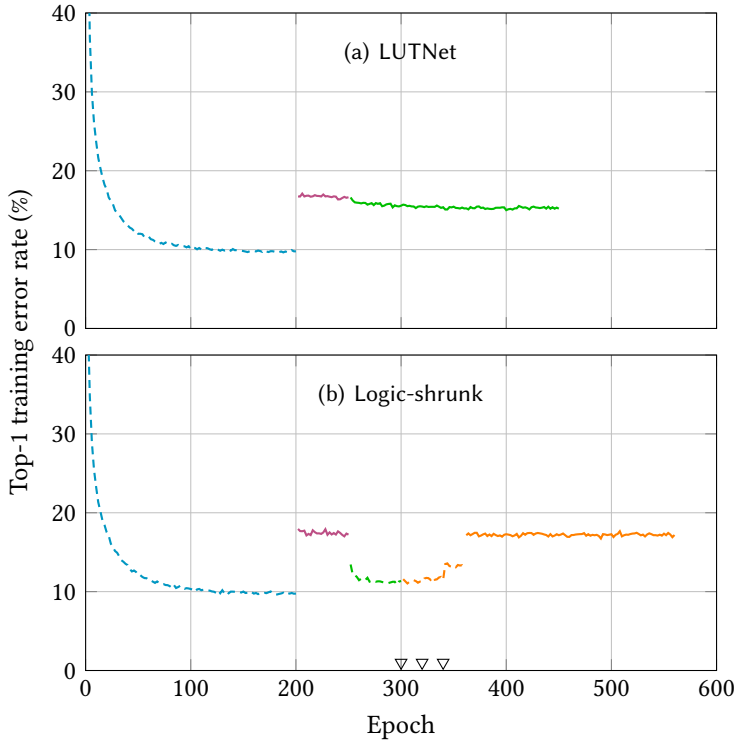


Fig. 3. Training error for CNV classifying CIFAR-10 using LUTNet (a) and logic-shrunk (b) architectures, during initial training (- - - -), post-node pruning retraining (——), post-logic expansion retraining (——) and post-logic shrinkage retraining (- - - -). Phases with binarized forward propagation are denoted with solid lines; those with high-precision (`float32`) forward propagation are shown dashed. Annotations (▽) mark epochs at which logic shrinkage was applied.

trained as we described in its original publication [29]. Where possible, we also maintained the BNN baseline, ReBNet [7], used as the starting point for LUTNet's logic expansion, and considered its test accuracy to be a performance floor.

For fairness of comparison to vanilla LUTNet (and ReBNet), we used identical experimental settings to those employed for its evaluation with MNIST, SVHN and CIFAR-10 [29]. Implementations for these datasets included all layers: those selected for logic expansion (and subsequent shrinkage) were unrolled, with the remainder left identical to the BNN starting point. For those datasets, the same set of layers were selected for unrolling as vanilla LUTNet. For ImageNet, our design encompassed the target layer only due to the complexity of implementing the remaining layers. In all cases, layers selected for logic expansion and shrinkage are marked in bold in Table 2.

5.3 Training Specifics

5.3.1 Small-Scale Datasets. For our experiments with MNIST [13], SVHN [21] and CIFAR-10, pre-trained ReBNet BNNs were first node-pruned and logic-expanded following the LUTNet approach (described in Section 3) before being logic-shrunk (Section 4). We inserted four new retraining phases between the post-node pruning (——) and post-logic expansion (——) phases performed for LUTNet shown in Figure 3a. These are reflected in Figure 3b. After logic expansion, we performed

50 epochs of retraining with high-precision forward propagation (----), with a further 20 (----) performed following each of three logic shrinkage iterations. Finally, 200 epochs with binarized forward propagation (——) were performed, matching the final phase of LUTNet training. We chose these numbers of epochs and logic-shrinkage iterations since, as exemplified in Figure 3, training accuracy saturation was achieved at or before the end of each phase. All training phases were executed in TensorFlow and accelerated using Nvidia RTX 3090 GPUs.

5.3.2 ImageNet. We also experimented with the ImageNet dataset. For this task, we prepared a pretrained Bi-Real Net model [17], Bi-Real-18, as our starting point, and then performed the retraining process outlined in Section 5.3.1. Here, we ran post-logic expansion retraining for 32 epochs (rather than 50), post-logic shrinkage retraining for eight epochs per iteration (rather than 20) and final, binarized retraining for 64 epochs (rather than 200). These numbers were again chosen due to our observance of accuracy stability.

5.4 Area Efficiency

In line with the prior FPGA-tailored DNN works detailed in Section 2.1, our primary objective was to maximize the area efficiency of our implementations. We define this as the number of device LUTs required to construct a network able to achieve a particular test accuracy for a given dataset while operating at a given classification rate. In all of our experiments, throughput remained fixed, thus we need only to consider area *vs* accuracy.

5.4.1 Pruning Sparsity Tuning. We began by seeking to understand the interplay between the sparsity afforded to us through BNN node sparsification (by tuning θ) and LUT input pruning (δ). To this end, Figure 4 shows the achieved whole-network area *vs* top-1 test accuracy for LUTNet and logic-shrunk implementations of the CNV network trained to classify the CIFAR-10 dataset. Each point marks the mean of five differently seeded training runs, with an error bar indicating its range. For reference, the mean test error rate of ReBNet without pruning—again averaged over five training runs—is also shown (◁). Filled markers (● ♦ ▲ ■ ⬟) reflect results for LUTNet, split into those with LUT size $K = 2$ (Figure 5a), 4 (5b) and 5 (5c). Each color/shape represents a distinct node sparsity θ. Unfilled markers (○ ♢ △ □ ⬠) capture area *vs* accuracy for logic-shrunk implementations with varying LUT input sparsity δ. Along each colored line, implementations all had the same K and θ, varying only in δ. Logic-shrunk designs used the respective fixed-K LUTNet architecture as the starting point for logic shrinkage, after which they contained LUTs up to size K.

By comparing across data points of different shapes/colors, one can clearly observe that the error rate increases as more aggressive node pruning is applied. This trend is consistent across both the LUTNet and logic-shrunk implementations. Figure 4 also reveals relatively consistent area-accuracy tradeoffs exposed through the variance of LUT input sparsity δ for each combination of K and θ. As δ increases, connection pruning becomes more aggressive, pushing data points to the left. The error rate decreases at first due to the removal of redundant logic from the netlist. Beyond each curve's inflection point, the pruning becomes too harsh; we thus begin to see the error rate rise. Also notice that, in some cases, K-LUT-based implementations outperform unpruned ReBNet (660196 LUTs) despite occupying as little as a quarter of its area. This speaks to the increased expressiveness of these architectures over BNNs.

Inspection of Figures 5a and 5b reveals that, in some cases, logic-shrunk implementations consume more area than the LUTNet architectures they were shrunk from. This is counterintuitive since logic shrinkage reduces netlist complexity by severing LUT connections; it never adds them. We attribute this effect, which is more pronounced in denser networks (higher θ) of smaller LUTs (lower K), to Vivado's heuristic-based placement and routing algorithms.

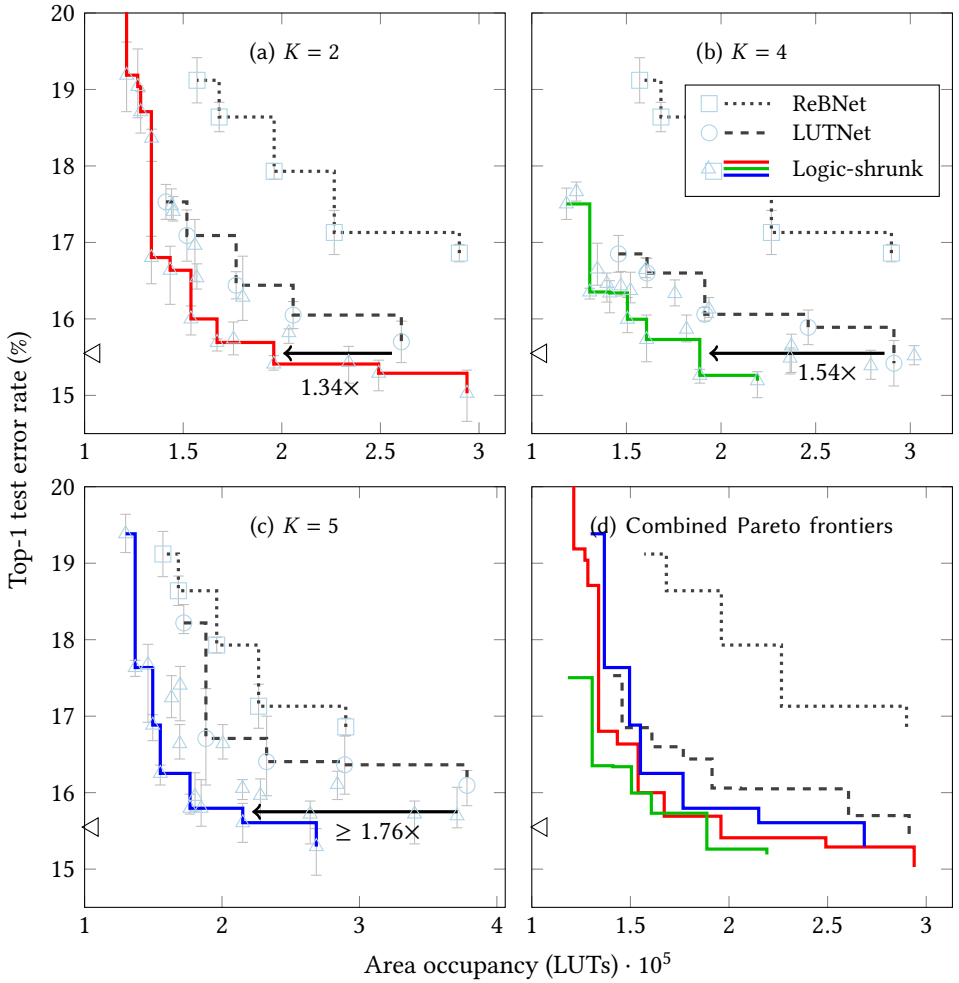


Fig. 4. Pareto-optimal frontiers of the LUTNet (○) and logic-shrunk (△) data points from Figure 4, with LUTNet (- - -) frontiers shown alongside those for logic-shrunk implementations with (initial) LUT size $K = 2$ (a, ——), 4 (b, ——) and 5 (c, ——). Pruned ReBNet data (□ ·····) are also present. All frontiers are overlaid in (d). Arrows indicate the area decrease between the best-performing LUTNet and logic-shrunk implementations with accuracy bounded within ±0.3 pp of unpruned ReBNet's (◁).

These experiments suggest that the performance of logic-shrunk networks is more sensitive to the tuning of node sparsity θ than LUT input sparsity δ. Figure 4 contains design points with θ ranging from 91.0 to 98.0% and δ in the range 0.0–87.5%. We can see that a 7 pp change in θ has a larger impact on area-accuracy behavior than a change in δ more than 10× in magnitude. We thus recommend that θ be fine-tuned with $\delta = 0$ prior to increasing δ with fixed θ. We have found $\delta = 75\%$ to be a reasonable starting point.

5.4.2 Pareto-Optimality Analysis. Figures 4a–4c feature the data points taken from Figures 5a–5c with the addition of Pareto-optimal frontiers for the LUTNet (○ - - -) and logic-shrunk (△ ——) implementations with identical (initial) LUT size K. For reference, points for the pruned ReBNet

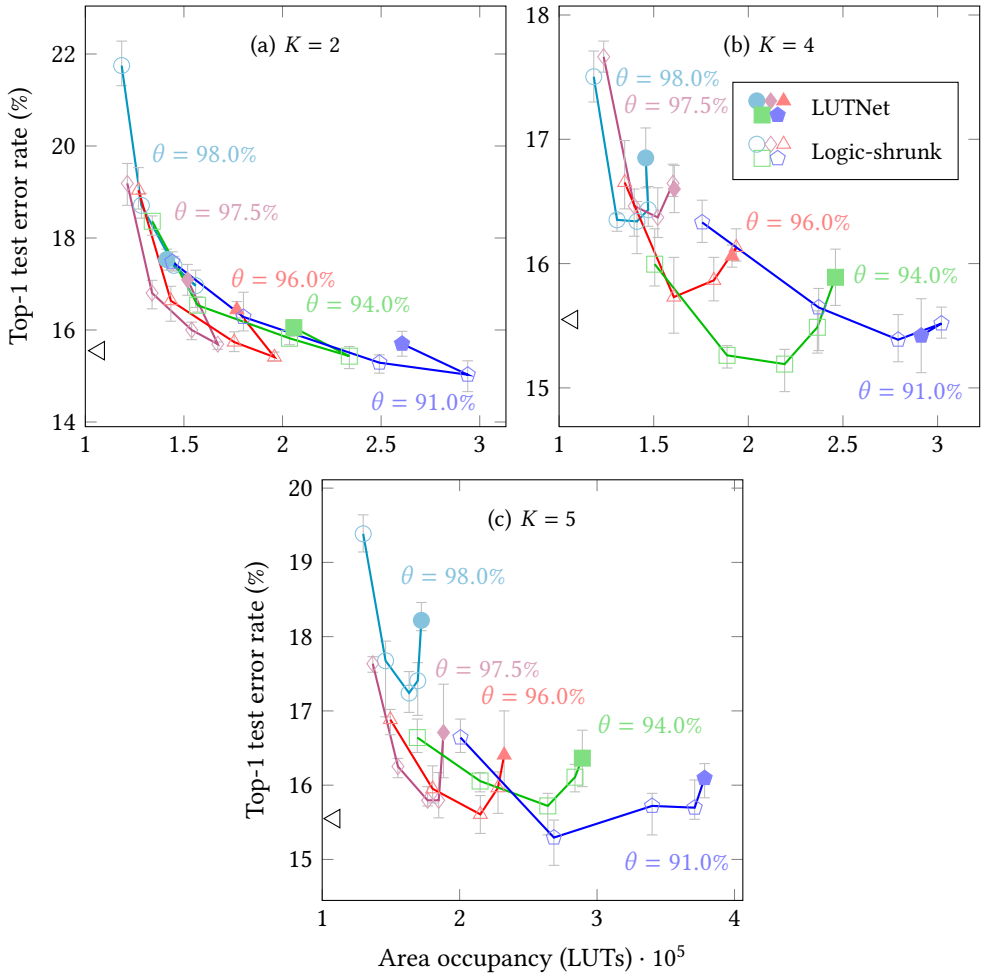


Fig. 5. Area-accuracy tradeoff for LUTNet (●♦▲■⬟) and logic-shrunk (○◇△□⬠) implementations of the CNV network classifying the CIFAR-10 dataset with (initial) LUT size $K = 2$ (a), 4 (b) and 5 (c). Each color/shape reflects a distinct node sparsity θ. Along a given curve, each logic-shrunk point is representative of a different LUT input sparsity δ. The reference accuracy—that for unpruned ReBNet—is annotated on each y-axis (◁).

implementations used as starting points for logic expansion are also included (□ ·····). From these plots, we can quickly establish that logic shrinkage facilitates a significant area improvement—savings of up to 1.76× while remaining bounded within ±0.3 pp of the unpruned ReBNet accuracy—over LUTNet. As K increases, the area gap between LUTNet and logic-shrunk designs increases, indicating that netlists of fixed-K-LUTs with higher K are more redundant. Since logic shrinkage removes this redundancy, we would expect implementations with differing initial K reaching comparable accuracy to be similar in size. We explore this hypothesis in Figure 6, in which the pairs of Pareto-optimal LUTNet and logic-shrunk implementations that resulted in the savings marked by dashed lines in Figure 5 are featured. As expected, the area of the logic-shrunk designs is relatively stable.

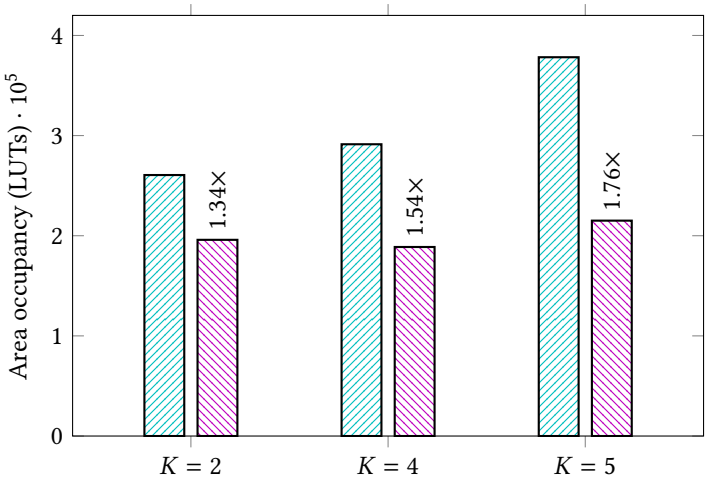


Fig. 6. Post-implementation LUT requirements of the best-performing LUTNet (▨) and logic-shrunk (▨) implementations from Figure 4 with accuracy ±0.3 pp from that of unpruned ReBNet. Annotations indicate LUT decreases.

In Figure 4d, we overlay the frontiers across all K taken from Figures 4a–4c. The LUTNet frontier (- - -) in Figure 4d captures all Pareto-optimal LUTNet points from the preceding subfigures. In comparison to that for pruned ReBNet (·····), its placement demonstrates the significant area efficiency gain when moving from XNOR- to LUT-based networks for deployment on FPGAs. However, with logic shrinkage, we go further: all three logic-shrunk frontiers reflect improvement over LUTNet, with that using $K = 4$ as the starting point (——) performing the most favorably. While logic-shrunk implementations with initial $K = 5$ exhibit the greatest area savings over LUTNet, those with $K = 4$ have the best area-accuracy tradeoff. The superiority of designs with initial $K = 4$ can be attributed to the presence of 5-LUTs within those logic-shrunk from a netlist with $K = 5$. The LUTs physically present in the target device are 6-LUTs, each capable of implementing either a single six-input function or two k-input functions with at least five (for $k = 5$), three ($k = 4$) or one ($k = 3$) shared inputs. There is less opportunity for packing of pairs of 5-LUTs than with LUTs taking four inputs or fewer, hence the lower area efficiency of designs logic-shrunk from the starting point with $K = 5$. We thus recommend $K = 4$ as the starting point for exploration with new benchmarks.

5.4.3 Comparison to Random Pruning. To verify that logic shrinkage is an efficient sparsification method, we compared it against random LUT input pruning as a sanity check. The process for this was identical to that for logic shrinkage, but LUT inputs were removed at random. Our results for this set of experiments are shown in Figure 7. As evidenced by their Pareto fronts, logic-shrunk (△ ——) implementations consistently outperformed those with random pruning (⬠ ——), the former achieving a 1.50× area saving *vs* the latter at the unpruned ReBNet accuracy (◁).

5.4.4 LUT Distribution. In order to better understand the source of our area savings, we inspected the post-shrinkage distribution of LUT sizes K'_n for each LUT n in both pre-and post-synthesis netlists. To facilitate our investigation, we disabled design hierarchy optimization in Vivado, preventing the synthesis engine from flattening across modules. Table 3 shows the breakdown in LUT sizes across the implementations shown in Figure 4 with (initial) LUT size $K = 4$ and node

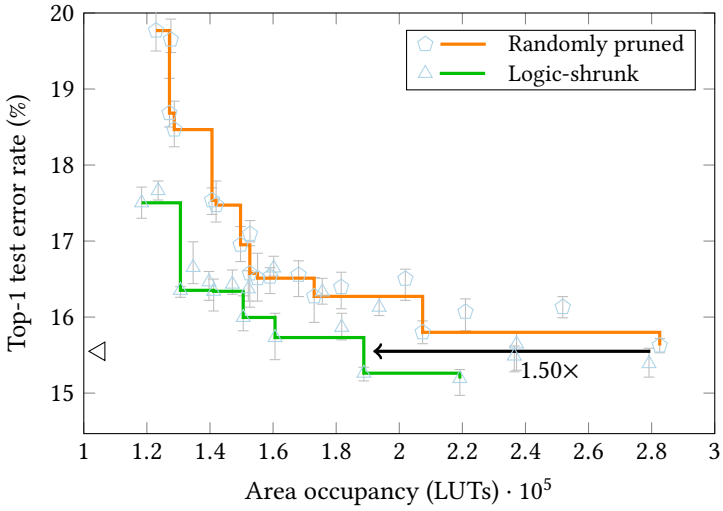


Fig. 7. Area-accuracy tradeoff for randomly pruned (⬠) and logic-shrunk (△) CNV implementations trained to classify CIFAR-10 with initial LUT size $K = 4$. Each point reflects a distinct LUT input sparsity δ. Pareto frontiers for logic-shrunk (——) and randomly pruned (——) designs are overlaid for comparison. The annotated arrow indicates the area saving between the best-performing implementations with accuracy ±0.3 pp from unpruned ReBNet's (◁).

Table 3. Pre-and post-synthesis LUT size distributions for the LUTNet (■) and logic-shrunk (□) implementations with (initial) LUT size $K = 4$ and node sparsity $\theta = 94\%$ reported in Figure 5b. Shaded cells mark the post-synthesis LUT size in the majority.

LUT input sparsity δ (%)	Top-1 test error rate (%)	Σ LUTs		4-LUTs		3-LUTs		2-LUTs		1-LUTs	
		Pre	Post	Pre	Post	Pre	Post	Pre	Post	Pre	Post
0.0	15.89	70 778	53 430	70 778	49 541	0	3654	0	233	0	2
25.0	15.49	70 778	40 209	29 758	24 405	20 908	13 629	10 466	1852	9646	323
50.0	15.18	70 778	21 451	2642	2293	18 130	12 289	26 592	6375	23 414	494
75.0	15.26	62 518	3212	0	0	998	816	6264	1708	55 256	688
87.5	16.00	35 262	945	0	0	6	2	116	32	35 140	911

sparsity $\theta = 94.0\%$ (■ □) as an example. The implementation with LUT input sparsity $\delta = 0$ is the LUTNet design; all of those with $\delta > 0$ were logic-shrunk from that starting point. Pre-synthesis netlists were those generated as output from the logic shrinkage (or vanilla LUTNet) toolflow, while post-synthesis netlists were extracted from Vivado before implementation.

Two key features are apparent from the data in Table 3. Firstly, there is a downward (towards high sparsity) and rightward (small LUTs) shift in LUT counts. Diminishing returns are seen when increasing K in LUTNet architectures [29], indicating that the inputs added with higher K tend to be of decreasing value. These are generally severed first, making it increasingly unlikely that all inputs of large LUTs will remain unpruned as δ rises. As a result, we see that larger LUTs are usually reduced in size before smaller ones, giving rise to the reduction in majority LUT size with increasing δ highlighted with shading. We can also infer from these data, along with reference back to Figure 4, that equally sparse designs perform better under logic shrinkage than when

Table 4. Top-1 test error rate and area—post-synthesis and post-implementation—for LUTNet and logic-shrunk designs with various models classifying various datasets. (Initial) LUT size K was 4 in all cases.

Dataset (network)	Architecture	Node sparsity θ (%)	LUT input sparsity δ (%)	Error rate		Area (post-synth.)		Area (post-impl.)	
				%	Δ (pp)	LUTs	Δ (× ↓)	LUTs	Δ (× ↓)
MNIST (LFC)	LUTNet	99.9	–	2.13	–	62 919	–	58 192	–
	Logic-shrunk	90.0	75.0	2.53	0.40	63 928	0.98	54 647	1.06
SVHN (CNV)	LUTNet	95.0	–	3.80	–	201 644	–	154 814	–
	Logic-shrunk	98.0	75.0	3.75	−0.05	179 236	1.13	137 610	1.13
CIFAR-10 (CNV)	LUTNet	91.0	–	15.42	–	339 479	–	291 349	–
	Logic-shrunk	94.0	75.0	15.26	−0.16	220 060	1.54	188 765	1.54
ImageNet (Bi-Real-18)[1]	LUTNet	30.0	–	45.13	–	1 840 666	–	–[2]	–
	Logic-shrunk	30.0	75.0	46.60	1.47	690 357	2.67	665 720	–

[1] Target layer only. Designs for other datasets included all network layers.
[2] Design could not fit onto target device.

constructed using the vanilla LUTNet flow. For the same θ, logic shrinkage with initial $K = 4$ and $\delta = 0.5$ generates a netlist with the same number of total LUT inputs as a LUTNet design with $K = 2$. However, the logic-shrunk implementation has an error rate of 15.18% (Table 3): lower than all LUTNet designs with $K = 2$ (Figure 4). It is thus evident that selectively shrinking to a smaller implementation from a larger one through consideration of LUT input salience is preferable to the creation of an equally sized architecture from scratch.

Secondly, there are large gaps between pre-and post-synthesis LUT counts, with this phenomenon becoming more pronounced as δ increases. This is attributable to the logic optimization central to synthesis, opportunities for which increase as LUT size falls. The effects of optimization are particularly marked for 1-LUTs, the majority of which were optimized away. Three of the four possible functions performable by a 1-LUT ($y = 0$, $y = 1$, $y = x$) are free to implement. Only $y = \overline{x}$ requires device resources, but in most cases can be absorbed by the downstream logic. Consequently, we see increasing LUT removal as the average LUT size decreases. Overall, we can conclude that logic shrinkage successfully promotes sparsity in such a way as to suit the optimizations performed during synthesis, resulting in highly area-efficient implementations.

5.4.5 Other Benchmarks. We also benchmarked logic shrinkage using other popular datasets and models: MNIST (with LFC), SVHN (with CNV) and ImageNet (with Bi-Real-18). Table 4 shows the post-synthesis and post-implementation LUT requirements of each of these model-dataset combinations when implemented with LUTNet and logic-shrunk architectures with (initial) LUT size $K = 4$. The same layers for all pairs of designs were unrolled and pruned, with the node sparsity (and LUT input sparsity) tuned in an effort to keep their accuracy as close as possible.

For CNV classifying CIFAR-10, our use of logic shrinkage saw an area reduction of 1.54×. With the smaller datasets, the gains realized via logic shrinkage were less pronounced. The SVHN-CNV and MNIST-LFC combinations are more tolerant of sparsity, thus the majority of nodes in these networks were able to be removed prior to logic expansion. This left relatively little room for further improvement by logic shrinkage. Despite this, we still achieved area reductions of around 10% for these simpler tasks. For ImageNet on Bi-Real-18, the LUTNet layer was too large to fit our target FPGA, the XCVU9P (1182240 LUTs). Logic shrinkage with node and LUT input sparsity of 30% and 75%, respectively, saw its post-synthesis area reduced by 2.67×, thus leading to success in implementation.

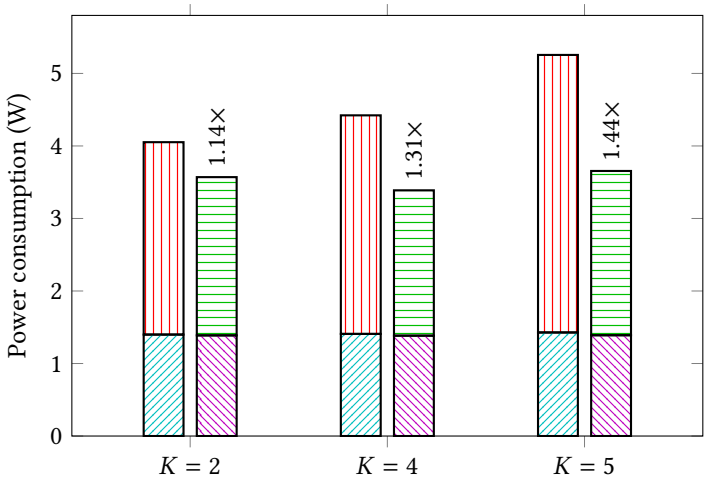


Fig. 8. Post-implementation power consumption estimates for the LUTNet (▨ ▢) and logic-shrunk (▧ ▤) designs in Figure 6. Power is broken into static (▨ ▧) and dynamic (▢ ▤) components. Annotations reflect the decrease in total power between each pair of implementations.

5.5 Energy Efficiency

We also sought to quantify the energy efficiency impact attributable to logic shrinkage. To do so, we obtained power consumption estimates of both LUTNet and logic-shrunk implementations using the AMD Power Analyzer (XPA) tool with default settings: vectorless mode and 12.5% primary input switching probability. The resultant power estimates, for the same designs as captured in Figure 6, are shown in Figure 8. All were obtained post-placement and -routing. Power consumption is equivalent to energy efficiency here since all implementations have identical throughput.

Since dynamic power consumption is directly related to area occupancy, Figures 6 and 8 show similar trends. The static power remains consistent across all implementations. Overall, it can be concluded that the significant area reductions of logic shrinkage also result in energy efficiency improvements.

5.6 Training Efficiency

Logic shrinkage introduces additional matrix-vector multiplications for every forward propagation during retraining in order to ensure that pruned inputs remain severed. Thanks to the highly optimized linear algebra routines provided by GPUs, the slowdown in training speed with logic shrinkage is minor. This is evident in Figure 9, in which we capture per-epoch logic shrinkage overheads.

6 APPLICATION SHOWCASE

With the benefits of logic shrinkage demonstrated using standard image classification benchmarks, we moved to further verify the generality and flexibility of our approach using a topical, real-world application: real-time face mask detection. The implementations described in this section represent the first fully functional end-to-end deployments of the ReBNet, LUTNet and logic-shrunk architectures on real devices. Through this section, we demonstrate with ample details on how users can drop-in implement logic shrinkage with an arbitrarily selected machine learning application, and observe instant and hardware-verifiable inference performance boost.

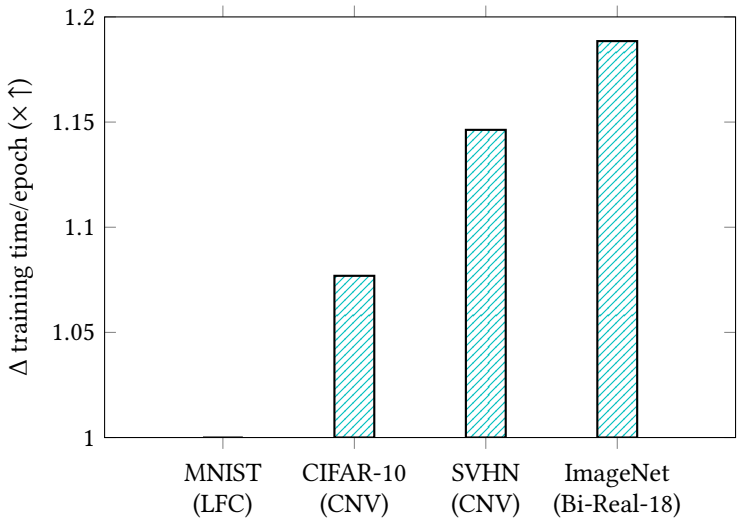


Fig. 9. Training time increase for the logic-shrunk designs over LUTNet taken from Table 4.

6.1 Context

Interest in the automated detection of face mask wearing peaked at the onset of the COVID-19 pandemic [22], when many governments imposed rules requiring their use to limit the spread of the virus. Manually monitoring compliance with such rules is infeasible in locations with high population density, and the problem becomes more challenging when trying to determine *correct* wearing, *i.e.* complete covering of the nose, mouth and chin. It is natural to cast face mask detection as an image classification problem, for which convolutional neural networks (CNNs) are known to perform particularly well [11].

Agrawal *et al.* presented cloud-based classification for a range of personal protective equipment, including face masks [1]. However, reliance on network transmission and remote processing raise data protection concerns, particularly for public deployment. Wang *et al.* [34] and Hammoudi *et al.* [8] presented detectors running on personal computers and mobile phones, but these implementations require users to self-initiate them; passive and continuous surveillance are not possible. Nvidia performed face mask detection on 960×544 input images using ResNet-18 with eight-bit fixed-point and half-precision floating-point data [12]. ResNet-18 is a large model, however, and even eight bits a high precision in the context of edge inference. On a Jetson Nano embedded GPU board, which typically within a power envelope of 10 W, throughput was limited to 21 classifications per second (cl/s). Operation at 508 cl/s was shown by moving to a Jetson AGX Xavier board, but this came at the cost of increasing power consumption to 25 W. Moreover, the authors only predicted the presence of face masks on faces; they were not able to discern *correctness* of wearing.

Many implementations of low-precision CNNs with high classification rates and energy efficiency using FPGAs and application-specific integrated circuits can be found in the literature [32]. Fasfous *et al.*'s BinaryCoP is a low-power BNN-based classifier for correct face mask wearing and positioning [5]. The authors targeted an AMD PYNQ-Z1 development board, which features an embedded-scale Zynq device, achieving up to 6400 cl/s while consuming 2 W of power. Such throughput is high enough to support real-time classification using multiple cameras. These attributes led us to select BinaryCoP as our showcase application.

6.2 BinaryCoP

6.2.1 Dataset. Mask-wearing during the COVID-19 pandemic presented researchers with an opportunity to collect image data suitable for model training. Ge *et al.* released one of the first such datasets, assembled from real photographs of people wearing masks, but its scale makes it unsuited to the training of large networks [6]. Wang *et al.* synthetically generated mask-wearing samples by drawing masks onto existing images taken from natural face datasets [33]. MaskedFace-Net, presented by Cabani *et al.*, improved on the quality of existing synthetic datasets using facial key-point matching, which enables the generation of deformable face mask overlays [2]. The latter adds flexibility to the generation process, allowing the creation of images with parts of the face (chin, nose, mouth, *etc.*) left exposed. MaskedFace-Net is split into two subsets: correctly and incorrectly masked faces.

Fasfous *et al.* used MaskedFace-Net for BinaryCoP, but split the latter subset in three, resulting in a total of four detection classes [5]:

(1) correctly masked face, with full coverage of the nose, mouth and chin;
(2) incorrectly masked face with uncovered chin;
(3) incorrectly masked face with uncovered mouth and nose; and
(4) incorrectly masked face with uncovered nose.

Larger classes were randomly sampled such that the size of all classes were approximately equal. The 138486 images in the resulting balanced dataset were then randomly augmented with a combination of contrast and brightness balance, Gaussian noise injection and flip and rotate operations, resized to 32×32, and split into (∼110000) training and (∼28000) test samples.

The augmented dataset was not available in the BinaryCoP repository[2] at the time of writing, but we are grateful to the authors for sharing this with us privately. We did not receive separate training and test sets, so performed our own random sampling to create these according to the aforementioned proportions.

6.2.2 Network Description. BinaryCoP's authors used CNV, along with two successively slimmed-down versions they proposed named n-CNV and μ-CNV, as their network models [5]. When implemented using the FINN architecture [28], they reported top-1 test accuracy of 98.10%, 93.94% and 93.78% on their augmented MaskedFace-Net dataset for CNV, n-CNV and μ-CNV, respectively.

We chose to use μ-CNV for our implementations. Since (logic-shrunk) LUTNet layers must be unrolled, use of the smallest model gave us the greatest scope for design-space exploration. Table 5 shows the μ-CNV model along with the folding factors for each layer. These designate the amount of parallelism that exists across output (number of processing elements, denoted "PE") and input (number of single-instruction multiple-data lanes, "SIMD") channels, respectively. The iteration interval (II) of a given layer decreases with PE $\times$ SIMD; unrolled layers have an II of 1.

6.3 Implementation

We recreated μ-CNV using the ReBNet, LUTNet and logic-shrunk architectures. While Fasfous *et al.* used FINN for their implementations [5], we chose to use ReBNet as our baseline for consistency with the experiments in Section 5 and since the latter generally outperforms the former.

During hardware verification, we found minor errors in ReBNet's official GitHub release, which our work depends on [7]. We fixed them and verified the hardware design generated from both ReBNet and logic shrinkage. We have included the corrected version of ReBNet in our github release, so as to facilitate the community reproduction of our work.

[2]https://github.com/NaelF/BinaryCoP

Table 5. μ-CNV network model proposed by Fasfous *et al.* [5]. $\text{Conv}_{x,\,y,\,z}$ 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. $\text{MaxPool}_{x,\,y}$ is an $x \times x$ maximum-pooling layer with stride y, and BatchNorm and SoftMax are batch normalization and normalized exponential layers, respectively. The number of PEs and SIMD lanes for SoftMax are omitted since this layer does not need to be implemented for inference.

Layer	Folding factor	
	PE	SIMD
$\text{Conv}_{16,\,3,\,1}$, BatchNorm	4	3
$\text{Conv}_{16,\,3,\,1}$, BatchNorm	4	16
$\text{MaxPool}_{2,\,2}$	1	1
$\text{Conv}_{32,\,3,\,1}$, BatchNorm	4	16
$\text{Conv}_{32,\,3,\,1}$, BatchNorm	4	32
$\text{MaxPool}_{2,\,2}$	1	1
$\text{Conv}_{64,\,3,\,1}$, BatchNorm	1	32
FConn_{128}, BatchNorm	1	16
FConn_{4}, BatchNorm	1	1
SoftMax	–	–

6.3.1 Target Platforms. We targeted AMD's PYNQ platforms, implementing our designs as PYNQ "overlays". The Zynq FPGAs on PYNQ boards feature embedded, hardened Arm cores that run Linux with a Web server hosting Jupyter Notebooks used for configuring, communicating with, and commanding user circuitry in soft logic. We implemented a Python dynamic library to initiate the execution of each inference job. Our high-level, Jupyter Notebook-based interface reports run statistics including classification and error rates in real time, and imposes no requirement on users to have exposure to the back-end C or Verilog codebases. This setup enables easy and rapid deployment and evaluation on real devices.

In common with Fasfous *et al.*, we used the PYNQ-Z1 board as our primary verification and evaluation platform. This features a Zynq XC7Z020 FPGA with 53200 LUTs. To give us more room for pruning parameter tuning, we also targeted a larger, 242400-LUT Kintex UltraScale XCKU040.

6.3.2 Data Preprocessing. Following the protocols proposed by Umuroglu *et al.* [28] and Fasfous *et al.* [5], we performed a series of preprocessing steps on the test set before using it for on-board inference. Each image was converted from JPEG to a series of raw red-green-blue (RGB) pixels, these were scaled by mapping $[0, 255] \rightarrow [-1, {}^{127}/_{128}]$, and the pixels were finally packed into a 64-bit-wide packet stream. Unlike the aforementioned authors, we performed image conversion in TensorFlow rather than using the Python Imaging Library, facilitating verification by reducing inconsistency between hardware and software behaviors.

6.3.3 Data Movement. We implemented our top-level architecture in the same style as FINN's, in which data movement is managed by direct memory access cores that stream data into and out of the network using AXI-Stream interfaces [28]. Each RGB image is streamed in sequentially as 384 64-bit packets. Again following Umuroglu *et al.*'s approach, we read 16 images at a time whenever at least this many are available in order to make good use of the available bandwidth.

6.3.4 Target Layer Selection. For logic expansion and subsequent shrinkage, we targeted the final convolutional layer of μ-CNV. With 32 input and 64 output channels, this layer is the largest; it thus has the greatest potential to demonstrate the advantages of our approach. Our primary baseline was ReBNet with the target layer unrolled and pruned to the same level as LUTNet.

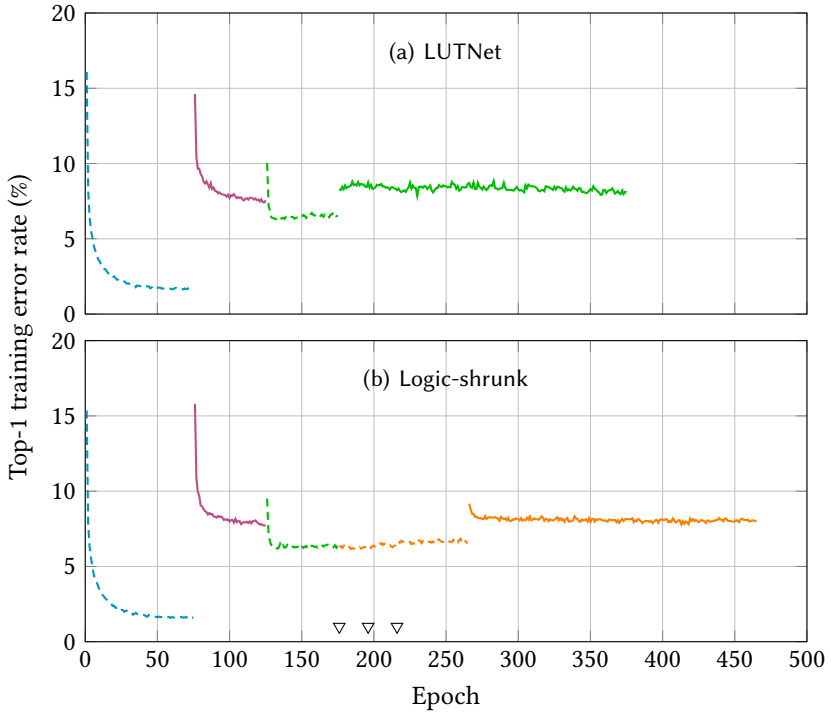


Fig. 10. Training error for μ−CNV classifying the augmented MaskedFace-Net dataset using LUTNet (a) and logic-shrunk (b) architectures, during initial training (----), post-node pruning retraining (——), post-logic expansion retraining (----) and post-logic shrinkage retraining (----). Phases with binarized forward propagation are denoted with solid lines; those with high-precision (`float32`) forward propagation are shown dashed. Annotations (▽) mark epochs at which logic shrinkage was applied.

6.3.5 Verification. We performed both layer-level unit testing for the ReBNet, LUTNet and logic-shrunk architectures by co-simulating our implementations in Vivado HLS, ensuring that their results matched those from TensorFlow. We also verified complete system behavior, again for all three architectures, on the PYNQ-Z1 board by running inference on the whole test set.

6.4 Evaluation

6.4.1 Training Specifics. We trained all of our implementations on the augmented MaskedFace-Net dataset. Our choices of training phase duration matched those for CNV classifying CIFAR-10 described in Section 5.3.1 with a few exceptions based on our observations of training error saturation. As reflected in Figure 10, we dropped our initial `float32` training (----) from 200 to 75 epochs. We equalized the LUTNet and logic-shrunk post-logic expansion retraining by inserting a 50-epoch, `float32` phase (----) after each. Following each of the three increasingly aggressive logic shrinkage steps, we retrained for 20 epochs, as before, but extended the final `float32` retraining phase (----) from 20 to 50 epochs. The initial LUT size K was set to 4 since it proved to be a good starting point for the experiments discussed in Section 5.4.

6.4.2 Area Efficiency. Table 6 captures the results of our analysis of the competing architectures. To start with, we noted that our recreation of μ-CNV using the ReBNet architecture, rather than FINN, for all convolutional and fully connected layers achieved a 1.77 pp boost over the 93.78% top-1 test

accuracy reported by Fasfous *et al.* [5]. Note that for both of our target devices, we unrolled the target layer, including for ReBNet. This allowed for direct comparison between all architecture types since their frequency, throughput and latency are identical. Further note that, for ReBNet, unrolling has no accuracy impact.

The size of the PYNQ-Z1's FPGA, the XC7Z020, makes it challenging to accommodate unrolled layers. Indeed, as shown in Table 6, our ReBNet baseline came nowhere close to fitting; we had to prune the target layer to a node sparsity δ of 95% in order to arrive at an implementable design. This aggressiveness in pruning led to a 4.81 pp accuracy degradation, placing it 3.04 pp below the FINN-based equivalent. We found that replacement of the target layer with a LUTNet version allowed us to regain 1.70 pp of that loss at the cost of a small (∼3%) area increase. Proceeding to logic-shrink the LUTNet layer with a LUT input sparsity of $\delta = 50\%$ led to a further 0.71 pp accuracy recovery while nullifying nearly all of the aforementioned resource cost. Overall, with logic shrinkage, we bettered ReBNet's accuracy by 2.57 pp with negligible (∼0.1%) area overhead. This phenomenon—of increasing accuracy despite reducing network complexity—was also observed in Figure 5 in cases of low δ.

For the XC7Z020, we could not further reduce area without harming accuracy by pushing δ beyond 50%. This is because the choice of θ required to allow pruned ReBNet to fit (95%) leaves little room for logic shrinkage to have beneficial effects. Owing to this, we then moved to the larger XCKU040 device, which allowed us to reduce θ to a more favorable 60%. Here, ReBNet performs much better, degrading by only 1.20 pp versus its unpruned counterpart. We found almost all (1.12 pp) of this drop to be recoverable by moving to the LUTNet architecture, but this comes at the more significant cost of an ∼8% area increase. This increase is larger than the equivalent observed for the XC7Z020 since the node density was 8× that ($\theta = 60\%$ rather than 95%) of the target layer on the XCKU040. Unlike unrolled ReBNet's XNOR gates, LUTNet's inference nodes cannot in general be absorbed by the adder trees that follow them, and with higher density this effect becomes more pronounced. However, we were able to recoup all of this overhead, and more, by logic-shrinking the LUTNet layer with an aggressive $\delta = 87.5\%$. The end result was a logic-shrunk network that performed equivalently to ReBNet, with the target layer unrolled and pruned, in terms of accuracy while consuming ∼8% fewer resources. We see much more benefit for this choice of δ than for the 50% used with the XC7Z020. High LUT input sparsity not only allows more opportunity for inference node packing but also, and usually more significantly, results in reductions to the number of inputs per adder tree.

6.4.3 Latency. We measured the end-to-end inference latency of the XC7Z020-based logic-shrunk implementation as shown in Table 6. With a batch size of one, our implementation can inference at a latency of 1.70ms per image. On an Nvidia RTX3090 GPU, we measured the inference latency of the same network to be 0.82ms per image with a batch size of 100. While not exactly an apple-to-apple comparison in terms of batch size, our implementation on a low-end FPGA device, priced at around $170 to date, is able to perform inference at a comparable speed as a high-end GPU.

7 LIMITATIONS

While logic shrinkage implementations typically reach higher logic density than XNOR-based BNNs and LUTNet, our proposal's greatest current limitation is that it requires full unrolling of the target layers due to lacking support for time multiplexing. While this may be acceptable in deployment scenarios where throughput and energy efficiency are of paramount importance, it nevertheless limits the scalability of our proposal.

We previously showed that time-multiplexing could be introduced to LUTNet by sacrificing some LUT inputs to enabling tiling by switching in inference operator behaviour over each clock

Table 6. Top-1 test error rate and area—post-synthesis and post-implementation—for ReBNet, LUTNet and logic-shrunk designs with μ-CNV classifying the augmented MaskedFace-Net dataset.

Device	Architecture	Node sparsity θ (%)	LUT input sparsity δ (%)	Error rate		Area (post-synth.)		Area (post-impl.)	
				%	Δ (pp)[1]	LUTs	Δ (× ↓)[1]	LUTs	Δ (× ↓)[1]
XC7Z020	ReBNet	–	–	4.45	–	96 435	–	–[2]	–
	ReBNet	95.0	–	9.26	–	44 398	–	39 188	–
	LUTNet	95.0	–	7.40	−1.86	45 658	0.97	40 419	0.97
	Logic-shrunk	95.0	50.0	6.69	−2.57	44 455	1.00	39 214	1.00
XCKU040	ReBNet	–	–	4.45	–	73 693	–	68 878	–
	ReBNet	60.0	–	5.65	–	60 533	–	55 708	–
	LUTNet	60.0	–	4.53	−1.12	65 562	0.92	61 611	0.90
	Logic-shrunk	60.0	87.5	5.66	0.01	56 090	1.08	51 357	1.08

[1] Versus pruned ReBNet.
[2] Design could not fit on target device.

cycle [30]. We will explore the impact of introducing time-multiplexing to logic shrinkage in our future work.

Modern FPGA clusters feature high-throughput inter-FPGA links, enabling the mapping of networks across multiple FPGA boards without going through external memory. These clusters could be ideal platforms to deploy our work, in which the resource consumption requirement of logic shrinkage is less of a concern. We will explore this in our future work.

8 CONCLUSION

In this article, we introduced logic shrinkage: the automated search for, and implementation of, LUT-based neural network inference accelerators in which LUT sizes and inputs are learned during training. We showed our realization of logic shrinkage to be lightweight and to result in the production of netlists that well suit the logic optimizations performed by FPGA synthesis tools. We analyzed hundreds of experimental results, finding significant area-accuracy tradeoff improvement over homogeneous LUT-based networks. We validated the generality of logic shrinkage using a topical machine learning application—face mask detection—on real devices, and achieved higher accuracy than the state-of-the-art BNN baseline.

The authors of prior NAS works pursued a top-down approach, learning an intermediate representation—a network topology—while leaving its hardware mapping as a separate task. In contrast, we propose a bottom-up, hardware-aware alternative: directly learning a netlist as the topology, with the flexibility of the target platform exposed to the training process. We chose to focus on the search for efficient node functions in this work, but in the future we will extend our scope to other components—accumulators, activation functions, *etc.*—in order to allow for greater structural learning by the training software and to further drive up the area and energy efficiency of the resulting inference engines.

ACKNOWLEDGMENTS

The authors are grateful for the support of the United Kingdom EPSRC (grant numbers EP/S030069/1 and EP/P010040/1).

REFERENCES

[1] Tushar Agrawal, K Imran, Matteo Figus, and C Kirkpatrick. 2020. *Automatically Detecting Personal Protective Equipment on Persons in Images Using Amazon Rekognition.* https://aws.amazon.com/cn/blogs/machine-learning/automatically-detecting-personal-protective-equipment-on-persons-in-images-using-amazon-rekognition/

[2] Adnane Cabani, Karim Hammoudi, Halim Benhabiles, and Mahmoud Melkemi. 2021. MaskedFace-Net–A Dataset of Correctly/incorrectly Masked Face Images in the Context of COVID-19. *Smart Health* 19 (2021), 100144.

[3] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2018. Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges. *IEEE Signal Processing Magazine* 35, 1 (2018).

[4] Jia Deng, Wei Dong, Richard Socher, Jia Li, Kai Li, and Feifei Li. 2009. ImageNet: A Large-scale Hierarchical Image Database. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[5] Nael Fasfous, Manoj-Rohit Vemparala, Alexander Frickenstein, Lukas Frickenstein, Mohamed Badawy, and Walter Stechele. 2021. BinaryCoP: Binary Neural Network-based COVID-19 Face-mask Wear and Positioning Predictor on Edge Devices. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.

[6] Shiming Ge, Jia Li, Qiting Ye, and Zhao Luo. 2017. Detecting Masked Faces in the Wild with LLE-CNNs. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[7] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. 2018. ReBNet: Residual Binarized Neural Network. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*.

[8] Karim Hammoudi, Adnane Cabani, Halim Benhabiles, and Mahmoud Melkemi. 2020. Validating the Correct Wearing of Protection Mask by Taking a Selfie: Design of a Mobile Application" CheckYourMask" to Limit the Spread of COVID-19. (2020).

[9] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Network. In *Conference on Neural Information Processing Systems*.

[10] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images.* Master's thesis. University of Toronto.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Conference on Neural Information Processing Systems*.

[12] Amey Kulkarni, Amulya Vishwanath, and Chintan Shah. 2020. Implementing a Real-time, AI-based, Face Mask Detector Application for COVID-19. *NVIDIA Developer Blog* 13 (2020).

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* (1998).

[14] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. 2018. SNIP: Single-Shot Network Pruning Based on Connection Sensitivity. In *International Conference on Learning Representations*.

[15] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. 2018. FP-BNN: Binarized Neural Network on FPGA. *Neurocomputing* 275, C (2018).

[16] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations*.

[17] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. 2018. Bi-Real Net: Enhancing the Performance of 1-bit CNNs with Improved Representational Capability and Advanced Training Algorithm. In *European Conference on Computer Vision*.

[18] Jieru Mei, Yingwei Li, Xiaochen Lian, Xiaojie Jin, Linjie Yang, Alan Yuille, and Jianchao Yang. 2019. AtomNAS: Fine-Grained End-to-End Neural Architecture Search. In *International Conference on Learning Representations*.

[19] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *International Conference on Learning Representations*.

[20] Mahdi Nazemi, Ghasem Pasandi, and Massoud Pedram. 2018. NullaNet: Training Deep Neural Networks for Reduced-Memory-Access Inference. *arXiv preprint arXiv:1807.08716* (2018).

[21] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. 2011. Reading Digits in Natural Images with Unsupervised Feature Learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

[22] World Health Organization. 2020. *Archived: WHO Timeline - COVID-19.* https://www.who.int/news/item/27-04-2020-who-timeline---covid-19

[23] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized Evolution for Image Classifier Architecture Search. In *AAAI Conference on Artificial Intelligence*.

[24] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. 2021. A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions. *Comput. Surveys* 54, 4 (2021).

[25] Abigail See, Minh-Thang Luong, and Christopher D. Manning. 2016. Compression of Neural Machine Translation Models via Pruning. In *SIGNLL Conference on Computational Natural Language Learning*.

[26] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017).

[27] Yaman Umuroglu, Yash Akhauri, Nicholas J. Fraser, and Michaela Blott. 2020. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. In *International Conference on Field-Programmable Logic and Applications*.

[28] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip H. W. Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[29] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. 2019. LUTNet: Rethinking Inference in FPGA Soft Logic. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*.

[30] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. 2020. LUTNet: Learning FPGA Configurations for Highly Efficient Neural Network Inference. *IEEE Trans. Comput.* 69, 12 (2020).

[31] Erwei Wang, James J. Davis, Georgios-Ilias Stavrou, Peter Y. K. Cheung, George A. Constantinides, and Mohamed Abdelfattah. 2022. Logic Shrinkage: Learned FPGA Netlist Sparsity for Efficient Neural Network Inference. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[32] Erwei Wang, James J. Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter Y. K. Cheung, and George A. Constantinides. 2019. Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going. *Comput. Surveys* 52, 2 (2019).

[33] Zhongyuan Wang, Guangcheng Wang, Baojin Huang, Zhangyang Xiong, Qi Hong, Hao Wu, Peng Yi, Kui Jiang, Nanxi Wang, Yingjiao Pei, et al. 2020. Masked Face Recognition Dataset and Application. *arXiv preprint arXiv:2003.09093* (2020).

[34] Zekun Wang, Pengwei Wang, Peter C Louis, Lee E Wheless, and Yuankai Huo. 2021. Wearmask: Fast In-browser Face Mask Detection with Serverless Edge Computing for COVID-19. *arXiv preprint arXiv:2101.00784* (2021).

[35] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*.

Received 25/09/2022; revised 02/12/2022; accepted 26/01/2023