2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-7281-9586-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/CCGrid51090.2021.00064

Fused DSConv: Optimizing Sparse CNN Inference for Execution 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*—Accelerating CNN on resource-constrained edge devices is becoming an increasingly important problem with the emergence of IoT and edge computing. This paper proposes an execution strategy and an implementation for efficient execution of CNNs. Our execution strategy combines two previously published, but not widely used, ideas – direct sparse convolution and fusion of two convolution layers. Together with a scheme for caching intermediate results, this results in a very efficient mechanism for speeding up inference after the model has been sparsified. We also demonstrate an efficient implementation that uses both multi-core and SIMD parallelism. Our experimental results demonstrate that our scheme significantly outperforms existing implementations on an edge device, while also scaling better in a server environment.**

I. Introduction

Deep Neural Networks (DNNs) have lately become the predominant component for smart applications on off-the-shelf mobile devices. Applications like smart assistants (e.g., Apple Siri and Google Assistant), powered by speech and NLP models [27], and auto portrait retouching camera, using image processing DNNs [29], are among the most popular modern computing applications. This work has been powered by a variety of optimizations that tune DNN inferencing to mobile devices [8], [25], [33].

With the trend towards Internet of Things (IoT), there is a growing interest in pushing this further to the *edge devices*. Unlike mobile phones, these devices are typically less expensive, have less computation and memory resources, and the battery life is an even bigger constraint. Specifically, unlike smart phones equipped with dedicated neural network accelerators, edge devices typically have only CPUs. Fortunately, with increasing transistor density, parallelism has become extremely common on even small edge devices. For example Raspberry Pi 3 and NXP's i.MX 8M Nano all have quad-core CPUs.

Given these trends, deep learning frameworks have started to support those platforms [5], [8]. However, many challenges remain – particularly, the most popular applications of Convolution Neural Networks, i.e., computer vision and speech recognition, are real-time tasks and thus very sensitive to inference latencies. The topic of designing light-weight convolutional neural network (CNN) models and optimizing their execution on resource-limited devices is beginning to receive attention [8], [24], [30].

Two general methods for improving DNN inference time have been *sparsification* and *memory-related restructuring*, specifically, *loop fusion*. Sparsification prunes the original dense model to remove many of the weights, creating a sparse model that requires less computations [22], [28], [32], [37], [39]. While the existing proposals differs considerably in the sparse patterns chosen, the basic goals remains trying to improve efficiency while maintaining accuracy. Loop fusion, on the other hand, focuses on utilizing memory (hierarchy) more effectively [3], [8], [9], [13]. By fusing two (or more) consecutive layers, one can eliminate unnecessary materialization of some of the intermediate results and reduce unnecessary scans of the data.

To date, the two directions listed above, i.e., sparsification and loop fusion, have worked in isolation, i.e., loop fusion has not been applied to sparse models. In this paper, our goal is to design an inference scheme that works with irregularly pruned (sparse) models to accommodate for strict hardware constraints of small (edge) devices. Our work includes combining *Direct Sparse Convolution* (DSConv) [35] with layer fusion, with implementation optimizations for multi-core and SIMD execution.

Specifically, the contributions of this work include the following. We show that it is possible to fuse the execution of multiple sparse convolution layers. We modified the execution flow of DSConv and designed a novel tiling based depth-wise execution of convolution layers. Such design avoids the replication of input in GeMM based CNN inference and complicated cache maintenance in previous approach of convolution layer fusion [9]. Next, for resource-limited IoT devices, we reorder the loop and lower the memory requirement for inference on large inputs as loading of the inputs and off-loading of output can both be tiled. This reordering also gives better memory/cache locality on small devices. Furthermore, we show how tiled execution can be correctly parallelized, and how the inference loops can benefit from SIMD parallelism.

Finally, we compare the performance of Fused DSConv with conventional GeMM-backed approach and previous work

978-1-7281-9586-5/21/$31.00 ©2021 IEEE
DOI 10.1109/CCGrid51090.2021.00064

on DSConv. The results show that Fused DSConv significantly outperforms GeMM approach in terms of efficiency and scalability. Particularly, on memory-limited devices where Caffe [26] fails to carry out the inference, Fused DSConv can still run and scale well. Fused DSConv also exhibits around 1.5× speedup compared with DSConv, and performs even better when there are stricter memory constraints. We also show that in a server environment, our implementation scales better than Caffe.

II. Background and Motivation

A. CNN Structure

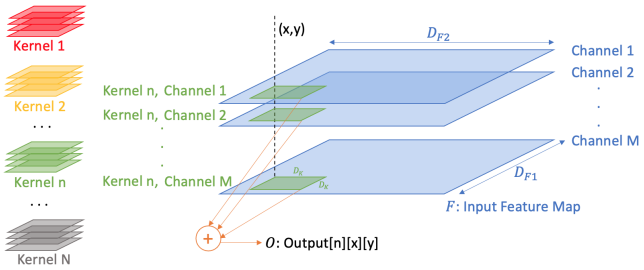


Fig. 1: Convolution Layers

A (2-D) CNN model typically comprises several different types of layers. The core block, a *convolution layer*, takes the input of a M-channel feature map $\mathbf{F} \in \mathbb{R}^{M \times D_{F1} \times D_{F2}}$, where D_{F1} and D_{F2} denote the dimensions of each 2-D channel ($D_{F1} = D_{F2}$ in most cases). The weights, i.e. N convolution kernels each with M channels of D_K by D_K filters are in the form $\mathbf{W} \in \mathbb{R}^{N \times M \times D_K \times D_K}$. In the inference phase, the output $\mathbf{O} \in \mathbb{R}^{N \times D_o \times D_o}$, where $D_o = D_F - D_K + 1$, is calculated by a convolution process:

$$O_{n,x,y} = \sum_{c,i,j} W_{n,c,i,j} \cdot F_{c,x+i-1,y+j-1} \tag{1}$$

This process is illustrated in Figure 1.

Apart from the convolution, other commonly used layers are as follows. *Normalization* layers will normalize the output in different directions (intra-channel, inter-channel, or across samples in a batch), depending on the algorithm used. *Activation* layers will transform the input through various activation functions. *Pooling* layer can extract max/minimum values within a given window of input. Finally, a *fully connected* layer will perform a matrix-vector multiplication on the flattened input.

The complexity of evaluating one convolution layer is $M \times D_F \times D_F \times N \times D_K \times D_K$. For a relatively simple CNN, VGG-16, it can total up to 1.55×10^{10} Floating Point Operations (FLOPs). The challenge is hardly limited to computing – the model itself can take up to hundreds of MBs in storage and the intermediate results also requires substantial memory. This can severely challenge the resources at inexpensive edge devices. Fortunately, it is possible to make the model more efficient in terms of spacial and computational complexity. We next introduce two categories of complexity reduction approaches closely related to our work.

B. Existing CNN Optimizations

Weight Pruning: Pruning was first proposed by Han *et al.* [22] and is based on the idea that there exists a large portion of small and redundant weights in convolution kernels. They employ a iterative threshold-based weight pruning and retraining scheme to reduce the connection density. This approach is termed as *non-structured pruning*, as it leaves irregular sparsity (0-Dimensional) in weight tensors. Later efforts [37], [39] employ ADMM regularization [12] to join the process of quantization and weight pruning and achieved high sparsity and accuracy. On the other hand, *structured pruning*, which tries to prune vectors (1-Dimensional), tiles/blocks (2-Dimensional), channels (2-Dimensional), or kernels (3-Dimensional), has been explored to improve the CPU/GPU performance, as regular sparsity allows for simpler software/hardware implementations [19], [28], [32]. A more recent work [18] leverages the sparsity by optimizing sparse matrix multiplication (SpMM) on GPUs.

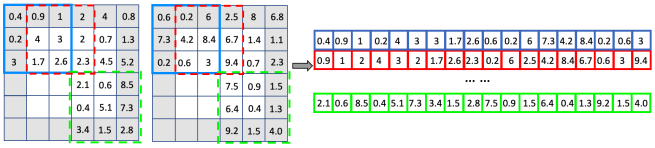


Fig. 2: Lowering (Pre-processing) for Convolution with GeMM

Next, Direct Sparse Convolution (DSConv) [35] is designed to replace the underlying General Matrix Multiplication (GeMM) libraries, which require that the input feature map be replicated several times. This replication, known as *lowering* or *im2col* operation (Figure 2), has demonstrated substantial overheads [4], [20], especially for kernels with high sparsity [35]. Even worse, the basic memory requirement for lowering operation will be several times the input size, and will also increase quadratically with the problem size [16], [40].

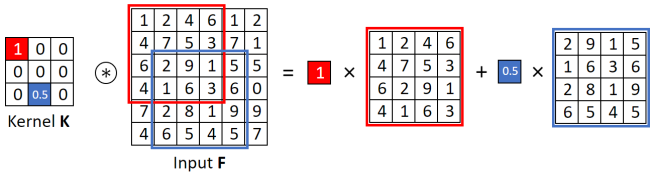


Fig. 3: Direct Sparse Convolution [36]

The DSConv approach, on the other hand, falls back to the conventional definition of the convolution process with a reordered loop. Specifically, the DSConv takes as input convolution kernels (weights) in 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 on according to Equation 1. We call this sub-matrix the *window to apply* for the corresponding non-zero weight, as illustrated in Figure 3. Here, each highlighted box in F is the *window to apply* for the non-zero weight in same color in K. This DSConv approach is later used in several works on optimizations for CNNs, e.g. TIRAMISU [10] compiler and Escoin [14].

Other related works also include special architecture designed for sparse CNN [34], [41] and GPU optimization for sparse CNN [14]. More recent work [31] leverages Sparse Convolution Patterns (SCP) for filter level pruning and incorporates channel-level (structured) pruning.

Layer Fusion: Layer fusion is widely used in frameworks as Tiramisu [10], TensorRT [2], and SkimCaffe [3] to optimize the performance of evaluation. The idea is to combine the evaluation of several adjacent layers. A common implementation of layer fusion is to pixel-wise fuse 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.

A special (but less common) case is to fuse multiple convolution layers, as proposed by Alwani *et al.* [9]. This technique employs tiling and reordering to help feed partial results from one layer directly to its downstream layer to generate partial output, forming a *depth-first* pattern of execution.

C. Motivation

To further motivate this work, we show the computational and memory requirements of popular DNN models on an edge device. We use Caffe for this motivating study with results reported in Table I (Raspberry Pi 3B, with the input of $3 \times 227 \times 227$ integers). Caffe [26] (and other frameworks like cuDNN [15]), uses traditional General Matrix Multiply (GeMM), requiring replication of dense input matrices. Table I shows resources requirements for popular models, and specifically that CaffeNet, GoogleNet, RCNN, and Vgg can take up to 690-1160 MB of runtime memory. This can easily constraint even the high-end edge devices. While memory availability of edge devices can increase in the future, we can also expect models and inputs of even larger size, and even the possibility that multiple models may process the same input simultaneously. In addition, the $16-49$ second inference time obviously cannot meet the needs of real-time tasks.

TABLE I: Memory Consumption and Latency for Running Caffe On Raspberry Pi

Model Name	Model Size (MB)	Peak Mem (MB)	Latency (s)
CaffeNet	233	843	16
GoogLeNet	51	806	30.12
Rcnn	220	690	16
Vgg_cnn_s	392.6	1162	48.78

Although DSConv algorithm [35] shows a promising direction of fast inference on small devices, its implementations today [14], [35] are mostly on Intel architectures or GPUs. Because of its low memory requirements, it does seem suitable for edge device. For the layer-fusion process, current implementations, e.g., Tiramisu, TensorRT [2], and SkimCaffe [3], have pixel-wise fusion between a convolution layer and its following bias, normalization, and activation layers. However, they have not yet employed fusion between convolution layers, mainly due to the incompatibility of fusion with the underlying GeMM libraries and overhead of additional control flow. Such layer-fusion technique can potentially help to reduce the size of intermediate results and improve cache/memory locality.

Thus, to summarize, our goal is to create an efficient inference schemes for sparse models on edge devices, where we combine the largely unused techniques of DSConv and layer fusion between (sparse) convolution layers, together with a careful design that can support parallel execution.

III. Fused SConv Inference Scheme Design

In this section, we present the design of our CNN inference scheme, Fused DSConv.

A. Preliminaries

Building on top of the existing ideas like Direct Sparse Convolution (DSConv) and loop fusion, we aim to design a novel scheme with the following goals:

- Optimizing for devices with limited memory, and in particular, limiting the size of intermediate results to also improve the cache performance.
- To utilize the power of multi-core devices.
- Keep memory accesses and control flow sufficiently simple to facilitate optimization with SIMD operations.

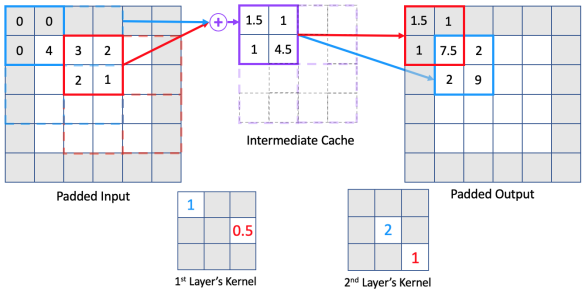


Fig. 4: Fused Direct Sparse Convolution with Simple 1-Channel Input and Output

The general idea of this process is shown in Figure 4. Our design fuses the evaluation of two convolution layers together, since: 1) it is very common to see two successive convolution layers in popular network architectures such as ResNet and VGGNet; and 2) fusing more than three layers requires extensive control code, which can result in overheads and an impediment to SIMD parallization.

The salient aspects of our scheme include the following:

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] × [0, D_F2] do
16:                 iter ← h × (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
```

- A tile-based execution with less implementation overheads, compared with the previous work on fusion of convolution layers [9] that employed a sliding window based execution,
- Avoiding the overheads of retention/re-calculation of intermediate results while combining loop fusion and direct convolution,
- A design that can be efficient both when the CNN is compute-bound and I/O bounded (the latter when the input size and number of weights is large).

These features of our design are achieved by two key innovations. First, we leverage a tiled version of direct sparse convolution on the 1st convolution layer of the fused execution. Second, we introduce an *inverse* version of sparse convolution to evaluate the 2nd convolution layer. By doing this, we can fully propagate influence of an input tile to the output map. The result is that sliding window of input or retention of intermediate results are not needed.

B. Algorithmic Details

Before describing our overall scheme that incorporates the above ideas, we first review the original direct sparse convolution algorithm and then describe our approach. The implementation of DSConv, as shown in Algorithm 1, takes the input of a dense feature map f_map and compressed sparse weights W in a kernel-major CSR format. As a background, CSR is comprised of three arrays, *W.kernel_ptr[]* storing the number of non-zero values in each kernel, *W.kernel_offset[]* storing the offsets of each non-zero values in the kernel, and the *W.value[]*, storing the actual non-zero weights. In order to prepare the weights for the DSConv procedure, a transformation (lines 1-8), is done to map each non-zero value's offset in the array *W.kernel_offset[]* to the initial offset of its *window to apply* in the input feature map. The intuition of this step can be illustrated through Figure 3, where the *kernel_offset* ($1 + 2 \times D_k = 7$) of the weight marked by blue, with value 0.5 in the kernel, will be transformed to the offset of the first pixel ($1 + 2 \times D_{F2} = 13$) in its corresponding *window to apply* (blue box) in the input feature map. This step pre-calculates the initial *offset* used in each step of the direct sparse convolution, as shown in line 17.

From line 10 starts the process of the DSConv. It iterates over the N kernels on the outer loop and each kernel's non-zero weights in the inner loop. There is no multiplication-by-zero involved and the process is highly regular, meaning it can be tiled and optimized with multi-threading and SIMD instructions.

Now, our novel algorithm, Fused DSConv, is shown as Algorithm 2. Before the 1st convolution, the borders of row-major input feature map are padded with zeros, as in convention evaluation process. Then, we perform a tiled version of DSConv on the first convolution layer (lines 12-20). The idea is that for each non-zero weight, its *window to apply* will be partitioned into tiles $t_{00}, t_{01}, \ldots, t_{PQ}$, each with size $W_T \times H_T$. As illustrated in Figure 4, in the padded input, *window to apply* of the red/blue pixel is divided into four tiles respectively, where the solid 2×2 tile corresponds to t_{00}. In each step u, v, we multiply all non-zero weights with their corresponding $t_{u,v}$, whose results will be accumulated to an intermediate tile. Upon the generation of this intermediate tile, we perform convolution on it again with weights of the second layer to produce (partial) output. This immediate processing of intermediate cache can ensure better locality. However, here we cannot simply apply DSConv again on the cache without inter-tile retention/recalculation, due to the differences in the offset for each non-zero weight's *window to apply*, and because the intermediate result is partial.

To resolve this, we introduce the *inverse* sparse convolution to align the input window for different non-zero weights, as shown in the second convolution process of Figure 4. This relies on the property that convolution is a linear system, i.e., given an input pixel (value and coordinate) and the kernels, one can fully determine this pixel's contribution to the output feature map, and contributions from different input pixels can be simply added up for complete output. Using this altered convolution process, we are able to easily calculate and accumulate the full contribution of a given tile, as shown in lines 26-30. More convenient is that the offset shifting (in line 24) will be applied to the output (line 28), rather than input (line 17), implying that different non-zero weight will actually be multiplied to the same *window to apply*. This property significantly enhances the locality of intermediate cache and regularity of accesses. To make this happen, the offsets need to be pre-calculated differently from the DSConv. As we can see from padded output of Figure 4, the offset, if represented in (row offset, col offset), is (center row - row, center col - col) = $\{(1, 1)$ for blue, $(0, 0)$ for red$\}$ in $inverse$ sparse convolution,

as opposed to (row, col) = $\{(1,1)$ for blue, $(2,2)$ for red$\}$ if it were in DSConv.

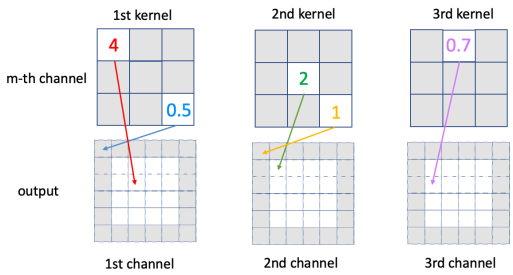
Such *only-once* processing makes it much simpler to maintain the cache. Specifically, it can be de-allocated or cleared all at once, and there is no need for more complicated retention policy as in the original layer fusion work [9]. This also decouples the processing of different tiles, allowing for easier multi-threading optimizations. Additionally, the layers that may appear in-between two convolution layers can be added to the fused convolution process - including activation layers and normalization layers.

C. Other Optimizations

To further optimize the process of Fused DSConv and address the third challenge, we can perform *inverse* sparse convolution by input channel. To achieve this, we reformat the original CSR into *W.channel[]*, *W.offsets[]*, and *W.value[]*, to reflect the non-zero values by channel, as shown in Figure 5. In this new format, *W.channel[m]* stores channel m's beginning position in *W.offsets[]* and *W.value[]*. Within a channel, *W.offset[j]* stores the first offset O_j of the output region of $j-th$ non-zero value, across different kernels. The above format can be employed when the original model is stored, or generated by a one-time procedure during the deployment.

Based on this reformatted input, we can iterate over the M channels of weight tensor on the outer loop instead of N kernels. Within a given channel, we go through all non-zero weights across N kernels, those weights will be multiplied with the same input window/tile, and the results will be accumulated onto corresponding output regions specified by *W.offsets[]*. In this process, reads to the same tile of intermediate results are packed together, as will greatly improve the locality of access and the potential for further optimization with SIMD instructions.

In Algorithm 2, we only apply this channel-major evaluation to the 2nd convolution layer, while the 1st convolution layer remains kernel-major. This allows us to iterate over channels



W.channel: channel[m]=j, channel[m+1]= j+5

	j	j+1	j+2	j+3	j+4
W.offset:	14	0	36	43	85
W.value:	4	0.5	2	1	0.7

Fig. 5: Channel-Major CSR Format

Algorithm 2 Fused Direct Sparse Convolution

```
1:  // in: input feature map
2:  // W: compressed weights for the 1st conv layer
3:  // W2: compressed weights for the 2nd conv layer
4:  // out: output feature map
5:  procedure FUSED-DSCONV(in, W, W2, out)
6:      for u in [0, P] do // tile row id
7:          for v in [0, Q] do // tile col id
8:              offset_tile = f(u,v)
9:              for n in [0,N] do //n-th intermediate channel
10:                 // 1st conv layer
11:                 cache.clear()
12:                 for j in [W.kernel[n], W.kernel[n+1]) do
13:                     offset ← W.offset[j]
14:                     val ← W.value[j]
15:                     for (h,w) in [0, H_T] × [0, W_T] do
16:                         iter ← h × (D_F2 + padding) + w
17:                         cache[h][w] +=
                              val × in[offset + offset_tile + iter]
18:                         //offset is applied on the input
19:                     end for
20:                 end for
21:                 // 2nd conv. layer
22:                 // instant propagation of intermediate results
23:                 for j in [W2.channel[n], W2.channel[n+1]) do
24:                     offset ← W.offset[j]
25:                     val ← W.value[j]
26:                     for (h,w) in [0, H_T] × [0, W_T] do
27:                         iter ← h × D_T + w
28:                         output[offset + offset_tile + iter] +=
                              val × cache[iter]
29:                         //offset is applied on output
30:                     end for
31:                 end for
32:             end for
33:         end for
34:     end for
35: end procedure
```

of intermediate cache, meaning that it is sufficient to only keep one channel ($W_T \times H_T$) of intermediate results in cache. Note that in lines 12 - 20, we first generate the result for the n-th kernel in the 1st convolution, corresponding to the n-th channel of intermediate results. This channel is instantly fed to the second convolution, lines 21 - 31, and the partial results are accumulated to the output layer. After this the cache can be cleared as the contributions of this channel is fully propagated to the output layer.

IV. Mapping to Multi-Core and SIMD Parallelism

In this section, we present the implementation details of our inference scheme, specifically, the implementation of tile-based fusion is described with an emphasis on how the fused execution process is parallelized on multi-core structures. Later, the optimization with SIMD instructions is discussed.

A. Execution and Parallelism

In the execution, we iterate through all the tiles in a row major fashion, i.e. $t_{0,0}, t_{0,1}, \ldots, t_{0,P-1}, t_{1,0}, t_{1,1} \ldots, t_{P-1,Q-1}$.

Within each tile, the DSconv, Algorithm 2 lines 12-20, is carried out by kernel, while the *inverse* sparse convolution, lines 26-31, is done by channel.

This will always pack access to the intermediate results by the channel (continuous $H_T \times W_T$). As a side effect, the access to input/output will be cross-channel; however, with row-major tiling, the input/output tied to the active tiles will be small enough to reside in cache. For example, when we are processing tiles $t_{0,0}, t_{0,1}, \ldots, t_{0,P-1}$ in the 1st convolution layer, access to input feature map will only reach first $H_T + D_K - 1$ rows of every channel. Similar rules apply for the output feature map in the second convolution layer. At any moment, the active memory region for evaluation will be $(H_T + D_{K,1}) \times D_{F2,1} \times M_{in}$ for input (1 in subscript means the first layer), $H_T \times W_T \times n_{threads}$ for intermediate cache, $(H_T + D_{K,2}) \times D_{F2,2} \times M_{out}$ for output, plus the compressed sparse weights. Furthermore, first H_T rows of input/output will not be accessed again during the processing of later tiles. Such loading process can greatly reduce the memory footprint compared with previous DSConv implementation, which accesses the full input map in every loop. To conclude, the above execution can both improve the cache locality and performance under lower memory, comparing to either the GeMM based execution or original DSConv approach.

To make full use of the multi-core architecture, we parallelize the above process using the OpenMP library. We start by allocating a cache of size $H_T \times W_T$ for each thread, denoted by $C_0, C_1, \ldots, C_{n\_thread-1}$. Then, the first $H_T + D_K^1 - 1$ rows of all input channels are loaded. We assign $t_{0,0}, t_{0,1}, \ldots, t_{0,P-1}$ to different threads. For a given tile $t_{0,j}$, a worker thread carries out line 12-31. Since different non-zero weights in the 2nd convolution have their own output region, as shown in Figure 6, there exists a potential race condition where writing the output of tiles on different cores will be in conflict. We resolve this by: 1) ensuring tile width W_T is larger than the kernel size D_K, so that conflict only happens between adjacent tiles, and 2) equally partitioning the tiles to cores and adding conditional variables to make sure no adjacent windows will be concurrently processed. This solution is efficient since D_K in most practical models is relatively small. Thus, with different cores are working at approximately the same speed, threads will rarely wait on conditional variables.

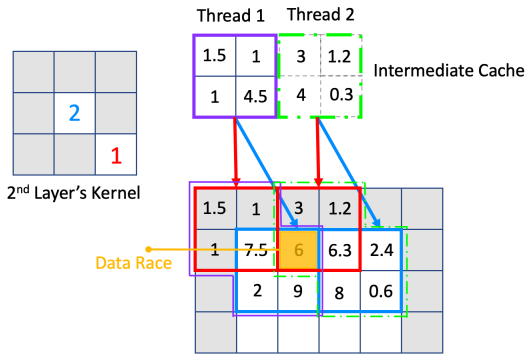


Fig. 6: Data Race Between two Adjacent Tiles

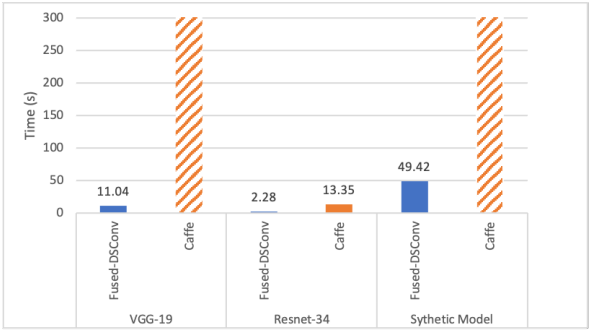


Fig. 7: Comparing Fused DSConv with Caffe (GeMM based Inference)

B. Optimizations with SIMD

Today, even the small and inexpensive IoT devices are equipped with SIMD lanes. For example, Arm Cortex-A architecture has a well-developed Neon instruction set, and Helium, the lightweight vector extension for Arm Cortex-M micro-controller architecture is constantly evolving for DSP and Neural Network (NN) usages. Therefore, we utilize SIMD instructions and further optimize the execution. Specifically, we choose the 64 bit ARMv8-A as an example. This micro-architecture requires 32 separate SIMD registers, each 128-bit wide per core [1]. Therefore, we make our tile width a multiple of 4 (assuming float32 used in the model). On top of this, we are able to apply scalar-vector multiply-accumulate (MAC) instructions. More importantly, using different loop ordering in the two fused convolution layers allows us to maintain a very small set of intermediate results ($H_T \times W_T$) for each thread, which can reside in the per-core SIMD registers. Consequently, one operand of all multiplication instructions – the accumulator for the 1st layer and the input vector for the 2nd layer, will always be in the registers. This greatly improves the efficiency of processing.

V. Experimental Results

In this section, we demonstrate that Fused DSConv is efficient for executing sparse models on edge devices by comparison with a previous DSConv implementation and also a standard deep learning library Caffe). Furthermore, we also show that Fused DSConv also shows good scalability on a multi-core server and better performance under limited memory.

A. Experiment Setup

Our experiments are conducted using two different hardware platforms. The first is a Raspberry Pi (RPi) 3B, equipped

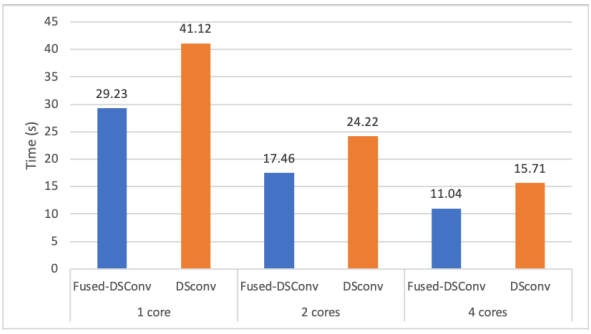


Fig. 8: Comparing Fused DSConv with DSConv (Sparsified Vgg-19)

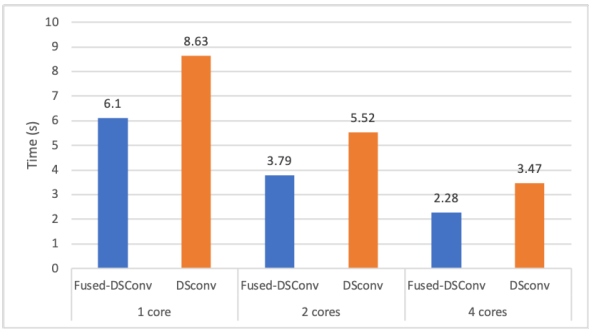


Fig. 9: Comparing Fused DSConv with DSConv (Sparsified ResNet-34)

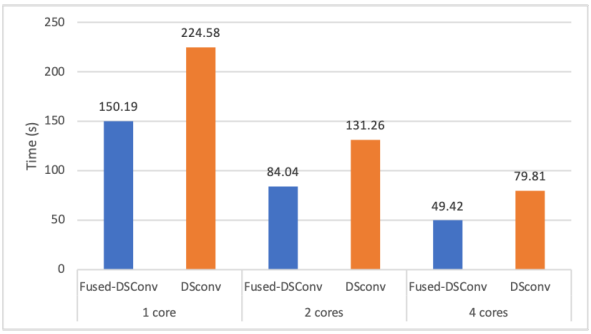


Fig. 10: Comparing Fused DSConv with DSConv (Synthetic Sparse Model)

with a quad-core Cortex A53 processor clocked at 1.2GHz, implementing ARMv8-A 64-bit instruction set, and execution Ubuntu 18.04 LTS Arm64. It has SIMD engine on all cores, each with 32×128 bit registers. The device also has 1GB LPDDR2 SRAM. This device costs roughly $30 at the time of writing this paper, and thus is a good example of an inexpensive IoT device in the edge computing setup. The second is a 16 core multi-core machine with 2.9Ghz Intel Xeon E5-2666 v3 processor and over 60 GB memory, execution Ubuntu 18.04 x86_64, that we accessed on the Amazon Elastic Compute Cloud (Amazon EC2). This platform is used for scalability study beyond four cores.

One of the challenges for our work was the unavailability of sparse models as benchmarks. However, since our emphasis was on performance and not accuracy after sparsification, we created sparse models by applying random pruning on two popular models that use many convolution layers. Thus, the models we are using include sparsified versions of VGG-19 [38], ResNet-34 [23]. In addition, we also used a synthetic model. Specifically, VGG-19 has 19 weight layers, of them 16 are convolution layers. All convolution layers have 3×3 filters, and the number of kernels ranges from 64 to 512. The parameters total up to $\sim 143M$, and the FLOPs $\sim 20G$. ResNet-34 also uses 3×3 filters, but with significantly less parameters ($\sim 21M$) and FLOPs ($\sim 4G$). These two models take $3 \times 224 \times 224$ input, and reports comparable accuracy [11]. To simulate irregular pruning, we take the dense models and randomly prune out connections until a certain level of sparsity is reached. The input set for these two models are 50 random images selected from ImageNet [17], each resized to $3 \times 224 \times 224$. The inference batch size is set to one.

The synthetic model was designed to keep the layers simple, but increase the input size (which is the upcoming trend). Specifically, the input is $3 \times 1000 \times 1000$. The model itself consists of 4 convolution-3 layers with ReLU activation, with $\{64, 64, 32, 32\}$ kernels for each layer. For our experiments, the images are up-scaled from the same input set as the previous models. The sparsified models are generated by randomly pruning out weights in the CNN kernels, forming models with 70% - 97.5% sparsity, i.e. 30% - 2.5% non-zero weights. Previous work [7], [21], [22] as shown that even at this sparsity levels, models tend to have moderate or no accuracy loss. In our experiments, 5 randomly pruned versions for each model are generated and the final reported results are the average over these versions.

The inference schemes in this benchmark include Fused DSConv, DSConv, and Caffe. The previous implementation of DSConv is part of SkimCaffe [3], which depends on Intel tool-chain and libraries; therefore, we use our own version of DSConv by disabling fusion in Fused DSConv. Thus, the only difference in implementation between Fused DSConv and DSConv is fusion of convolution layers whereas the same approach to parallelization and SIMD is used for both 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-BatchNorm-ReLu-Conv. We build BVLC Caffe v1.0 from source on both platforms, enabling support for OpenBLAS and OpenMP.

The experiments we conducted on RPi are: (1) comparing the performance of Fused DSConv with DSConv using high (95%) sparsity on 1, 2, and 4 cores; (2) comparing the

performance of Fused DSConv with Caffe using high-sparsity model, on 4 cores; (3) showing the trend of performance of DSConv and Fused DSConv under different sparsity levels on 4 cores; (4) observing the performance of DSConv and Fused DSConv when there is memory contention. The experiment on 16-core platform was conducted to compare the scalability of Fused DSConv and Caffe beyond four cores.

B. Experimental Results

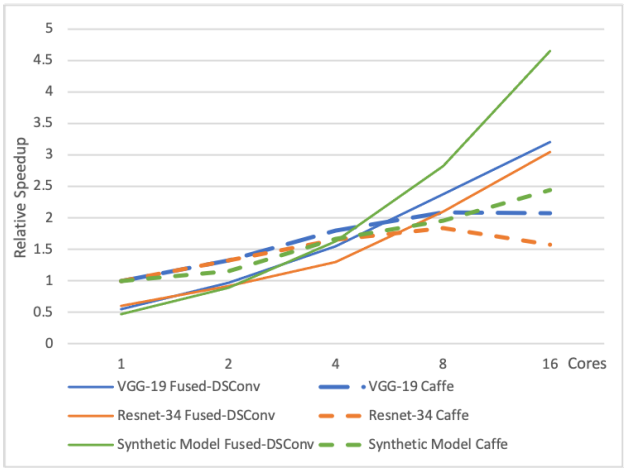


Fig. 11: Scalability on a Multi-core Server. For each of the models, Caffe version on 1 core is used as baseline for reporting performance.

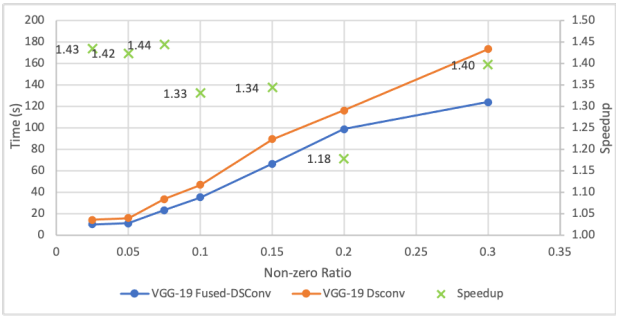


Fig. 12: Execution Time under Different Sparsity Levels, VGG-19

First, we benchmark the performance of Fused DSConv against Caffe [26], using models with 95% sparsity. In this test, Caffe uses Generalized Matrix Multiplication (GeMM) for sparse convolutions. Both implementations utilize all four cores of RPi. As shown in Figure 7, for ResNet-34, Fused DSConv shows a speedup of $5.85\times$. For VGG-19, Caffe will take around 6000 seconds to finish due to the memory limitation, while Fused DSConv can take advantage of the high weight sparsity and finish the inference in ~ 10 seconds. For the synthetic model, the GeMM-based evaluation cannot

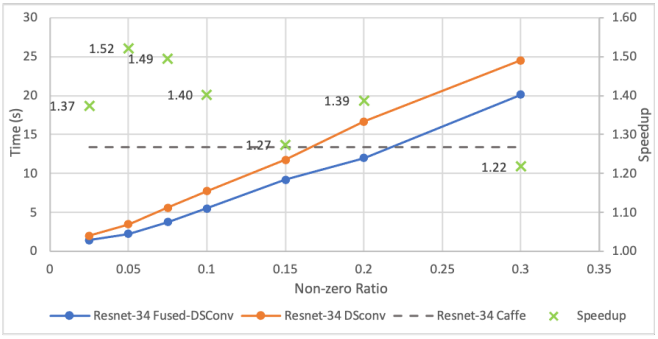


Fig. 13: Execution Time under Different Sparsity Levels, ResNet-34

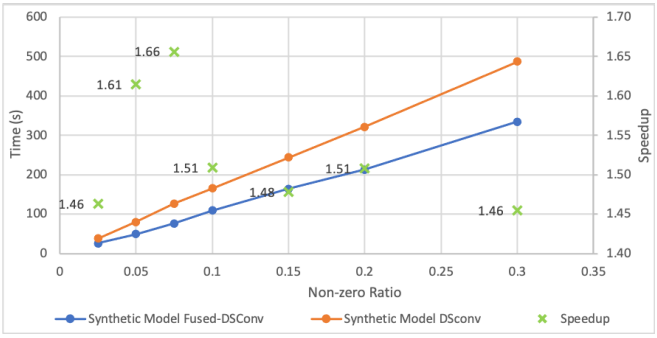


Fig. 14: Execution Time under Different Sparsity Levels, Synthetic Model

allocate the memory needed ($\sim 2GB$) to perform the lower operation *im2col*, while Fused Conv finishes in ~ 80 seconds. It is obvious that compared with Caffe's GeMM (OpenBLAS), Fused DSConv has significant performance gain while lowering the memory requirements, benefiting from sparsity and compression of models.

Next, we evaluate the performance of Fused DSConv and DSConv on RPi using same 95% sparsity models. We also studied the scalability by using 1, 2, and 4 cores of RPi. As illustrated in Figures 8, 9, and 10, it can be observed that Fused DSConv shows $1.34 - 1.61\times$ performance boost compared with DSConv on RPi across all settings. Specifically, on four cores, the speedup is $\{1.42\times, 1.52\times, 1.61\times\}$ for VGG-19, ResNet-34, and synthetic model, respectively. As we have discussed in Section III and Section IV, this performance gain mainly comes from the improved locality and not outputting the intermediate results between two fused layers. Since the size of both input and intermediate results are significantly larger (1000×1000 vs 224×224) for the synthetic model, it benefits more from the efficient input access pattern and intermediate results handling.

Next, focusing on scalability, when the number of cores increase from 1 to 4 on RPi, Fused DSConv demonstrates comparable speedup to DSConv, with $\{2.65\times, 2.68\times, 3.03\times\}$

for Fused DSConv, and $\{2.61\times, 2.49\times, 2.81\times\}$ for DSConv. These results show that loop fusion is not limiting parallelism but aiding it, because of our careful design and implementation.

To further demonstrate the scalability of Fused DSConv beyond four cores, we switch to the server platform with 16 cores, where we can study the performance of Caffe when memory is not an issue. Figure 11 shows these results, with dashed lines denoting Caffe, solid lines denoting Fused, and different models in different colors. For clarity, we normalize the execution time of Caffe on one core as one unit for each model, and other results are reported as relative speedups. On one core, we find that Fused DSConv takes $67\% - 111\%$ more time to finish. This is because Fused DSConv is not optimized for Intel processors, whereas OpenBLAS-backed Caffe is able to better utilize the SIMD instructions. Nevertheless, Fused DSConv generally shows better scalability, especially from 4 - 16 cores; therefore, when all cores are used, Fused DSConv will achieve a speedup of $\{1.54\times, 1.94\times, 1.90\times\}$. This shows that our parallelization is very effective.

Additionally, to study how the performance is affected by sparsity, we evaluate the execution time on RPi of three models using DSConv and Fused DSConv with different degrees of irregular sparsity. Figures 12, 13, and 14 report that under different sparsity, Fused DSConv is consistently faster than DSConv, with speedup of $\{1.17-1.44\times, 1.21-1.49\times, 1.45-1.65\times\}$ for VGG-19, ResNet50, and synthetic model, respectively. As the sparsity increases, generally we can observe higher advantage of Fused over DSConv for each model. The main reason behind this is the channel-major execution of the 2nd fused layer that packs the sparse access to the same intermediate channel together. Putting Caffe into the picture, the results show that above the sparsity of $\sim 80\%$ for ResNet-34 Fused DSConv exhibits better performance on RPi.

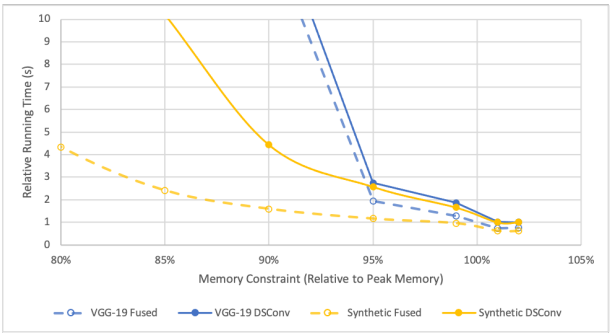


Fig. 15: Execution Time under Different Memory Constraints

Finally, we compare the performance of Fused DSConv and DSConv under limited memory. This corresponds to the real-world scenario where strict memory constraint is imposed by device capacity and/or the need for executing multiple models simultaneously. In this experiment, we execute the 95%-sparsity version of VGG-19 and synthetic models, since they have comparable peak memory consumption ($\sim 500MB$). However, the synthetic model takes a much larger input (16x) as compared to VGG-19, which is a trend we can expect to see more often in the future. The experiment is carried out on RPi (4 cores) while varying memory bounds. This is facilitated using the *control groups* tools [6] (*libcgroup* package on Ubuntu). limiting the physical memory usage while allowing for sufficient swap size.

The results are reported in Figure 15, with dashed lines denoting DSConv, solid lines denoting Fused DSConv, and different models in different colors. We normalize the memory boundary by the peak memory consumption, and the execution time by DSConv's execution time under unrestricted memory. Comparing the results of 99% vs 101% memory bound, it's easy to see that the performance will start to degrade once we impose a weak memory restriction, by a factor of $0.5 - 0.6\times$. As the restriction comes to 95% of peak memory, there is an obvious trend that Fused DSConv suffers less from such constraint, with a relative speedup of $1.41\times, 2.21\times$ for VGG-19 and synthetic model, respectively, over DSConv. From 90% onwards, the performance gap of DSConv v.s. Fused grows steadily for synthetic model, with Fused version having an advantage of $4.2\times$ at 85% memory. As we have discussed in Section III, by accessing in/out feature map a few rows at a time, Fused DSConv has smaller memory footprint, and will translate to better performance under limited memory. VGG-19, on the other hand, sees a big leap in execution time at 90% for both DSConv and Fused DSConv. This is mainly due to that for VGG-19, most memory consumption is to store the model weights, the weights of dense fully connected layer in particularly. Both DSConv and Fused DSConv is not optimized for such dense layers; therefore, a comparable performance loss is to be expected.

VI. CONCLUSIONS

Deploying sparse CNNs on IoT devices has become a promising area with the rise of smart and pervasive edge applications. However, efficient inference of sparse CNN models remains a significant challenge in this context as one must accommodate for the limited cache/memory, low CPU speed, and lack of accelerators. In this work, we present a novel inference design that enhances the locality of evaluation by fusing two convolution layers and leverages the irregular sparsity using a non-GeMM based convolution procedure. We further tune our implementation using multi-core parallelism and by exploiting SIMD features. We show an average of 1.5x speedup from fusion alone (Fused DSConv over DSConv, with later still having the same parallelization and other optimizations), while outperforming previous GeMM based implementations by at least 6x. We also show better relative speedups on a server machine.

Acknowledgements. This research was partially supported by NSF awards CCF-1629392; CCF-2007793 and OAC-2034850.

REFERENCES

[1] Armv8-a aarch64 isa overview. https://developer.arm.com/-/media/Files/pdf/graphics-and-multimedia/ARMv8_InstructionSetOverview.pdf. (Accessed on 12/04/2020).

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

[3] Github - intellabs/skimcaffe: Caffe for sparse convolutional neural network. https://github.com/IntelLabs/SkimCaffe.

[4] Github - soumith/convnet-benchmarks: Easy benchmarking of all publicly accessible implementations of convnets. https://github.com/soumith/convnet-benchmarks.

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

[6] libcgroup/libcgroup. https://github.com/libcgroup/libcgroup.

[7] Reducing deep learning model size without effecting it's original performance and accuracy with tensorflow model optimization toolkit on real world dataset | by janibasha shaik | analytics vidhya | oct, 2020 | medium. https://medium.com/analytics-vidhya/reducing-deep-learning-model-size-without-effecting-its-original-performance-and-accuracy-with-a809b49cf519.

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

[9] 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.

[10] 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.

[11] S. Bianco, R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[12] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

[13] 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.

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

[15] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[16] M. Cho and D. Brand. Mec: memory-efficient convolution for deep neural network. *arXiv preprint arXiv:1706.06873*, 2017.

[17] 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.

[18] T. Gale, M. Zaharia, C. Young, and E. Elsen. Sparse gpu kernels for deep learning. *arXiv preprint arXiv:2006.10901*, 2020.

[19] 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.

[20] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré. Caffe con troll: Shallow ideas to speed up deep learning. In *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, pages 1–4, 2015.

[21] 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.

[22] 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.

[23] 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.

[24] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[25] L. N. Huynh, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95, 2017.

[26] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[27] V. Kepuska and G. Bohouta. Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home). In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 99–103. IEEE, 2018.

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

[29] J. Li, C. Xiong, L. Liu, X. Shu, and S. Yan. Deep face beautification. In *Proceedings of the 23rd ACM international conference on Multimedia*, pages 793–794, 2015.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. IEEE, 2017.

[35] 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.

[36] J. Park, S. R. Li, W. Wen, H. Li, Y. Chen, and P. Dubey. Holistic sparsecnn: Forging the trident of accuracy, speed, and size. *arXiv preprint arXiv:1608.01409*, 1(2), 2016.

[37] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 925–938. ACM, 2019.

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

[39] S. Ye, X. Feng, T. Zhang, X. Ma, S. Lin, Z. Li, K. Xu, W. Wen, S. Liu, J. Tang, et al. Progressive dnn compression: A key to achieve ultra-high weight pruning and quantization rates using admm. *arXiv preprint arXiv:1903.09769*, 2019.

[40] J. Zhang, F. Franchetti, and T. M. Low. High performance zero-memory overhead direct convolutions. *arXiv preprint arXiv:1809.10170*, 2018.

[41] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 20. IEEE Press, 2016.