2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-1016-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/HIPC53243.2021.00060

A Fused Inference Design for Pattern-Based Sparse CNN on Edge Devices

Jia Guo
Computer Science and Engineering
Ohio State University
Columbus OH 43210
Email: guo.980@osu.edu

Radu Teodorescu
Computer Science and Engineering
Ohio State University
Columbus OH 43210
Email: teodores@cse.ohio-state.edu

Gagan Agrawal
Computer and Cyber Sciences
Augusta University
Augusta GA 30912
Email: gagrawal@augusta.edu

***Abstract*—Weight pruning approaches for Convolution Neural Networks (CNN) has been well developed in the past years. Compared with traditional unstructured and structured pruning, the new state-of-the-art sparse convolution pattern (SCP) based pruning uses certain patterns that lead to both high pruning rate and low accuracy loss. This paper introduce a novel inference scheme to accelerate the execution of SCP-pruned models on IoT devices with limited resources. This inference scheme applies and combines ideas from direct sparse convolution and layer fusion. To fully utilize the power of modern IoT processors, the inference is also mapped to all available cores and optimized with SIMD instructions. The experimental results show good performance improvement as well as scalability of our scheme on an edge device.**

I. Introduction

Beyond the success and popularity of smartphones and their applications, AI and ML are moving towards smaller devices, such as wearable computing and Internet of Things (IoT). Broadly, such *edge* devices are cheaper but with lower processing capabilities and larger constraints. Thus, there is a growing need to further optimize DNN inferencing to match these devices. Specifically, unlike smart phones equipped with dedicated neural network accelerators, edge devices typically have only CPUs.

Research has started towards supporting DNN inferencing on these platforms [2], [3]. However, with more complex models (and thus requiring more memory and computing power) required for achieving high accuracy, many challenges still exist in using edge devices for DNNs. Much attention has been given to the problem of reducing the high computational needs associated with the inferencing step while using Deep Neural Networks (DNNs). To this end, various DNN *model compression* techniques have been proposed [17], [19], [20], [23], [29], [31]. Weight pruning is a representative model compression technique, where 90% or more weights are reduced to zero. What is often done can be referred to as *fine-grained non-structured pruning* and has the advantage of significant compression or reduction in the number of operations needed for inferencing, while maintaining high accuracy [9], [11], [14], [20]. However, this also has the disadvantage that dense computations for inferencing are now replaced by sparse computations. The result is that despite a significant reduction in the number of operations involved, speedups obtained are marginal at best.

Lately, it has been shown that it is possible to prune the DNN in a fashion that resulting computations can be more suited for optimized execution on complex architectures Along these lines, to preserve fine structure and high accuracy in pruned models, while not sacrificing to much regularity, researchers introduce the *Sparse Convolution Pattern (SCP)* based pruning [25]. This approach requires that each 2-D kernel pattern, characterized by shape of non-zero weights, should be selected from a limited set of patterns. Therefore, it grants some freedom of pruning within one kernel, but also ensures regularity among all kernels, which can be leveraged by parallelism in modern architectures.

Independent of sparsification, a different class of techniques for improving execution has been loop fusion, where the goal is utilizing memory (hierarchy) more effectively [3], [4], [6]. By fusing two (or more) layers, one can reduce the number of memory accesses and improve cache reuse.

Building on top of the success of both pattern-based pruning and loop fusion for DNNs, this paper examines techniques where the two are combined. To improve the inference efficiency of sparse models on IoT devices while maintaining model accuracy, we propose a novel inference scheme that is able to leverage the pattern regularity in pruned CNN layers. Additionally, building on top of our previous work on the combination of direct sparse convolution and layer fusion [13], we further improve the locality by fusing the execution. We conduct several experiments using popular models and demonstrate that pattern based optimization can improve the efficiency of execution by up to 19% on top of direct sparse convolution. Furthermore, the SCP optimization can be combined with fusion to further speedup the inference.

II. Background and Motivation

A. Existing CNN Optimizations

Weight Pruning: Han *et al.* [15] introduces pruning, which was based on the insight that the extensive weights in convolution models are small and/or unimportant. Thus, pruning out these weights through heuristic methods can yield high model sparsity with moderate accuracy loss. Latter work [27] further develops this idea by introducing ADMM regularization. This type of *unstructured pruning* approaches leave irregular zeros in the weight tensor. To avoid such inefficiency, *structured pruning* is proposed to prune weights with regular patterns altogether. For example, the most common of these, *channel* *pruning*, will prune out an entire 2-D kernel as the lowest granularity. The resulting models are more amenable to optimized execution [12], [19], [22], with some even capable of reusing the highly optimized BLAS process [19].

Sparse Convolution Patterns (SCP): A more recent trend is to find the sweet spot between fully structured and unstructured pruning, so that the pruned models can be both highly accurate and regular. The state-of-the-art pruning scheme [21], [25] introduces a novel sparsity dimension, Sparse Convolution Patterns (SCP). To elaborate, authors first defined is a pool of SCPs, where the number of non-zero weights in each pattern is equal. This pool contains all the candidate patterns that can be taken by 2-D kernels in the sparsified CNN models, i.e., the shape formed by non-zero weights in each kernel should be found in this pool. In practice, this pool is either selected statically from the most frequently used patterns in pre-trained models [25], or dynamically generated during training [30]. A empirical number of patterns is $4 - 8$ [25]. Additionally, the SCP-based pruning also incorporates channel pruning, meaning certain relatively unimportant 2-D kernels can be pruned out entirely. This enables a widely adjustable range of sparsity for pruned models, not limited by fixed sparsity of the SCP pool.

This pattern based pruning design demonstrates comparable accuracy to non-structurally pruned models and far outperforms structured pruning under the same compression rate [25]. Furthermore, it is more flexible in terms of patterns and pruned channels. It also demonstrates desirable inference speedup compared with non-structured pruning due to higher access regularity and similarity among the patterns.

Direct Sparse Convolution (DSConv): Another important background for our presentation is Direct Sparse Convolution (DSConv) [26]. This method is designed to replace the underlying General Matrix Multiplication (GeMM) libraries, which require that the input feature map be replicated several times, and has significant overheads both computationally and in terms of memory requirements.

As described in Algorithm 1, the DSConv takes as input convolution filters (weights) in a popular sparse representation, i.e., the compress sparse row (CSR) format. Each non-zero value is then multiplied with a sub-matrix in the input map, where this non-zero value will be applied and we call this sub-matrix the *window to apply* for the corresponding non-zero weight. Since the introduction of this concept [26], it has seen deployment in popular compilation systems Tiramisu [5] and Escoin [7].

Layer Fusion: Layer fusion is a concept based on the popular loop fusion idea from optimizing compilers. In context of DNNs, it has been used in several projects, including Tiramisu [5] and TensorRT [1]. A common implementation involves a pixel-wise fusion of a convolution layer with its following bias, normalization, or activation layers. In SkimCaffe, for example, the convolution layer is fused with its following activation, pooling, and normalization layers. Much more limited is the possibility of fusing two consecutive convolution layers (see Alwani *et al.* [4]). In Alwani *et al*'s work, tiling and reordering are applied first, and used to feed partial results from one layer directly to its following layer. We can also view it as replacing a breadth-first or layer-by-layer execution with an execution that is depth-first.

Algorithm 1 Direct Sparse Convolution

```
1: procedure PREPROCESSING(W, $D_{F1}$, $D_{F2}$)
2:     for n in [0,N] do
3:         for j in [W.kernel_ptr[n], W.kernel_ptr[n+1]) do
4:             W.kernel_offset[j] ← offset(j, $D_{F1}$, $D_{F2}$, $D_K$)
5:             //Calculate the first offset of window to apply
6:         end for
7:     end for
8: end procedure
9:
10: procedure DSCONV(in, W, out)
11:     for n in [0,N] do
12:         for j in [W.kernel_ptr[n], W.kernel_ptr[n+1]) do
13:             offset ← W.kernel_offset[j]
14:             val ← W.value[j]
15:             for (h,w) in $[0, D_{F1}] \times [0, D_{F2}]$ do
16:                 iter ← $h \times (D_{F2} + padding) + w$
17:                 out[n][h][w] += val × in[offset + iter]
18:             end for
19:         end for
20:     end for
21: end procedure
```

B. Motivation

Our preliminary research shows that the computational workload and memory requirement for dense CNN is unrealistic for deployment on resource-limited edge devices. Specifically, we executed four popular models {CaffeNet, GoogleNet, RCNN, Vgg_cnn_s} using Caffe on a Raspberry Pi 3B. The latency of execution is $\{16s, 30s, 16s, 48s\}$ for four models, respectively, and the peak memory consumption ranges from 690MB to 1160MB. Such latency level obviously fails requirements for real-time services. Meanwhile, extensive computational/spacial complexity leads to high cost of single device and consumption of power, which diminishes the coverage of edge network. Therefore, extensive sparsification and optimization is obviously necessary for CNN models in the IoT context.

Among the three pruning approaches we discussed in Section II-A, non-structure pruning is the one with most generality and accuracy, but it suffers from efficiency problem in the inference process. Specifically, unpredictable zeros in the weight tensor makes it hard to optimize. Meanwhile, existing sparse matrix multiplication (SpMM) routines typically targets scientific applications with extremely high sparsity ($> 99\%$) [18], [24] and do not fit for sparse convolutional neural networks (70% - 95%) sparsity. On the other hand, although structured pruning leaves highly regular sparsity in the kernels, facilitating optimization based on general matrix multiplication (GeMM), its accuracy is often compromised [8], [21]. In comparison, SCP-based pruning shows desirable accuracy and disable access regularity. Its adaptability and performance under high sparsity makes a good fit with the IoT context.

DSConv demonstrates good potential in improving the locality and reducing the memory requirement of sparse CNN inference. Yet its implementations [7], [26] are not on IoT platforms. Layer fusion is commonly adapted in many cases [1], [5], except while fusing two dense CNN layers due to the implementation overhead. However, we believe this will not be a problem in sparse layer inference and its ability to reduce intermediate results is valuable in the edge environment.

Therefore, we aim to improve the access locality and memory efficiency of SCP-based sparse models on edge devices, using the techniques of DSConv and fusion between CNN layers. We also want to effectively use parallelism that is seen even in inexpensive IoT devices.

III. Fused SConv Inference Scheme Design

A. Preliminaries

Based on the previous work of Sparse Convolution Patterns and layer fusion, we propose a novel CNN inference scheme, aiming to provide a efficient way of mapping a CNN models with pattern-based sparsification on multi-core IoT platforms. Specifically, we set the following targets: 1) Regularizing access patterns in sparse convolution inference for better cache performance and vectorization potential, and 2) Improving memory locality and efficiency using layer fusion while avoiding its overhead, and 3) Achieving effective parallelism on edge devices with multi-core processors.

To achieve the goals above, we complete our design in four key steps. First, a format to compress models trained with SCP is designed. This format groups the kernels across different filters by SCPs. Based on this format, we propose a single-layer CNN inference scheme leveraging DSConv concept, with special optimizations for the grouped weights. Then, we introduce the idea of *scattering DSConv* [13] and show how it can be used to fuse the evaluation of two convolution layers. Finally, we show how this new scheme can be tiled and mapped to multi-core structure and optimized with SIMD instructions.

B. Compression Format for Sparse filters

As we have discussed in Section II, the sparse filters in DSConv are compressed in filter-major CSR format. In our work, a new format is introduced to support efficient execution of Fused SCP. Our new format comprises of two parts, as show in Figure 1. The first part defines all patterns by their window offsets. The second part stores the kernels ordered by channel id and pattern id. First, we allocate an array *W.pattern_offset[]* of size *NNZ* $\times N_{scp}$ to store all the patterns, where *NNZ* is the pre-defined number of non-zeros in each pattern and N_{scp} denotes the count of distinct patterns. We identify each pattern using *p_id* $\in \{0, 1, \ldots, N_{scp} - 1\}$. Naturally the pattern with *p_id* is recorded in W.pattern_offset[*NNZ*×*p_id* : *NNZ*×(*p_id*+1)]. Values in *W.pattern_offset[]* are the *window to apply* offsets of non-zero weights. However, unlike the *W.filter_offset[]* in standard DSConv, *W.pattern_offset[]* does not include any channel offset information, since they are only used to describe patterns in 2-D kernel. In another way, they are calculated as if their corresponding *windows to apply* all locate in Channel 0.

Now that the patterns are defined, we can use them to describe the filters. Instead of standard filter-major arrangement of weights in DSConv, we group the weights by channel and pattern. For a given Channel $c \in \{0, 1, ..., M-1\}$, we first collect 2-D kernels of Channel c across all filters, and group these kernels according to their patterns. Suppose P_c distinct patterns appear in Channel c, then we write $\sum_{i=0}^{c} P_i$ to W2.channel[c+1] to record this number. Then, each of these P_c patterns can be addressed by $p \in \{P_{c-1}, P_{c-1}+1, ..., P_{c-1}+P_c-1\}$. For n_k kernels sharing the same pattern p with pattern id p_id, we store their own filter ids and non-zero weights in *W2.weights[]*, indexed by *W2.woffset[p]*. We also record the p_id and n_k in *W2.patterns[p]*.

With this format, we can easily access kernels sharing the same pattern within a given channel, enabling us to leverage pattern regularity.

C. Single-layer CNN Inference with SCP

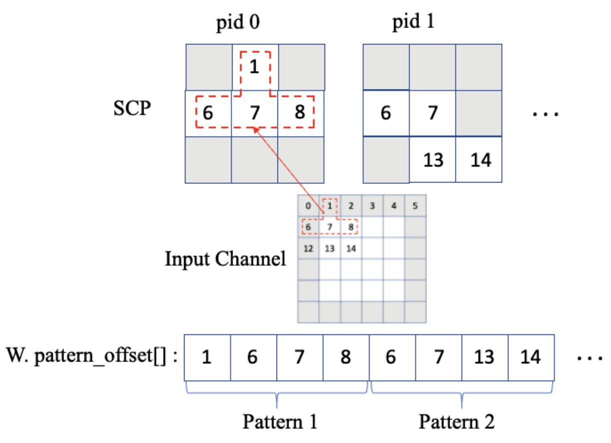
Algorithm 2 Direct Sparse Convolution with SCP

```
// in: input feature map
// W: compressed SCPs
// W2: compressed Channels
// out: output feature map
procedure SCP-DSCONV(in, W, W2, out)
    for c in [0,C] do //c-th input channel
        for p in [W2.channel[c], W2.channel[c+1]) do
            pid = W2.patterns[p] && 127
            count = W2.patterns[p] >> 7
            weights = W2.weights + W2.woffset[p]
            for i in [0, NNZ-1] do
                offset = W.pattern_offset[i+NNZ*pid]
                V = weights + count*(i+1), N=weights
                for j in [0, count-1] do
                    n=N[j], v=V[j]
                    for (h,w) in $[0, D_{F1}] \times [0, D_{F2}]$ do
                        iter ← $h \times (D_{F2} + padding) + w$
                        out[n][h][w] += v × in[offset + iter]
                    end for
                end for
            end for
        end for
    end for
end procedure
```

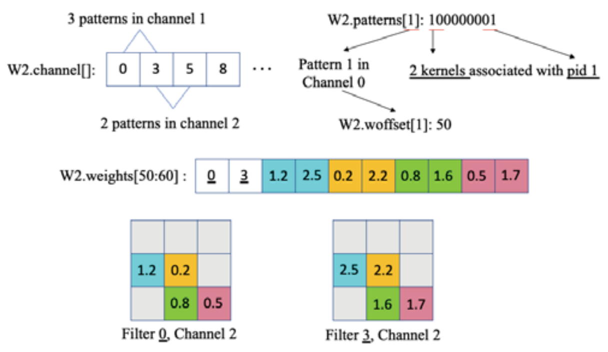
Once the input feature map is padded with zeros, we start to process it channel-by-channel. As listed from Line 7 of Algorithm 2, we iterate over all patterns used in a given channel. For each pattern, we extract its pattern id in W and the total number of kernels associate to it in Lines 8-10. Then a sequence of *windows to apply* is determined by the *W.pattern_offset[]*. As there can be several kernels sharing the same pattern, each window will likely have multiple non-zero weights to be applied on it. The average number of these weights can be estimated by

$$\frac{\#Filters \times Sparsity}{\#Patterns \times PatternSparsity}$$

As $\#Filters$ is usually several hundred, $\#Patterns$ is below 10, and $PatternSparsity$ is below 0.5, even under high sparsity, there can be quite a few weights associated with the same window. In this case, we can load the *window to apply* once (Line 12) and multiply it with all the weights(Line 13 to Line 20). Since this scalar-matrix multiplication is highly regular, it has good potential to be unrolled and vectorized.



(a) Structure of W



(b) Structure of W2

Fig. 1: Format of SCP-Based Filters

D. Fusion of Two Convolution Layers

In the original work of convolution-layer fusion [4], the authors propose a design using a sliding window on the input feature map. After processing the previous input, the window will slide for one stride (typically one pixel) to include new input. Clearly, there will be a large overlap between the neighboring windows. To avoid redundant calculations, a cache to hold results across neighboring windows has to be maintained. Similar to the sliding input window, the cache has to evict stale results to include new ones at every stride, adding extra overhead.

Fortunately, such overhead is unnecessary if we carefully design the fusion on top of the inference scheme in Section III-C. First, we introduce the idea of *scattering* DSConv, as described in our previous work [13]. Based on this idea, we modify Algorithm 2 and get the *scattering* version. Fusing it after a standard *gathering* Algorithm 2, the output/input windows on the intermediate results can both be aligned. This design resolves dependencies between adjacent input windows of the second convolution layer, so that the contributions of non-overlapping intermediate results can propagate individually to the output.

In the entire process, access to input is limited to one channel at a time, and the influence of an input area is full propagated after being processed once. This optimizes the cache efficiency for input. Furthermore, the size of intermediate results is also limited to $N_{th} \times N \times H_T \times W_T$, take an typical $N_{th} = 4, W_T = 4, H_T = 2$ in IoT settings, and a large $N = 512$ for CNN model, the intermediate result size on each core will be 8 kB, which is very manageable for IoT devices.

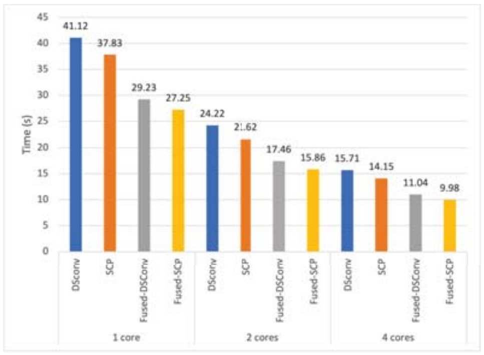
As a final remark, we note that this work has build on authors prior study [13] on applying convolution layer fusion and DSConv to CNN inference with random sparse patterns. Additionally, implementation details of tiling are mostly inherited, and therefore omitted in this work. However, this work explicitly exploits the regularity with SCP pruning. In next section, one of our baselines will the scheme from previous work, and we will demonstrate the benefits associated with the new scheme.

IV. Experimental Results

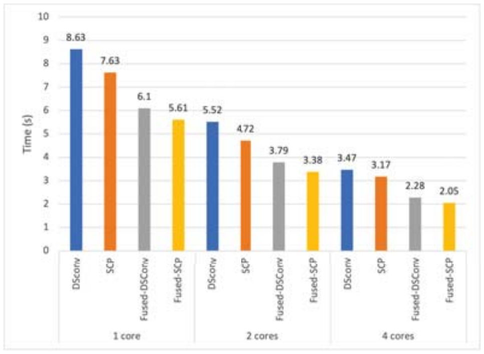
We run all experiments on a a Raspberry Pi (RPi) 3B. This single-board computer has a quad-core Broadcom BCM2837 processor clocked at 1.2GHz, supporting ARMv8-A 64-bit instruction set. It is also equipped with 32 128-bit SIMD registers on each core. The device has 1GB LPDDR2 RAM. We deploy Ubuntu 18.04 LTS Arm64 on this device. We believe that this device can represent pervasive and inexpensive edge devices at the time of writing this paper, given its retail price of around $30 and capability of serving a wide range of sensors and motors.

Due to the unavailability of pre-trained SCP models and the goal of creating inference scheme for general SCP-based CNN layers, we fix the following: (1) the target model sparsity and (2) the number of non-zeros in each 2-D kernel. Based on (2), we randomly generate a set of kernel patterns and populate the filters till we reach the target sparsity in (1). 5 models are generated for each setting and the final results reported are always the average across executions on them. The network structures tested include Vgg-19 [28], ResNet-34 [16], and a synthetic model. The size of 2-D kernels in each model is 3×3, and the input is $3 \times 224 \times 224$. The synthetic model has 4 convolution layers, each with 256 filters. This model is generated to study the performance of models with large filter size on each layer, as this factor can impact the number of repeating patterns in a channel. The input images are resized from 50 random images selected from ImageNet [10]. As we have elaborated in Section II, previous studies have proved that CNN models can still achieve desirable accuracy under high sparsity; therefore, we skip the examination of accuracy in our experiments to focus more on the validation of sparse inference under various sparsity.

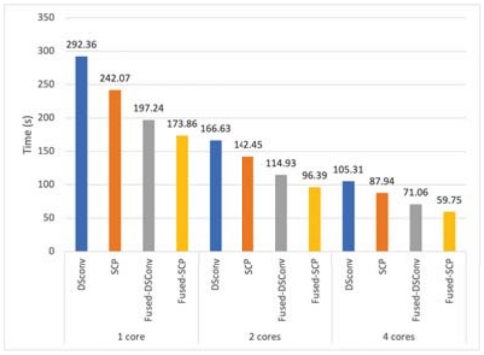
We benchmark the performance of four inference schemes : DSConv, Fused DSConv, SCP DSConv, and Fused SCP. In this way, we study the combinations of two independent optimizations: fusion and SCP on top of the direct sparse



(a) Vgg19



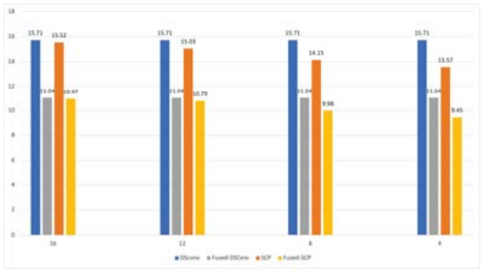
(b) ResNet-34



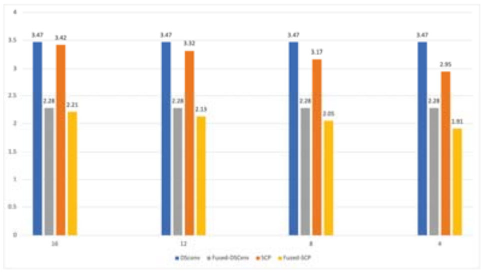
(c) Synthetic Model

Fig. 2: Scalability of Four Inference Scheme on RPi

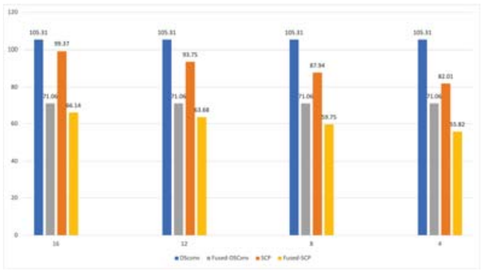
convolution DSConv, as shown in Table I. Specifically, we implement our own DSConv due to unavailability of DSConv in the Arm-based IoT environment. Fused DSConv is from our previous work [13]. SCP DSConv is as described in Algorithm 2. The last case is combined optimization of SCP and fusion, introduced in Section III-D. The same approach to parallelization and SIMD is used for all four implementations. In Vgg-19 (16 convolution layers) and the synthetic model(4 convolution layers), each odd numbered convolution layer is fused with its successive even numbered convolution layer; whereas in ResNet-34, we fuse each *basic block* of Conv-



(a) Vgg19



(b) ResNet-34



(c) Synthetic Model

Fig. 3: Performance Under Different Number of SCPs

BatchNorm-ReLu-Conv. The experiments we conducted are: (1) comparing the scalability of four schemes on RPi (2) varying the number of sparse patterns used and examining how the performance varies.

Sparse Pattern / Fusion	Default	Optimized
Default	DSConv	SCP DSConv
Optimized	Fused-DSConv	Fused-SCP

TABLE I: Optimizations Used in Different Schemes

First, we compare both the relative performance and scalability of four inference schemes on RPi using 1, 2, and 4 cores under high (95%) sparsity. The results are displayed in Figure 2. We fix the number of SCPs to 8 and the number of non-zeros in each kernel to 4, as suggested by original authors [25]. For Vgg-19 on 1 core, normalizing the efficiency of basic DSConv as 1, including optimizations for SCP can improve it to around $1.09\times$, and fusion to $1.41\times$. Applying two optimization altogether, we achieve a speedup of $1.51\times$.

This shows that fusion and SCP optimizations are mostly orthogonal and can work together to further improve the performance. The results for ResNet-34 and the synthetic data are either comparable (or even slightly better).

Next, we vary the number of sparse patterns in the models to see how the results are impacted. Specifically, we fix 95% sparsity and generate weights with $4, 8, 12$, and 16 sparse patterns for each model. We execute these models with all four inference schemes on RPi using 4 cores. The results are illustrated in Figure 3. It is obvious that the results for DSConv and Fused DSConv (blue and grey bars) are not influenced by the number of patterns, as they are not optimized for sparse patterns. On the other hand, the performance of SCP DSConv and Fused DSConv (orange and yellow bars) improve as the number of patterns decrease, since the number of kernels sharing the same pattern grows.

V. Conclusions

Pruning redundant weights in convolution layers has become a common practice to generate light-weight models. In this context, the state-of-the-art pattern based pruning makes a good trade-off between accuracy and implementation-efficiency. In this work, we proposed new fused inference scheme leveraging this pattern based approach. We optimize the inference using the combined approach of SCP and fusion, and provide an efficient implementation on multi-core edge devices. In the experiments, we demonstrate that applying SCP-based optimizations can improve the inference efficiency of DSConv on edge devices by up to 19%, and fusion can be used to further speedup the process. The Fused SCP inference also shows desirable parallelism on edge devices.

Acknowledgements: This work was partially supported by the following NSF grants: 1629392, 2007793, 2034850, 2131509, and 2018627.

References

[1] Developer guide :: Nvidia deep learning tensorrt documentation. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[2] Home | pytorch. https://pytorch.org/mobile/home/.

[3] Tensorflow lite | ml for mobile and edge devices. https://www.tensorflow.org/lite.

[4] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 22. IEEE Press, 2016.

[5] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 193–205. IEEE Press, 2019.

[6] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[7] X. Chen. Escoin: Efficient sparse convolutional neural network inference on gpus. *Matrix*, 4(5):7–8, 2019.

[8] E. J. Crowley, J. Turner, A. Storkey, and M. O'Boyle. A closer look at structured pruning for neural network compression. *arXiv preprint arXiv:1810.04622*, 2018.

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[9] X. Dai, H. Yin, and N. K. Jha. Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*, 68(10):1487–1497, 2019.

[11] Y. Dong, R. Ni, J. Li, Y. Chen, J. Zhu, and H. Su. Learning accurate low-bit deep neural networks with stochastic quantization. *arXiv preprint arXiv:1708.01001*, 2017.

[12] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. *arXiv preprint arXiv:2008.13006*, 2020.

[13] J. Guo, R. Teodorescu, and G. Agrawal. Fused dsconv: Optimizing sparse cnn inference for execution onedge devices (in press). In *2021 CCGRID Proceedings*. IEEE, 2021.

[14] Y. Guo, A. Yao, and Y. Chen. Dynamic network surgery for efficient dnns. *arXiv preprint arXiv:1608.04493*, 2016.

[15] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[18] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[19] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[20] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

[21] X. Ma, F.-M. Guo, W. Niu, X. Lin, J. Tang, K. Ma, B. Ren, and Y. Wang. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. *arXiv preprint arXiv:1909.05073*, 2019.

[22] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[23] C. Min, A. Wang, Y. Chen, W. Xu, and X. Chen. 2pfpce: Two-phase filter pruning based on conditional entropy. *arXiv preprint arXiv:1809.02220*, 2018.

[24] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç. High-performance sparse matrix-matrix products on intel knl and multicore architectures. In *Proceedings of the ICPP*, 2018.

[25] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 907–922, 2020.

[26] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*, 2016.

[27] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang. Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers. In *Proceedings of the ASPLOS*, 2019.

[28] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[29] B. Wen, S. Ravishankar, and Y. Bresler. Learning flipping and rotation invariant sparsifying transforms. In *Image Processing (ICIP), 2016 IEEE International Conference on*, pages 3857–3861. IEEE, 2016.

[30] C. Zhang, G. Yuan, W. Niu, J. Tian, S. Jin, D. Zhuang, Z. Jiang, Y. Wang, B. Ren, S. L. Song, et al. Clicktrain: Efficient and accurate end-to-end deep learning training via fine-grained architecture-preserving pruning. In *The 35th ACM International Conference on Supercomputing (ICS 2021)*, 2021.

[31] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian. Variational convolutional neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2780–2789, 2019.