JointNF: Enhancing DNN Performance through Adaptive N:M Pruning across both Weight and Activation

Sai Qian Zhang
New York University
New York, USA

Thierry Tambe
Stanford University
Stanford, USA

Gu-Yeon Wei
Harvard University
Cambridge, USA

David Brooks
Harvard University
Cambridge, USA

Abstract

Balancing accuracy and hardware efficiency remains a challenge with traditional pruning methods. N:M sparsity is a recent approach offering a compromise, allowing up to N non-zero weights in a group of M consecutive weights. However, N:M pruning enforces a uniform sparsity level of $\frac{N}{M}$ across all layers, which does not align well sparse nature of deep neural networks (DNNs). To achieve a more flexible sparsity pattern and a higher overall sparsity level, we present *JointNF*, a novel joint N:M and structured pruning algorithm to enable fine-grained structured pruning with adaptive sparsity levels across the DNN layers. Moreover, we show for the first time that N:M pruning can also be applied over the input activation for further performance enhancement.

CCS Concepts

• **Hardware → Hardware accelerators**.

Keywords

Hardware accelerator, Pruning, Transformer

ACM Reference Format:
Sai Qian Zhang, Thierry Tambe, Gu-Yeon Wei, and David Brooks. 2024. JointNF: Enhancing DNN Performance through Adaptive N:M Pruning across both Weight and Activation. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '24), August 5–7, 2024, Newport Beach, CA, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3665314.3670813

1 Introduction

DNN pruning has been studied extensively in the previous literature [3, 4, 7, 10–12, 17], which can be broadly divided into two categories: *unstructured pruning* and *structured pruning*. Unstructured pruning allows for a high degree of model sparsity without compromising accuracy, but the nonuniform placement of the nonzero weights makes hardware implementation challenging [17]. On the other hand, structured pruning removes entire blocks of model parameters (e.g., filters), making it more suitable for efficient hardware implementation. However, this approach sacrifices some flexibility in the sparsity patterns and may lead to suboptimal model accuracy. Recently, a new class of fine-grained structured sparsity called N:M sparsity has received a growing amount of attention [11, 15, 18]. This approach, illustrated in Figure 1(a), restricts the number of nonzero weights to at most N within a group of consecutive M (where $N \leq M$). N:M sparsity has been found to result in significant speedup on commodity hardware platforms (e.g., Nvidia V100 GPU [1]) compared to the original dense DNN.

Although N:M sparsity has shown promise in terms of performance, it imposes a strict sparsity ratio that is not greater than or equal to $\frac{N}{M}$ (e.g., 20%) across all DNN layers. This constraint on sparsity ratio may result in suboptimal accuracy performance because the sparsity ratios of various DNN layers tend to differ. Previous studies have demonstrated that under a fixed total amount of pruned weights, nonuniform sparsity among layers is crucial for achieving high test accuracy [3]. To better understand this, we conduct a simple experiment to evaluate the sparsity distribution across layers within ResNet-18 on the ImageNet dataset. We apply global magnitude pruning, which involves initially sorting all the weights in a DNN according to their magnitudes, followed by removal of the smallest 80% weights. Figure 1(c) illustrates the uneven sparsity ratio distribution of each ResNet-18 layer, with the later layers tending to have much greater sparsity ratios than the earlier layers. This result confirms layer-wise adaptive sparsity plays an important role on the test accuracy for a sparse DNN.

In this work, we propose a novel joint N:M and structured pruning solution termed *JointNF* to achieve more flexible fine-grained structured sparsity while maintaining high hardware efficiency. JointNF utilizes both N:M and structured pruning in a complementary fashion: N:M pruning is used to achieve fine-grained sparsity with a consistent sparsity ratio across all layers. On the other hand, structured pruning is applied together to facilitate adaptable sparsity across the layers. JointNF takes advantage of the structured characteristics of both N:M pruning and structured pruning, resulting in superior hardware efficiency while attaining optimal test accuracy. This performance exceeds that of N:M pruning or structured pruning applied alone. Moreover, we demonstrate that N:M pruning can also be applied to input activation to produce fine-grained structured transient sparsity. To accomplish this, we introduce a novel activation function called *NMReLU*, which imposes N:M sparsity on input activation and can be used in conjunction with N:M weight pruning to further speed up DNN inference. Overall, our contributions are summarized as follows:

- We introduce JointNF, an innovative pruning approach that combines N:M and structured pruning on DNN weights for superior accuracy and hardware efficiency. It employs an iterative magnitude-based method that minimizes changes to the L1 norms with a theoretical guarantee.
- We propose *NMReLU*, a novel activation function that allows for N:M sparsity on the DNN intermediate data.

ISLPED '24, August 5–7, 2024, Newport Beach, CA, USA

ACM ISBN 979-8-4007-0688-2/24/08…$15.00
https://doi.org/10.1145/3665314.3670813

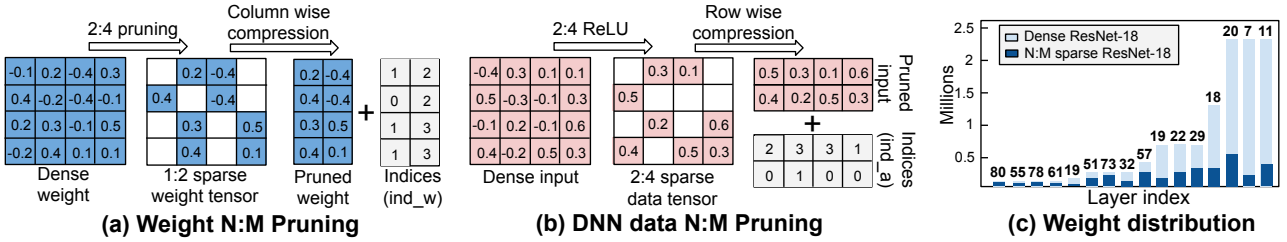

Figure 1: (a) 2:4 pruning on DNN weights. A 4×4 weight matrix is column-wise compressed into a 4×2 matrix. (b) 2:4 pruning on DNN inputs. A 4×4 data matrix is column-wise compressed into a 2×4 matrix. (c) Sparsity ratios on ImageNet, with lighter/darker bars denoting weights before/after magnitude pruning, and nonzero weights percentages indicated atop each bar.

- We design a JointNF compute engine that enables efficient matrix multiplication between N:M sparse weight and data.
- The evaluation results indicate the proposed JointNF solution can obtain a 2%-7% higher accuracy than other baseline pruning algorithms while reducing energy consumption by more than 5×.

2 Backgrounds and Related Work

DNN pruning can be categorized into two types: unstructured and structured pruning. Unstructured pruning [3, 4] involves removing each parameter individually, resulting in an uneven distribution of nonzero weights. While this method yields a highly sparse model, the irregularity of the sparsity pattern makes computing the sparse DNN challenging. To mitigate this, structured pruning was proposed, which removes DNN weights at a higher granularity level, such as filter-wise pruning [6, 12] within Convolutional Neural Networks (CNNs) and block-wise pruning for Transformer [14]. Although structured pruning generally leads to higher hardware utilization, it may come at the cost of suboptimal accuracy.

A middle ground between unstructured and structured pruning schemes is the N:M fine-grained structured sparsity, which has been recently proposed in literature such as [8, 15, 18]. In [18], the authors introduce an effective method called sparse-refined straight through estimator (SR-STE) to train the N:M sparse network by addressing gradient mismatch issues during backpropagation. However, this method enforces uniform sparsity across all layers of DNN, which can result in reduced accuracy when the sparsity level is high. In [8], the authors demonstrate that N:M sparsity can also accelerate the DNN training process. In comparison, our work proposes JointNF that allows for an adaptive sparsity pattern for the entire DNN, resulting in a sparse DNN that is amenable to efficient hardware implementation.

3 Joint Pruning Algorithms

The large solution space for pruning pattern selection makes joint N:M and structured pruning a challenging problem. To solve it, we first solve a simpler version of this problem: given the target number of filters to prune in a single layer, how does one perform joint N:M and structured pruning while minimally impacting the L1 norm? We propose an algorithm to solve this problem, and then describe our iterative JointNF algorithm in Section 3.1. Finally, we describe the NMReLU activation function in Section 3.2.

3.1 Joint N:M and Filterwise Pruning for CNN

Algorithm 1: NFFS Algorithm

Input: W is the weight matrix. N weights are kept for every M weights. U filters are removed from W.

1 **for** *every group of M weights in* W **do**
2 Find the top N weights with largest magnitude, remove the other smaller weights. // **Perform N:M pruning.**
3 Sort the rows of the N:M sparse matrix based on L1 norm.
4 Remove U filters with the smallest L1 norm.

We first consider the pruning strategy for CNN. Consider a four-dimensional weight tensor with C_{out} output channels, C_{in} input channels, and a kernel size of $k_w \times k_w$. We employ the *im2col* operation to transform the weight tensor into matrix format. This will convert a $C_{out} \times C_{in} \times k_w \times k_w$ weight tensor into a two-dimensional $C_{out} \times D$ weight matrix W, where $D = C_{in} k_w^2$. Under this notation, pruning a filter is equivalent to eliminating a whole row from W. Assume U filters are targeted for removal in W, and N:M pruning is applied to the remaining weights. We define a binary mask $\mathbf{S} \in \{0,1\}^{C_{out} \times D}$, where $S_{ij} = 0$ indicates that W_{ij} is removed and vice versa. To limit the impact of magnitude pruning, we aim to minimize the total change in L1 norm. Therefore, we need to find the mask S such that the removed weights have the smallest L1 norm. The detailed algorithm for solving this problem is described in Algorithm 1. We name this algorithm *N:M pruning First, Filterwise pruning Second* (NFFS).

The NFFS algorithm generates the optimal pruning selection with the minimum changes on L1 norm. To show this, let P denote the set that contains all of the smallest $M - N$ weights across the entire weight groups. The set Q is all the weights in the pruned filters. Our goal is then minimizing the L1 norm of all the weights contained in $P \cup Q$. $P \cup Q$ is equal to $P \cup ((Q \cap P) \cup (Q \cap P^c)) = P \cup (Q \cap P^c)$. This indicates that the set of eliminated weights includes all the weights in P (i.e., weights removed by N:M pruning) and $(Q \cap P^c)$. Since N and M are predefined, the total L1 norm of the weights in P is fixed. To minimize the L1 norm of the weights in $(Q \cap P^c)$, we can search for the weight filters with the smallest L1 norm on the resulting N:M sparse weight matrix.

As suggested by the NFFS algorithm, we have designed an iterative pruning approach, called JointNF, to progressively eliminate the smallest magnitude weights. An example of JointNF is given

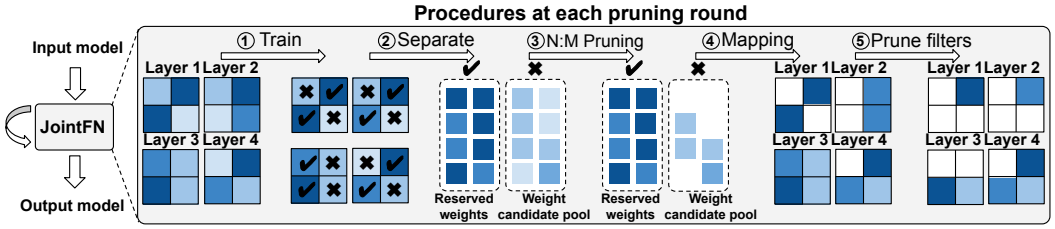

Figure 2: The JointNF algorithm for an illustrative two-layer DNN. The step numbers are shown in the circle. The weights with larger magnitude are shown with darker blue color.

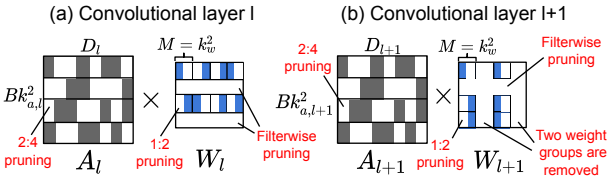

Figure 3: Computational cost and storage cost offered by JointNF with 1:2 weights and 2:4 data.

in Figure 2. For illustration purposes, we assume JointNF operates on a DNN with two layers and all the two weight matrices have the same shape of 2×4. The 2:4 pruning (N=2, M=4) is applied across the two layers and $U = 1$ filter will be removed at the current pruning round. At the beginning of a pruning round, the weights in two layers are first trained until convergence (step 1 in Figure 2). Within each row, the top N=2 weights with the largest magnitude (highlighted by ✓ in Figure 2), are separated from the other weights, called *local weight candidates* (X in Figure 2). The local weight candidates are then aggregated in a weight candidate pool (Step 2). After that, $p = 50\%$ of the smallest weights are pruned from the weight candidate pool (Step 3), where p is a hyperparameter that specifies the percentage of removed weights for N:M pruning at each pruning round. Then, each row of the intermediate weight matrix is sorted by their average L1 norm and the smallest filter, which corresponds to the second row of the weight matrix in Layer 1, will be eliminated (Steps 4, 5). This iterative process will repeat for multiple rounds until the weight candidate pool is exhausted and the target number of filters are removed. Between each round, the weights selected as local weight candidates may be different, but there are always at least N weights reserved for every consecutive group of M weights.

To identify and eliminate insignificant weight filters, we leverage the *network slimming* approach introduced in [12]. In network slimming, L1 regularization is applied to the scaling factors and bias factors in the Batch Normalization (BN) layers, pushing the unimportant scaling factors to a small value, which further allows us to identify the unimportant filters associated with the scaling factors. However, the original network slimming method does not consider N:M pruning and filterwise pruning in conjunction. In order to involve the impact of the scaling factors on the filter weights for performing JointNF pruning, we integrate the scaling factors to the filter weights by multiplying each scaling factor with its associated filter. The resulting weight filters will be pruned iteratively as shown in Figure 2.

3.2 N:M Pruning on the Data Activations

The extensive use of ReLU and ReLU6 in Deep Neural Networks (DNN), particularly in Convolutional Neural Networks (CNN), involves setting all negative outputs to zero. This process leads to a sparse input for subsequent layers. However, due to the irregular distribution of zeros, harnessing this sparsity for hardware efficiency becomes challenging. To address this issue, we introduce a new activation function called *NMReLU*. NMReLU ensures that within every consecutive group of M activation values, there are at most N nonzero values.

An example of NMReLU is shown in Figure 1(b). A 2:4 ReLU is applied to a 4×4 data matrix A. All negative data values are first converted to zeros by ReLU. In addition, the last two positive values on the forth column are also removed by the N:M sparsity constraint on this column. During the DNN training stage, only the gradients of the nonzero values are propagated to the earlier layers for weight updates. For iterative pruning, we progressively apply NMReLU to all the layers during the DNN training process.

3.3 Joint Pruning for Transformer

A similar strategy as described in Section 3.1 can be applied to the Transformer architecture. Specifically, we employ the JointNF technique across all the linear layers within both the self-attention and feedforward layers. In addition to achieving N:M sparsity in the linear layer weights, we also introduce structured pruning by eliminating entire rows from the weight matrices. This effectively reduces the embedding dimension for subsequent transformer blocks, resulting in a reduction in computation costs. However, it's important to note that since ReLU activation functions are not typically used within transformers, the input activations are generally not sparse. As a result, we do not apply NMReLU to transformers.

3.4 Computational and Storage Analysis

In this section, we provide a numerical analysis of the savings on computational cost and storage cost generated by JointNF over CNN. Let $\mathrm{A}_l \in \{\mathcal{R}^{B \times C_{in,l} \times k_{a,l} \times k_{a,l}}\}$ and $\mathrm{W}_l \in \{\mathcal{R}^{C_{out,l} \times D_l}\}$ denote the 2D data matrix and weight matrix of layer l, where B and $k_{a,l}$ represent the batch size and kernel size of A_l, respectively. Define $D_l = C_{in,l} \times k_w \times k_w$. $C_{in,l}$, $C_{out,l}$ and k_w represent the number of channels, number of filters and kernel size of W_l, respectively. Further assume that an N_w:M_w and N_a:M_a sparsities are applied on the weight matrices and data matrices, respectively, and U_l filters are removed from W_l. Then the number of operations in l-th convolutional layer is $\frac{N_w N_a}{M_w M_a} B k_{a,l}^2 C_{in,l} (C_{out,l} - U_l) k_w^2$, and the associated storage cost of

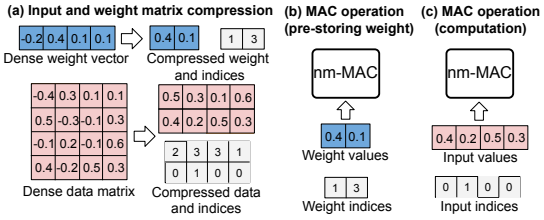

Figure 4: (a) Both weight and input matrix are pruned with 2:4 sparsity. (b) The nonzero weights are pre-stored at the nm-MAC before the computation. (c) The compressed data values will be delivered to the nm-MAC during DNN processing.

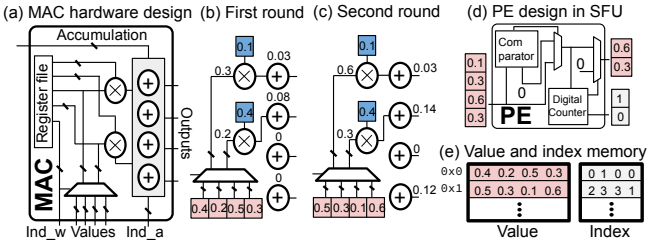

Figure 5: (a) shows the nm-MAC hardware design. (b) and (c) describe the operation of nm-MAC during the two rounds. (d) The special function unit design. (e) Memory pattern.

the weight and data will be $\frac{N_w}{M_w} C_{in,l}(C_{out,l} - U_l)k_w^2(\frac{N_w\lceil log_2 M_w\rceil}{M_w} + b_w)$ and $\frac{N_a}{M_a} BC_{in,l}k_{a,l}^2(\frac{N_a\lceil log_2 M_a\rceil}{M_a} + b_a)$, where b_w and b_a are the storage cost (in bits) for a single weight and data value, respectively. $\frac{N_w\lceil log_2 M_w\rceil}{M_w}$ and $\frac{N_a\lceil log_2 M_a\rceil}{M_a}$ are average number of bits per value to record the index information of N:M sparsity (See Figure 3 (a)). Additionally, the elimination of the i-th filter in Layer l will lead to the removal of the i-th feature map in the convolutional output. Since the scaling factors and biases associated with the pruned filters in the BN layer will also be tiny due to the L1 regularization, and the activation layer involves only elementwise operations, the input to the $(l+1)$-th convolutional layer will also has its i-th feature map eliminated. This will further induce the removal of the weight kernels that apply on the removed input feature maps. In JointNF, we make the group size M equal to the weight kernel size (M=k_w^2) so that the removal of the weight kernels will cause the whole weight groups to be pruned, generating a further reduction on computational and storage cost at layer $l+1$. The amount of operations in $(l+1)$-th convolutional layer will be $\frac{N_w N_a}{M_w M_a} Bk_{a,l+1}^2(C_{in,l+1} - U_l)(C_{out,l+1} - U_{l+1})k_w^2$, and corresponding weight storage cost will be $\frac{N_w}{M_w}(C_{in,l+1} - U_l)(C_{out,l+1} - U_{l+1})k_w^2(\frac{N_w\lceil log_2 M_w\rceil}{M_w} + b_w)$. A similar results can be derived for transformer, and we eliminate this due to space limit.

4 JointNF Hardware System Design

We now describe the design of the processor array that performs efficient inference by leveraging the N:M sparsity presented in both DNN weight and data values (if any). To enable a simple system design, we use a 2D systolic array to implement the computation engine. However, we note that our JointNF pruning paradigm can also support other computational engine designs.

The systolic array consists of multiple nm-MACs, where each nm-MAC performs the dot products between $M_a \times M_w$ data matrix and $M_w \times 1$ weight vector, where the data matrix and weight matrix are applied with $N_a : M_a$ and $N_w : M_w$ sparsities, respectively. A hardware design of a nm-MAC for 2:4 weight and 2:4 data is shown in Figure 5 (a), which mainly consists of two multipliers, four adders, and a register file. Figure 4 (a) shows an example of nm-MAC operation between a 4×4 data matrix and 4×1 weight vector, where 2:4 pruning is applied to both of them (i.e., $N_w = N_a = 2$ and $M_w = M_a = 4$). The compressed data and weight matrices will have a dimension of 2×4 and 2×1, respectively. Before the operation, N_w nonzero weight values and their associated indices are first buffered inside the register file (Figure 4 (b)). During operation, the nonzero data values together with their indices will be sent to the nm-MAC (Figure 4 (c)) for multiply-accumulate computation. The computation is performed within $N_a = 2$ rounds. During the first round of operation (Figure 5 (b)), the first row of the compressed data matrix, together with their indices, is sent to the nm-MAC, and $N_w = 2$ data values are selected according to the weight indices. The selected data will multiply with the pre-stored weight values, and the intermediate results will be directed and buffered at the corresponding output location according to the data indices. In the second round of operation (Figure 5 (c)), the second row of the data matrix will enter the MAC and similar operations will be executed, and the final output will be accumulated and forwarded to the nm-MAC on the right side.

The proposed nm-MAC design, coupled with N:M sparse weight and data matrices, can achieve significant hardware savings. Specifically, the nm-MAC design can perform matrix-vector multiplication between 4×4 data matrix and 4×1 weight vector only using two multipliers within two rounds. In comparison, matrix-vector computation between the dense 4×4 data matrix and 4×1 weight vector using a standard MAC would require eight multipliers in order to finish the computation in two rounds. In general, nm-MAC requires N_w multipliers to perform matrix multiplications between a $M_a \times M_w$ data matrix and $M_w \times 1$ weight vector with N_a rounds, while the standard MAC requires $M_a \times M_w$ multipliers to perform the multiplication. This results in large hardware savings given N_a and N_w are much smaller than M_a and M_w, respectively. More evaluation results can be found in Section 5.2.

The Special Function Unit (SFU) takes systolic array output and performs NMReLU. Each SFU has multiple PEs, each processing the outputs from a single row of the systolic array. A PE's architecture is shown in Figure 5 (d). A digital comparator compares input to zero, truncating negatives. Nonzero results feed a counter tracking positive values. Output is set to zero if the number of positive values exceed N_a out of every M_a data values. We also developed an efficient storage format for the N:M sparse input and weights, where the values and their indices are stored separately. Figure 5 (e) provides an example of storing a 2:4 sparse input matrix shown in Figure 4 (a). The weights are stored in a similar fashion, where non-zero values and corresponding indices are stored separately.

5 Performance Evaluation

We evaluate the performance of JointNF on multiple DNNs for different applications. For image classification task, we evaluate

Table 1: Test accuracies evaluation on different CNN-based Models.

Methods	ResNet-18		ResNet-50		MobiletNet-v2		VGG-16		Masked-RCNN	
	Accuracy	GFLOPs	Accuracy	GFLOPs	Accuracy	GFLOPs	Accuracy	GFLOPs	AP	GFLOPs
Dense	69.15 (0.00)	1.80	75.78 (0.00)	4.076	71.90 (0.00)	0.580	71.62 (0.00)	15.506	37.11 (0.00)	275.6
SR-STE	67.43 (-2.02)	0.618	74.93 (-0.85)	0.926	69.35 (-2.55)	0.227	70.87 (-0.75)	3.831	34.54 (-2.57)	97.7
ASP	66.79 (-2.36)	0.618	74.66 (-1.14)	0.926	68.93 (-2.97)	0.227	70.11 (-1.51)	3.831	33.07 (-4.04)	97.7
NS	62.58 (-6.57)	0.632	71.40 (-4.38)	0.920	65.12 (-6.78)	0.238	68.86 (-2.76)	3.187	29.15 (-2.05)	88.84
ThiNet	62.28 (-6.87)	0.590	71.33 (-4.45)	0.914	67.08 (-4.82)	0.244	68.62 (-3.00)	3.176	30.51 (-6.6)	90.48
GReg-2	62.71 (-6.44)	0.616	71.67 (-4.11)	0.931	67.35 (-4.55)	0.260	68.96 (-2.66)	3.202	29.45 (-7.66)	91.18
JointNF-4-4-0.2	68.45 (-0.70)	0.302	75.58 (-0.20)	0.653	71.22 (-0.68)	0.102	71.33 (-0.29)	2.787	35.89 (-1.22)	48.50
JointNF-6-4-0.1	69.03 (-0.12)	0.491	75.71 (-0.07)	0.919	71.72 (-0.18)	0.164	71.50 (-0.12)	3.660	36.43 (-0.68)	72.59

Table 2: Accuracies of ViT on ImageNet.

Methods	Dense		JointNF-4-0.3		SR-STE		ASP	
	Acc.%	GFLOPs	Acc.%	GFLOPs	Acc.%	GFLOPs	Acc.%	GFLOPs
ViT-base	75.88	4.56	75.57	1.092	74.13	1.16	74.23	1.16
ViT-large	76.95	15.48	76.43	4.80	75.25	4.92	75.02	4.92

ResNet-18, ResNet-50, MobileNet-V2, VGG-16 and vision transformer (ViT) [2] on the ImageNet (ILSVRC 2012) dataset (Section 5.1). We also evaluate JointNF on Mask R-CNN [5] with the COCO dataset for the objection detection task. Specifically, we compare JointNF with two categories of pruning algorithms: 1) Standard N:M pruning algorithms on DNN weights only (Nvidia ASP [1]) and their advanced variants (SR-STE [18]); 2) Filterwise pruning algorithms (Network Slimming (NS) [12], ThiNet [13], and GReg-2 [16]). For fair comparisons, we strictly apply the same pruning pipeline for all methods. All DNNs are trained for 60 epochs using a batch size of 128 with three pruning rounds. A uniform amount of weights are removed for either N:M pruning or filterwise pruning (or both) across the pruning rounds. For MobileNet-V2, we only prune the weights in the pointwise layer and keep the depthwise layer intact. For Masked R-CNN, we exclusively prune the backbone network. For ViT-base and ViT-large, we prune the linear layers within the self-attention layers and feedforward layers.

5.1 Performance Evaluation on DNNs

In this section, we evaluate the accuracy performance of JointNF together with NMReLU across different DNNs. Table 1 shows the performance of each approach on ImageNet for ResNet-18, ResNet-50, MobileNet-v2, VGG-16 and ViT. Since the kernel size of the pointwise convolutional layer in MobileNet-v2 is 1×1, we group the weights along the input channel dimension. For Masked R-CNN, we use ResNet-50 as the backbone network, and use Average Precision (AP) as the evaluation benchmark. For SR-STE and ASP, we apply a 3:9, 2:9, 3:9, 2:9, 3:9 and 3:9 pruning on DNN weights for ResNet-18, ResNet-50, MobileNet-v2, VGG-16, ViT and Masked R-CNN, respectively. For Network Slimming (NS), GReg-2 and ThiNet, we prune 60%, 70%, 60%, 80%, 70% and 70% of weight filters (or hidden dimension) on ResNet-18, ResNet-50, MobileNet-v2, VGG-16, ViT and Masked R-CNN, respectively. We evaluate two types of JointNF algorithms by using different pruning ratios: for the first type of JointNF (JointNF-4-4-0.2), we apply 4:9 pruning (NMReLU activation function with 4:9 ratio) on activations, and 4:9 pruning in conjunction with 20% filters pruned on DNN weights for all the DNNs. We also evaluate the performance of JointNF-6-4-0.1, which applies 6:9 pruning on activations and 4:9 pruning in conjunction with 10% filters pruned for weights.

From Table 1 we notice that, compared with the dense DNN implementation, JointNF-4-4-0.2 leads to an average of 3.60%, 2.41%, 3.38% and 1.638% improvements on test accuracy compared with the other baseline approaches for ResNet-18, ResNet-50, MobileNet-v2 and VGG-16 and ViT. JointNF-4-4-0.2 also outperforms all other algorithms in terms of AP and FLOPs for Masked R-CNN. Table 1 also shows JointNF-6-4-0.1, with a less aggressive pruning ratio, can achieve accuracy comparable to dense DNN, while incurring an average of 4.1× savings in FLOPs on average compared with the dense DNN. JointNF achieves the optimal performance because it leverages the sparity presented in both DNN weight and data, which is in contrast to the other approaches that only prune DNN weights. In addition, JointNF also enables flexible pruning patterns across different layers via filterwise pruning, thereby achieving superior test accuracy compared to other structured methods that exhibit the same level of sparsity.

Table 2 shows the accuracy evaluation over ViT-base and ViT-large. For ViT, we evaluate 4:8 pruning with 30% row-wise pruning (JointNF-4-0.3), and apply 3:8 pruning for the rest methods. NMReLU is not applied to ViT. We also observe a similar trend as CNN, where JointNF achieve the highest accuracies than other N:M pruning approaches, while resulting in the highest saving on computation cost.

5.2 Hardware Evaluation

In this section, we evaluate the performance of the nm-MAC described in Section 4. We have implemented and synthesized the nm-MAC using Catapult High-Level Synthesis (HLS) in a commercial 16nm process technology. The register-transfer level (RTL) design is synthesized from SystemC, and a C++ testbench was developed to validate functionality. We evaluate the efficiency of our nm-MAC design by comparing it against two other MAC designs: (1) bit-parallel implementations of a conventional MAC (pMAC) and (2) a MAC design that supports N:M sparsity only on the DNN weight (bMAC). We evaluate the two designs for the following multiply-accumulate operation: $Y_{out} = AW + Y_{in}$, where Y_{in}, Y_{out}, and W all have a shape of 9×1, and A have a shape of 9×9. Figure 6 (a) shows the layout for a pMAC, which performs multiplication between the weight and data and sums the result with the input accumulation in one cycle. To compute the results, we adopt an array of 9 pMACs, which can compute the dot product between two 9×1 vectors in one cycle. Therefore it will take 9 cycles to compute the results. The bMAC implementation which supports 3:9

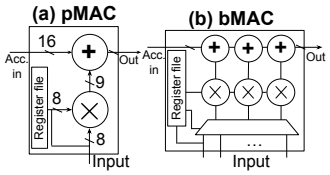

Figure 6: MAC designs.

MACs	Area efficiency	Energy efficiency
nm-MAC-4-4	1×	1×
nm-MAC-6-4	0.62×	0.64×
bMAC	0.56×	0.60×
9 pMACs	0.23×	0.21×

Table 3: Evaluation on MAC designs.

sparse weight multiplication is shown in Figure 6 (b), whose architecture is based on the design proposed by [9]. It consists of three multipliers, three adders and additional routing hardware. With a bMAC, it takes 9 cycles to complete the processing of the results. Finally, we implements two different nm-MACs for JointNF-4-4-0.2 and JointNF-6-4-0.1, respectively. The nm-MACs-4-4 enables matrix multiplication between a 4:9 input and a 4:9 weight, necessitating 4 cycles for computing the results. Similarly, the nm-MACs-6-4 supports matrix multiplication between a 6:9 input and a 4:9 weight, requiring 6 cycles to complete the process.

Given the processing latencies described above, we further measure the area and power consumption of each MAC design and compute the area efficiency and energy efficiency. As shown in Table 3, compared with pMAC and bMAC, nm-MAC-4-4 achieves the best performance in terms of both area efficiency and energy efficiency. Compared with nm-MAC-4-4, nm-MAC-6-4 has lower area efficiency and energy efficiency due to the higher processing latency, but it can still achieve on average 1.2× and 2.86× higher area and energy efficiency compared with bMAC and pMAC arrays.

We now evaluate the performance of our JointNF system described in Section 4. Specifically, we evaluate the performance of the JointNF system by comparing against the other DNN inference systems implemented with bMAC and pMAC. Additionally, we evaluate two different JointNF systems that are equipped with two types of nm-MACs described in Table 3. We call these two systems JointNF-4-4-0.2 and JointNF-6-4-0.1, respectively. We configure all hardware systems to have the same total area. Specifically, we are able to fit JointNF-4-4-0.2 and JointNF-6-4-0.1 systems with 16×16 and 15×15 nm-MAC systolic arrays, respectively. In contrast, with the same area, we can fit a 20×20 bMAC systolic array and 32×32 pMAC array. Each bMAC can multiply the dense input data with 3:9 weights. We quantize all the weight and data to 8 bits so that 8-bit multipliers and adders can be used in each type of MAC design. For the bMAC and pMAC system, the special function unit performs ReLU operation instead of NMReLU on the DNN outputs, and the dense outputs will be saved in the data SRAM with the original format without encoding. We implement the resulting sparse DNNs generated with JointNF-4-4-0.2 and JointNF-6-4-0.1 described in Table 1. In comparison, we deploy original dense DNN on bMAC and pMAC systems, respectively. The left side of Figures 7 show the area breakdown of the proposed JointNF-4-4-0.2 system. Notice that the majority of the area is consumed by the systolic array and memory subsystem, which take 37.2% and 31.3%, followed by the accumulator (18.5%) and special function unit (13.0%).

The right side of Figure 7 compares the energy consumption of JointNF-4-4-0.2, JointNF-6-4-0.1, pMAC, and bMAC hardware systems to process one ImageNet input sample. Specifically, the JointNF-4-4-0.2 system consumes an average of 3.9×, 4.3×, 3.5×,

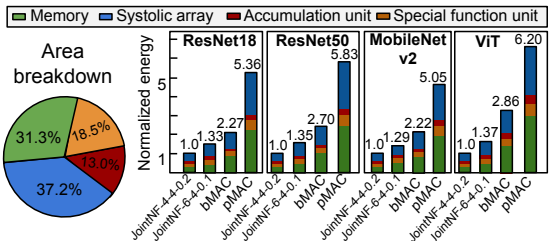

Figure 7: Energy consumption and the breakdown of the different hardware systems under different DNN workloads.

and 4.7× less energy for ResNet-18, ResNet-50, MobileNet-v2, and ViT, respectively, than the other baseline systems. The JointNF system achieves superior performance for two major reasons. Most importantly, the JointNF pruning algorithm, in conjunction with NMReLU, significantly reduces multiplication operations between the DNN data and weight. Second, given the N:M sparsity pattern presented in both DNN data and weight, nm-MAC designs can achieve significant latency and power reduction.

6 Conclusion

In this paper, we propose JointNF for efficient learning of joint N:M and filterwise sparsity. We also propose a JointNF hardware system to process the N:M sparse DNNs with high efficiency. Evaluations over multiple DNN models and datasets demonstrate that JointNF can achieve superior test accuracy while offering significant hardware performance improvements.

REFERENCES

[1] 2020. NVIDIA A100. https://www.nvidia.com/en-us/data-center/a100/.
[2] Alexey Dosovitskiy et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).
[3] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635* (2018).
[4] Song Han and et al. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *ICLR* (2016).
[5] Kaiming He and et al. 2017. Mask r-cnn. In *Proceedings of ICCV*.
[6] Yihui He and et al. 2017. Channel pruning for accelerating very deep neural networks. In *ICCV*.
[7] Yang He, Ping Liu, and Ziwei Wang. 2019. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *CVPR*. 4340–4349.
[8] Itay Hubara and et al. 2021. Accelerated sparse neural training: A provable and efficient method to find n: m transposable masks. *NeurIPS* (2021).
[9] HT Kung et al. 2019. Systolic building block for logic-on-logic 3d-ic implementations of convolutional neural networks. In *2019 IEEE ISCAS*. IEEE, 1–5.
[10] HT Kung, Bradley McDanel, and Sai Qian Zhang. 2018. Adaptive tiling: Applying fixed-size systolic arrays to sparse convolutional neural networks. In *2018 24th International Conference on Pattern Recognition (ICPR)*. IEEE, 1006–1011.
[11] HT Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 821–834.
[12] Zhuang Liu and et al. 2017. Learning efficient convolutional networks through network slimming. In *ICCV*. 2736–2744.
[13] Jian-Hao Luo et al. 2017. Thinet: A filter level pruning method for deep neural network compression. *arXiv preprint arXiv:1707.06342* (2017).
[14] Jiachen Mao et al. 2021. Tprune: Efficient transformer pruning for mobile devices. *ACM Transactions on Cyber-Physical Systems* (2021).
[15] Junghun Oh and et al. 2022. Attentive fine-grained structured sparsity for image restoration. In *CVPR*. 17673–17682.
[16] Huan Wang, Can Qin, Yulun Zhang, and Yun Fu. 2020. Neural pruning via growing regularization. *arXiv preprint arXiv:2012.09243* (2020).
[17] Wei Wen et al. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*.
[18] Aojun Zhou et al. 2021. Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch. In *International Conference on Learning Representations*.