PatterNet: Explore and Exploit Filter Patterns for Efficient Deep Neural Networks

Behnam Khaleghi*, Uday Mallappa*, Duygu Yaldiz, Haichao Yang, Monil Shah, Jaeyoung Kang, Tajana Rosing
Department of Computer Science and Engineering, UC San Diego, La Jolla, CA 92093
{bkhaleghi, umallapp, hcyang, m3shah, j5kang, tajana}@ucsd.edu

Abstract

Weight clustering is an effective technique for compressing deep neural networks (DNNs) memory by using a limited number of unique weights and low-bit weight indexes to store clustering information. In this paper, we propose `PatterNet`, which enforces shared clustering topologies on filters. Cluster sharing leads to a greater extent of memory reduction by reusing the index information. `PatterNet` effectively factorizes input activations and post-processes the unique weights, which saves multiplications by several orders of magnitude. Furthermore, `PatterNet` reduces the add operations by harnessing the fact that filters sharing a clustering pattern have the same factorized terms. We introduce techniques for determining and assigning clustering patterns and training a network to fulfill the target patterns. We also propose and implement an efficient accelerator that builds upon the patterned filters. Experimental results show that `PatterNet` shrinks the memory and operation count up to 80.2% and 73.1%, respectively, with similar accuracy to the baseline models. `PatterNet` accelerator improves the energy efficiency by 107× over Nvidia 1080 1080 GTX and 2.2× over state of the art.

ACM Reference Format:
Behnam Khaleghi*, Uday Mallappa*, Duygu Yaldiz, Haichao Yang, Monil Shah, Jaeyoung Kang, Tajana Rosing. 2022. PatterNet: Explore and Exploit Filter Patterns for Efficient Deep Neural Networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530422

1 Introduction

The ever-increasing efficacy of DNNs in diverse application domains is coupled with the increase in the size and computations of their models [1]. Extensive research has been done to alleviate the memory and computational burden of DNNs. Primary compression techniques include weight quantization [2–4], pruning [5, 6], clustering [6, 7], and filter pruning [8, 9], especially with a slant toward hardware efficiency such as hardware-aware quantization [10] and structured pruning [11].

In weight quantization, the network parameters take values from a set of predetermined values (e.g., -2^{k-1} to $2^{k-1}-1$ in uniform quantization), while weight clustering groups the weights into abstract clusters, where all weights of a cluster share the same value. Thus, by clustering we can simply store the cluster index/id of each weight (in *index table*), along with a small table that maps the indexes to weight values. Previous works [4, 12] show that ~16 unique weights can retain the accuracy, which results in 2× memory compression by storing $\log_2 16$ = 4-bit indexes instead of the primary 8-bit weights.

*Both authors contributed equally to this research.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530422

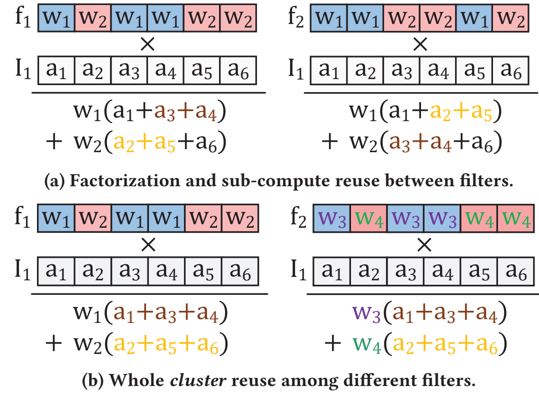

Figure 1: Weight clustering with (a) computation reuse and (b) cluster reuse between filters.

As shown in Figure 1, the convolution operation in CNNs is essentially a window-wise dot-product between a multi-dimensional filter and the input activations to generate output feature maps. Since clustering uses a limited number of unique weights, it can be leveraged for computation efficiency by factorizing weights. The example of Figure 1(a) shows filters clustered with two unique weights w_1 and w_2. Clustering can reduce multiplications (MULs) by first accumulating the inputs based on the weights clusters and applying MULs on the sum of factorized terms. For a filter with n_w weights (typically $O(10^3)$), the number of MULs reduces from n_w to **G** (the number of unique weights or clusters), where usually **G**=16 unique weights is sufficient as alluded earlier.

Factorization also results in common sub-groups of inputs. In Figure 1(a), both filters f_1 and f_2 computations have overlapping sub-groups $a_3 + a_4$ and $a_2 + a_5$. A previous study, UCNN [12], attempts to form compound sub-computations to be reused among multiple filters. Nevertheless, UCNN achieves less than 30% energy improvement due to the complexities involved in dealing with fine-grained sub-groups. FuseKNA [13] uses weights in a bit-serial fashion to slice MULs into ADDs. When processing each bit of multiple filters, FuseKNA reuses the overlapping ADDs among kernels.

In this paper, we take an unorthodox approach to increase computation reuse and reduce memory by enforcing filters to share the same clustering pattern. Filters f_1 and f_2 in Figure 1(b) share the same clustering. That is, a particular weight at index i of both f_1 and f_2 belong to the same cluster. This is distinguished by using the same background colors for clusters of f_1 and f_2. However, unlike baseline clustering [6] that uses the same unique weights, in

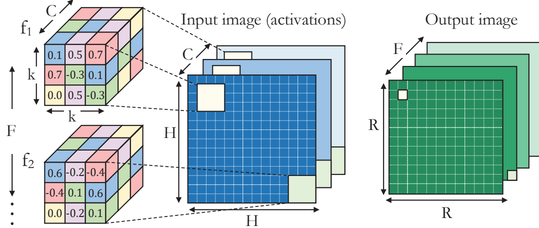

Figure 2: Convolution operation. Filters f_1 and f_2 use the same pattern with different unique weights.

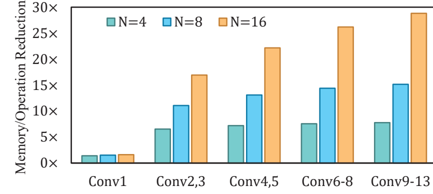

Figure 3: Memory and operation saving of patterned weight clustering when N filters share one pattern.

PatterNet each filter can have a different set of **G** unique weights; hence, *filters share clustering patterns*, not exact data. As a result of pattern sharing, along with the reuse of whole activation groups between f_1 and f_2 (e.g., $a_1 + a_3 + a_4$ is repeated for both filters), the same cluster-index information can be used for both filters. Therefore, f_2 only needs to store its unique weights set (which is negligible compared to the eliminated index information), and carry out only **G** MULs on the pre-computed input sub-groups that have already been accumulated when processing f_1.

Our main contributions are as follows. In Section 2, we explore the potentials of patterned filters, propose a mathematical formulation to identify the patterns, and a training strategy to enforce the desired patterns without deteriorating the model accuracy. To the best of our knowledge, this is the first work that introduces patterned filters to save memory and computation of DNNs. In Section 3, we elaborate the dataflow, architecture, and processing units of PatterNet accelerator that supports networks with both patterned and conventional weight clustering. As weight quantization is a special case of clustering, our architecture supports quantized networks, as well. In Section 4, we examine the efficiency of PatterNet on various datasets and networks in terms of computation and memory reduction and compare the functionally-verified synthesized PatterNet accelerator with previous works.

2 Patterned Neural Network

2.1 Motivation

Figure 2 shows the parameters of a convolution layer that comprises **F** filters of $C{\times}k{\times}k$ dimension. The depth of each filter, **C**, is equal to the number of channels (feature maps) of the input activations. An output pixel (activation) of output channel ℓ is created by applying the filter F_ℓ over a particular $C{\times}k{\times}k$ window of the input. Thus, the number of output feature maps is equal to the number of filters, **F**. Multiplication of a filter and input window is essentially a dot-product by flattening them. For an input with $H{\times}H$ channels, the output image has a dimension of $R{\times}R$, for $R = \frac{H-k}{S} + 1$, where S is the stride size (i.e., the sliding step of the filters).

Assuming every n_f subset of a layer's filters share the same clustering pattern, the total parameter memory consists of $C{\times}k{\times}k{\times}\log \mathbf{G}$ bits to store the common index table (i.e., cluster indexes of weights instead of values), and $n_f{\times}\mathbf{G}{\times}8$b bits to store the actual weights of n_f filters assuming 8-bit weights. The total number of operations include total $C{\times}k{\times}k$ ADD (in **G** groups/clusters), accompanied with **G** MULs and ADDs for each filter to generate an output.

That being said, Figure 3 shows the parameter memory and operation reduction of patterned VGG-16 layers over the 8-bit quantized model for $N{\in}\{4, 8, 16\}$ filters sharing one pattern assuming **G**=16 unique weights per filter. For intermediate layers, saving ranges from 7.2× to 22.1× depending on N, and up to 28.8× in the last layers. We assumed that a fixed number of filters share a single pattern. In practice, each pattern may contribute to a different number of filters of a layer as we elaborate in Section 2.2. Note that **G**=16 and 8-bit weights are a special case that leads to the same memory and operation savings; otherwise, the savings can be different.

2.2 Pattern Selection

Pattern selection involves determining the *number* of clustering patterns, the patterns themselves, and the assignment of patterns to filters. Exploring inter-filter structural similarities is a proper starting point in determining the common patterns and the filters that share these patterns. Patterning is more complicated than other problems such as filter pruning that considers the filters exclusively (e.g., pruning based on l_1 norms [8] or ranks of filters [9]).

We use Figure 4 to elaborate our proposed pattern selection approach. Using a pre-trained model, we cluster the weights of each filter to **G** groups using any conventional approach such as k-means. Note that this step is a simple one-shot clustering merely to reduce the number of unique weights of filters. In this illustrative example, filters f_1 and f_2 are clustered into four groups, distinguished by different colors. In patterned clustering, for flexibility, each filter can have an arbitrary set of unique weights different than other filters (denoted by G_{1-4} for f_1 and H_{1-4} for f_2 in Figure 4). The goal is to find the filters with most *similar* clustering, indicated by how many same-cluster weight indexes in f_i are also in the same cluster in f_j. A naive approach is to correspond each cluster of f_i with a cluster in f_j and count the overlaps, which results in **G**! combinations ($2{\times}10^{13}$ for **G**=16).

We formulate the "similarity finding" as the **Hungarian matching** problem. For each pair of filters f_i and f_j, we create the table of longest common subsequences between all groups, ending up in a **G**×**G** table. For instance, in Figure 4, cluster G_2 of f_1 has three common indexes with cluster H_4 of f_2, namely, indexes 2, 14, and 20. The Hungarian matching algorithm, with a time complexity of $O(\mathbf{G}^3)$, finds the best matching of f_i and f_j groups that maximizes the score (shared elements). The example of Figure 4 obtained a score of 20, meaning that by replacing clustering of f_1 with f_2, 20 (out of 27) weights of f_1 will be still in the same cluster as before (i.e., only 7 of f_1 weights get a different value).

We obtain the similarity scores between all pairs of filters and create an **F**×**F** distance matrix (distance defined as 1/score). Finally we use the distance matrix to find **P** (number of patterns) collections of filters, where filters of a collection have smaller distances to each other than to other collections. For this end, we use the **k-medoids** algorithm [14] to cluster **F** filters into **P** collections. Unlike k-means that calculates the Euclidean distance between data points, k-medoids works with custom cost functions, e.g., a distance matrix. In addition, unlike k-means, k-medoids returns actual data points of the collection as the center points, leading to a greater interpretability of the centers. This is essential in pattern selection as the returned centers will be the filters with their clustering pattern selected to be shared. Note that the number of filters in each of the **P** pattern collections can be different.

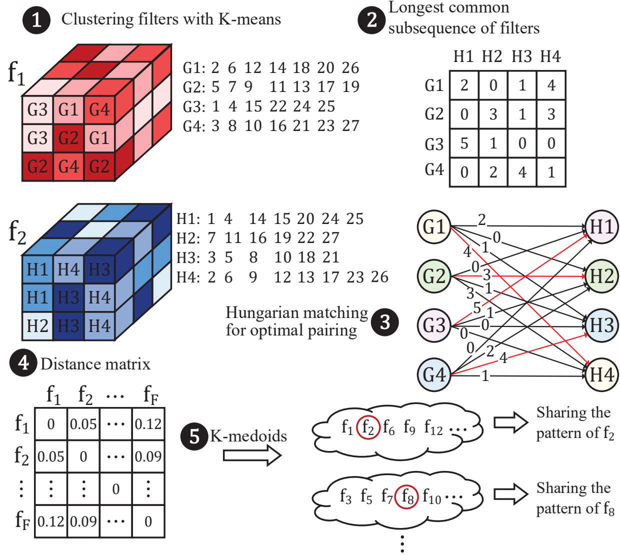

Figure 4: Selecting and sharing weight patterns.

2.3 Free Filters

Although imposing a limited number of patterns among all the filters works for simpler datasets such as Fashion-MNIST, in more complex datasets such as CIFAR100 we observe accuracy degradation. This is a result of failing to extract certain pixel patterns because of the cluster-sharing constraint between the filters. Therefore, we relax the constraint of pattern sharing on certain filters in a layer, dubbed as **free filters**. Free filters still comply with weight clustering (hence they still benefit from factorization) but do not follow an enforced pattern.

To select the free filters, in the original pretrained model, we sort the filters based on the singular value decomposition (SVD) of their output feature maps using the train data, according to [9]. SVD value indicates how many rows of a feature map are linearly independent. The overall rank score of a filter is the mean of the generated feature maps SVDs. Filters with a rank higher than a threshold are deemed as more informative filters and selected as pattern-free (or indeed single-pattern) filters.

2.4 Patterned Model Training

After identifying the patterns associated with each filter, we use **projected gradient descent** (PGD) to calibrate the model toward the determined patterns. PGD solves constrained optimization problems, which in our case is "the solution W of the DNN must belong to pattern constraints Q", formally, $\min_{W \in Q} f(\{W^i\}_{i=1}^{\mathcal{L}}, \mathcal{X})$, where $\mathcal{L}$ is the layers and $\mathcal{X}$ is the input data. Starting from an initial $W_0 \in Q$ (e.g., by cluster-wise averaging of pre-tarined weights) PGD proceeds as follows:

$$W_{k+1} = P_Q\Big(W_k - \lambda \nabla f(W_k, \mathcal{X})\Big) \tag{1}$$

P_Q projects the gradients such that $W_{k+1} \in Q$ as well. The projection of the gradients itself is an optimization problem:

$$P_Q(W_k) = \arg \min_{W \in Q} |W - W_k|_2^2. \tag{2}$$

Meaning that the new weights need to minimize $|W - W_k|_2^2$ while also adhering to Q. Since weights of the solution W are clustered, i.e. all weights of a cluster get the same value, the solution of (2) translates to minimizing $\sum(x - w_i)^2$ for each cluster, in which

Algorithm 1: Training process in `PatterNet`

Inputs: model (trained), $\mathcal{X}$, free_filters, pattern_dict, **G**
Output: `PatterNet` model
1: **for** *iter* from 1 to epochs×batches **do**
2: model ← **SGD**(model, $\mathcal{X}$)
3: **for** ℓ in model.convlayers **do**
4: **for** f in model.filters(ℓ) **do**
5: **if** f in free_filters[ℓ] **then**
6: model.weight[ℓ][f] ← *k*-means(f, **G**)
7: **end if**
8: **if** f in pattern_dict[ℓ] **then**
9: model.weight[ℓ][f] ← **project_weights**(f, pattern_dict)
10: **end if**
11: **end for**
12: **end for**
13: **end for**
14: **return** model

w_is are the post-gradient weights and x is the new weight of the cluster. Thus, $x = \overline{w_i}$ yields the optimal solution. Therefore, after backpropagation of each batch, we simply replace each updated weight with the average of its cluster. Algorithm 1 summarizes the `PatterNet` training, where the **project_weights** function of line 9 carries out the weight projection explained above.

3 PatterNet Architecture

3.1 Overview

Figure 5 shows the details of `PatterNet` architecture and data flow. The architecture comprises an $R_a{\times}C_a$ array of processing elements (PEs). Each PE is responsible for one pattern (which is shared with one or multiple filters) and generates one/multiple output pixels. PEs gradually receive all the inputs and *pattern cluster indexes* of a window, accumulate each input in the proper group based on the index, and eventually multiply the unique weights (for all filters sharing the pattern) on the accumulated groups.

To reduce the memory accesses, `PatterNet` uses a *pattern-stationary* data flow while trying to maximize the data reuse, as well. To this end, the PE array is logically split into *row-groups*, made up of two consecutive rows (total $R_a/2$ row-groups in our architecture). All PEs in a row-group operate on the same inputs *(intra* row-group data sharing), but each PE possesses a different pattern. Thus, a row-group generates multiple channels of an output. The corresponding PEs in all row-groups (e.g., PE_1, PE_{33}, etc.) possess the same pattern (*inter* row-group data sharing), but use different inputs. Therefore, in a given time, the same channels of $R_a/2$ outputs are on progress. Once all the channels associated with the running patterns are produced, `PatterNet` scans another input window to generate the next $R_a/2$ outputs. After scanning all input rows, `PatterNet` starts over with the next set of patterns (if any) and repeats the same procedure to generate all the channels.

3.2 Data Flow

We elaborate the data flow of `PatterNet` using the 3×3 example convolution of Figure 5 ❸. A brick is a complete 1×1 window that includes all the channels (z dimension). `PatterNet` fetches the input activations as *sub-bricks*. The number of channels (pixels) in a sub-brick is architectural parameter (e.g., four pixels). As shown in the figure, the convolution involving the input activation window $w1 = \begin{pmatrix} 13 & 12 & 11 \\ 8 & 7 & 6 \\ 3 & 2 & 1 \end{pmatrix}$ and the associated filter generates the right-most pixel of the output feature map. To do this, fetching of inputs starts from the bottom-right brick toward to top-left in a column-wise fashion (i.e., $1 \rightarrow 6 \rightarrow \cdots 13$) by fetching all sub-bricks commencing the next brick. This facilitates a great degree of data reuse as explained next in subsection 3.3. Once a sub-brick is fetched, it is broadcast

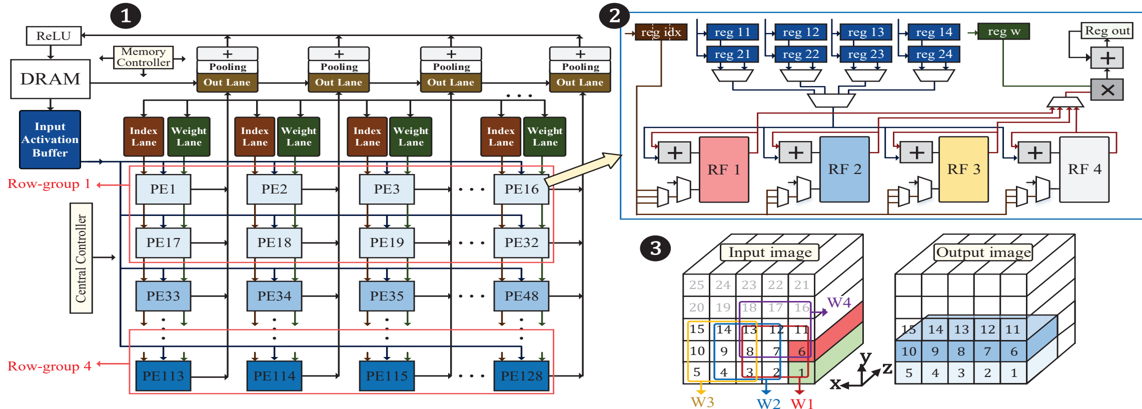

Figure 5: PatterNet (1) architecture, (2) processing element, and (3) data flow.

to all PEs in a row-group. Along with the inputs, each PE receives the pattern index corresponding to the fetched activations.

To recap, we first create activation sub-groups by adding cluster-specific activations, before multiplying with the cluster's weight value. To implement this, in every cycle, a PE processes one activation and adds it to the corresponding cluster group (out of **G**). After fetching and accumulating all the input bricks of an input window, each PE fetches the actual weights associated with the processed pattern. For each filter that shares the current pattern, the PE fetches its **G** unique weights cycle by cycle and multiplies with the accumulated values of group-1 to group-**G**. The aforementioned window $w1$ produces the output pixels associated with 32 patterns of PE_1 to PE_{32} of output brick 1 (i.e., at least 32 channels of the output feature map). The convolution window is then shifted left. Hence, the row-group 1 will generate the same channels of output brick 2 as it did for output brick 1.

Multiple row-groups generate multiple output rows simultaneously. As row-group 1 processes input window $w1$, row-group 2 processes window $w4$ to generate 2nd output *row*. All row-groups generate the same channels since they use the same patterns (hence, filters). Once the row-groups finish scanning the current input rows (i.e., the windows reach the left edge), each input window moves up by $^{R_a}/_2$ (number of row-groups) rows. After scanning all the rows, `PatterNet` starts over from the first row with a new set of patterns until all output channels are created.

3.3 Data Reuse

`PatterNet` takes advantage of multiple levels of data sharing. The input activations are shared among all PEs of a row-group, and clusters index data are shared between all corresponding PEs in the row-groups (e.g., PE_1, PE_{33}, PE_{65}, etc.). In addition, except the edge of the image, in a 3×3 convolution window, an input brick is shared between three windows of the same row. E.g., in Figure 5❸, input brick 3 is used in windows $w1$, $w2$, and $w3$ (processed by `RF1`, `RF2`, and `RF3` as explained in the next subsection). Therefore, once a sub-brick of input brick 3 is fetched, `PatterNet`'s PE processes computations for all the three windows (the PE also fetches three index data in a cycle). This results in ~3× speed-up in addition to memory access reduction. Furthermore, the kth row-group processes one input row ahead of its previous row-group $k-1$. `PatterNet` buffers the input to be reused later for row-group $k-1$ and avoids DRAM accesses with a small buffer. For instance, row-group 2 starts by operating on input brick 6, which will be immediately required by row-group 1 upon finishing input brick 1. Similarly, row-group 3 starts by input brick 11, which will be required by row-group 2 after processing input brick 6. This efficient data reuse is possible due to `PatterNet`'s data flow that simultaneously runs multiple vertically-adjacent windows, and processing each window in a column-wise fashion. Thus, when scanning the input image for the current patterns, each input is fetched only once from the DRAM.

3.4 Processing Units

Processing Elements: Figure 5❷ shows the internals of the PE. Top blue boxes are temporary registers `reg11` to `reg24` that store the activations sub-bricks fetched from the input buffer. PEs of different row-groups use the same input buffer bus in a time-multiplexed fashion. Thus, these registers are required to store enough inputs until the round-robin arbiter grants access to a row-group to fetch the next sub-brick after $^{R_a}/_2$ cycles.

Register Files: As explained in subsection 3.3, an input brick may participate in several adjacent windows. The register files `RF1` to `RF4` receive one input activation as data, along with several cluster indexes as the address to accumulate the input with the proper group. One of the `RF`s is spare to avoid stalls, explained below. The `reg idx` (index register) continuously fetches these index data from the `Index Lane` buffer. Since the windows sharing an input are adjacent (i.e., an activation only differs in x dimension within the windows), the index data of these windows can be aligned in one memory row. Note that since corresponding PEs of row-groups process the same pattern, the fetched index data is broadcast to all of $^{R_a}/_2$ corresponding PEs of all row-groups using the common index bus of a column.

Accumulator: Once all inputs of a window are accumulated in an `RF`, the PE loads unique weights w_1 to $w_{\mathbf{G}}$ one-by-one from the `Weight Lane` to the `reg w`, and reads the accumulated sum of group-1 to group-**G** from that `RF`, accumulates the multiplications in the `reg out`, and finally transfers the output to `Out Lane`. Since each filter sharing a pattern has its own unique weights, these multiplications need to be repeated for all filters sharing the pattern. The key benefit of `PatterNet` is that, once the input sub-groups are computed a pattern, producing new output channels (of shared filters) takes just **G** cycles per filter. Since the first window (of horizontally-adjacent windows) is several input bricks ahead from the other two, in a given time, the results of only one window becomes ready in a PE. A PE contains one extra `RF`, so when an `RF` is stuck to finalize the multiplications, the fourth `RF` replaces it to process new input bricks and avoid stall.

Output Lanes: PEs in a column time-multiplex the same output bus to transfer the output activation to the `Out Lane`. The bus

is granted in a round-robin fashion, but it does not cause performance overhead as outputs of all PEs of a column can be transferred to `Out Lane` before generating the outputs of next window. The `Out Lane` temporarily stores a few adjacent horizontal outputs (from the same PE), or adjacent vertical outputs (from the *corresponding* PEs of different row-groups) for pooling operation before writing to DRAM. The output data layout written into the DRAM is the same as input bricks, i.e., continuous pixels of an output brick are written in the same DRAM row.

4 Experiments and Results

4.1 Experimental Setup

We implemented `PatterNet` concepts (i.e., pattern and rank-based free filter selection and training) using PyTorch. For training, we used SGD optimizer, momentum of 0.9 with weight decaying, and learning rate from 0.1 down to 0.0008 over 100 epochs. For parameter **G** (number of unique weights or clusters per pattern) we found **G**=16 sufficient to retain accuracy by sweeping across a spectrum of values. Similarly, we tried a range of values for **P** (number of patterns) and found **P**=16 sufficient for accuracy.

We implemented `PatterNet` accelerator in SystemVerilog and verified its functionality with Modelsim. We synthesized it using TSMC 40 nm standard cell library at 0.9 V using Synopsys Design Compiler for a target frequency of 500 MHz. We used Artisan memory compiler with the same technology to generate SRAM buffers and register files. Power consumption of all elements is obtained using Synopsys Power Compiler. For DRAM access energy model we used Destiny [15]. Our primary architecture consists of R_a=8 rows (four row-groups) and C_a=16 (32 PEs per row-group).

4.2 Operation and Memory Reduction

We evaluate the effectiveness of `PatterNet` by comparing it with a state-of-the-art filter pruning approach dubbed Hrank [9]. We use VGG16, Resnet18, and Resnet50 networks with CIFAR10 and CIFAR100 datasets as used in [9], and a 200-class subset of ImageNet (Tiny ImageNet). To recap from Section 2.1, the *patterned* filters run ADDs to accumulate the input activations for **P** filters, followed by MULs of their unique weights on the resulted groups. The *free* filters are special cases of patterned filters, where a free filter has one independent pattern. Thus, free filters also benefit from factorization to reduce the number of MULs, as well as weight clustering to reduce memory.

Table 1 summarizes the accuracy, operation count (ADD and MUL), and memory for the aforementioned models and datasets. The *Base* column indicates the baseline 8-bit model, and *Hrank* column is the state-of-the-art filter pruning [9]. We selected the pruning ratios of Hrank layers according to its original work [9].

CIFAR10: As compared to the baseline VGG16 network, while HRank provides 56.1% reduction in operation count and 62.2% reduction in parameters, our method offers 72.4% reduction in operation count and 77.9% reduction in parameters, with 0.3% better accuracy. For residual networks such as ResNet18, while the operation reduction in HRank is 54.4%, our method offers 69.4% reduction. We observe a similar trend for ResNet50; 68% operation reduction in PatterNet as compared to 46% reduction of HRank. PatterNet shrinks parameters size significantly (80.2% vs HRank's 66.8%) for Resnet18 and (64.1% vs HRank's 45.7%) for Resnet50, along with better accuracy metrics as compared to HRank.

CIFAR100: For CIFAR100, we achieve 73.1% operation count reduction using VGG16, 61.5% using ResNet18 and 68.6% using ResNet50. The reduction in parameters is considerably better than HRank's reductions (77.4% vs 61.1%, 71% vs 48.8% and 64% vs 46.2%) for VGG16, ResNet18 and ResNet50 respectively.

TinyImageNet: We observe a similar trend with the Tiny ImageNet dataset. Along with an improved operation reduction (up to 72%) and parameter reduction (up to 70.7%) as compared to the baseline, our improvements are better than HRank while achieving improved accuracy metrics (1-2%) over HRank.

In summary, PatterNet shrinks the model memory up to 80.2% and operation count up to 73.1%, with a similar accuracy as compared to the 8-bit baseline models.

Table 1: Comparing PatterNet with baseline and Hrank [9].

	Model	Accuracy			Operation (M)			Parameters (MB)		
		Base	Hrank	PatterNet	Base	Hrank	PatterNet	Base	Hrank	PatterNet
CIFAR10	VGG16	91.73%	91.89%	92.19%	314.0	137.7 (56.1%)	86.7 (72.4%)	14.28	5.39 (62.2%)	3.14 (77.9%)
	Resnet18	93.84%	93.14%	93.59%	555.5	253.2 (54.4%)	169.7 (69.4%)	10.64	3.53 (66.8%)	2.10 (80.2%)
	Resnet50	94.65%	94.12%	94.64%	1298	701.2 (46.0%)	416.5 (68.0%)	22.3	12.16 (45.7%)	8.04 (64.1%)
CIFAR100	VGG16	70.45%	69.84%	70.15%	312.0	132.4 (57.6%)	84.1 (73.1%)	14.26	5.55 (61.1%)	3.22 (77.4%)
	Resnet18	75.30%	74.19%	74.83%	555.5	341.1 (38.6%)	213.8 (61.5%)	10.69	5.47 (48.8%)	3.09 (71.0%)
	Resnet50	77.21%	76.12%	76.92%	1298	682.1 (47.4%)	407.5 (68.6%)	22.5	12.10 (46.2%)	8.1 (64.0%)
Tiny-ImageNet	VGG16	56.95%	53.16%	55.90%	1272	549 (56.8%)	355 (72.0%)	22.79	14.1 (38.2%)	11.7 (48.4%)
	Resnet18	62.28%	60.97%	62.20%	2221	1364 (38.6%)	854 (61.5%)	10.74	5.52 (48.5%)	3.14 (70.7%)
	Resnet50	64.20%	62.65%	63.88%	5192	2727 (47.5%)	1629 (68.6%)	22.75	12.3 (45.8%)	8.31 (63.4%)

Table 2: Memory size of baseline PatterNet architecture.

Input buffer	Index lane	Weight lane	Out lane	Register File
2048×32b (8 KB)	768×24b (2.25 KB)	64×8b (64 B)	512×20b (1.25 KB)	16×20b (40 B)

Table 3: Characterization of PatterNet components.

Module (one)	Area (μm^2)	Leakage (mW)	Dynamic (mW)
Input Buffer	33,284	0.454	0.563
Index Lane	15065	0.171	0.426
Weight Lane	1,744	0.030	0.034
Out Lane	10,961	0.118	0.356
PE (with RFs)	9,472	0.173	0.470
Controller	151,029	1.701	2.518

4.3 PatterNet Accelerator Details

The baseline `PatterNet` architecture consists of four row-groups (R_a=8) and 16 columns (C_a=16). Table 2 reports the size of `PatterNet` memories. As explained in Section 3.3, the *input buffer* stores the entire brick of a row-group for reuse by the preceding row-group. The image depth goes up to 2048 channels in Resnet50, thus, the input buffer should store 2048×4 input activations of four row-groups, packed as 2048×32b (four inputs of a brick are packed in a row and fetched at once to a row-group). The *index memory* stores all 4-bit indexes, which is 512×3×3 for the largest filter. Since three indexes per pattern is read in a column (and there are two patterns in a column), the memory has a 768×(6×4) layout. The *weight memory* supplies the unique weights of a column's filters. Each pattern is shared with up to 32 filters, thus, it stores up to 64 weights. Similarly, the *out lane* stores all outputs generated by a column (four row-groups and 64 filters). In addition, it stores the adjacent pixels for pooling, requiring a total of 512 rows and 20-bit per row for each output pixel. Finally, each *RF* has 16 rows for accumulation of **G**=16 groups.

Table 3 shows the per-component area and delay of the `PatterNet` using the setup of subsection 4.1. The 8×16 architecture of `PatterNet` occupies an area of 1.84 mm^2 (at 40 nm). The compact area is mainly due to sharing a weight index lane and an output lane within an

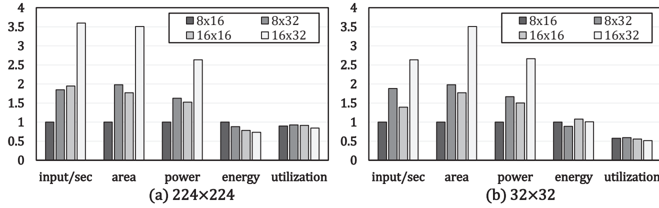

Figure 6: Scalability of performance, power, and energy of PatterNet running VGG16 with (a) 224×224 (b) 32×32 images.

Table 4: Metrics of 8×16 array for large and small images.

Image size	Input/sec	Area (mm^2)	Power (mW)	Energy/input (mJ)
224×224	11.3	1.84	133.6	11.9
32×32	415.3	1.84	96.3	0.23

entire column, and a small input activation memory that buffers the inputs for reuse so `PatterNet` uses only 70 KB on-chip memory. The design consumes a peak (wost-case) power of 145.7 mW: 29.4 mW leakage, and maximum dynamic power of 116.3 mW (at 500 MHz), 34% of which is the DRAM access power. The data reuse of `PatterNet` makes an effective DRAM access rate of ∼1 Byte/cycle, the same rate as PEs consume inputs in a shared fashion.

4.4 PatterNet Scalability

Figure 6 shows the scalability of `PatterNet` (implementing VGG16 model) as the array size increases from 8×16 to 8×32 (2× columns), 16×16 (2× rows), and 16×32 (2× columns and rows) for 32×32 images and ImageNet-scale 224×224 images. The area in both cases is the same and input-independent. Except for the PE utilization that shows the actual quantities, the other parameters are normalized to 8×16 array values (Table 4 shows the actual values of the baseline 8×16 architecture). For large images, `PatterNet` architecture shows better scalability, i.e., 3.6× higher performance (input/sec) when both rows and columns duplicate. However, for small images, PE utilization rate reduces down to 46% in the 16×32 array. As a result, it achieves only 2.6× performance gain. The average utilization rate for large images is 90% in the baseline 8×16 array and 76% in the largest array. The area is *not* scaled by 4× since the size of index lane and weight lane buffers remains the same, and their number only increases by 2×. Finally, for large images, the largest array (16×32) shows better energy/input. This is mainly because the DRAM access power ratio significantly reduces (down to 9.3%) because the fetched inputs are reused between more row-groups.

4.5 Comparison with Previous Work

We compare performance-per-watt of `PatterNet` with the state-of-the-art FuseKNA [13] which also reuses the overlapping ADDs among kernels in a bit-serial accelerator, and with SCNN [16] which is a MAC-based sparse (zero-skipping) accelerator (results compiled from [13]). Figure 7 shows the performance-per-watt (energy per image) normalized to Nvidia 1080 GTX GPU, all designs running 224×224 images. `PatterNet` surpasses GPU energy efficiency by 107×, SCNN by 3.6×, and FuseKNA by 2.2×.

5 Conclusion

In this paper, we introduced the concept of patterned cluster sharing between DNNs filters, which achieves memory reduction by reusing the clustering indexes, and operation reduction by using weight factorization and reusing the result among the filters of the same cluster. We proposed techniques to determine and assign the patterns over the filters, as well as a training approach

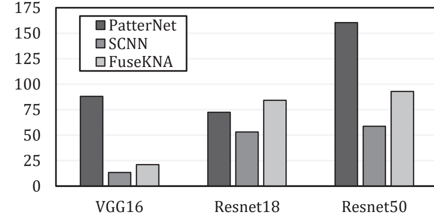

Figure 7: Comparison of PatterNet, SCNN [16], and FuseKNA [13] energy efficiency normalized to GPU.

to yield the target patterns. We evaluated the filter patterning using different datasets and networks, which revealed its effectiveness in significant memory and operation reduction (by 80.2% and 73.1% respectively), which surpassed the state-of-the-art filter pruning technique while achieving better accuracy. We also proposed `PatterNet` accelerator based on the above ideas, which obtained 2.2× better energy efficiency than state-of-the-art accelerators.

Acknowledgements

This work was supported by TSMC, in part by CRISP, one of six centers in JUMP (an SRC program sponsored by DARPA), SRC Global Research Collaboration (GRC) grant, and NSF grants #1911095, #1826967, #2100237, and #2112167.

References

[1] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.
[2] M. Rastegari *et al.*, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*, pp. 525–542, Springer, 2016.
[3] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.
[4] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *arXiv preprint arXiv:1702.03044*, 2017.
[5] A. Ren, T. Zhang, *et al.*, "Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 925–938, 2019.
[6] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
[7] S. Ye, T. Zhang, *et al.*, "A unified framework of dnn weight pruning and weight clustering/quantization using admm," *arXiv preprint arXiv:1811.01907*, 2018.
[8] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.
[9] M. Lin *et al.*, "Hrank: Filter pruning using high-rank feature map," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1529–1538, 2020.
[10] S.-E. Chang, Y. Li, *et al.*, "Mix and match: A novel fpga-centric deep neural network quantization framework," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 208–220, IEEE, 2021.
[11] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 1389–1397, 2017.
[12] K. Hegde *et al.*, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *International Symposium on Computer Architecture (ISCA)*, pp. 674–687, 2018.
[13] J. Yang *et al.*, "Fusekna: Fused kernel convolution based accelerator for deep neural networks," in *International Symposium on High-Performance Computer Architecture*, pp. 894–907, 2021.
[14] L. Kaufman and P. J. Rousseeuw, "Partitioning around medoids (program pam)," *Finding groups in data: an introduction to cluster analysis*, vol. 344, pp. 68–125, 1990.
[15] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1543–1546, IEEE, 2015.
[16] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.