This CVPR Workshop paper is the Open Access version, provided by the Computer Vision Foundation. Except for this watermark, it is identical to the accepted version; the final published version of the proceedings is available on IEEE Xplore.

# DSC: Dense-Sparse Convolution for Vectorized Inference of Convolutional Neural Networks

Alexander Frickenstein BMW Group Autonomous Driving alexander.frickenstein@bmw.de Manoj Rohit Vemparala BMW Group Autonomous Driving manoj-rohit.vemparala@bmw.de

Christian Unger BMW Group Autonomous Driving christian.unger@bmw.de

Fatih AyarWalter StecheleTechnical University MunichTechnical University MunichElectrical and Computer EngineeringElectrical and Computer Engineeringfatih.ayar@tum.dewalter.stechele@tum.de

Abstract

The efficient applications of Convolutional Neural Networks (CNNs) in automotive-rated and safety critical hardware-accelerators require an interplay of DNN design optimization, programming techniques and hardware resources. Ad-hoc pruning would result in irregular sparsity and compression, leading to in very inefficient real world applications. Therefore, the proposed methodology, called Dense-Sparse Convolution, makes use of the right balance between pruning regularity, quantization and the underlying vectorized hardware. Different word-length compute units, e.g. CPU, are used for low-latency inference of the sparse CNNs. The proposed open-source<sup>1</sup> CPU- kernel is enabled to scale along with the vector word-length and the number of cores.

# 1. Introduction

In the automotive industry, CNNs have become a common tool for solving complex problems in the field of computer vision, speech recognition and autonomous driving. CNNs have already proven their superiority in the resource demanding fields of image classification [1], semantic segmentation [2] and object detection [3]. Albeit impressive advances in the recent past, the deployment of CNNs on embedded devices is still challenging. Especially, modern CNNs in the context of computer vision are computationally expensive and require immense memory resources for storing their parameters. For instance, the efficient object detection algorithm YOLO [4] requires around 140 billion multiply-accumulate operations (MACs) and 240MB for storing the parameters. When applying DNNs for safety critical automotive systems, four main challenges can be identified. (1) A high-performant hardware (HW) and lower power consumption is needed to allow the prevention of accidents in real time with minimal latency. (2) The programmability of automotive HW plays a major role due to short time to market and cost pressure. (3) Reprogrammable HW, providing flexibility for being used for different tasks in the future, is needed. (4) The reliability of AD applications and HW used in safety critical environments is crucial to prevent accidents and currently not provided at a satisfying level. Consequently, it is clear that there is no single hardware which is on the top of the list for every criterion.

Making use of CNN optimization techniques combined with better utilization of HW resources and efficient algorithms, a compact solution would be obtained for efficient deployment. By combining pruning, quantization and vectorization of convolutional operations, the efficient use of instruction, data and thread-level parallelism is enabled. The key contributions of this paper are in particular:

- Making use of CNN optimization techniques (i.e. pruning and quantization) along with algorithmic optimization (Winograd) for an efficient vectorization.
- A methodology to accelerate the sparse convolutional layer on different word-length for vectorized compute units, *e.g.* CPU, is presented.
- Introduce an open-source CPU-kernel for low latency inference (batchsize=1) of CNNs.

# 2. Related Work

Different types of optimization techniques are applied on a structural, algorithmic and HW-level to reduce computational complexity and efficiently process DNNs.

https://github.com/HAPPI-Net/DSC.git

**Pruning** simply aims to reduce the number of connections of a CNN by removing redundant and unused parameters. As removing weights weakens the CNN, retraining of the given model is obligatory to maintain the accuracy. Han et al. [5] initially train the dense network and gradually prune it, if the individual weight value is below a certain threshold. They show that a majority of weights in the CNN can be set to zero with no loss in accuracy. Extending their element-wise pruning, Han et al. [6] proposed in *Deep Compression* further compression (weight sharing, approximation and data coding) leading towards more irregular memory accesses, which is impracticable for general purpose computing platforms such as CPUs. Recently, He et al. [7] proposed an automation technique for model compression using Reinforcement Learning (RL). They claim that their automation tool AutoML results in higher compression rates with better accuracy, compared to hand crafted rule-based pruning techniques. However, pruning regularity (see Fig. 1) is an important metric which is highly related to the actual acceleration of vectorized compute engines. This holds for most general purpose GPUs, CPUs and FPGAs, except for custom ASIC-architectures, such as *Efficient Inference Engine* [8] or SCNN [9], where the irregular sparsity is inefficient for other target platforms.



Figure 1. Illustration of pruning regularities, From (l) Irregular data pattern to (r) structured pattern.

Anwar *et al.* [10] argue that the irregular network pruning cannot achieve equally good speed up rates in terms of computation, as the compression causes irregular parallelism. Even though the high degree of sparsity could be achieved with element-wise pruning, deploying the pruned network on a fixed hardware results in overheads during computation and the obtained overall acceleration is considerably low. Thus, they analyze structured pruning on various scales, namely intra-filter level, filter level and channel level. Fig. 1 illustrates the pruning regularity occurring during pruning. He *et al.* [11] use the Lasso regression method to determine redundant channels instead of individual weights. They prune the redundant channels iteratively layer-wise and achieve a  $5 \times$  speed up for VGG16 with a 0.3% drop in the accuracy. However, as the regularity of pruning increases, the amount of sparsity decreases [12]. As channel-wise pruning discards entire channels of a CNN design, the compressed model is dense. Therefore, this paper discusses about sparse convolution including the trade-off between element-wise pruning, filter-wise pruning and regularity for vectorized inference.

Quantization is used to reduce the word-length of weights and activations, thus, resulting in lower external memory bandwidth and local memory demand. By using flexible modern vector-compute units, like those of CPUs, the number of MAC operations, performed in parallel, can be increased. There are mainly two approaches to quantize a DNN. The first approach is to convert floating-point to fixed-point without training. This approach is feasible in real world applications, reducing the time required for finetuning the CNN model. The fixed-point implementation proposed by Lin et al. [13] focuses primarily on a mixed precision, which allocates different bit widths and scaling factors to the weights and activations of different layers in DNNs. The bit width allocation is based on achieving an optimal output signal to quantization noise ratio (SQNR). Zhou et al. [14] convert a floating-point CNN into a network whose weights are either powers of two or zero. The three operations, namely weight partition, group-wise quantization and retraining, are iteratively performed until all the weights are converted into low precision. Vogel et al. [15] argue that the power of two quantization schemes suffer from high accuracy degradation and represent weights and input feature maps (activations) with powers of arbitrary log bases. Without fine-tuning, their approach observes less than 3% accuracy degradation using VGG-16. As the emerging target platforms support computation of flexible bit width for each layer, Wang et al. [16] use RL to determine the fixed-point parameters out of the vast design space, considering HW latency and model accuracy. However, most automotive-rated HW accelerators do not support flexible bit width.

**Binary Neural Networks** with bipolar parameters and activations require more intensive training as all the above quantization schemes. Rastegari *et al.* [17] use binary weights and activations to convert the expensive convolution to groups of XNOR and a pop count operation. However, there is a significant drop in accuracy levels due to binary transformation, which is not acceptable for most safety critical applications in the field of AD.

**Efficient convolution algorithms** are chosen to accelerate the convolutional layer of a CNN. The algorithm must be based on the target HW platform, the layer type and optimization schemes. The General Matrix Multiplication (GEMM) based convolution involves the transformation of the input and output feature map. Different schemes



Figure 2. Structure of the Dense-Sparse Convolution. A Conv. layer with dense (red), zero (gray) and sparse filter (blue) is shown left. The vectorized sparse convolution and Winograd convolution is shown right. Feature maps  $A_l$  are displayed in orange and green.

are analyzed by Anderson *et al.* [18]. The most common transformations *im2col* and *col2im* are available in the well known BLAS libraries for CPUs and GPUs with optimized portioning and memory access strategies. These GEMM based convolution performs superior when processing large batches.

The Fast Fourier Transform (FFT) based convolution [19] converts the convolution operation to element-wise multiplications in the frequency domain. Additional transformations are required as well (time domain  $\leftrightarrow$  frequency domain) and it is ideal for filter sizes close to the spatial input size.

Winograd convolution, generalized for CNNs by Lavin and Gray [20], transforms the filter and input feature maps to perform element-wise multiplication, causing a reduction between  $2.25 \times$  and  $4 \times$  compared to basic convolution. It also introduces intermediate transformation which can be computed using addition, subtraction and bit shift operations. Winograd convolution is useful for small batch and filter sizes as the transformation is applied to a small 2D tile. According to the input transformation, it is not able to utilize sparsity along with Winograd. To overcome this challenge, Liu *et al.* [21] proposed to prune the network in the Winograd domain, taking advantage of the sparsity. However, pruning in the Winograd domain in not easy to implement as most of the Deep Learning frameworks do not provide access to the intermediate transformations.

The direct sparse convolution algorithm proposed by Park *et al.* [22] allows to perform the convolution of sparse CNN models on various CPUs. The sparse matrices are stored in the Compressed Sparse Row (CSR) format which specifies the location and value of the non zero element. In their approach, a high degree of sparsity should be obtained to achieve a significant speed up, which eventually degrades the accuracy of a CNN model. This paper proposes to accelerate irregular pruned weights by separating them into sparse and dense components. The dense weight-filters perform best with the Winograd convolution. The sparse filters follow a similar convolution approach as *Direct Sparse Convolution* [22]. Therefore, the proposed approach is called *Dense-Sparse Convolution*. Another straight forward approach follows a much simpler path by pruning entire filters. As the Winograd transformation is a linear operation, a complete zero filter is also zero in the Winograd domain and does not have to be computed. In order to take advantage of this property, the proposed method of this paper prune the network filterwise and store the non zero filters in a special format which is designed to perform Winograd convolution efficiently.

# 3. Quantization and Vectorization

## 3.1. Vectorized Compute Units

The approximation of DNNs for a resource-aware implementation is beneficial for two reasons: (1) Reducing the memory footprint and (2) increasing arithmetic intensity of the application. Quantization of parameters can also be seen as a very structured way of pruning, as the CNN is stored and computed using less bits. At the same time, bandwidth requirements are reduced. From a technical point of view, fetching a 8-bit integer of a 32-bit float from memory demands the same effort. Modern memory systems never fetch a single data element. For instance, the smallest data transfer between the memory and an i7 CPU is 64-bits and for an Altera Arria 10 it is 32-bit which need to be aligned. Fig. 3 displays the difference between an aligned and an unaligned data access, causing different number of fetch operations. Moreover, the typical size of a CPU data cache line is 64-bytes (most Intel or ARM processors).

Altera Arria 10's minimal BRAM<sup>2</sup> access is also 64-



Figure 3. Aligned data vs. unaligned data for vectorized memory access instructions. (Top) Fetching  $16 \times 32$ -bit worlds at once, see red block. (Bottom) Two read operations are required.



Figure 4. Vectorized processing examined on varying CPUs (128bit, 256-bit, 512-bit) and word-lengthes (32-bit, 16-bit, 1-bit).

bytes. That means,  $64 \times 8$ -bit integer or  $16 \times 32$ -bit values can be fetched to registers in parallel, as they are aligned. Hence, quantization enables fetching more data, during the aligned process with the same time and energy effort. From a transistor level's point of view, arithmetic operations' the low bit width approximations cover less area on the chip than full precision ones. Therefore, more arithmetic operations can be performed on the same area. Fixed HW, such as CPU, GPU or DSP is limited to a dedicated wordlength, *e.g.* 8-bit, 16-bit, 32-bit, 64-bit operations. Throughout this work, data level parallelism is carried out using CPU-based SIMD instruction. In particular, different wordlengths of 128-bit, 256-bit and 512-bit are evaluated on 32 full-precision and 16-bit approximations. A SIMD instruction based data level parallelism is shown in Fig. 4. With the right HW-support computing a vectorized operation takes the same time as a scalar one.

#### 3.2. Quantization of the CNN Model

The main concern of low precision representations of DNNs lies in preserving numerical correctness for the particular tensors by minimizing the rounding and conversion errors. Whereas the quantization of a DNN to FP16 or Int16 works out of the box, the conversion to a shorter word-length requires more effort. Weights w are quantized according to the rounding function rd(w). In contrast to stochastic rounding, applied by Gupta *et al.* [23], round-tonearest is applied as expressed in Eq. 1, where IB and FB are the integer and fractional-bits. The precision  $\epsilon = 2^{-FB}$ is limited by FB and the upper and lower bounds of  $\tilde{w}$  are  $\pm 2^{\text{IB}} - 2^{\text{FB}}$ . As the precision range is limited, batch normalization [24] is used before every convolutional layer.

$$\tilde{w} = rd(w) = \begin{cases} -2^{\mathrm{IB}} - 2^{\mathrm{FB}} & \text{if } w < -2^{\mathrm{IB}} - 2^{\mathrm{FB}} \\ 2^{\mathrm{IB}} - 2^{\mathrm{FB}} & \text{if } w > 2^{\mathrm{IB}} - 2^{\mathrm{FB}} \\ \lfloor w \rfloor & \text{if } \lfloor w \rfloor \le w \le \lfloor w \rfloor + \frac{\epsilon}{2} \\ \lfloor w \rfloor + \epsilon & \text{if } \lfloor w \rfloor + \frac{\epsilon}{2} < w \le \lfloor w \rfloor + \epsilon \end{cases}$$
(1)

# 4. Methodology

#### 4.1. Dense Convolution

As the trend for the application of CNNs is moving towards deeper topologies with small filters, the dense convolution can be implemented more efficiently using the Winograd algorithm [20]. Every element in the output feature map is separately computed, whereas the Winograd-based convolution generates output maps in tiles. It reduces the number of multiplication by reusing the intermediate outputs and is suitable for small filter sizes and stride values. In the proposed work, the 2D Winograd Convolution  $F(2 \times 2, 3 \times 3)$  is used, where the generated output tile size is  $2 \times 2$  and the filter size is  $3 \times 3$ . The required input tile size is  $4 \times 4$  (4 = 3 + 2 - 1). The output of the Winograd convolution for a convolutional layer can be expressed as shown in Eq. 2. Here, g is a tile of the input feature map and bis a convolutional filter before the Winograd transformation respectively. Due to linearity in the Winograd transformation, Eq. 2 can be expressed as Eq. 3. The transformed filters and images are flattened into single vectors to fit into the appropriate register sizes, scalable across various CPU platforms as demonstrated in Fig. 2. Winograd convolution is efficient for smaller filters. Larger filters and fully connected layers could profit from common GEMM libraries. For  $F(2 \times 2, 3 \times 3)$ , the matrices required for transformation matrices G, B, and A are shown in Eq. 4. Moreover, g is a

<sup>&</sup>lt;sup>2</sup>BRAM called Block-RAM, embedded memory resource in FPGA, which can be accessed every clock cycle.

 $4 \times 4$  tile of the input feature map and b is a  $3 \times 3$  filter of the convolutional layer.

$$Y = \sum_{channels} A^T \Big[ G^T g G \odot B^T b B \Big] A \tag{2}$$

$$Y = A^T \Big[ \sum_{dense filters} [G^T g G \odot B^T b B] \Big] A$$
(3)

$$G = \begin{bmatrix} 1 & 0 & 0\\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2}\\ \frac{1}{2} & \frac{-1}{2} & \frac{1}{2}\\ 0 & 0 & 1 \end{bmatrix} B = \begin{bmatrix} 1 & 0 & -1 & 0\\ 0 & 1 & 1 & 0\\ 0 & -1 & 1 & 0\\ 0 & 1 & 0 & -1 \end{bmatrix} A = \begin{bmatrix} 1 & 0\\ 1 & 1\\ 1 & -1\\ 0 & -1 \end{bmatrix}$$
(4)

A superior data level parallelism can be achieved by using lower precision data and increasing the arithmetic intensity of the executions. The vector operations can be efficiently applied on the assembly level with low level intrinsics. For instance, MMX, SSE, AVX<sup>3</sup> extensions exist for ARM-based CPUs. SSE extension consists of 128-bit wide register, which can store four single-precision floating-point numbers. However, the same register can also store two 16-bit and four 8-bit fixed-point integers. SSE extension only supports 16-bit, 32-bit, 64-bit multiplication, whereas 8-bit multiplication is not supported. Other extensions such as MMX, AVX and AVX-512 support 64-bit, 256-bit and 512-bit wide registers respectively.

The Winograd transform of a sparse filter is not necessarily sparse but the transformation of a completely pruned filter will be zero and does not need to be multiplied. Thus, the summation should only cover channels where the corresponding filter is assumed to be dense. The transformed filters are converted into a single vector and fit into the SIMD register, based on the quantization parameters of the layer.

### 4.2. Sparse Convolution

Based on the number of zeros in the filter, the filter is assigned as a completely sparse matrix. These sparse matrices are stored in the CSR format, and direct sparse convolution as presented by [22], is adopted. The data array only contains the non zero elements, and the core idea lies in determining the location of these elements and multiply those with the corresponding input tiles. The data level parallelism for the sparse convolution is utilized by duplicating the non zero elements onto the SIMD registers, as shown in the Fig. 2.

### 4.3. Dense-Sparse Convolution

The element-wise sparsity with magnitude-based pruning results in a reasonable amount of completely pruned filters, using only direct sparse convolution benefits from fully removed filters or filters with one or two non zero elements (see Fig 2 top left). Whereas the convolution of slightly pruned filters, using a sparse algorithm, reduces the overall acceleration. Thus, the direct sparse convolution is performed for the filters with less than a certain number of non zero elements and dense convolution for the remaining ones. The threshold, used to consider the filter as sparse, is a heuristic process which depends on the spatial size of the input, filter and the overall distribution of sparsity. The overall algorithm for Dense-Sparse Convolution is demonstrated in the supplementary material. As the convolution is based on multiplication and summation through the input channels, the results of sparse and dense are accumulated together, as shown in Eq. 5.

$$A_{li,k,x,y} = \sum_{c}^{sparse} \sum_{v=1}^{R} \sum_{u=1}^{S} A_{l-1i,c,x+u,y+v} W_{lk,c,u,v} + \sum_{c}^{dense} \sum_{v=1}^{R} \sum_{u=1}^{S} A_{l-1i,c,x+u,y+v} W_{lk,c,u,v}$$
(5)

These two operations are orthogonal from the HW's point of view and they can be simultaneously executed over different cores. Dense-Sparse Convolution can be implemented for any kind of layers, independent of the regularity of sparsity. This methodology could be further adopted on various HW platforms like CPUs, GPUs and FPGAs. In case of Nvidia-GPUs and FPGAs, the partitioning of CUDA cores and DSP blocks, must be performed for the sparse and dense operations, respectively. When pruning is performed filter-wise, the number of completely sparse filters could be further increased. In this case, the Dense-Sparse Convolution could be transformed to filter-wise Winograd convolution.

#### **5. Experimental Results**

The CNN design of VGG16 [25] is compressed with respect to structural and algorithmic level optimization. A batch size of 1 is used for inference. As one level has an impact on the other level, an ablation study is carried out hereinafter. Three different CPUs are used throughout the experiments, which are in particular:

- ARM-Cortex A53 (Raspberry Pi3), which operates at 1.2 GHz and has 4 cores. The CPU has 32 kB L1 and 512 kB L2 caches and supports NEON SIMD extensions of ARM.
- Intel i5-6300, which is a dual-core processor with hyperthreading capability and it is operating at 2.4 GHz. The CPU has 128 kB L1, 512 kB L2 and 3 MB L3 caches and supports AVX2 SIMD extensions.
- Intel Xeon Gold 6152 which has 22 cores with hyperthreading capability. The CPU operates at 2.1 GHz and

<sup>&</sup>lt;sup>3</sup>MMX : Multi Media Extension, SSE : Streaming SIMD Extensions, AVX : Advanced Vector Extensions

has 1.375 MB L1, 22 MB L2 and 30.25 MB L3 caches and supports AVX512 SIMD extensions.

All kernels are written in low-level C, using Intel or ARM intrinsic functions and OpenMP library. These aspects are covered in Sections S2 and S3 of the supplementary material. The entire firmware is compiled either using Intel or Gnu Compiler Collection (GCC).

**Ablation Study:** Element-wise pruning removes redundant parameters from the filters and between 50-90% sparsity is obtained for each layer after pruning, see Fig. 5.



Figure 5. Pruning rate and speed up with varying pruning regularity. Accuracy is preserved for element and filter-wise pruning.

Rather than measuring the acceleration (inference time) with respect to a basic tile based approach, the performance of sparse convolution is compared to the fastest dense algorithm. In case of VGG16, using the Winograd algorithm represents the fastest algorithm (black curve), as shown in Fig. 6.



Figure 6. Theoretical acceleration of sparse convolution with varying precision. Model accuracy is not taken into consideration.

The experiment is performed with an input feature size of  $224 \times 224 \times 64$  and an filter of size  $3 \times 3 \times 64 \times 64^4$ . In order to reduce the computation demand, a pruned parameter

is replaced by zero and is not to be included in the calculation. The sparse convolution algorithm is based on direct sparse convolution, as described in Section S1 of the supplementary material. The effect of HW optimization methods and algorithms on dense convolution is compared, concluding that a significant amount of acceleration is achievable with choosing the suitable algorithm and the appropriate optimizations. However, the direct sparse convolution is faster than the dense Winograd convolution in fullprecision, when the sparsity lies over 92%. The first reason is that the sparse convolution algorithm is based on basic tile-based convolution, wherein the Winograd convolution has the advantage of less multiplications required during computation. Secondly, while the Winograd convolution has efficient and regular memory access, sparse convolution has irregular memory access due to unstructured pruning. In order to further increase bandwidth utilization, we implemented low-precision convolution with 16-bit weights and activations. It appears that in low-precision diminishes the bandwidth problem of sparse convolution. By reducing the precision from 32-bit floating-point to 16-bit fixedpoint, bandwidth utilization significantly increases and the amount of cache misses drastically decreases. As a result, sparse convolution becomes faster than Winograd in case of 70% sparsity.

By keeping the spatial size of the filter fixed as  $3 \times 3$ , one wide, large feature ( $224 \times 224$ ) and one small one ( $28 \times 28$ ) is analyzed, as illustrated in Fig. 7. Aligned with the first assumption, one can observe that sparse convolution is not that efficient on large input features, due to the memory bandwidth problem. Contrarily, on small input sizes, sparse convolution becomes faster than Winograd, even with 50% sparsity in full precision. In conclusion, the convolution algorithm has to be chosen based on the sparsity and feature size. Concludingly, rather than utilizing the same algorithm for all convolutional layers, different algorithms should be chosen for efficient deployment on vectorized compute units, likewise CPUs.

The threshold is determined heuristically and three convolution methods are compared for various layers of VGG-16 as shown in Fig. 8. The *Conv1.2* layer of VGG16 has 80% overall sparsity, however direct sparse convolution leads to the lowest speed, compared to full-dense and Dense-Sparse Convolutions. Dense-Sparse Convolution with a threshold equal to zero offers around  $2\times$  acceleration compared to direct sparse convolution. For *Conv3.3*, layer the overall sparsity is 75%. The threshold is chosen as *one* and Dense-Sparse Convolution performs around  $1.5\times$  faster compared to full-dense and direct sparse convolutions. Thus, Dense-Sparse Convolutions provides the optimal computation strategy for convolution of irregularly sparse filters. Depending on the size of the spatial input and the filter size, the sparsity rate and distribution of sparsity

<sup>&</sup>lt;sup>4</sup>These are layer dimensions of the first convolutional layer in VGG16.



Figure 7. Comparison of sparse and dense convolutions with varying feature sizes.

over the filters an optimum threshold value can always be determined. For layers where spatial sizes are small and overall sparsity is high, direct sparse convolution is preferred. Moreover, layers where the spatial size is large and the sparsity is low *,e.g. conv1.1*, can be computed with full dense convolution.



Figure 8. Comparison of *sparse*, *dense* and *dense-sparse* convolutions.

It is clear that the main performance improvement goes along with entirely pruned filters, see Fig. 2 (Zero Filter). By adopting the filter-wise pruning approach from the beginning, higher sparsity can be attained on the filter level. That would solve two main challenges of unstructured pruning: memory required to store indexes and deceleration arising from decoding compressed parameters. With filter-wise pruning, fully pruned and fully dense filters are obtained. Thus, the Dense-Sparse Convolution does not have to perform sparse convolution for filters with only a few pruned weights. Only dense convolution is calculated on filters, remaining after pruning, see Fig.5. The performance of the Dense-Sparse Convolution algorithm is measured by evaluating its acceleration with respect to the filter-wise sparsity ratio.

**Benchmark:** The results in this work are compared to those of several other works in Table. 1. The results obtained from different HW are compared to the Dense-Sparse variants run on an Intel i5 CPU. The accuracy is maintained similar to the full precision dense model throughout all experiments. The execution time is at its best by applying the filter-wise pruning strategy on the CNN model combined with highly optimized Winograd-based dense convolutional algorithm. The performance of CPU based realization of the *Dense-Sparse* methodology is very competitive against state-of-the-art dedicated HW solutions.

Table 1. Low latency application of VGG16 trained on ImageNet with different optimization methods. Usually, accuracy is maintained throughout our experiments, which is 68.5% Top1 and 88.7% Top5. Irregular pruning has 76.7% sparsity and regular pruning 68.5% sparsity.

Implementation	Prec.	Lat. [ms]	Acc. [%]
FPGA Virtex-7 [26]	Int16	151.8	66.5/86.9
FPGA XC7Z045 [27]	Int16	224.6	64.6/86.9
FPGA Stratix-V [28]	Int8/16	262.9	66.5/87.5
GPU JTX1 [29]	Fp32	200	68.5/88.7
CPU (Intel i7) [30]	Fp32	858	/88.7
Ours, Intel i5-6300:	•		
Irreg. Pr., Dense Conv.	Fp32	529	68.4/88.7
Irreg. Pr., Sparse Conv.	Int16	403	68.5/88.7
Filt. Pr., Dense-Sparse	Fp32	295	68.3/88.6
Filt. Pr., Dense-Sparse	Int16	174	68.5/88.7

# 6. Conclusion

Network optimization techniques, such as pruning and quantization, can reduce computational and memory demand. However, there is no significant advantage without realizing an efficient convolution scheme. It is evident that the dense Winograd convolution can simply outperform direct sparse convolution during inference, even with sparse filters. This work develops a novel algorithm called Dense-Sparse Convolution which enables the acceleration of irregularly pruned CNNs. By providing the flexibility of separating sparse and dense filters, we are able to accelerate sparse convolution. Furthermore, handcrafted HW-level optimizations maximize the efcient resource utilization, using vectorized SIMD intrinsics and multithread parallelization. The best performance can be extracted by incorporating the structured filter-wise pruning in the pruning strategy. Finally, a comparison of the proposed Dense-Sparse Convolution with state-of-the-art papers reveals a  $2.6 \times$  acceleration on an Intel i5 and  $5 \times$  speedup on an embedded ARM Cortex-A53.

# References

- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 10437–10453, 2016.
- [2] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *TPAMI*, 2017. 1
- [3] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *NIPS*. 2015. 1
- [4] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *CVPR*, 2016. 1
- [5] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. NIPS, 2015. 2
- [6] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2016. 2
- [7] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *ECCV*, September 2018. 2
- [8] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. ISCA, 2016. 2
- [9] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. ISCA, 2017. 2
- [10] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *JETC*, 2017. 2
- [11] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, Oct 2017. 2
- [12] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. Exploring the regularity of sparse structure in convolutional neural networks. *CoRR*, abs/1705.08922, 2017. 2
- [13] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *ICML*, pages 2849–2858. JMLR.org, 2016. 2
- [14] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *CoRR*, abs/1702.03044, 2017. 2
- [15] Sebastian Vogel, Mengyu Liang, Andre Guntoro, Walter Stechele, and Gerd Ascheid. Efficient hardware acceleration of cnns using logarithmic data representation with arbitrary log-base. *ICCAD*, pages 1–8, 2018. 2

- [16] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: hardware-aware automated quantization. *CoRR*, abs/1811.08886, 2018. 2
- [17] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In ECCV, 2016. 2
- [18] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. Low-memory gemm-based convolution algorithms for deep neural networks. *CoRR*, abs/1709.03395, 2017. 3
- [19] Michaël Mathieu, Mikael Henaff, and Yann Lecun. Fast training of convolutional networks through ffts. In *ICLR*, 2014. 3
- [20] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *CVPR*, pages 4013–4021, 2016. 3, 4
- [21] Xingyu Liu, Jeff Pool, Song Han, and William J. Dally. Efficient sparse-winograd convolutional neural networks. *CoRR*, abs/1802.06367, 2018. 3
- [22] Jongsoo Park, Sheng R. Li, Wei Wen, Hai Li, Yiran Chen, and Pradeep Dubey. Holistic sparsecnn: Forging the trident of accuracy, speed, and size. *CoRR*, abs/1608.01409, 2016. 3, 5
- [23] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. ICML, 2015. 4
- [24] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, pages 448–456. JMLR.org, 2015. 4
- [25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. 5
- [26] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *ISLPED*, pages 326–331, New York, NY, USA, 2016. ACM. 7
- [27] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA*, pages 26–35, New York, NY, USA, 2016. ACM. 7
- [28] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *FPGA*, pages 16–25, New York, NY, USA, 2016. ACM. 7
- [29] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016. 7
- [30] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *TPAMI*, 38(10):1943–1955, October 2016. 7