3PXNet: Pruned-Permuted-Packed XNOR Networks for Edge Machine Learning

WOJCIECH ROMASZKAN, University of California Los Angeles, United States
TIANMU LI, University of California Los Angeles, United States
PUNEET GUPTA, University of California Los Angeles, United States

As the adoption of Neural Networks continues to proliferate different classes of applications and systems, edge devices have been left behind. Their strict energy and storage limitations make them unable to cope with the sizes of common network models. While many compression methods such as precision reduction and sparsity have been proposed to alleviate this, they don't go quite far enough. To push size reduction to its absolute limits, we combine binarization with sparsity in Pruned-Permuted-Packed XNOR Networks (3PXNet), which can be efficiently implemented on even the smallest of embedded microcontrollers. 3PXNets can reduce model sizes by up to 38X and reduce runtime by up to 3X compared with already compact conventional binarized implementations with less than 3% accuracy reduction. We have created the first software implementation of sparse-binarized Neural Networks, released as an open-source library targeting edge devices. Our library is complete with training methodology and model generating scripts, making it easy and fast to deploy.

CCS Concepts: • **Computing methodologies → Neural networks**; *Object recognition*; • **Computer systems organization → Embedded software**.

Additional Key Words and Phrases: neural networks, image recognition, embedded systems

ACM Reference Format:
Wojciech Romaszkan, Tianmu Li, and Puneet Gupta. 2019. 3PXNet: Pruned-Permuted-Packed XNOR Networks for Edge Machine Learning. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2019), 22 pages. https://doi.org/10.1145/3371157

1 INTRODUCTION

With increasing need for intelligence in Internet-of-things devices, there is a growing interest in deploying neural networks on mobile and edge devices. However, state-of-the-art deep learning models have sizes in tens or hundreds of megabytes and require millions of multiply-accumulate operations, making them impossible to use on heavily resource-constrained platforms.

To cope with this, various model compression schemes have been developed in recent years, chief among them, quantization and pruning [15]. Multiple works have shown that decreasing precision of underlying computation through quantization does not affect accuracy, while significantly improving storage and runtime [17, 25, 37, 38, 68, 77].

Authors' addresses: Wojciech Romaszkan, University of California Los Angeles, Department of Electrical and Computer Engineering, 56-125B Engineering IV Building, 420 Westwood Plaza, Los Angeles, CA, 90095-1594, United States, wromaszkan@ucla.edu; Tianmu Li, University of California Los Angeles, Department of Electrical and Computer Engineering, 56-125B Engineering IV Building, 420 Westwood Plaza, Los Angeles, CA, 90095-1594, United States, litianmu1995@ucla.edu; Puneet Gupta, University of California Los Angeles, Department of Electrical and Computer Engineering, 56-125B Engineering IV Building, 420 Westwood Plaza, Los Angeles, CA, 90095-1594, United States, puneetg@ucla.edu.

In the most extreme case of precision reduction, binarization, can be used without significant drop in accuracy [16, 19, 34, 40, 62].

Exploiting redundancy through sparsity has been studied since the advent of neural networks [20, 28], and recent work [27] has shown over 10x compression on popular network models with same accuracy. Although model compression through pruning significantly reduces the required computational complexity, it is hard to efficiently exploit, especially on highly-parallel hardware [74]. Combining binarization and pruning, is the next logical step when pushing the limit of model compression [73]. Unfortunately, naively induced sparsity makes exploiting binarization-enabled parallelism prohibitively costly. To achieve actual performance gains over dense binarized implementation, a form of structured pruning needs to be employed [54]. However, this enforced structure might in turn constrain pruning flexibility and negatively affect the accuracy of the network, which in itself is undesirable.

1.1 A Case for Sparse XNOR Networks

Table 2 shows available memory in some of the common embedded microcontroller platforms. This is usually limited to few hundred kilobytes at most. Such severe resource constraints make implementation of reasonable deep learning networks on these platforms very challenging. For example consider few network models shown in Table 1. Most floating point and even 8-bit fixed point implementations are 10-1000X off from where they need to be for these platforms.

By constraining weights and activations to binary values, Binarized Neural Networks can perform 32 multiply-accumulate operations using XNOR and population count (popcount) instructions in a 32-bit processor (with appropriate "*packing*" of weights and activations), which gives it a potential 32x storage and computation saving compared to a 32-bit implementation (see Table 1). While this in itself is impressive, it might not be enough to use common models on typical embedded development platforms. Further compression is therefore necessary to use large models on those devices, or in case of the most memory-constrained ones, make it feasible to deploy them at all.

Table 1. Weight storage requirements of different networks depending on precision.

Network	Weight Memory [MB]		
	F32	FP8	XNOR
ILSVRC VGG-D [64]	553.4	138.3	17.3
ILSVRC AlexNet [42]	227.5	56.9	7.1
MNIST MLP [19]	147.2	36.8	4.6
MNIST MLP Small (This work)	0.40	0.10	0.01
CIFAR-10 CNN [19]	56.1	14	1.7
CIFAR-10 CNN Small [45]	0.36	0.09	0.01

In this paper, we propose a Pruned-Permuted-Packed XNOR Neural Network (3PXNet) model aiming to combine binarization and pruning in a way that is computationally efficient and does not significantly degrade accuracy. We specifically target resource-constrained edge devices, and provide implementation results on a range of embedded platforms. Our pruning method allows further model size reduction and speedup compared to a binarized networks. Contributions of this work are as follows.

- We develop methods to prune binarized XNOR networks aware of the need for packing them into words for computational efficiency.

- We develop training methods for such 3PXNets and open-source the training routines using PyTorch framework [60].
- We show that 3PXNets offer some of the most compact networks with good accuracy: 3x-38x (22x-307x) size reduction versus dense binary (8-bit), with 0-5.2% (0.3-10.4%) accuracy drop on MNIST and 2.3-3.8% (3.5-5%) on Google Speech dataset, depending on the level of sparsity.
- We develop the *first* software implementation of sparse binarized networks and open-source an implementation of 3PXNets.
- We make multiple design optimizations, like loop ordering, fused kernels and implicit padding, which result in very low memory footprint and runtime. 3PXNet implementation can be as much as 3X (25X) faster and more energy efficient than dense binarized (8-bit fixed point) networks enabling real-time inference on IoT platforms.

2 RELATED WORK

2.1 Binarized Neural Networks

Reducing computational complexity of Neural Network training and inference has become a major research topic in recent years [15]. Multiple works have shown that decreasing precision of underlying computation through quantization does not affect accuracy, while significantly improving storage and runtime [17, 25, 37, 38, 68, 77]. To reduce computational complexity to its limit, researchers have proposed a variety of implementations which binarized both weights and activations [16, 19, 34, 40, 62]. Those Networks are commonly referred to as XNOR-Nets, because multiplication can be implemented using a bitwise XNOR operation.

The promise of significant performance and storage improvements given by XNOR-Nets has resulted in multiple software and hardware implementations. Umuroglu et. al. [67] have created FINN, a framework for binarized FPGA accelerators, which was further expanded to larger models by Fraser et. al. [23]. Other binarized accelerators have been proposed, both targeting FPGAs [49, 51, 72, 76], ASIC [2, 10, 18, 63], and in-memory compute [11, 36]. Yang et. al. [71] have developed BMXNet, an extension of MXNet [13] based on binarized GEMM kernel. Depth-first binarized convolution implementations have also been shown for both CPU and GPU [33, 56, 61]. Our implementation leverages the same depth-first approach to convolutional layers, while also incorporating coarse intra-kernel pruning.

2.2 Weight Pruning

Weight pruning in Neural Networks was first proposed over 20 years ago as a way of improving generalization and reducing computational complexity for both training and inference [20, 28]. Recently, Han et. al. [27] have shown over 10x compression on popular network models with no increase in error rates. By further coupling pruning with quantization and efficient coding, in a scheme called Deep Compression, they achieved up to 49x size reduction [26]. However, deploying pruned models on highly-parallel architectures has proven problematic due to storage overhead and irregular memory access patterns of sparse matrix multiplication [65, 74].

To make pruning more regular, multiple forms of "structured" pruning have been proposed. Lebedev and Lempitsky [46] proposed group-wise sparsification. Foroosh et. al. [55] hard-coded the sparsity patterns into the source code, achieving up to 6.88x speedup on CPUs. Anwar et. al. [3, 4] explored different granularities of pruning: feature map, kernel and intra-kernel and introduce kernel strided sparsity. Sredojevic et. al. [65] have proposed an algorithmic way of inducing regularity in sparse networks. Yu et. al. [74] have developed a hardware-aware pruning method called Scalpel, which matches the coarseness of pruning to the parallelism of underlying hardware. Our approach to packing is based on Scalpel, but applied to binarized models and using CPU bitwidth as packing granularity, while also permuting layer inputs to improve packing opportunities. Wang et. al. [69] have used structured sparsity in unrolled kernels after im2col conversion. Pruning has been successfully exploited in custom accelerators by using compressed storage, skipping memory accesses, gating computation and exploiting novel dataflows [14, 39, 59, 75]. Crossbar-aware pruning has also been proposed given recent emergence of analog crossbar-based accelerators [52].

2.3 Sparse Binary Networks

While traditional implementations of ternary networks can be considered as binary-sparse, they store numbers in 2-bit format and don’t skip computation, which makes it impossible to capitalize on the benefits of either binarization (XNOR multiplication) or sparsity (storage and computation reduction) [1, 30, 44, 48, 57, 73, 78]. Thus we would like to make a distinction between explicitly Ternary Networks, using 2-bit representation, and Binary-Sparse Networks, leveraging XNOR multiplication and size compression, like ours. As an example of the latter, Lin et. al. [53], exploited Singular Value Decomposition to reduce kernel sizes in BNNs. Certain software and hardware implementations rely on operand-gating XNOR multiplication [21, 31], however they still require 2-bits of information per weight: value and mask. Li and Ren [50] decomposed first-layer activations into "bit-slices" and explore pruning opportunities in those, but their scheme does not extend over the whole network. Faraone et. al. [22] have discussed implications of exploiting sparsity in binarized FPGA accelerators, but not software ones.

2.4 Machine Learning on Embedded Systems

Edge Machine Learning inference on embedded platforms has been explored in recent years as a way to remove the communication energy and latency involved in offloading it to the servers. Due to severe memory and energy constraints of such devices, various model compression techniques have been used to make such applications feasible. Compressed Neural Network models like SqueezeNet [35] and MobileNets [32] have been developed, specifically targeting low memory footprints. Lai et. al. [45], have developed CMSIS-NN, a software library for ARM Cortex-M microcontrollers with 8- and 16-bit fixed-point support. Microsoft is developing EdgeML, a Machine Learning library containing algorithms optimized for low storage, energy and latency [24, 43].

3 THE 3PXNET APPROACH

In the following sections we describe the principal components of the Pruned-Permuted-Packed XNOR Networks.

3.1 XNOR Networks

Binarized neural networks reduce network size by having only one bit for each weight, allowing multiple weights to be packed in binary vector, e.g. a processor 32-bit word. By constraining activations to binary values, they can also be packed, and 32 multiply-accumulate operations (MAC) can be performed in parallel using a bitwise XNOR and popcount instructions on the activation and weight packs in a 32-bit processor (or 64 MACs in a 64-bit processor). In theory, this can reduce weight storage and computation requirement by a factor of 32 compared to a 32-bit floating point implementation.

3.2 Challenges in pruning XNOR networks

Dense binarized or XNOR networks are relatively straightforward to pack for both weights and activations thereby getting close to the "32x" leverage [34]. However, introducing sparsity on top of binarization will not necessarily improve the results further. We first illustrate how naively pruning binarized networks actually worsens storage and runtime. Consider a "small" binarized MLP used for MNIST (see Table 1) classification with the input layer of size 784 , one hidden layer with 128 neurons, and an output layer of size 10, both followed by batch normalization. If we prune it without any constraints and store the sparse weight matrix using compressed-sparse-row (CSR) format, binary weight packing and "SIMD" XNOR multiplication cannot be leveraged easily. We refer to this scheme as Naively-Pruned (NP) network. If the number of non-zero weights for each kernel is constrained to be the same multiple of 32, binary weights can now be packed to save storage, but activations need to be fetched individually and packed separately for each kernel during computation. We refer to this scheme as Naively-Pruned, Packed Network (NPP). NPP scheme has two advantages over NP in terms of storage overhead. First is packing weights into binary vectors instead of storing values individually. Second is getting rid of row extent values in the CSR format - having the same number of packs per kernel means that only one row extent value per layer needs to be stored. This will have more profound impact on high levels of sparsity, because while the number of column indices goes down with sparsity, the number of rows, and therefore row extents stays the same. Figure 1 shows total storage required for NP and NPP implementations, compared to a dense implementation. NPP offers significant storage reduction over NP, mainly through reduction in weight storage itself. However, to break even with Dense XNOR, sparsity levels of over 90% and 95% are required for NPP and NP respectively.

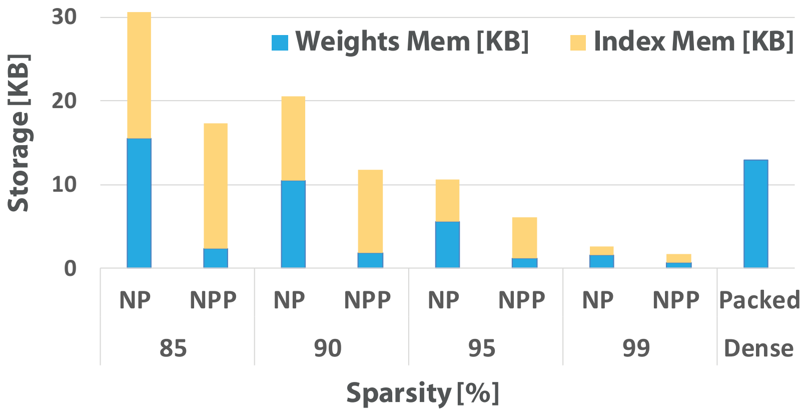

Fig. 1. Storage requirements for Dense, NP and NPP XNOR "small" MNIST MLP for varying levels of sparsity.

While NPP allows for packing non-zero weights, there is no easy way to leverage input packing. As runtime is usually a concern for convolutional layers, we implemented both Dense and NPP kernels for the Large CNN model (Table 3) for CIFAR-10. Even with sparsity set to 87.5% for each convolutional layer, except for the first one, which is kept as dense Binary-Weight (BWN - using full-precision activations and binarized weights), NPP version is 15X-29X slower compared to the unpruned dense XNOR Net on the different convolutional layers, running on a Raspberry Pi 3. This clearly shows that naively pruning binarized networks, even with packing, is not beneficial at best, and possibly detrimental to both their size and runtime.

3.3 Pruning a packed XNOR network

In order to make inference of pruned binarized neural network more efficient, inputs (activations) need to be fetched in packs, and those packs need to work for all kernels in a layer. This leads us to constrain the non-zero, or "active",

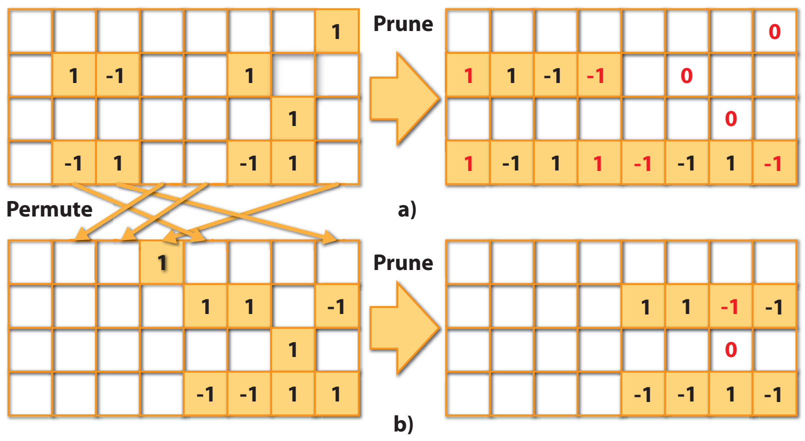

Fig. 2. Pruning with packing constraint of 4 bits a) without permutation, b) with permutation

weights of each kernel to packs of 32 consecutive positions. These 32-bit packs are aligned across all kernels of a layer so that the activations only needs to be packed once. This packing constraint reduces the number of indices to store by a factor of 32 compared to NPP implementation.

Forcing the packing constraint reduces the flexibility of the network, and can result in excessive pruning in some packs and insufficient pruning in others. In order to alleviate this effect, we propose to permute the weight matrix so that weights inside the 32-bit packs are more likely to be all zeros in the ternary network. Figure 2 illustrates the effect of permutation on packing for pack size of 4. A similar approach is also used in [52] albeit in context of crossbar neuromorphic systems. Weight permutations are performed on input channels (NI) of a fully-connected or convolutional layer. For a convolutional layer with weight shape (KN, KZ, KY, KX), the weight is first flattened in all dimensions except KZ to (KN_flat, KZ), and then treated as a fully-connected layer.

Permutation tries to group similar input channels into packs of 32 in a method resembling Prim's algorithm. Before grouping, weights are first ternarized to {-1,0,+1}. For each pack, a random input channel is chosen as starting point. A similarity score is calculated by counting the overlap of 0 positions between the existing input channels in the pack and all other channels that have not been grouped, and the channel with the most overlap is added to the pack. If a group has both 0s and {-1, +1} values in a position, it is considered as a non-zero position, as it's unclear if weights in the pack will be pruned or not, and either choice will result in some weights being forced to change. On the other hand, positions filled with 0s in a pack will definitely be pruned, and the pruning action will not force any weight change. Once a pack is filled, another random input channel is chosen as the starting position for the next pack. This process continues until all input channels are grouped into packs. Input channel permutations can be directly translated to output channel reordering of the previous layer, so it is completely free in terms of inference except for the first layer. Figure 3 shows the effect of using permutation in a small MLP, where maximal benefit (>4%) is observed with very high sparsity.

3.4 Training 3PXNet

Network pruning is usually performed by training a dense network and then deleting some edges or kernels. Since 0 cannot be represented in a binarized network, and the binary values contain no information about the importance of each weight, we start the training with ternary weights to introduce zero values. The training algorithm is adapted

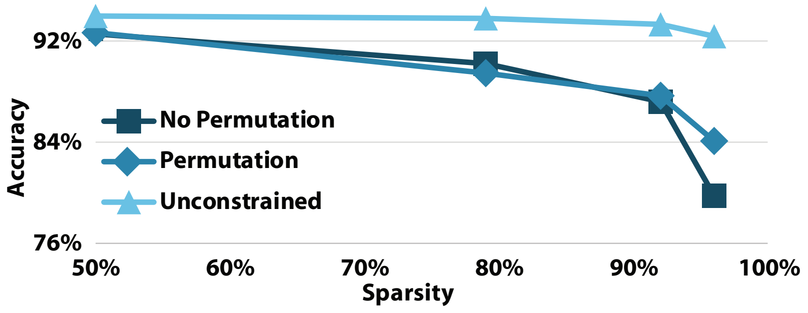

Fig. 3. Comparison of training results with and without permutation for different sparsities.

from the one in [34] where full-precision weights are kept during training and binarized during inference, except that the weights are ternarized using a threshold function instead of binarized.

Algorithm 1. Pseudocode for calculating threshold for NPP.

```
weight_sorted = sort(weight, descending=True)
index = ceil((1-sparsity)*size/word_width) * word_width
thres = weight_sorted[index-1]
```

The threshold used is calculated using Algorithm 1. The weight tensor is first flattened and sorted based on the floating point values. Threshold value is then chosen to make sure that the sparsity roughly align with the sparsity constraint of that layer, and that the number of active weights align with the word width of the processor (typically set to 32). The result of this phase of training follows the NPP scheme, which allows efficient storage of weights but requires high indexing overhead. We then permute the weight matrices using the method mentioned above, and enforce the packing constraint.

Algorithm 2. Pseudocode for calculating threshold for packed pruning.

```
weight_split = split(weight, split_size=word_width)
// For every split weight pack
for sp = 0 to size/word_width
    weight_sum[sp] = sum(abs(weight_split[sp]))
weight_sorted = sort(weight_sum, descending=True)
index = ceil((1-sparsity) * size / word_width)
thres = weight_sorted[index-1]
```

Similar to the NPP phase, a threshold is calculated using Algorithm 2 to determine the packs to prune. For each kernel in the weight matrix, we split the weight values into packs aligned to the word width of the processor, and sum the absolute value of weights inside each pack. We then calculate the threshold so that the sparsity roughly matches the sparsity requirement of the layer, and prunes away the packs with sums below the threshold. For packs that are not pruned, the weights inside are forced into {-1, +1}, even if they were originally 0. Before pruning is fixed, the pruned packs can still be unpruned if sum of another pack drops lower than the pruned pack. The network is trained for a few more epochs to determine which packs to prune. Finally, the packs to be pruned are fixed, and the model is fine-tuned to further improve accuracy. The final pruned, permuted and packed binarized network is referred to as 3PXNet.

As shown in Figure 3, forcing the packing constraint reduces network accuracy compared to unconstrained pruning, and permutation cannot fully recover accuracy loss. Permutation of inputs changes the allowed topology of the final network, which is the case for MLPs. For CNN we are not pruning the input layer, so permutation doesn't change the achievable topology. Due to the fact that we are permuting entire kernel planes, permutation is also more restrictive for convolutional layers. Immediately after permuting, packing and pruning a trained network, permutation has a significant advantage in accuracy (as much as 20% from 23.0 to 43.2%) but we fine-tune the network after permutation and it largely recovers irrespective of permutation indicating that the networks have significant redundancy unless the sparsity is very high (>90%). Because of the negligible effect of permutation on training and inference runtime, all networks are trained with and without permutation, and the better performing one is chosen as result.

4 THE 3PXNET IMPLEMENTATION

In this section, we detail the implementation choices which make 3PXNet one of the most compact and efficient neural networks. Before doing that however, we want to establish a consistent terminology for describing both fully-connected and convolutional layers. We will be using uppercase variables to describe dimensions, and lowercase ones to describe indices within those dimensions. Fully-connected layer is essentially a matrix-vector product between a vector of *NI* activations, or inputs, and a matrix of weights with size *NOxNI*, producing *NO* outputs, as shown on Figure 4. Number of outputs is the same as the number of kernels, and each kernel is a row in the weight matrix, with a size equal to the number of inputs.

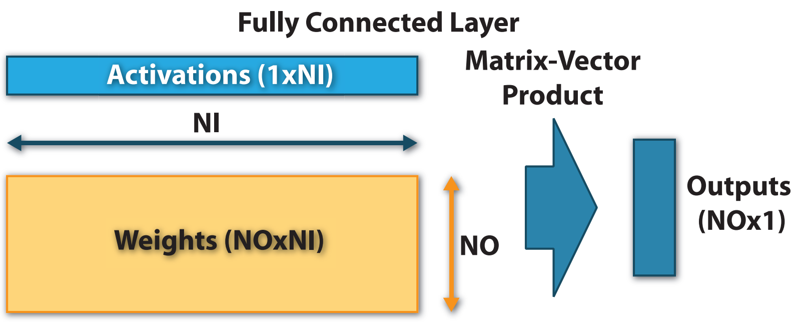

Fig. 4. A schematic view of a fully-connected layer with NI inputs and NO kernels.

Convolutional layer is shown schematically on Figure 5. Input to the layer is a 3D activation tensor of height *Y*, width *X* and depth *Z*. Activations are often padded in the *X* and *Y* dimensions, creating a "halo". Padding size, *PD*, is applied on both sides of each dimension, creating a tensor of height *Y+2PD*, width *X+2PD* and depth *Z*. This tensor is then convolved with *KN* kernels, each of them with height *KY*, width *KX* and depth *KZ*. We only consider cases where input (*Z*) and kernel (*KZ*) depths are the same. Each kernel generates an output of height *Y+2PD-KY+1*, width *X+2PD-KX+1* and depth of 1. Those outputs are then "stacked" depth-wise, creating a tensor of height *Y+2PD-KY+1*, width *X+2PD-KX+1* and depth of *KN*. Certain convolutional layers will be followed by pooling operations to reduce output dimensionality. Pooling, most commonly max pooling, uses a kernel with size and stride *PL* in the *Y* and *X* dimensions, producing an output of height *OY=(Y+2PD-KY)/PL + 1*, width *OX=(X+2PD-KX)/PL + 1* and depth *KN*. Those equations can be used for any type of layer - if padding is not used, *PD* is set to 0, and if there is no pooling, *PL* is set to 1. For pooling operations, kernel size and stride might sometimes be different, but that is not the case for any of the networks implemented here [42]. For simplicity we also do not discuss strided convolutions [29].

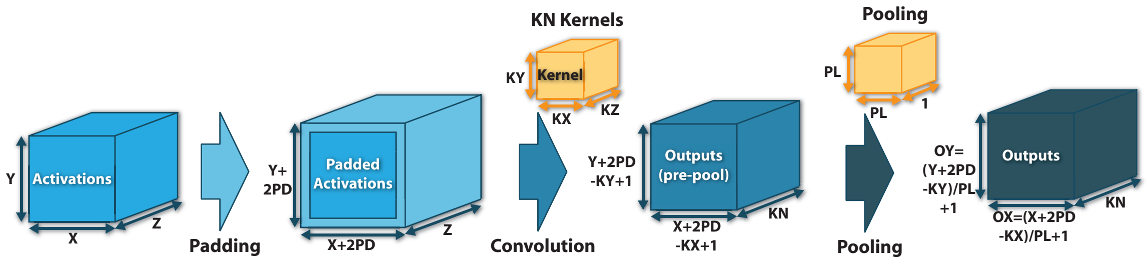

Fig. 5. A schematic view of a padded convolutional layer followed by pooling operation.

4.1 Fully-Connected Layers

As mentioned above, a fully-connected layer is essentially a general matrix-vector multiplication (GEMV) kernel. As it is easy to obtain close to theoretical speedups even with a straightforward GEMV implementation, we use it for our dense binarized reference [71]. Activations and weights are packed into binarized vectors which size is a multiple of 32 or 64, depending on the bitwidth of the processor used. In all subsequent sections we assume a 32-bit pack width. The exact alignment to the multiple of 32 is not necessary and could be handled by masking and adjusting the last pack to arbitrary size. However, for common network topologies, like the one presented in this work, layer sizes are generally a multiple of 32. We plan to support arbitrary layer sizes in future releases. We use the outputs/kernels as an outer loop, and input packs as an inner loop, as shown in Algorithm 3 for 32-bit packs. After all popcounts for a given output are completed, it needs to be adjusted. This is because popcount result is the number of ones, whereas the actual result is the number of ones minus the number of zeroes, since zeroes represent -1. The advantage of using outputs as an outer loop is that only one output is accumulated to at a time, and its partial result can be kept locally. On the other hand, it can make input reuse in caches harder. However, binarized input vectors are usually in the order of few hundred bytes, meaning even the smallest caches can fully fit them. After a single output is computed, we then perform on-the-fly binarization, to pack outputs into vectors for the next layer, as shown on Algorithm 3.

Algorithm 3. Pseudocode showing dense FC layer processing.

```
// For every output pack
for no = 0 to NO/32-1
    // For every output in a pack
    oPack = 0
    for pCnt = 0 to 32-1
        output = 0
        // For every input
        for ni = 0 to NI/32-1
            // XNOR multiplication
            mult = xnor(inputs[ni],weights[no*NI+ni])
            // Popcount accumulation
            output += popcount(mult)
        // Correct output value
        output = output-(NI*32-output)
        // Binarize
        output = output >= 0
        // Shift and pack
        oPack |= output << (31-pCnt)
    outputs[no] = oPack
```

For 3PXNets, only active (non-zero) weights are stored. We constrain pruning in such a way that every kernel has the same number of non-zero weight packs, *NP*. We did not enforce this constraint in the beginning, but we found that introducing it does not hurt accuracy. Models trained with the same size of sparse kernels sizes never have more than

0.5% lower accuracy than models with without that constraint. Additionally, it enables a further reduction in indexing overhead, by not storing row extents of the CSR format. For example, for MLP-L with high sparsity we can achieve up to 50% reduction in indexing storage, and 16.7% overall. We call those active packs. Because we know the number of packs for each neuron, only one index per pack needs to be stored, making it more efficient than e.g. CSR format. We use 8-bit indices, which support layers of up to 8192 activations. A simplified representation of weight and index storage for 3PXNet FC layers is shown on Figure 6. The loop ordering is the same as for dense implementation, but the inner loop iterates only through the active packs, instead of all inputs, as shown on Algorithm 4. Both Dense and 3PXNet FC layer implementations have versions with and without output binarization, the latter used for the final classifier layer.

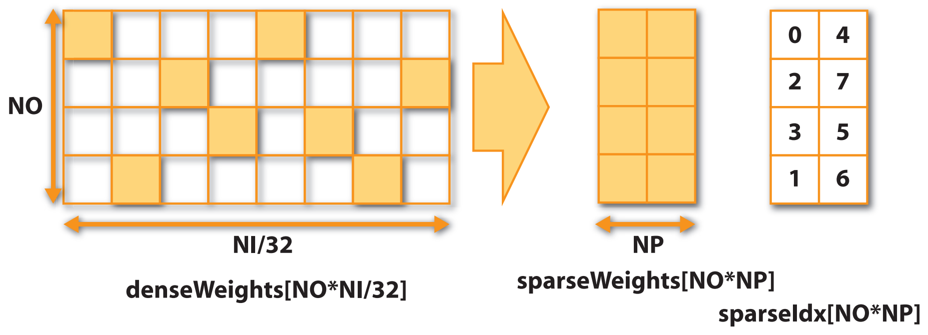

Fig. 6. 256x4 fully-connected layer weight and index storage with 75% sparsity (NP=2x 32-bit packs per output).

Algorithm 4. Pseudocode showing 3PXNet fully-connected layer processing.

```
// For every output pack
for no = 0 to NO/32−1
    oPack = 0
    // For every output in a pack
    for pCnt = 0 to 32−1
        output = 0
        // For all active packs
        for np = 0 to NP−1
            // Fetch correct input pack
            input = inputs[indcs[no*NP+np]]
            // XNOR multiplication
            mult = xnor(input, weights[no*NP+np])
            // Popcount accumulation
            output += popcount(mult)
        // Correct output value
        output = output−(NP*32−output)
        // Rest as in Alg 1
```

4.2 Convolutional Layers

Convolutional layers can be unrolled into matrix-matrix multiplication, as proposed by Chellapilla et. al. [12], which makes it possible to compute them in the same way as FC layers. However, the overhead of unrolling in binarized implementations offsets the arithmetic speedup [33]. To address that, we implement dense convolutional layers directly using the PressedConv approach proposed by Hu et. al.[33]. We capitalize on the fact that most convolutional layer activations have a depth (Z) which is a multiple of 32 and organize our data using depth as the innermost dimension for both activations and kernels. Because of that we can easily pack the actiavations and kernel weights in vectors of 32, irrespective of *X* and *Y* dimensions, which can have arbitrary values depending on the network. This is schematically shown for a *Y=3, X=3, Z=32* kernel on Figure 7. When implementing convolutional layers, loop ordering plays a particularly important role. There are at minimum 6 loops: output height (*OY*), output width (*OX*), output depth (*KN*), kernel height (*KY*), kernel width (*KX*) and kernel depth (*KZ*). For dense XNOR implementation, we use *OY-OX-KN-KY-KX-KZ* ordering, as shown on Algorithm 5. Kernel (*KN*) loop is split into two loops to facilitate packing, similarly to the output loop in fully-connected layers. The advantage of of this ordering is that it provides good locality on both activations and outputs. Figure 8 shows a comparison of speedups for a few different VGG-16D [64] for dense and 3PXNet with kernel as an outer (*K-YX*) and inner (*YX-K*) loop, normalized to dense *K-YX*. Using kernel as an inner dimension results in 8% geomean speedup.

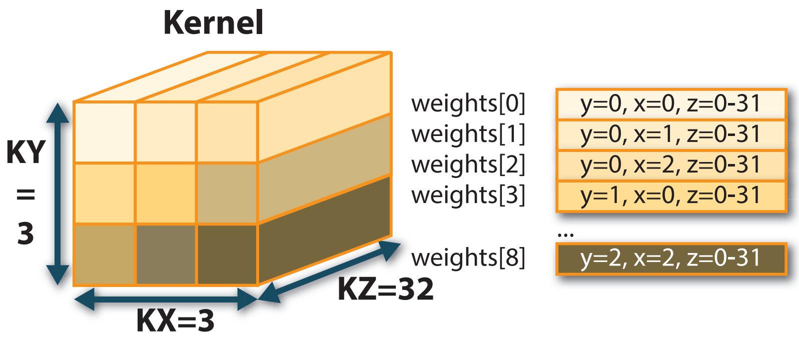

Fig. 7. 3x3x32 kernel packed into depth-first binarized vectors.

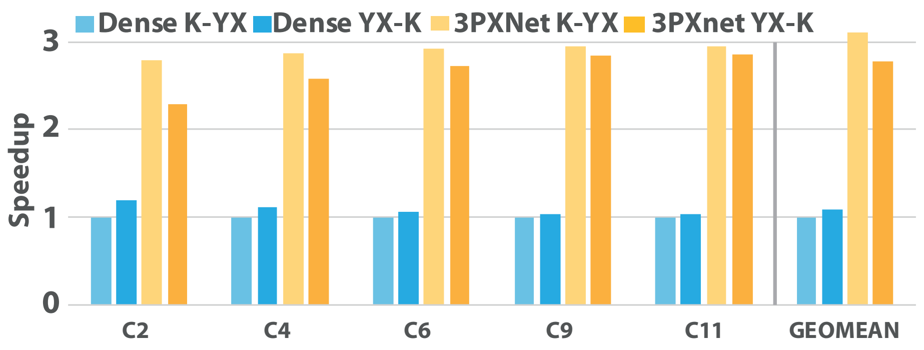

Fig. 8. Dense and 3PXNet (93.75% sparsity) speedups for kernel as an outer (K-YX) and inner (YX-K) loop for different VGG-16D [64] layers, normalized to dense K-YX.

Algorithm 5. Pseudocode showing dense conv layer loop ordering.

```
// For every output row
for oy = 0 to OY−1
    // For every output column
    for ox = 0 to OX−1
        // For every kernel packet
        for kn = 0 to (KN/32)−1
    // For every kernel in a packet
    for ks = 0 to 32−1
        // For every kernel row
        for ky = 0 to KY−1
            // For every kernel column
            for kx = 0 to KX−1
                // For every pack
                for kz = 0 to KZ/32−1
                    // XNOR multiplication
                    // Popcount accumulation
        // Output correction
        // Packing
```

Similarly to FC layers, for 3PXNets, we only store the non-zero weight packs, and their corresponding indices. We also constrain each kernel to have the same number (*KL*) of active packs to simplify indexing. This is shown schematically in Figure 9, for a kernel with *KY*=3, *KX*=3, *KZ*=32 and *KL*=3 active packs. We only use one index per pack, which combines information on all 3 dimensions, compressing index storage by a factor of 3. It comes at a cost to runtime, as indices need to be decoded for every kernel.

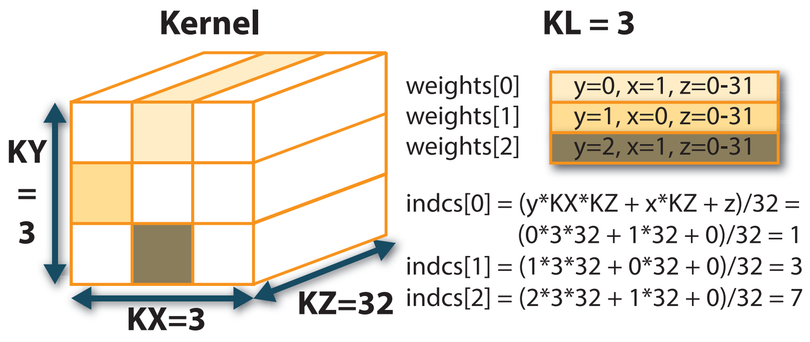

Fig. 9. 3x3x32 Convolution kernel weight packs and indices with KL=3 active packs.

For 3PXNet convolutional layers, we change the loop ordering compared to the dense implementation. As shown on Figure 8, for 3PXNet, using kernel as an outer dimension is actually faster. This is because it amortizes the cost of fetching and decoding indices - this way it only needs to be done once per each kernel. The downside of this approach is reduced output locality. Therefore we use kernels (*KN*) as an outer loop, as shown in Listing 6.

Algorithm 6. Pseudocode showing 3PXNet conv layer loop ordering.

```
// For every kernel packet
for k = 0 to (KN/32)−1
    // For every kernel in a packet
    for ks = 0 to 32−1
        // Decode indices
        // For every output row
        for oy = 0 to OY−1
            // For every output column
            for ox = 0 to OX−1
                // For every active pack
                for kl = 0 to KL−1
                     // XNOR multiplication
                     // Popcount accumulation
                // Output correction
                // Packing
```

For padding support, to further reduce storage requirements for intermediate activations, we opt to not store padded regions explicitly in memory, as they don't contribute to output values. During computation, we detect multiplications that fall under padded regions, and skip computation on those. This is possible, because convolution is done depth-first in packs of 32 or 64 values, and therefore each pack falls completely inside or outside the padded region. Skipping decision is therefore made on a single pack granularity. Other binarized works have proposed doing padding computation explicitly, through -1 or +1 padding, without significantly affecting accuracy [23, 76] albeit with increased storage and runtime. For example, using explicit padding in convolutional layers of CNN Large, listed in Table 3, adds between 12.8% and 56.3% additional activation storage, and increases computation by 4.1% to 31.9%, depending on the layer. Because padding is only ever applied in X and Y dimensions and packing is done along depth, every pack is either completely in the padding region or not. Since this approach makes computation irregular, potentially worsening runtime, we further split activations into 5 regions, where the middle one can be computed without padding related overheads, and the "halo" regions use computation skip, as shown on Figure 10.

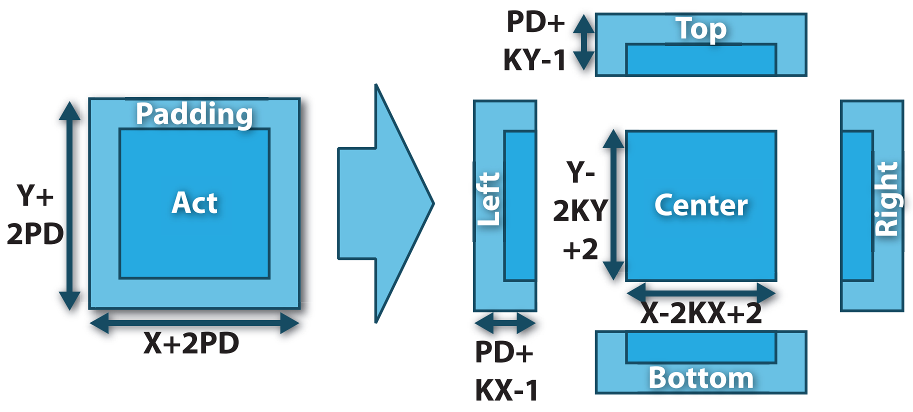

Fig. 10. Convolution splitting into padded and non-padded regions for efficient computation.

4.3 Fused kernels

Because we are targeting resource-constrained embedded devices, we want to limit intermediate storage used during the computation. In XNOR Networks, outputs of a given fully-connected or convolutional layers are generally passed through Pooling or Batch Normalization layers before they can be binarized, meaning that intermediate results need to be stored in full precision. To alleviate this issue, we use fused kernels, similar to McDanel et. al. [56]. Operations following fully-connected and convolutional layers, such as Pooling and Batch Normalization, are performed on-the-fly for each output. Specifically, they are implemented as following:

- Max Pooling layers - to enable on-the-fly pooling, we process the outputs of convolutional layers in groups matching the pooling patch size. For each patch we keep a running maximum value that is being updated as outputs are calculated.
- Batch Normalization and Binarization - Umuroglu et. al.[67] made an observation that in binarized networks, Batch Normalization followed by sign binarization can be reduced to a thresholding operation followed by a conditional sign change. By using this approach only one floating-point parameter needs to be stored per kernel. Signs can be stored in binary packed format and applied in batches using bitwise XNOR operations.

The idea behind fused kernels and threshold Batch Normalization is shown on Figure 11.

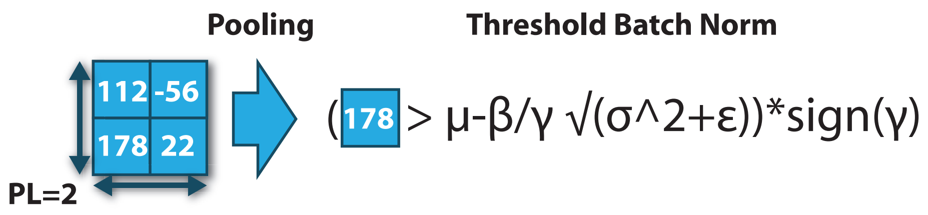

Fig. 11. A 2x2 fused Max Pooling followed by threshold Batch Normalization.

4.4 ARM NEON Support

For devices with ARM Cortex-A processors, like Raspberry Pi, we utilize NEON SIMD extension [6] to leverage hardware popcount support. Since our NEON-optimized functions use double- and quad-word (64- and 128-bit) support, we restrict sparsity to have a number of 32-bit packs that is a multiple of 2 or 4 for every kernel. Special care needs to be taken with padding in sparse convolutional layers, where within a single 64-/128-bit block, some packs may be within the padded region, while others are not. There is no easy way of skipping computation in that case, so we mask multiplication results that fall under the padding region. We use masking operations before popcounts to correct for that.

4.5 Binarization of the First Layer

In case of image processing, usually the only layer with a depth that is not a multiple of 32 is the first one, which makes it problematic to implement using depth-first convolutions. However, binarized implementations generally compute the first layer using full precision activations and binary weights to retain accuracy [19]. MLPs are used for MNIST which is almost black and white, so we binarize the inputs and keep the first layers as XNOR layers. For CNNs, we refrain from sparsifying the first layer, due to limited storage benefits.

5 EXPERIMENTAL SETUP

Hardware and software platforms, benchmark datasets and neural network architectures we use for our experiments are outlined below.

5.1 Platforms

We test our implementation on three different embedded development platforms from the STM Nucleo family [66], and a Raspberry Pi, with the configurations shown in Table 2. Our implementation is written in C and compiled with ARMCC version 5.06 for Nucleo devices, and GCC version 5.4 for Raspberry Pi, with -03 compiler optimizations enabled. For Cortex-M7 and M3 devices ("NUC Large" and "NUC Medium") we use the built-in cycle counter to measure execution time over 10 runs for each network [9]. For Cortex-M0 ("NUC Small") we use SysTick counter with a period of 1ms over 100 runs [8]. For Raspberry Pi we use C library routines to measure execution time over 100 runs. We use an off-the-shelf USB power meter with 2 decimal digit precision, to perform rough power measurement for each of the boards.

Table 2. Hardware platforms used for the runtime experiments. Only the *Small* platform has a DSP extension with hardware multiply-accumulate unit. All three microcontrollers are from the ST Nucleo family.

Name	Model	SRAM (KB)	Flash (KB)	Core Type	Clock (MHz)
NUC Large	F746ZG	320	1024	ARM CM7	216
NUC Medium	F103RB	20	128	ARM CM3	72
NUC Small	F031K6	4	32	ARM CM0	48
Name	Model	L2 (MB)	DRAM (GB)	Core Type	Clock (GHz)
RPi	Model B+	2	1	ARM CA53	1.2

5.2 Benchmarks

We evaluate our approach using two fully-connected networks (MLP) and a small convolutional neural network (CNN) on MNIST dataset, and two convolutional neural networks on CIFAR-10 and SVHN dataset as shown in Table 3. All datasets are image classification datasets with 10 classes. MLP-Large (MLP-L) and CNN-Large (CNN-L) are used in [19]. CNN-Medium (CNN-M) uses the same convolutional layers as CNN-L, but only has one fully-connected layer. CNN-Small (CNN-S) is a modified version of LeNet in [47], and MLP-Small (MLP-S) is a minimally sized MLP with one hidden layer. We also tested the same CNN-M for Google Speech Command dataset [70]. Networks are trained with PyTorch 0.4.1. All models are trained on the training set provided, and accuracies are measured on the testing sets. We used Adam optimizer [41] for training, with batch size of 256 and initial learning rate of 0.001.

Table 3. Benchmark models and datasets

Dataset	Model		Architecture
MNIST	MLP	Large [19]	784-4096-4096-4096-10
		Small	784-128-10
	CNN	Small	$32CONV5 - MP2$ $32CONV5 - MP2$ $10FC$
CIFAR-10 SVHN Speech[70]	CNN	Large [19]	$128CONV3 \times 2 - MP2$ $256CONV3 \times 2 - MP2$
		Medium	$512CONV3 \times 2 - MP2$ $(1024FC \times 2) - 10FC$

5.3 Baseline

On Nucleo boards, we use the ARM CMSIS-NN [7], version 5.3.0 optimized for Cortex-M processors. CMSIS-NN uses DSP extension present M7 processor for SIMD multiplication and accumulation. On Raspberry Pi we use Arm Compute Library [5], version 18.03, with NEON extension enabled, -O3 compiler optimizations, no batching and no multithreading for a fair comparison with 3PXNet. For both CMSIS-NN and Compute Library we use 8-bit precision: q7_t and QS8 datatypes respectively. First layer in CNNs is implemented using CMSIS-NN/CL routines with binarization overhead, since they are not packed and sparsified. We tried comparing to two existing Dense binarized implementations: BMXNet [71] and EBNN [56]. Unfortunately we were not able to extract and compile underlying C implementations for BMXNet, and eBNN runtimes we obtained were worse then CMSIS-NN 8-bit precision[1], therefore we refrain from reporting them. All results are generated for a batch size of *B=1*. Larger batch sizes increase storage requirements, and can make the models even more prohibitive to deploy on resource-constrained devices.

6 RESULTS AND DISCUSSION

We discuss results for 3PXNet accuracy separately from performance as only the latter depends on the hardware platform.

6.1 Accuracy & Model Size

Table 4. Accuracy and network size (KB, in brackets) comparison.

Dataset	MNIST			CIFAR-10		SVHN		Speech
Model	MLP-L	MLP-S	CNN-S	CNN-L	CNN-M	CNN-L	CNN-M	CNN-M
8-bit	98.67% (36.9k)	98.28% (102)	99.42% (32.7)	92.52% (14.1k)	92.51% (4.69k)	95.65% (14.1k)	95.15% (4.69k)	97.87% (4.70k)
XNOR	98.40% (4.64k)	93.15% (13.1)	97.83% (4.46)	89.07% (1.80k)	88.29% (592)	95.03% (1.80k)	95.00% (592)	96.64% (591)
3PXNet$_{low}$	98.37% (421)	90.35% (3.96)	96.60% (1.66)	84.74% (257)	82.49% (98.8)	93.51% (257)	92.57% (98.8)	94.32% (97.6)
3PXNet$_{high}$	96.58% (120)	87.93% (2.08)	96.27% (1.46)	81.40% (143)	78.28% (61.7)	92.25% (143)	90.61% (61.7)	92.81% (60.5)

Figure 12 compares accuracy vs. model size of 3PXNet to a dense binarized network eBNN [56]. 3PXNet achieves up to 4X (2.5X) reduction in MLP (CNN) model size with same or better accuracy than eBNN. While both implementations use quantization, for eBNN, additional size compression comes from tweaking the network structure itself, whereas for us it comes from pruning. In this work we only explore the interaction binarization and prunning, but we plan to extend our approach to also optimize across the network structure itself.

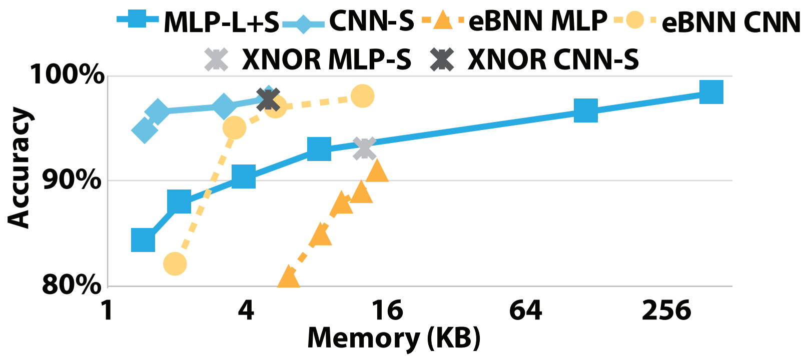

Fig. 12. Accuracy vs. Memory tradeoff compared to eBNN and dense XNOR.

Training results are shown in Table 4. Each network is trained with two sparsities, and the size of the network is shown in parenthesis next to accuracy. For most models, the 3PXNet$_{low}$ has target sparsity of 90%, while 3PXNet$_{high}$ is targeting 95%. For MLP-L on MNIST, 3PXNet$_{low}$ has target sparsity of 95%, while 3PXNet$_{high}$ is targeting 99%. Output layers of CNN-M and CNN-L is kept to 50% sparsity due to its negligible impact on storage and computation. Because

[1]eBNN was 8X and >80X slower than our dense implementation for FC and CNN layers respectively, which we believe is partially because it uses 8-bit packs for higher flexibility, but lower computational efficiency, and using floating-point accumulation.

we're not pruning entire neurons, batch normalization layers are also not pruned. Since these constant-sized layers take up varied proportion of the entire model and doesn't decrease in size with higher sparsity, actual compression rate varies a lot across models and we think it's more clear to list the actual sizes of the models, which are shown in parenthesis next to accuracy numbers. For binary/3PXNet implementations, weights are binary or ternary depending on sparsity of a layer, and activations are all binary except for inputs to the first layer in CNNs. While for some of the networks there is a noticeable drop in accuracy when comparing 8-bit with 3PXNet as in the case of MLP-S on MNIST and networks on CIFAR-10, we argue that the main benefit of 3PXnet is enabling deploying some of those models on heavily memory constrained devices, which would not be possible with 8-bit and, in some cases, even dense binary. As shown in Table 5 and 6, binarization enables implementation of the network in 3 cases, and 3PXNet enables another 2. When the model loses noticeable accuracy when binarized, it tends to lose more when pruned. Accuracy loss can be mitigated by using more permutations per layer in addition to the "free" one, or using smaller pack sizes like 8. For MLP-S on MNIST, reducing pack size to 16 and 8 for "3PXNet$_{low}$" increases accuracy to 88.42% and 90.09% respectively. The added indexing storage and runtime overhead is usually not a good trade-off when naively using packs of size smaller than 32, but we plan on exploring more efficient implementations of such networks in the future.

Network pruning can also be performed on higher precision models, so we compared our performance to magnitude-based weight pruning [27]. Apart from the processing difficulties resulting from irregular sparsity, the index of each active weight also needs to be stored. Since size of a filter can easily surpass 255, which is the limit of 8-bit indexing, we used 16 bits for each index, but 13 shows the accuracy comparison between sparse 8-bit networks and binarized networks. For MNIST, sparse 8-bit networks have worse accuracy-size trade-off compared to dense binarized networks, let alone the sparse ones. For CIFAR-10, sparse 8-bit networks have higher accuracy for the same size compared to dense binarized networks, but are worse than 3PXNet implementations. To achieve the model size of binarized networks, an 8-bit fixed-point network requires very high sparsity particularly due to the indexing required, and loses too many connections to sustain accuracy. Methods to reduce indexing overhead for fixed-point networks is proposed in [74], but is mostly limited to packing of 2, so the conclusion doesn't change. Under the size constraints of very small microcontrollers, 3PXNet offers better accuracy-size trade-offs, even when not accounting for the benefits in processing efficiency our structured pruning method brings.

6.2 Performance & Energy

Runtime and energy results for 8-bit, XNOR and 3PXNet, are shown in Tables 5 and 6 for MNIST and CIFAR-10/SVHN/Google Speech Command respectively. We report only one result for CIFAR-10, SVHN and Speech Command. The first two have exactly the same network structure, which yields the same runtime. For Google Speech Command, only the first and the last layers are different, and runtime differences are so small (<5%) we opt not to report them separately. For MLP-S 3PXNet shows between 10.7x and 25.2x runtime improvement over CMSIS-NN, and 1.8x to 3x over Dense XNOR implementation. MLP-L can fit one the NUC Large only after binarizing and sparsifying the network. On Raspberry Pi, the speedups obtained for both MLP-S and MLP-L are much larger than what we'd expect (76x-400x). For the former, the model is so small that overheads like memory management limit the performance of 8-bit (ARM CL) implementation. For MLP-L which has a model size in tens of MBs, the large latency might be caused by memory bandwidth limitations. This means that dense and sparse binary implementations can provide speedups beyond 8x (vs 8-bit binary) on memory bandwidth constrained models and architectures.

On CNN-S, 3PXNets show between 2x and 2.6x improvement over 8-bit implementations, and between 1.06X and 2.7x improvement over Dense XNOR, on Nucleo platforms, for low and high sparsity respectively. On Raspberry Pi,

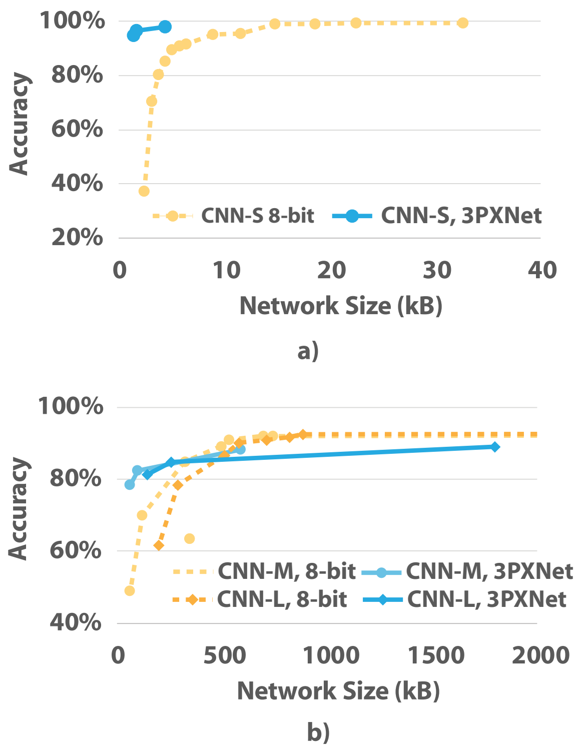

Fig. 13. Accuracy comparison between sparse 8-bit network and 3PXNet, for MNIST (a) and CIFAR-10 (b).

the speedups are 2.6x and 2.8x versus 8-bit, and 1.5x and 1.6x versus dense binary. 3PXNet CNN-M is 2.2x and 2.7x faster than dense binary, on the largest Nucleo device, where the 8-bit model cannot fit at all. On Raspbeery Pi, 3PXNet achieves 7.4-10.4x speedup vs ARM CL, and 2-2.75x speedup over dense binary. Energy reduction is proportional to runtime, which means 3PXNets could greatly extend battery life of edge devices.

There are a few factors limiting achievable speedups, for both Dense and 3PXNets. First is the lack of hardware popcount support on Cortex-M processors, which is the case for most embedded microcontrollers. Even heavily optimized software implementation, which we use, is quite inefficient [58]. On the small MLP, especially sparse versions, pure XNOR speedups are further limited by overhead of input binarization and batch normalization, particularly on NUC Medium and Small platforms which don't have hardware Floating Point Units. We plan on exploring fixed-point batch normalization in the future to alleviate that. CNN speedups are heavily limited by Binary-Weight first layer, especially for CNN-S, e.g. Dense XNOR on "NUC Large" spends 54% of total runtime in the first layer, whereas for 3PXNet$_{high}$ that number reaches 92%. In principle Binary-Weight layers have an advantage over full-precision ones, because multiplication can be removed with an up/down counter. However this comes at a cost of introducing data-dependent conditional statements, which are potentially worse than actual multiplication. An alternative is to preserve multiplication and compressed weights, which achieves storage reduction at a very small cost to runtime due to weight unpacking overhead. We plan on implementing more efficient BWN layers in the future work. Finally, 3PXNet performance improvements over dense XNOR implementations are limited by indexing-related overheads.

Table 5. Runtime (ms) and energy (mJ, in brackets) for MNIST networks. A dash indicates a given model could not fit in memory.

	MLP-S				MLP-L				CNN-S			
	8bit	XNOR	$3PXN_{low}$	$3PXNet_{high}$	8bit	XNOR	$3PXNet_{low}$	$3PXNet_{high}$	8bit	XNOR	$3PXNet_{low}$	$3PXNet_{high}$
RPi	2 (9.5)	≈5e-3 (0.02)	≈5e-3 (0.02)	≈5e-3 (0.02)	191.3 (908)	4.9 (23.2)	2.5 (11.8)	0.41 (1.9)	12.4 (58.9)	7 (33)	4.8 (23)	4.5 (21)
NUC Large	1.83 (2.03)	0.37 (0.41)	0.17 (0.19)	0.14 (0.15)	—	—	10.1 (11.2)	3.97 (4.41)	47.4 (52.6)	34.09 (37.8)	23.4 (26)	23.2 (25.7)
NUC Medium	19.9 (8)	2.4 (0.9)	1.05 (0.4)	0.79 (0.3)	—	—	—	—	1733 (693)	731.2 (292)	676.8 (271)	673.3 (269)
NUC Small	—	2.9 (0.7)	1.6 (0.4)	1.3 (0.3)	—	—	—	—	—	1176 (294)	1109 (275)	1102 (277)

Table 6. Runtime (ms) and energy (mJ, in brackets) for CIFAR-10/SVHN/Speech networks. A dash indicates a given model could not fit in memory.

	CNN-M				CNN-L			
	8bit	XNOR	$3PXNet_{low}$	$3PXN_{high}$	8bit	XNOR	$3PXNet_{low}$	$3PXNet_{high}$
RPi	551 (2.6k)	146 (700)	74 (350)	53 (250)	638 (3k)	154 (730)	75 (350)	54 (250)
NUC Large	—	3625 (4k)	1630 (1.8k)	1346 (1.5k)	—	—	1632 (1.8k)	1347 (1.5k)

7 CONCLUSION

We have developed the first software implementation and corresponding training methodologies for sparse binarized networks or 3PXNets. 3PXNets can deliver up to 300x (38x) smaller model sizes compared to 8-bit fixed point (dense binarized) networks allowing us to fit complex deep learning models on smallest of microcontrollers for the first time. Even in smaller models that can fit with conventional approaches, 3PXNets achieve up to 25x improvement in runtime and energy. We show multiple sub-ms and sub-mJ models on commodity low-end microcontrollers, which would not be possible without 3PXNet. We release 3PXNet as an open-source library targeting machine learning at the edge (link omitted for blind review). We plan to expand our approach to other structures like recurrent neural networks, and more carefully explore sparsity options within networks.

REFERENCES

[1] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frederic Petrot. 2017. Ternary neural networks for resource-efficient AI applications. *Proceedings of the International Joint Conference on Neural Networks* 2017-May (2017), 2547–2554. https://doi.org/10.1109/IJCNN.2017.7966166 arXiv:1609.00222

[2] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. 2018. Yoda NN: An architecture for ultralow power binary-weight CNN acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 1 (2018), 48–60. https://doi.org/10.1109/TCAD.2017.2682138 arXiv:1606.05487

[3] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured Pruning of Deep Convolutional Neural Networks. *ACM Journal on Emerging Technologies in Computing Systems* 13, 3 (2017), 1–18. https://doi.org/10.1145/3005348 arXiv:1512.08571

[4] Sajid Anwar and Wonyong Sung. 2016. Compact Deep Convolutional Neural Networks With Coarse Pruning. *CoRR* abs/1610.09639 (2016). arXiv:1610.09639 http://arxiv.org/abs/1610.09639

[5] ARM. 2018. ARM Compute Library. https://github.com/ARM-software/ComputeLibrary

[6] ARM. 2018. NEON. https://github.com/ARM-software/CMSIS{_}5

[7] ARM. 2019. ARM CMSIS-NN. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon

[8] ARM Limited. 2017. ARMv6-M Architecture Reference Manual. , 1138 pages. https://doi.org/ARMDDI0419D

[9] ARM Limited. 2018. ARMv7-M Architecture Reference Manual. , 1138 pages. https://doi.org/ARMDDI0403E

[10] A. A. Bahou, G. Karunaratne, R. Andri, L. Cavigelli, and L. Benini. 2018. XNORBIN: A 95 TOp/s/W hardware accelerator for binary convolutional neural networks. In *2018 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*. 1–3. https://doi.org/10.1109/CoolChips.2018.8373076

[11] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann. 2019. An Always-On 3.8 μ J/86% CIFAR-10 Mixed-Signal Binary CNN Processor With All Memory on Chip in 28-nm CMOS. *IEEE Journal of Solid-State Circuits* 54, 1 (Jan 2019), 158–172. https://doi.org/10.1109/JSSC.2018.2869150

[12] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. (Oct. 2006). https://hal.inria.fr/inria-00112631 http://www.suvisoft.com.

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:cs.DC/1512.01274

[14] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 367–379. https://doi.org/10.1109/ISCA.2016.40

[15] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A Survey of Model Compression and Acceleration for Deep Neural Networks. *CoRR* abs/1710.0 (2017), 1–10. https://doi.org/10.1109/ICRA.2016.7487304 arXiv:1710.09282

[16] Zhiyong Cheng, Daniel Soudry, Zexi Mao, and Zhen-zhong Lan. 2015. Training Binary Multilayer Neural Networks for Image Classification using Expectation Backpropagation. *CoRR* abs/1503.03562 (2015). arXiv:1503.03562 http://arxiv.org/abs/1503.03562

[17] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and Characterization of Inherent Application Resilience for Approximate Computing. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 113, 9 pages. https://doi.org/10.1145/2463209.2488873

[18] F. Conti, P. D. Schiavone, and L. Benini. 2018. XNOR Neural Engine: A Hardware Accelerator IP for 21.6-fJ/op Binary Neural Network Inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (Nov 2018), 2940–2951. https://doi.org/10.1109/TCAD.2018.2857019

[19] Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR* abs/1602.02830 (2016). arXiv:1602.02830 http://arxiv.org/abs/1602.02830

[20] Yann Le Cun, John S Denker, and Sara a Solla. 1990. Optimal Brain Damage. *Advances in Neural Information Processing Systems* 2, 1 (1990), 598–605. https://doi.org/10.1.1.32.7223 arXiv:arXiv:1011.1669v3

[21] Lei Deng, Peng Jiao, Jing Pei, Zhenzhi Wu, and Guoqi Li. 2018. GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework. *Neural Networks* 100 (2018), 49–58. https://doi.org/10.1016/j.neunet.2018.01.010 arXiv:1705.09283

[22] Julian Faraone, Nicholas Fraser, Giulio Gambardella, Michaela Blott, and Philip H.W. Leong. 2017. Compressing Low Precision Deep Neural Networks Using Sparsity-Induced Regularization in Ternary Networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10635 LNCS (2017), 393–404. https://doi.org/10.1007/978-3-319-70096-0_41 arXiv:1709.06262

[23] Nicholas J. Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Scaling Binarized Neural Networks on Reconfigurable Logic. In *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM '17)*. ACM, New York, NY, USA, 25–30. https://doi.org/10.1145/3029580.3029586

[24] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices. *34th International Conference on Machine Learning (ICML 2017)* 70 (2017), 1331–1340. http://proceedings.mlr.press/v70/gupta17a.html

[25] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 1737–1746. http://dl.acm.org/citation.cfm?id=3045118.3045303

[26] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. (2015), 1–14. https://doi.org/abs/1510.00149/1510.00149 arXiv:1510.00149

[27] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 1135–1143. http://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf

[28] Babak Hassibi, David G. Stork, and Gregory J. Wolff. 1993. Optimal brain surgeon and general network pruning. , 293–299 pages. https://doi.org/10.1109/ICNN.1993.298572

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[30] Z. He, B. Gong, and D. Fan. 2019. Optimize Deep Convolutional Neural Network with Ternarized Weights and High Accuracy. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 913–921. https://doi.org/10.1109/WACV.2019.00102

[31] Mattias P. Heinrich, Max Blendowski, and Ozan Oktay. 2018. TernaryNet: faster deep model inference without GPUs for medical 3D segmentation using sparse and binary convolutions. *International Journal of Computer Assisted Radiology and Surgery* (2018), 1–10. https://doi.org/10.1007/s11548-018-1797-4 arXiv:1801.09449

[32] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[33] Yuwei Hu, Jidong Zhai, Dinghua Li, Yifan Gong, Yuhao Zhu, Wei Liu, Lei Su, and Jiangming Jin. 2018. BitFlow: Exploiting Vector Parallelism for Binary Neural Networks on CPU. *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2018), 244–253. https://doi.org/10.1109/IPDPS.2018.00034

[34] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 4107–4115. http://papers.nips.cc/paper/6573-binarized-neural-networks.pdf

[35] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[36] L. Jiang, M. Kim, W. Wen, and D. Wang. 2017. XNOR-POP: A processing-in-memory architecture for binary Convolutional Neural Networks in Wide-IO2 DRAMs. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. https://doi.org/10.1109/ISLPED.2017.8009163

[37] Norman P. Jouppi, Al Borchers, and Rick et.al. Boyle. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17* (2017), 1–12. https://doi.org/10.1145/3079856.3080246 arXiv:1704.04760

[38] Patrick Judd, Jorge Albericio, and Andreas Moshovos. 2017. Stripes: Bit-Serial Deep Neural Network Computing. *IEEE Computer Architecture Letters* 16, 1 (2017), 80–83. https://doi.org/10.1109/LCA.2016.2597140 arXiv:arXiv:1011.1669v3

[39] Patrick Judd, Alberto Delmas, Sayeh Sharify, and Andreas Moshovos. 2017. Cnvlutin2: Ineffectual-Activation-and-Weight-Free Deep Neural Network Computing. (2017), 1–6. arXiv:1705.00125 http://arxiv.org/abs/1705.00125

[40] Minje Kim and Paris Smaragdis. 2016. Bitwise Neural Networks. 37 (2016). arXiv:1601.06071 http://arxiv.org/abs/1601.06071

[41] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. http://arxiv.org/abs/1412.6980 cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[42] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems* (2012), 1–9. https://doi.org/10.1016/j.protcy.2014.09.007 arXiv:1102.0183

[43] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. *34th International Conference on Machine Learning (ICML 2017)* 70 (2017), 1935–1944. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/06/kumar17.pdf{%}0Ahttp://proceedings.mlr.press/v70/kumar17a.html

[44] Abhisek Kundu, Kunal Banerjee, Naveen Mellempudi, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. 2017. Ternary Residual Networks. (2017), 1–16. arXiv:1707.04679 http://arxiv.org/abs/1707.04679

[45] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. (2018), 1–10. arXiv:1801.06601 http://arxiv.org/abs/1801.06601

[46] Vadim Lebedev and Victor Lempitsky. 2015. Fast ConvNets Using Group-wise Brain Damage. (2015). https://doi.org/10.1109/CVPR.2016.280 arXiv:1506.02515

[47] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. https://doi.org/10.1109/5.726791

[48] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary Weight Networks. Nips (2016). arXiv:1605.04711 http://arxiv.org/abs/1605.04711

[49] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. 2017. A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks (Abstract Only). *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17* March (2017), 290–291. https://doi.org/10.1145/3020078.3021786 arXiv:1702.06392

[50] Yixing Li and Fengbo Ren. 2018. Build a Compact Binary Neural Network through Bit-level Sensitivity and Data Pruning. (2018). arXiv:1802.00904 http://arxiv.org/abs/1802.00904

[51] Yixing Li, Kai Xu, and Hao Yu. 2017. A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks. *CoRR* (2017). arXiv:1702.06392

[52] Ling Liang, Lei Deng, Yueling Zeng, Xing Hu, Yu Ji, Xin Ma, Guoqi Li, and Yuan Xie. 2018. Crossbar-aware neural network pruning. (2018), 1–13. arXiv:1807.10816 http://arxiv.org/abs/1807.10816

[53] Jeng Hau Lin, Tianwei Xing, Ritchie Zhao, Zhiru Zhang, Mani Srivastava, Zhuowen Tu, and Rajesh K. Gupta. 2017. Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops* 2017-July, 1 (2017), 344–352. https://doi.org/10.1109/CVPRW.2017.48 arXiv:1707.04693

[54] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards Accurate Binary Convolutional Neural Network. 3 (2017), 1–14. arXiv:1711.11294 http://arxiv.org/abs/1711.11294

[55] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse Convolutional Neural Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[56] Bradley McDanel, Surat Teerapittayanon, and H. T. Kung. 2017. Embedded Binarized Neural Networks. (2017), 1–6. arXiv:1709.02260 http://arxiv.org/abs/1709.02260

[57] Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. 2017. Ternary Neural Networks with Fine-Grained Quantization. (2017). arXiv:1705.01462 http://arxiv.org/abs/1705.01462

[58] Wojciech Mula, Nathan Kurz, and Daniel Lemire. 2018. Faster Population Counts Using AVX2 Instructions. *Computer Journal* 61, 1 (2018), 111–120. https://doi.org/10.1093/comjnl/bxx046 arXiv:1611.07612

[59] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. (2017). https://doi.org/10.1145/3079856.3080254 arXiv:1708.04485

[60] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[61] Fabrizio Pedersoli, George Tzanetakis, and Andrea Tagliasacchi. 2018. Espresso: Efficient Forward Propagation for BCNNs. (2018), 1–10. arXiv:arXiv:1705.07175v2

[62] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *arXiv preprint* (2016), 1–17. https://doi.org/10.1007/978-3-319-46493-0 arXiv:1603.05279

[63] Shimpei Sato, Hiroki Nakahara, and Shinya Takamaeda-yamazaki. 2018. BRein Memory : A Single-Chip Binary / Ternary Reconfigurable in-Memory Deep Neural Network. 53, 4 (2018), 983–994.

[64] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014), 1–14. https://doi.org/10.1016/j.infsof.2008.09.005 arXiv:1409.1556

[65] Ranko Sredojevic, Shaoyi Cheng, Lazar Supic, Rawan Naous, and Vladimir Stojanovic. 2017. Structured Deep Neural Network Pruning via Matrix Pivoting. (2017), 1–16. arXiv:1712.01084 http://arxiv.org/abs/1712.01084

[66] STMicroelectronics. 2018. STM32 Nucleo-144 boards. https://www.st.com/resource/en/data{_}brief/nucleo-f746zg.pdf

[67] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, and Michaela Blott. 2017. FINN : A Framework for Fast , Scalable Binarized Neural Network Inference. February (2017). arXiv:arXiv:1612.07119v1

[68] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*.

[69] Huan Wang, Qiming Zhang, Yuehai Wang, and Roland Hu. 2018. Structured Deep Neural Network Pruning by Varying Regularization Parameters. (2018). arXiv:1804.09461 http://arxiv.org/abs/1804.09461

[70] Pete Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *CoRR* abs/1804.03209 (2018). arXiv:1804.03209 http://arxiv.org/abs/1804.03209

[71] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. 2017. BMXNet: An Open-Source Binary Neural Network Implementation Based on MXNet. (2017). arXiv:arXiv:1705.09864

[72] Li Yang, Zhezhi He, and Deliang Fan. 2018. A Fully Onchip Binarized Convolutional Neural Network FPGA Impelmentation with Accurate Inference. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 50:1—-50:6.

[73] Haruyoshi Yonekawa, Shimpei Sato, and Hiroki Nakahara. 2018. A Ternary Weight Binary Input Convolutional Neural Network: Realization on the Embedded Processor. *2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL)* (2018), 174–179. https://doi.org/10.1109/ISMVL.2018.00038

[74] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), 548–560. https://doi.org/10.1145/3079856.3080215

[75] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* 2016-Decem (2016). https://doi.org/10.1109/MICRO.2016.7783723

[76] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17* (2017), 15–24. https://doi.org/10.1145/3020078.3021741

[77] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. 1, 1 (2016), 1–13. arXiv:1606.06160 http://arxiv.org/abs/1606.06160

[78] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2016. Trained Ternary Quantization. (2016), 1–10. arXiv:1612.01064 http://arxiv.org/abs/1612.01064