ABM-SpConv-SIMD: Accelerating Convolutional Neural Network Inference for Industrial IoT Applications on Edge Devices

Xianduo Li, Xiaoli Gong, *Member, IEEE,* Dong Wang, *Member, IEEE,* Huayou Su, Jin Zhang, *Member, IEEE,* Thar Baker, *Senior Member, IEEE,* Jin Zhou, and Tingjuan Lu

Abstract—Convolutional Neural Networks (CNNs) have been widely deployed, while traditional cloud data-centers based applications suffer from the bandwidth and latency network demand when applying to Industrial-Internet-of-Things (IIoT) fields. It is critical to migrate the CNNs inference to edge devices for efficiency and security concerns. However, it is challenging to deploy complex CNNs on resource-constraint IIoT edge devices due to a large number of parameters and intensive floating-point computations. In this paper, we propose ABM-SpConv-SIMD, an on-device inference optimization framework, aiming at accelerating the network inference by fully utilizing the low-cost and common CPU resource. ABM-SpConv-SIMD first adopts a model optimizer with pruning and quantization, which produces *Sp*arse *Conv*olutional models. And then, the *A*ccumulation-*B*efore-*M*ultiplication mechanism is proposed to reduce multiplication operations. Additionally, the *SIMD* instructions, which are commonly available on cost-effective edge devices, are employed to improve the performance of convolutions. We have implemented *ABM-SpConv-SIMD* base on the ARM Compute Library software framework and evaluated on Hikey970 and Raspberry Pi devices with two representative models AlexNet and ResNet50. The results show that the ABM-SpConv-SIMD can significantly improve the performance, and achieve on average of 1.96x and 1.73x speedup respectively over the baseline implementation with negligible loss of accuracy.

Index Terms—Convolutional Neural Networks, Edge Devices, Industrial Internet-of-Things Applications, Single Instruction Multiple Data, Sparse Convolution

1 Introduction

The rapid development of sensor technology, network communication and cloud computing technology enables many industrial Internet-of-Things (IIoT) applications, covering a wide range of fields such as structural health monitoring and remote diagnosis [1], intelligent transportation [2], financial technology applications [3], and industrial control systems such as specific voice control [4]. In the past few years, Convolution Neural Networks (CNNs) has developed into one of the key branches of deep learning and paved the way in many domains such as computer vision [5] and language translation [6]. Based on its powerful feature extraction and data analysis capabilities, CNNs are also widely used in many IIoT applications. For example, CNN models have shown outstanding results for human activity recognition in a smart home environment [7], surface defect inspection in intelligent industrial production [8], and health symptoms monitoring such as elderly fall detection [9] and chronic diseases management [10] in smart healthcare. Besides, CNN can also be used as a fundamental model in IIoT applications based on other artificial intelligence algorithms, such as federated learning-based architecture for detecting Android malware applications [11].

Generally, CNN models should be trained with extensive data available and then used for inference in the specific scenario for numerous IIoT applications. Model training means incrementally modifying the weight values associated with connections in the network until a satisfactory error rate has been achieved, which is usually deployed on a customized data-center infrastructure. And model inference refers to flowing the input data through the network and getting the output such as classification result. In an edge-side environment, edge devices are often required to provide low inference latency for a single request to improve efficiency, rather than high inference throughput as in a data-center environment. Therefore, we focus on edge-side inference latency optimization in this paper.

Normally, edge-side CNNs inference relies on the computing resources in the cloud-side servers, while the edge devices which contain various sensors are only responsible for collecting data and then uploading it to the servers for further inference. However, the cloud-based approach may violate the real-time requirements of delay-sensitive IIoT applications due to network congestion and may cause resource conflict of remote computing servers while the

- *This work was supported in part by Natural Science Foundation of China (62172239), Beijing Natural Science Foundation (Grant 4202063), Fundamental Research Funds for the Central Universities (Grant 2020JBM020), and Open Fund of PDL(WDZC20215250123).*
- *Xianduo Li, Xiaoli Gong, Jin Zhang are with the College of Computer Science, Tianjin Key Laboratory of Network and Data Security Technology, Nankai University, China.*
- *Dong Wang is with Beijing Jiaotong University, Beijing, China.(Email: wangdong@bjtu.edu.cn).*
- *Huayou Su is with National University of Defense Technology, Changsha, China.*
- *T. Baker is with Department of Computer Science, University of Sharjah, UAE.(Email: tshamsa@sharjah.ac.ae.)*
- *Jin Zhou is with the Chest Hospital of Tianjin, China.*
- *Tingjuan Lu is with the 903rd Hospital of PLA, China.*
- *Jin Zhang is the corresponding author of this paper. (Email: nkzhangjin@nankai.edu.cn).*

Manuscript received on October-30, 2021.

local edge devices are left idle [12]. Even though the network performance has been significantly improved, such as 5G, Wi-Fi and UWB, it is still challenging to get a guaranteed response from the remote data-centers which are required to handle numerous requests from massively distributed edge devices. Besides, transmitting and storing personal health data with cloud servers may raise many privacy and security challenges. Therefore, deploying the CNNs inference on local edge devices rather than with the help of cloud servers is necessary, which makes no data transmission and then no privacy threat nor network latency.

Running CNNs inference requires intensive floating-point computing power and memory bandwidth, which is impractical on cost-effective edge devices. To address this problem, lots of research has been conducted. For example, many specifically designed hardware components, such as floating-point accelerators, Graphics Processing Units (GPUs), and Neural Network Processing Unit (NPU) [13], have been integrated into edge devices to provide turbo performance. At the same time, new software frameworks such as Tensorflow Lite [14] and Pytorch Mobile [15] are proposed. However, integrating these new hardware technologies into industrial IoT applications is still challenging due to the strict cost control, energy constraints, and infeasibility of hardware expansion on existing industrial devices.

In this paper, ABM-SpConv-SIMD, an on-device inference runtime optimization framework, is proposed to accelerate model inference for CNN-based IIoT applications by fully utilizing the common CPUs resource on the commercial-off-the-shelf cost-effective edge devices. ABM-SpConv-SIMD consists of two key steps. Firstly, a pruning and quantization approach is adopted to remove the redundant parameters and filters, which can reduce the computation power required during inference. And then, the expensive floating-point operations are converted to fixed-point integer operations through quantization. Secondly, an Accumulation Before Multiplication Sparse Convolution algorithm (ABM-SpConv) is designed based on the sparse filters after quantization, which can reduce the costly multiplication operations. After that, a multi-channel parallel convolution is implemented based on SIMD instructions, which indicates that the common CPU hardware resources are fully utilized.

In summary, our paper makes the following contributions:

- We propose a framework that employs offline pruning and quantization, and online optimization to fit CNN models for cost-effective edge devices.
- We design and implement ABM-SpConv-SIMD, which optimizes the sparse convolution algorithm with commonly available SIMD instructions, and then improves the CNNs inference performance on edge devices.
- We conduct a series of experiments to evaluate the performance of ABM-SpConv-SIMD, including ImageNet classification experiments of deploying AlexNet and ResNet50 on two ARM-based SoCs, Hikey970 and Raspberry Pi 3B. The results show that our framework can achieve a speedup of 1.96x on Hikey970 and 1.73x on Raspberry Pi with less than 2% accuracy loss, compared with the GEMM based convolution algorithm.
- ABM-SpConv-SIMD is not CNN model specific and is compatible with other CNNs for inference latency optimizations.

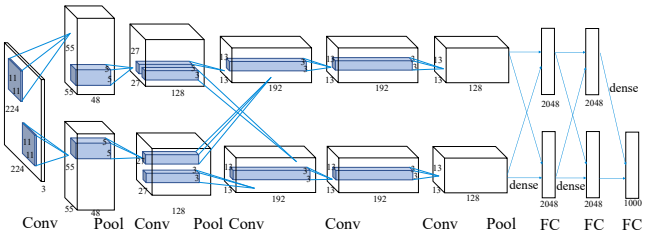


Fig. 1: An illustration of the architecture of AlexNet [5], where Conv, Pool, and FC represent the convolutional layer, pooling layer, and fully-connected layer, respectively.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 illustrates our ABM-SpConv-SIMD framework. Section 4 presents our experimental methodology. Section 5 describes the experiment results. Section 6 provides the related work. And Section 7 concludes the paper and considers future work.

2 Background

2.1 Convolutional Neural Networks

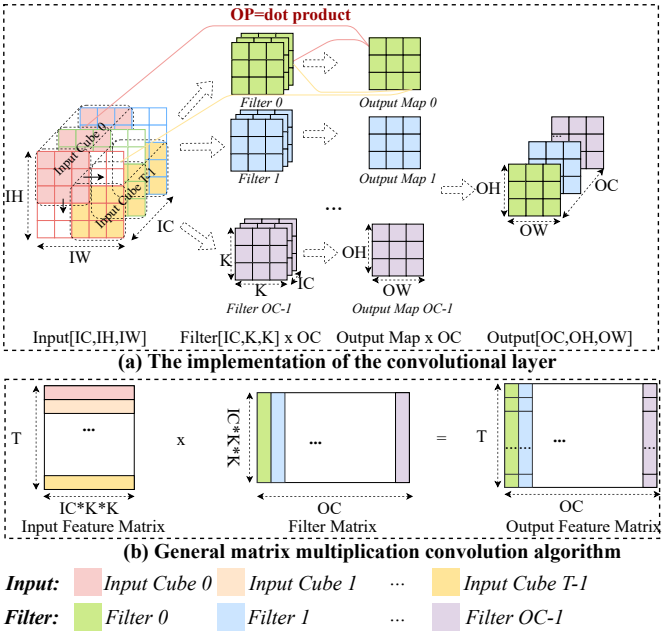


Fig. 2: The implementation of the convolutional layer and general matrix multiplication convolution algorithm.

Convolutional Neural Networks are wildly used in many domains, such as computer vision [5] and video analysis [16]. A CNN model generally consists of several layers which perform different operations on a given input, such as convolutional layers, pooling layers, and fully-connected layers. Fig.1 shows the structure of Alexnet [5], a classical CNN proposed in 2012. Specifically, the convolutional layers

TABLE 1: Classic convolutional neural networks and corresponding parameters, where MACs represent multiply-accumulate operations that compute the product of two numbers and add that product to an accumulator.

Year	Name	No. of Layers	No. of Conv Layers	No. of Parameters		No. of MACs		Accuracy
				Conv Layers	FC Layers	Conv Layers	FC Layers	
2012	AlexNet [5]	8	5	2.3M	58.6M	666M	58.6M	83.6%
2014	Overfeat [17]	8	5	16M	130M	2.67G	124M	85.8%
2014	VGG16 [18]	16	13	14.7M	124M	15.3G	130M	92.6%
2015	GoogLeNet [19]	22	21	6M	1M	1.43G	1M	93.3%
2016	ResNet50 [20]	50	49	23.5M	2M	3.86G	2M	96.4%

and fully-connected layers are the major component of the model. At the same time, these layers also contribute to the major floating-point computation and memory access.

With the rapid development of the neural network, deeper models are proposed to be integrated with more convolutional layers. Tab.1 shows the accuracy and parameters of the five classical CNN models including AlexNet [5], Overfeat [17], VGG16 [18], GoogLeNet [19] and ResNet50 [20]. It can be seen that as the accuracy of the model increases, the parameters or multiply-accumulate (MAC) operations in the convolutional and fully-connected layers are increasing. According to prior research [21], the execution time of convolutional and fully-connected layers account for over 90% of the total inference latency. Therefore, this research focuses on optimizing convolutional and fully-connected layers based on the architecture resource available on the CPUs wildly deployed in edge devices.

Convolution is a common operation in the two-dimensional data processing. Normally, there is an input data such as an image, and a filter, a small bitmap sliding over the input data to extract input features. Nowadays the input data generally contains multiply channels, which makes the input data as three-dimensional arrays. For example, there are three channels (RGB) for input data from camera. Therefore, the convolution operation on the input data is extended from two-dimensional to three-dimensional.

Fig.2-(a) shows the implementation of a convolutional layer. The input and filter of the convolutional layer are three-dimensional arrays, where IC, IH, IW, K represent the input channel, input height, input width, and filter height/width, respectively. It is worth noting that the filters have the same channel IC as the input data. As the Fig.2 shows, for the three-dimensional convolution, a filter (e.g., Filter 0) slides over the input, and the corresponding three-dimensional input data extracted at every position is donated as **input cube**. Then, multiple input pixels in the input cube are multiplied with corresponding weights in the filter, the results of which are accumulated and then stored as an output element. And the above operations of an input cube and a filter is donated as **dot product**. When a filter slides over the whole input data, multiple input cubes (e.g., Input Cube 0 to Input Cube T-1) will be extracted to do dot product with this filter to get multiple output elements, which are then arranged into a two-dimensional output map (e.g., Output Map 0) of size [OH, OW]. Assuming that there are OC filters of a convolutional layer, when repeating the above calculation for OC times, a three-dimensional output data of size [OC, OH, OW] is generated. In summary, the detailed implementation of the convolutional layer can be summarized as multiple dot product process, where each dot product corresponds to an input cube and a filter.

Traditionally, in the fully-connected layer, each filter **convolves** on the whole input image to extract the overall features of the input instead of local features, so the size of filters in the fully-connected layer is the same as the input image. Therefore, the fully-connected layer can be treated as a special case of the convolutional layer with input data of size [IC, IH, IW], filters of size [OC, IC, IH, IW], and output data of size [OC, 1, 1]. So among the various existing deep learning frameworks, the fully-connected layers share lots of the implementation details with the convolutional layers. Therefore, we use convolutional layers to generalize the two categories of convolutional and fully connected layers subsequently.

In many deep learning frameworks, the convolutional layer is implemented and optimized based on the General Matrix Multiplication (GEMM) convolution algorithm. First, convert the three-dimensional input data into a two-dimensional input feature matrix, a row of which represents an input cube. Then, convert OC three-dimensional filters into another filter matrix, a column of which represents a filter. Finally, the convolution is converted to the matrix multiplication of input feature matrix and filter matrix, as is shown in Fig.2-(b).

2.2 Pruning and Quantization

The toughest challenge for deploying CNNs inference computations on edge devices is the extremely high computational workload and storage costs. To overcome this obstacle, various schemes, including *model quantization (reducing arithmetic precision of the parameters)* [22], [23], [24], *low-rank factorization (factorizing the weight matrix into low-rank matrices)* [25], *knowledge distillation (distilling the knowledge learned from a complex network and pass it to a small network)* [26], [27], and *network pruning (trimming unimportant weights)* [28], [29], have been proposed and studied to compress the model size and reduce the computational workload effectively.

According to the granularity of pruning operation conducted in the algorithms, existing pruning schemes can be divided into two broad categories: structured pruning [29], [30], [31], [32], [33] and unstructured pruning (also referred to as weights pruning) [28], [34], [35], [36], [37], [38]. The granularity of unstructured pruning is a single neuron, while the granularity of structured pruning can be channels, filters, and even layers. Normally, for general accelerator hardware (such as CPUs and GPUs), structured pruning is more effective [39] in deployment due to the regular sparse granularity. For dedicated neural network processors [40], [41], [42], unstructured pruning methods are more attractive as they can normally achieve better reduction effects on both

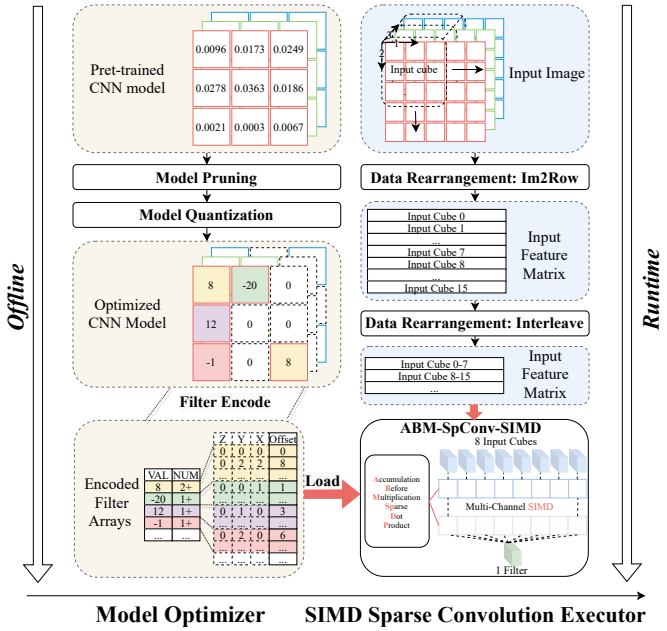


Fig. 3: Overview of the ABM-SpConv-SIMD framework.

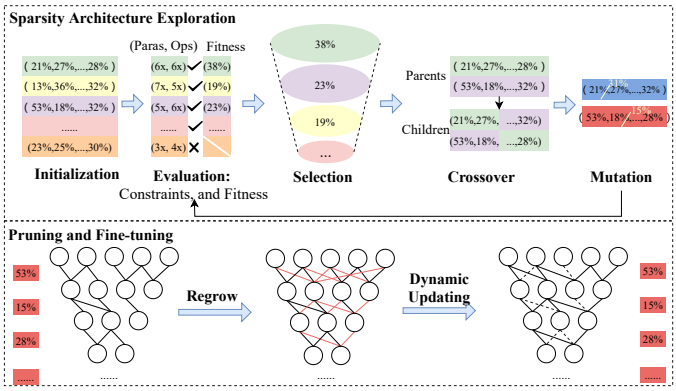


Fig. 4: Multi-objective CNN network pruning algorithm adopted by this work.

the memory footprint and computational workload of the DNN model over structured pruning approaches [30]. To achieve better reduction effects, we propose to use unstructured pruning schemes on embedded CPU processors. And to overcome the irregular sparsity, we design a new sparse convolution algorithm that can efficiently accelerate CNN models compressed by unstructured pruning schemes.

2.3 SIMD

Single instruction, multiple data (SIMD) technology such as Intel AVX [43] and ARM NEON [44] are widely used in processors to enhance multimedia user experiences, such as video watching, editing and enhancing, game processing, photo processing, and voice recognition, which are among the most common requirements of today's embedded edge devices such as smartphones and tablets. SIMD units refer to hardware components in processors that perform the same operation on multiple data operands concurrently. Typically, SIMD units perform operations directly on a vector register that contains several operands of the same length. The smaller the bit-width of each operand, the more the number of operands that can be processed in parallel. Therefore, SIMD is suitable for accelerating complex and intensive computational processes with high repeatability and simple logic, such as matrix multiplication. Compared with specially designed hardware accelerators, SIMD units are more cost-effective for resource-constrained edge devices. As we described in 2.1, there are many dot product operations between different input cubes and different filters in the convolutional layer, which are independent of each other and therefore suitable for NEON acceleration.

3 System Design

3.1 Framework Overview

Fig.3 shows the ABM-SpConv-SIMD framework, including a model optimizer in the front end and a SIMD sparse convolution executor in the back end. First, to make the CNN models suitable for edge devices with limited memory bandwidth and computation power, for a given pre-trained CNN model, the **model optimizer** performs model pruning to remove the redundant weights or connections and model quantization to convert floating-point weights into fixed-point. For the special filters after pruning and quantization, we redesign an *accumulation before multiplication sparse dot product (ABM-SpDP)* algorithm to replace the original *dot product*, which can remove costly multiplication computations. Moreover, in order to collect information of valid weights for sparse convolution, we encode each pruned filter into two arrays. All of the above processes are completed on the *offline* host machine. Second, to reduce CNNs inference latency on local edge devices, the back-end **SIMD sparse convolution executor** performs **runtime** optimization of the convolutional layer by utilizing common SIMD instructions. ABM-SpConv-SIMD first performs data rearrangement methods, including input cubes extracting and interleaving to meet the requirements of SIMD instructions to parallel accessing the memory. dThen, instruction-level parallelism is achieved by mapping multiple ABM-SpDP calculation corresponding to multiple input cubes in a convolutional layer to different channels of SIMD registers. Both the input rearrangement and SIMD instruction-level parallelism are executing on the *online* ARM devices. In summary, ABM-SpConv-SIMD makes the following runtime optimizations for each inference of CNN, including multiplication operation reduction, data layout optimization for vectorization, instruction parallelism by SIMD, and fix-point integers to fully utilize the SIMD registers.

3.2 Pruning and Quantization

Pruning and quantization are efficient approaches to compress CNN models and fit them into edge devices, which can determine the quality of CNNs by fine-tuning or even retraining the pre-trained models. Inspired by the studies of *Deep Compression* [45], *MetaPruning* [33] and *Ristrerro* [24], we have developed an efficient CNN pruning approach that can deliver optimal pruning results both in terms of memory footprint and computational workload, and thus can significantly improve the runtime performance of the pruned model when deployed on embedded processors.

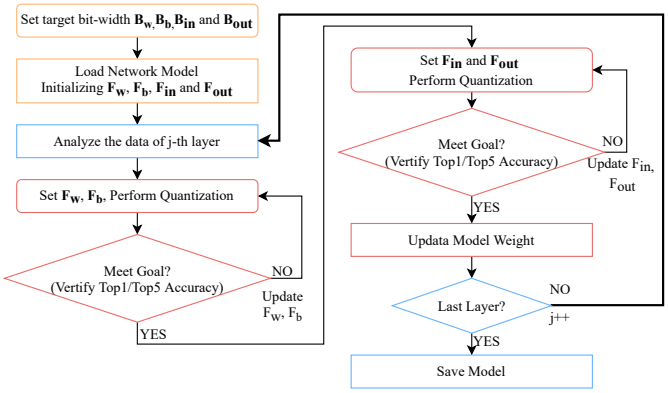


Fig. 5: Fixed-point model quantization flow adopted by this work.

Inspired by the recent studies of *Deep Compression* [45] and *MetaPruning* [33], we introduce a genetic algorithm-based multi-objective CNN pruning flow. As illustrated by Fig.4, the flow first performs sparsity architecture exploration to find the suitable sparsity ratio for each layer with given accuracy and performance constraints. And then, based on the optimal pruning ratio obtained in the first step, the second step conducts several iterations of weight-pruning and weigh-regrow to tune the CNN model and restore the accuracy.

The proposed pruning scheme maps the network pruning procedure as a multi-objective optimization problem. The optimization objectives include: CNN model size (memory cost), computational workload and accuracy. Given a CNN as $W = \{W_1, W_2, \cdots W_L\}$, where W_l represent the filter in each layer and $W_l \in \mathbb{R}^{OC_l \times IC_l \times K_l \times K_l}$, $1 \leq l \leq L$. L is the total number of layers. OC_l defines the number of filters of the size $IC_l \times K_l \times K_l$ in the l-th layer, where IC_l matches the number of channels of the input feature map. Further, the input/output feature maps are of the size $IC_l \times H_l \times W_l$ and $IC_{l+1} \times H_{l+1} \times W_{l+1}$, respectively. Assuming after pruning, the size of the model is reduced by $\mathcal{R}_{param}$ times, which can be quantitatively calculated by

$$\mathcal{R}_{param} = \frac{1}{\sum_{l=1}^{L} S_l^p \cdot p_l}, S_l^p = \frac{OC_l \cdot IC_l \cdot K_l \cdot K_l}{\sum_{l=1}^{L} OC_l \cdot IC_l \cdot K_l \cdot K_l} \quad (1)$$

where p_l represents the pruning rate of the weight in the l-th layer, and the optimal values of p_l will be searched by the genetic algorithm in the first step of the proposed algorithm. We also introduce S_l^p as a saliency score that measures the contribution of the l-th layer to the total model size. Similarly, we also quantitatively define the workload reduction ratio $\mathcal{R}_{ops}$ as

$$\mathcal{R}_{ops} = \frac{1}{\sum_{l=1}^{L} S_l^o \cdot p_l}, S_l^o = \frac{OC_l \cdot IC_l \cdot K_l \cdot K_l \cdot H_{l+1} \cdot W_{l+1}}{\sum_{l=1}^{L} OC_l \cdot IC_l \cdot K_l \cdot K_l \cdot H_{l+1} \cdot W_{l+1}} \quad (2)$$

where we introduce S_l^o as the computational workload saliency for the l-th layer. For a given network architecture, the distribution of layer-wise saliency can be obtained in advance. Both S_l^p and S_l^o can be use as an weights to balance the model's memory footprints and workloads during the searching procedures.

In this work, the goal of the pruning algorithm is to achieve both the desired memory footprint and computational workload goals at the same time during network pruning for given accuracy budget. Therefore, we formula the CNN pruning flow as the following multi-objective optimization procedure:

$$\begin{aligned} \underset{p_1, p_2, \ldots, p_L}{\arg\min} \quad & \mathcal{L}\left(Net\left(p_1, p_2, \ldots p_L; W\right)\right) \\ \text{s.t.} \quad & \mathcal{R}_{param} \geq \mathcal{P}_{set} \\ & \mathcal{R}_{ops} \geq \mathcal{F}_{set} \end{aligned} \quad (3)$$

where Net denotes the baseline network model. $\mathcal{P}_{set}$ and $\mathcal{F}_{set}$ represent the desired memory and workload reduction ratios, respectively.

Fig.4 shows the CNN pruning flow. The first stage is neural network sparsity architecture exploration. Based on this prior knowledge of the saliency score of each layer, a generic algorithm is used to explore the entire sparsity architecture space, i.e., $\{(p_1, p_2 \cdots p_L) : p_l \in [0, 1], l = 1, 2, \cdots L\}$, and then find all the good pruning ratio candidates that meet both model size and workload targets obtaining the Pareto solution set. The searching flow is conducted as follow: 1) randomly generating several chromosomes as the initial Pareto solution set; 2) pruning the network according to each sparsity setting (i.e., the chromosomes) and evaluating the accuracy as the fitness scores of all pruned sparse networks under both memory footprint and workload constraints. To prune the network, the network parameters are first sorted according to their absolute value, and the low ranking parameters are then trimmed (forced to zero values). In each layer, the total number of parameters to be trimmed are calculated from the sparsity setting; 3) chromosomes with the highest fitness scores are preserved and added to the elitists, and then mutation and crossover are performed to obtain a new population according to predefined probability; 4) repeatedly conducting the evaluation-selection-crossover-mutation procedure until the algorithm finds a satisfactory Pareto solution set that satisfies both memory footprint and workload constraints. Finally, the top-ranked pruning ratio combination is promoted as the most suitable sparsity architecture to meet the desired pruning goals. The second stage is the pruning and fine-tuning stage. We use the pruning rate obtained in the exploration stage to perform pruning and fine-tuning on the pre-trained model. To prevent some important parameters from being pruned incorrectly, we allow the trimmed weights to regrow and the model accuracy to improve correspondingly.

For network quantization, we adopt a uniform bit-width setting but variable fractional bit-width for each CNN layer. In each layer, the filter, input and output data are presented in fixed-point numbers as $Q_w \cdot 2^{-F_w}$, $Q_b \cdot 2^{-F_b}$, $Q_{in} \cdot 2^{-F_{in}}$ and $Q_{out} \cdot 2^{-F_{out}}$, respectively. The variable Q denotes the fixed-point binary word with B-bit length, while F denotes a bias presenting the number of fractional bits of the fixed-point number. Fig.5 illustrates the CNN quantization flow adopted in this work, which is based on the scheme used in [46] but with some improvements. For each layer, the quantization method first determines the F_w, F_b and performs quantization for the filters, then verifies whether the Top1/Top5 accuracy meets the preset threshold value. If the goal (accuracy) is satisfied, the quantization of input and output continues, otherwise, the F_w, F_b value is updated and the filter is re-quantized. The main difference with the scheme used in [46] is that, instead of minimizing the

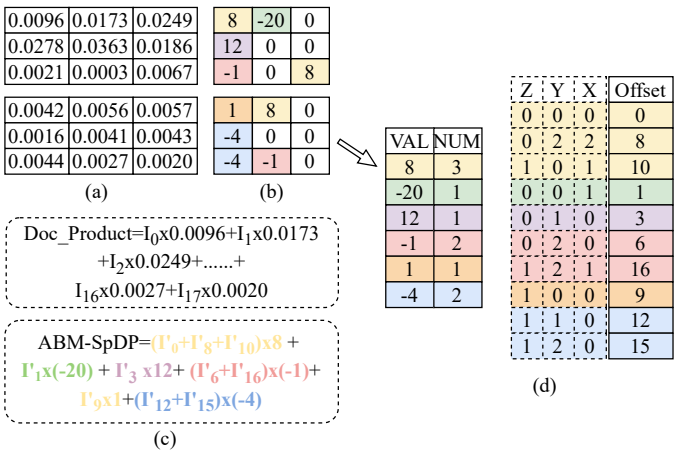


Fig. 6: An example of (a): original partial filter in the third convolutional layer of AlexNet, (b): pruned and quantized filter corresponding to the original filter, (c): *original dot_product* and *accumulation before multiplication sparse dot product (ABM-SpDP)* proposed in this paper, and (d): two arrays after filter encoding.

truncation error during quantization of the weights, our quantization flow searches for the group of parameters F_w, F_b, F_{in} and F_{out} jointly, which minimizes the hardware cost while satisfying the accuracy bound at the same time. Our modified scheme provides a faster algorithm runtime and better control of the desired classification accuracy.

3.3 The ABM-SpDP Algorithm

Through pruning and quantization, the sparse filters are filled with zero-value weights, and the 32-bit floating-point weights are converted to 8-bit fixed-point integers. For example, Fig.6-(a) and Fig.6-(b) show an original filter and a converted sparse filter respectively. The original dot product of the original filter is shown clearly in Fig.6-(c), where I_x represents the input pixel corresponding to the x_{th} weight when executing dot product. For the special sparse filter, there are a large number of weights with the same value after quantization. Thus the corresponding input pixels must be multiplied by the same value when executing the dot product. Therefore, we can **first accumulate** these input pixels and **then multiply** the accumulation result with the weight value. Take value 8 in Fig.6-(b) as an example, the dot product can be converted from $(I_0 \times 8 + I_8 \times 8 + I_{10} \times 8)$ to $(I_0^{'} + I_8^{'} + I_{10}^{'}) \times 8$, where the $I_x^{'}$ represents the quantized input pixels. Thus the multiplication operations are reduced from three to one. Furthermore, Fig.6-(c) shows the computation of all valid weights, where zero-value weights are not involved in the calculation, and 8-bit integer weights are used to do sparse convolution instead of floating-point weights.

Assuming that filters are quantized using q bits, there are a total of $Q = 2^q$ different values at most for the filter. Then, the sparse convolution is shown in the following formula, where w_p denotes the weight value and $\sum FI_p(w)$ represents all the input pixels in the input cube corresponding to the weights with value w_p at different positions. We call this redesigned dot product as **accumulation before multiplication sparse dot product, ABM-SpDP**. For each non-zero weight value w_p in the filter, ABM-SpDP first determines the position of multiple weights with value w_p and finds the input pixels corresponding to these weights. Then ABM-SpDP accumulates these input pixels and multiplies the results with w_p, which means that the valid weights with value w_p in this filter have been processed. Finally, for other non-zero weight values, repeat the above process and accumulate the results. An output FO, which represents the final convolution result of an input cube and a filter, can be obtained when processing all the valid values. Through accumulation before multiplication, the costly redundant multiplication operations corresponding to the weights with the same value are reduced, contributing to performance gains.

$$\begin{aligned} FO &= \sum FI_0(w) \cdot w_0 + ... + \sum FI_{Q-1}(w) \cdot w_{Q-1} \\ &= \sum_{p=0}^{Q-1} (w_p \cdot \sum FI_p(w)) \end{aligned} \tag{4}$$

As described above, the valid non-zero weight values and their occurrence positions are useful for the pruned filters, so we encode the pruned filters into two arrays to record the valid information. As Fig.6-(d) shows, for a non-zero valid value, we first store its fixed-point integer value *VAL* and the total number *NUM* of weights with value *VAL* in the filter. Then the *VAL* and *NUM* are merged into a 16-bit signed short integer, which is saved as an item in the first array. In addition, we use another array to store the occurrence positions *(Z, Y, X)* of valid weights. For *NUM* weights with the value *VAL*, *NUM* items are generated in the second array to represent the *NUM* valid positions. And the positions of different weights corresponding to different values are stored in the order in which the value appears.

Since the filter size is fixed for a CNN model, the offset of a valid weight with respect to the base address of the filter can be calculated as $Offset = Z \times (K \times K) + Y \times K + X$, where K represents the width and height of the filter. Thus the address of corresponding input pixel is calculated as follows: $Addr = P + Offset = P + Z \times (K \times K) + Y \times K + X$, where P represents the base address of an input cube. Therefore, we propose to store the *offset* into 16-bit unsigned short integers offline, rather than the *(Z, Y, X)* pairs. For example, Fig.6-d shows a simple case of filter encoding. There are three weights with value 8, so one entry in the first array is *VAL=8, NUM=3* and three entries in the second array are *0, 8, 10*, which are offsets corresponding to three valid occurrence positions *(0, 0, 0)*, *(0, 2, 2)*, *(1, 0, 1)*.

3.4 Accelerating Multiple ABM-SpDP Processes with SIMD Instructions

As described in 2.1, the detailed implementation of the convolutional layer can be summarized as multiple *dot product* processes, where each dot product corresponds to an input cube and a filter. And the original dot product can be optimized to *ABM-SpDP*, as described above. Therefore, all the computations in the convolutional layer can be summarized as multiple ABM-SpDP processes. Since each input cube is independent and performs the same operations in the convolutional layer, we choose to correspond multiple input

cubes with multiple channels of SIMD registers, i.e., one channel handles one input cube. , which enables parallelism of multiple ABM-SpDP processes associated with multiple input cubes. And the SIMD-based optimized convolution is donated as accumulation before multiplication sparse convolution (ABM-SpConv-SIMD). Therefore, we first need to extract all the input cubes and then effectively arrange multiple input cubes to meet the requirements of SIMD instructions to parallel access the memory. Based on the above analysis, we design the input rearrangement method.

3.4.1 Data Rearrangement Method to Meet the Requirements of SIMD Instructions

Since the input data is multidimensional, the input pixels in an input cube are discontinuous in memory, which impairs data locality when executing convolution and introduces data dependency of adjacent input cubes. To address this problem, we first design **Im2Row**. As is shown in Fig.7-(a), as the filter slides over the input data in the fixed stride, the input cube of size [IC, K, K] is extracted into a one-dimensional vector and then saved as a row in the input feature matrix. We extract input pixels in the order of width dimension, height dimension, and channel dimension, and then get a *ICxKxK* vector. When the filter is finished sliding over the whole input data, all the input cubes can be extracted and organized into a two-dimensional input feature matrix.

However, in the input feature matrix, even though the data of a single input cube is stored continuously, the arrangement of data among multiple input cubes does not satisfy the memory access patterns of NEON instructions. For example, when processing the first elements of multiple input cubes simultaneously, the existing input feature matrix cannot guarantee the continuity of those first elements, so loading the first elements of multiple input cubes cannot be achieved by one NEON instruction. To solve this problem, we design the second input rearrangement method **Interleave**. Considering the fixed length of NEON registers and overflow in subsequent addition and multiplication operations, we will divide the 128-bit NEON registers into eight channels. Therefore, we need to interleave eight input cubes. As is shown in Fig.7-(b), the Interleave method first rearrangements the first elements of eight input cubes and interleaves them into **consecutive** memory space, then processes the following elements until all the elements of the eight cubes are processed. After that, eight input cubes are stored alternately, but the eight elements in the same position are stored consecutively. As is shown in Fig.7-(c), by specifying the start address of every eight elements, they can be simultaneously loaded into eight channels of a SIMD register for further computation.

3.4.2 Multi-Channel Parallel ABM-SpDP Processes Based on SIMD

Despite the SIMD parallel optimization, the algorithm logic of ABM-SpConv-SIMD is the same as the original ABM-SpDP logic, which is briefly summarized as follows: (1) for each non-zero weight value *VAL* in the filter, ABM-SpConv-SIMD first determines *NUM* positions of weights with value *VAL* and loads *NUM input pixels groups* corresponding to these weights, (2) then ABM-SpConv-SIMD accumulates these *input pixels groups* correspondingly, (3) multiplies the accumulation result with *VAL*, (4) finally, for other non-zero weight value, repeats the above process and accumulates the multiplication results to get the final results, which equals to the dot products of eight input cubes and one filter. Now we introduce the details of each step.

First, load input pixels groups corresponding to multiple weights with the same value by using **LD1** instructions. As Fig.8 step-1 shows, to determine the positions of these input pixels groups, a *VAL-NUM* pair in the first encoded array and *NUM* positions ((*Z, Y, X)* pair or *offset*) for weights with value *VAL* in the second encoded array should be loaded. Then, for each input pixels group corresponding to a valid weight, the address can be calculated as $addr = P + 8 \times offset$, where P represents the base address of the eight input cubes after input rearrangement. Then LD1 instructions are used to load consecutive 64-bit input data into eight-channel SIMD registers, with each channel storing an 8-bit input pixel corresponding to an input cube. In order to avoid data overflow in subsequent accumulation, we expand the 8-bit signed integer in each channel into 16-bit by using the **SSHLL** instructions.

Second, accumulate *NUM* groups of input pixels corresponding to *NUM* weights with the same value by using **ADD** instructions. The ADD instruction adds the data stored in the corresponding channels of the two source registers, which indicates that each input pixels stored in each channel of a SIMD register are accumulated with corresponding pixels stored in the rest *(NUM-1)* SIMD registers. Eight different channels in a register are processed independently, thus enabling the parallel processing of eight input cubes.

Third, multiply the accumulated input result with the current valid weight value by using **MUL** instructions. When multiplying the 16-bit accumulated input result with the 8-bit valid weight, the multiplication of the two must be stored using more than 24 bits to prevent data overflow. Therefore, we use 32 bits to represent the multiplication results. We first expand the 16-bit accumulated input into 32-bit by using **DUP** instructions, and the original eight-channel register is divided into two four-channel registers, as shown in Fig.8 step-3. Then, we expand the valid 8-bit weight value *VAL* stored in the first encoded array into 32-bit and use **DUP** instructions to copy the valid weight into two four-channel registers. At last, we perform 32-bit multiplication operations and get the corresponding 32-bit multiplication results for the current valid weight value. Similar to the ADD instruction, the MUL instruction multiples the data stored in the corresponding channels of the two source registers independently, which means the accumulated input result of each input cube can multiply with corresponding valid weight separately.

Finally, repeat the above three steps and get multiple temporary multiplication results corresponding to multiple valid weight values, and then we can get the final convolution results of eight input cubes and a filter by accumulating all the temporary multiplication results. The results are stored in two four-channel SIMD registers, with each channel corresponding to an input cube, as is shown in Fig.8. Then the convolution results of input and filter should be added with bias.

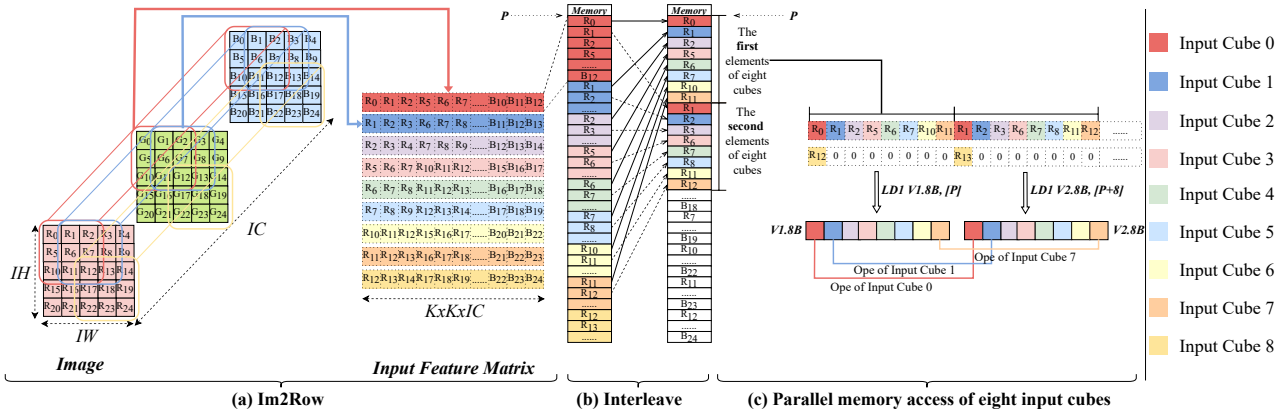


Fig. 7: The overview of input rearrangement methods, including Im2Row and Interleave.

Since the precision of input data and filter is F_{in} and F_w respectively after quantization, the precision of the above convolution result is $F_{in} + F_w$, which is not the same as the precision F_b of bias, so the precision control needs to be considered when adding with bias. Besides, the precision F_{out} of output data in this layer is equal to the precision $F_{in}^{'}$ of input data in the next layer, which is not the same as the precision $F_{in} + F_w$ of the convolution result and the precision F_b of bias, so the precision control needs to be considered when saving the output data too.

3.4.3 Precision Control for Fixed-Point Computation

As mentioned in Section 3.2, the input, filter, bias and output data are presented in fixed-point numbers as $Q_{in} \cdot 2^{-F_{in}}$, $Q_w \cdot 2^{-F_w}$, $Q_b \cdot 2^{-F_b}$ and $Q_{out} \cdot 2^{-F_{out}}$, respectively. The variable Q denotes the fixed-point binary word used for calculation in ABM-SpConv, while F denotes a bias presenting the number of fractional bits of the fixed-point number. Therefore, the above convolution result of input and filter can be represented as $I \cdot 2^{-(F_{in}+F_w)}$, with I representing the fixed-point integer of convolution result stored in SIMD registers. The bias can also be represented as $B \cdot 2^{-F_b}$, with B representing the fixed-point integer of bias. When adding I and B, we first use **SHL** instructions to convert bias to integers with the same number of fractional bits as I. That is, load B and shift B left by $F_{in} + F_w - F_b$ bits and get $B^{'} = B \cdot 2^{F_{in}+F_w-F_b}$, as is shown in the *first five instructions* of Fig.8 step-4. Then, the bias can be represents as $B^{'} \cdot 2^{-(F_{in}+F_w)}$. Through adding I and $B^{'}$, we can get the output result, which is represent as $(I + B^{'}) \cdot 2^{-(F_{in}+F_w)}$, as the *sixth and seventh instructions* of Fig.8 step-4 show.

The last step is converting the fractional bits of output from $F_{in} + F_w$ to F_o, so we shift $(I + B^{'})$ left by $F_o - (F_{in} + F_w)$ bits, and get $O = (I + B^{'}) \cdot 2^{F_o-(F_{in}+F_w)}$, as the *eighth and ninth instructions* show. It is worth noting that when executing shifting, the fixed-point output integer should be rounded up to the nearest integer instead of being directly floored. And to achieve this, we add $C = 2^{(F_{in}+F_w)-F_o-1}$ to $(I + B^{'})$, and then shifting left $(I + B^{'})$. When finishing the above steps, the output result can be represented as $O \cdot 2^{-F_o}$, with O representing the fixed-point integer of output.

Then, as the last five instructions show, we use **XTN** instructions to narrow the 32-bit output into 8-bit and use **ST1** instructions to store the output to the corresponding position of the output feature matrix, with a row representing output pixels corresponding to a filter and a column representing output pixels corresponding to an input cube. When we process all the input cubes and filters, a complete output feature matrix can be obtained, and then we can use Row2Im to convert the output feature matrix to three-dimensional output data, which is the inverse process of Im2Row.

4 Experimental Methodology

4.1 Experimental Setup

To evaluate the effectiveness of ABM-SpConv-SIMD, we measure the latency of several CNN models on two representative modern edge devices: Hikey970 and Raspberry Pi 3B. First, Hikey970 consists of four high-performance 2.36-GHz ARM Cortex A73 cores and four energy-efficient 1.8-GHz ARM Cortex A53 cores. Its hardware specification represents the high-end edge SoCs. Second, Raspberry Pi represents the edges SoCs of low-end devices and consists of four 1.2-GHz ARM Cortex A53 cores. The OS used in our experiment is Ubuntu18.04.

4.2 CNN Models

We choose to evaluate two representative CNN models from two different CNN classes, AlexNet [5] and ResNet50 [20]. AlexNet [5] represents early CNNs with large filter size. ResNet50 represents CNNs with complex network structures, which have a large number of convolutional layers and floating-point operations. We choose Pytorch pre-trained AlexNet and ResNet50 model as our baseline model. The number of parameters, weight size, number of MAC operations, and top-1/5 accuracy of the baseline model is shown in Tab.2 and Tab.3. It is worth noting that the number of MAC operations we count for the baseline AlexNet model is the sum of all MAC operations in the two-way group convolution, rather than the MAC operations in the only one-way group convolution as is shown in Tab.1.

We utilize ARM Compute Library [47], which provides optimized CNN layer implementations such as GEMM convolutional layer and Relu activation layer for ARM CPU cores, to construct CNN models and execute CNNs inference. We use two different strategies to construct CNN

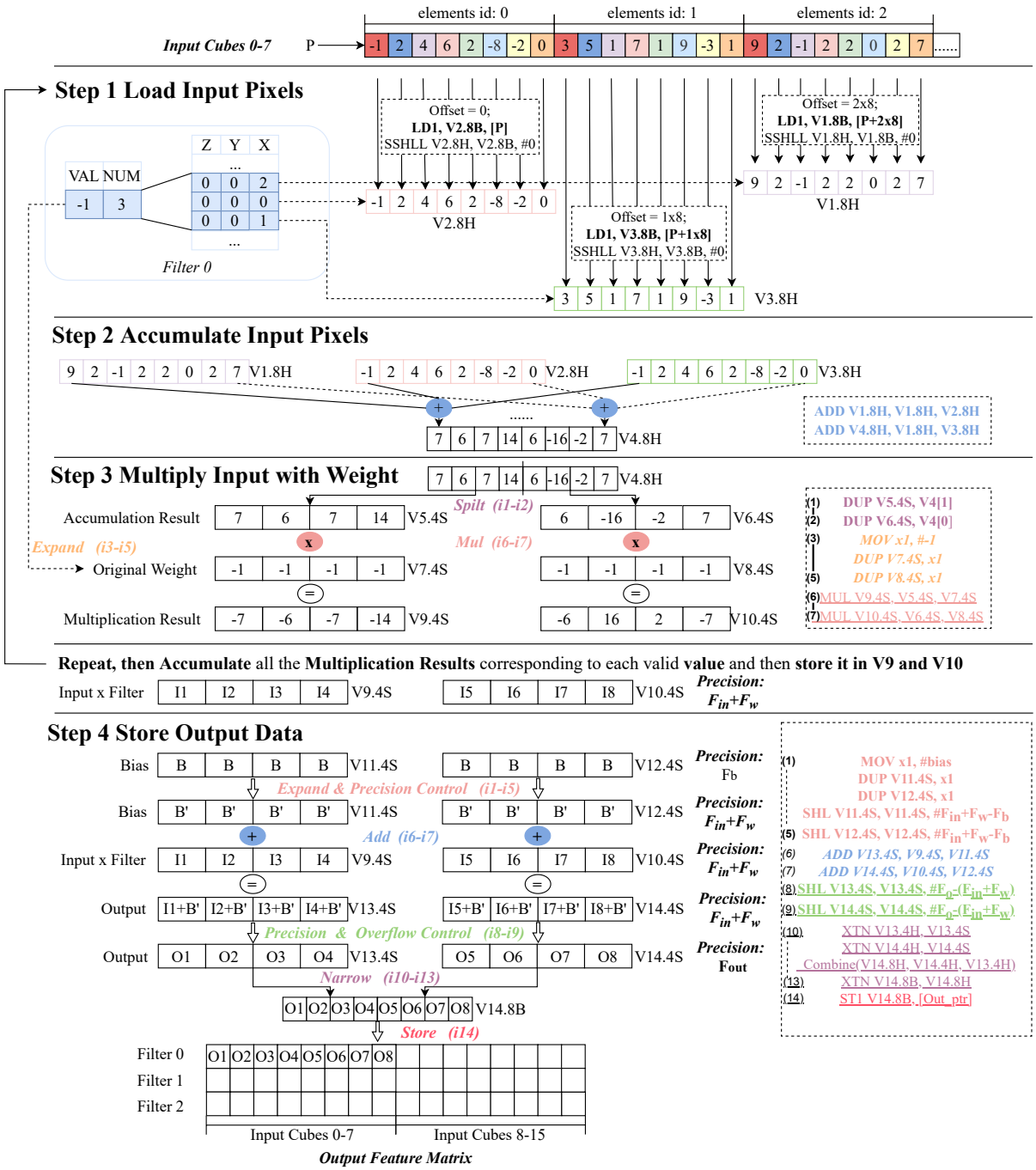


Fig. 8: Overview of multi-channel parallelism of eight ABM-SpDP processes based on SIMD instructions.

models, one is using **ABM-SpConv-SIMD** framework and loading the optimized model parameters after pruning and quantization, the other is using **GEMM** convolution algorithm provided by ARM Compute Library and loading original pre-trained model parameters. Both CNN models are designed for image classification, specifically images in the ImageNet dataset.

5 Experiment Results

5.1 Pruning and Quantization Results

Tab.2 first shows the number of parameters after pruning, weight size after quantization and filter encoding, and number of operations of our optimized models. Compared with the baseline model, our approach can effectively reduce the number of parameters, thus reducing storage usage. In addition, after pruning and quantization, a large number of floating-point operations are converted to a very small number of fixed-point MUL and ADD operations, especially the number of multiplication operations is greatly reduced.

Fig.9 shows the number of parameters for each layer of AlexNet and partial ResNet50. It can be seen that the parameters of each layer are effectively reduced after model optimization. Moreover, the pruning rate of each layer is also different. Take Alexnet as an example, the fully-connected layers contain the main parameters, as is shown

TABLE 2: The number of parameters, weight size, and number of operations of the baseline and optimized CNN models.

Model	No. of Parameters		Weight Size		No. of Operations		
	Baseline	After Pruning	Baseline	Encoded	Baseline (MACs)	After Pruning (MUL)	After Pruning (ADD)
AlexNet	60.95M	5.77M	233MB	11.65MB	1.14G	19M	107M
ResNet50	25.50M	4.70M	97MB	10.08MB	4.09G	173M	753M

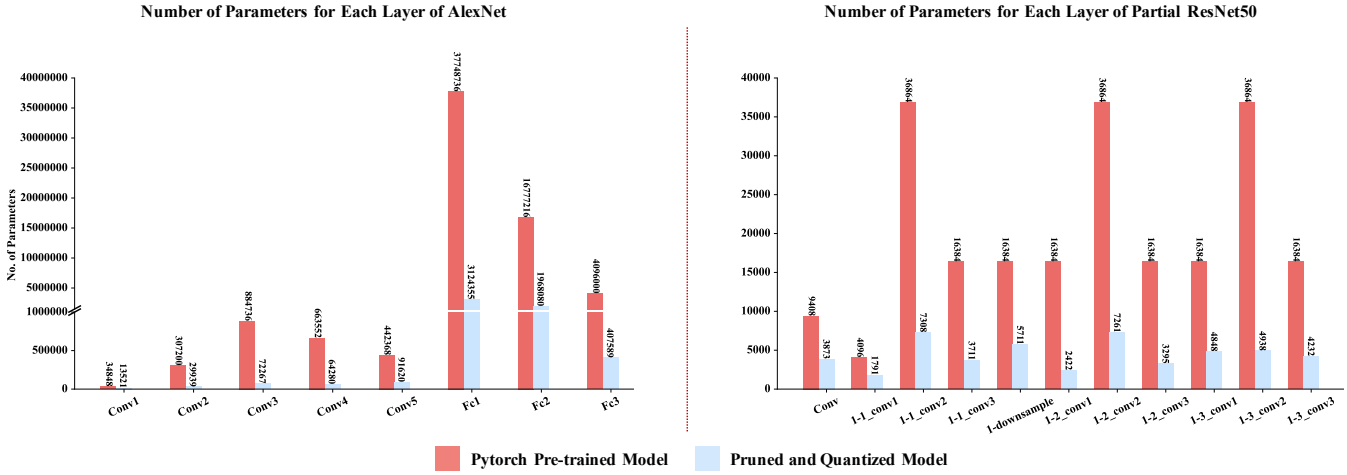


Fig. 9: Layer-wise parameter scale of AlexNet and partial ResNet50.

TABLE 3: Inference Accuracy of the baseline CNN models and the models processed by the model optimizer.

Model	Top-1/5 Accuracy	
	Baseline	After Pruning
AlexNet	56.55% / 79.09%	54.67% / 78.83%
ResNet50	76.15% / 92.87%	74.36% / 91.33%

in Fig.9, after model optimization, the parameters of the fully connected layer are reduced by a factor of 10.

5.2 Inference Accuracy

As is shown in Tab.3, for the inference accuracy of classification tasks on the validation set of ImageNet, ABM-SpConv-SIMD introduces less than 2% accuracy loss compared with original pre-trained models.

5.3 Inference Latency

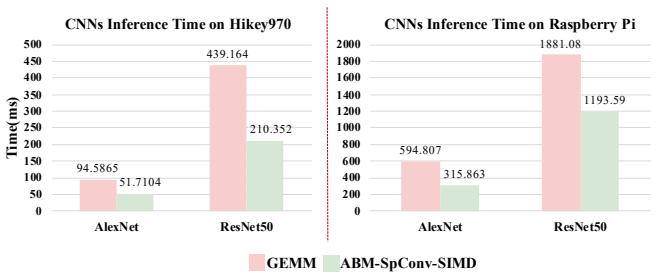


Fig. 10: Average inference latency of AlexNet and ResNet50 on Hikey970 and Raspberry Pi, respectively.

Fig.10 shows the average end-to-end inference latency of AlexNet and ResNet50 on Hikey970 and Raspberry Pi. It clearly shows that the inference latency of models built based on ABM-SpConv-SIMD is significantly reduced compared with those built on GEMM. More precisely, ABM-SpConv-SIMD achieves 1.83x and 2.09x speeds up for AlexNet and ResNet50 on Hikey970, respectively. Moreover, on Raspberry Pi, ABM-SpConv-SIMD can achieve 1.88x and 1.58x speeds up for AlexNet and ResNet50, respectively.

For further analysis, we evaluate the inference latency for each layer of AlexNet and partial ResNet50, as is shown in Fig.11 and Fig.12. First, the performance improvement of end-to-end inference mainly comes from the convolutional and fully-connected layers. In addition, for the activation layers in two models and eltwise layers in ResNet50, using 8-bit fixed-point integers instead of floating-point numbers can also reduce execution time. This is because SIMD instructions can process more channels simultaneously when processing 8-bit integers, thus achieving higher parallelism. Second, using ABM-SpConv-SIMD exhibits different degrees of performance improvement for the convolutional and fully-connected layers. For example, for the first convolutional layer in AlexNet, our framework has almost no performance gains. However, for the fully-connected layers in AlexNet, our framework can obtain 3.48x performance improvement on Hikey970 and 3.87x performance improvement on Raspberry Pi. Furthermore, the performance improvements of fully-connected layers come from the front-end model optimizer reducing most of the parameters and a lot of memory bandwidth usage when executing convolution online. As the original fully-connected layers in AlexNet have a large number of parameters, the model optimizer achieves greater pruning strength for fully connected layers than other convolutional layers. In summary, we believe that ABM-SpConv-SIMD can gain performance improvement in edge devices where computation and band-

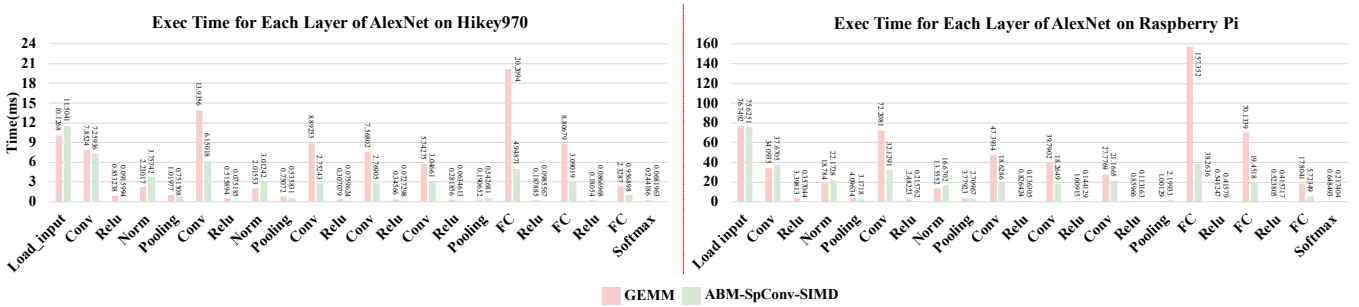


Fig. 11: AlexNet layer-wise inference latency on Hikey970 and Raspberry Pi.

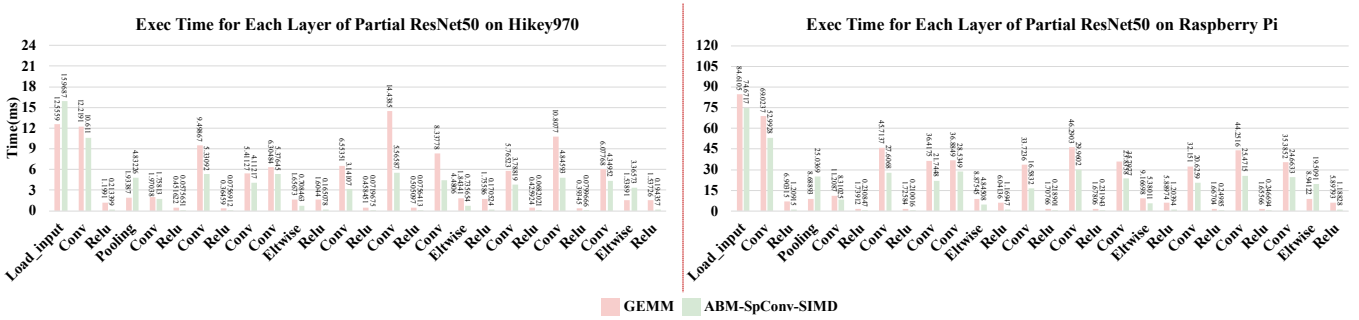


Fig. 12: Partial ResNet50 layer-wise inference latency on Hikey970 and Raspberry Pi.

width are very limited.

5.4 Insight at Micro-architectural Level

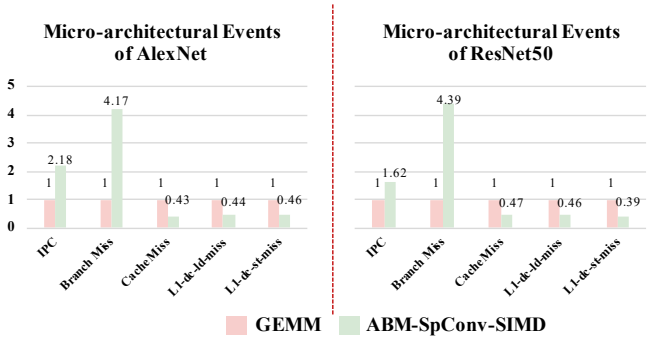


Fig. 13: Micro-architectural events comparison between GEMM and ABM-SpConv-SIMD of AlexNet and ResNet50 on Hikey970.

After obtaining significant performance improvements, we use *perf*, a performance analysis tool provided by the linux kernel, to analyze the sparse convolution algorithm from a micro-architectural insight. Fig.13 compares our approach with classical GEMM on five micro-architectural events of AlexNet and ResNet50 on Hikey970. As can be seen, the execution of the ABM-SpConv-SIMD improves the IPC during program execution, which indicates that our approach can exploit the processor performance more fully. In addition, the ABM-SpConv-SIMD can significantly reduce the L2 Cache Miss, L1 Dcache Load Miss, and L1 Dcache Store Miss, which indicates that compared with GEMM, our approach can better improve the data locality. However, ABM-SpConv-SIMD introduces deeper loops and more conditional judgments since we need to traverse the two arrays for each encoded filter to load all the valid weights and process different numbers of valid positions for a valid weight, resulting in a higher branch miss, as is shown in Fig.13.

5.5 Comparison with other work

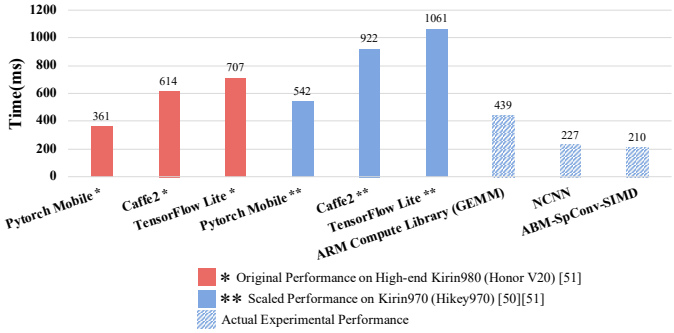


Fig. 14: Performance of ResNet50 with several framework.

We compare the inference latency of ResNet50 based on the ABM-SpConv-SIMD framework against other CNN frameworks developed for edge mobile devices. Fig.14 shows the performance of several frameworks. For ARM Compute Library (GEMM), NCNN [48] and ABM-SpConv-SIMD, we conduct actual experiments on our platform Hikey970. For Pytorch Mobile [15], Caffe2 [49] and Tensorflow Lite [14], the inference time are taken from other resources [51]. Then the borrowed experimental results are scaled approximately to compensate for differences in platforms by using AI-benchmark [50]. AI-benchmark [50] presents the real-world performance results on various

mobile SoCs, which cover all main existing deep learning frameworks, AI models, and hardware configurations. Furthermore, AI-benchmark scores each mobile SoC to characterize their AI performance, known as *AI_score*, which is the reference we chose to compensate for performance differences between hardware platforms. This model performance scaling approach is also adapted in another work called *Pipe-it* [52], where it can be proven to be effective. In summary, as shown in Fig.14, ABM-SpConv-SIMD can provide the highest performance among all the frameworks.

6 Related Work

Application of CNNs in IIoT fields. Many studies have recently proposed deploying CNNs in IIoT applications based on its powerful feature extraction ability. In the field of intelligent industrial manufacturing, Yuanbin Wang et al. proposed a CNN-based visual sorting system for accurate part model classification in flexible manufacturing systems [53]. Fei Wang et al. designed a cascade CNN (C-CNN) with progressive optimization for motor fault diagnosis [54]. Tanveer Hussain et al. presented a lightweight CNN and IIoT-based computationally intelligent multiview video summarization system [55], which can be used for security monitoring and intelligent transportation. In addition, a CNN-based camera management system is designed to work as a substitute for the academic filming crew in online classes or examinations [56]. Mingdong Zhang et al. designed a residual-based COVID-19 detection network [57], which can efficiently extract the lung features through small COVID-19 samples and remove the pretraining requirement on other medical datasets.

Optimizations of CNN inference on edge devices. CNN inference optimization on edge devices has become a rich research area. Some works focus on compressing the CNN model and reducing computations to enable CNNs on edge devices. Han et al. proposed a model compression method including pruning, trained quantization and huffman coding, which can effectively remove redundant parameters [45]. Based on the sparsity feature of CNN after pruning, some studies propose to trim out redundant computations. SparCE [58] proposed to extend the CPU's five-level pipeline mechanism and then skip redundant instruction sequences with operand zero (sparsity) that do not contribute to the final result, thus obtaining performance gains. UCNN [59] is designed to exploit computation reuse with the same weights and further improve performance by exploiting sparsity in weights, as reducing computation due to repeated zero weights is a special case of reducing computation due to repeated weights. Unfortunately, the hardware extensions are costly and not scalable.

In addition to exploiting the sparsity of CNNs to improve performance, some studies accelerate the on-device computation by fully utilizing on-ship hardware resources, such as SIMD units and GPUs. A robust work proposed by Yizhi Liu employed a full-stack and systematic scheme of optimizations, including single-thread optimization of the convolutional layers using SIMD instructions [60]. FeatherCNN [61] designed a highly efficient generalized matrix multiplication routine based on SIMD instructions to accelerate Winograd convolution on ARM CPUs. In addition to using SIMD units, DeepX [62] and uLayer [63] proposed to decompose deep model network architectures into unit-blocks of various types and deploy them on heterogeneous local device processors (e.g., CPUs, GPUs). DeepMon [64] proposed to use various optimization techniques including the convolutional layer caching, decomposition, and matrix multiplication optimizations to efficiently offload convolutional layers to mobile GPUs and accelerate the processing. Unfortunately, hardware accelerators like GPUs are not common and energy-efficient, and the software approaches mentioned above can not reduce the number of operations, so the performance improvement is limited.

7 Conclusion and Future Work

Executing CNN's inference tasks for many IIoT applications on local edge devices can provide real-time performance and protect users' privacy. Resource-constrained edge devices must fully utilize their hardware resources to accelerate CNNs inference. In this paper, we propose ABM-SpConv-SIMD, a low latency on-device inference runtime optimization framework. ABM-SpConv-SIMD first adopts offline model pruning and quantization approaches to optimize models and then employs a series of runtime optimizations including multiplication operation reduction, data layout optimization for vectorization, instruction parallelism by SIMD, and fix-point integers to fully utilize the commonly available and cost-effective CPU SIMD architectures to accelerating CNN inference. The experimental results show that ABM-SpConv-SIMD can achieve 1.96x (high-end) and 1.73x (low-end) performance improvement on average and 2.09x (high-end) and 1.88x (low-end) at most when executing inference on two representative edge devices.

There are several interesting directions for future work. First, ABM-SpConv-SIMD considers latency-aware pruning. We wish to design an energy-aware pruning strategy based on energy estimation methodology [65], which considers not only the memory footprint and computation workload but also each layer's energy consumption. For data-driven neural network models, high accuracy, good generalization ability, and good robustness often rely on a large amount of training data, which means generating a more complex, deeper, and larger network model. Knowledge distillation can transfer the multi-domain knowledge learned from a large complex integrated model to a small lightweight model which can be better applied to downstream inference tasks [66]. Therefore, in the future, we consider employing the knowledge distillation method to complete the knowledge migration from more complex models to lightweight models, and then use ABM-SpConv-SIMD framework to complete the model structure simplification, so as to further improve the inference accuracy and reduce the inference time. For example, huge Natural Language Processing (NLP) models such as Bert, GPT, and ELMo can be compressed by distilling the complex transformer structure contained in NLP models into simple and lightweight transformer, or simple LSTM and textCNN model [67], [68], [69], [70], thus facilitating the implementation of some natural language processing applications such as speech recognition, machine translation, and voice recognition on edge IoT devices.

REFERENCES

[1] A. K. Tripathi, K. Sharma, et al., "A Parallel Military-Dog-Based Algorithm for Clustering Big Data in Cognitive Industrial Internet of Things," in *IEEE Transactions on Industrial Informatics*, vol. 17, no. 3, pp. 2134-2142, March. 2021.

[2] A. A.Brincat, F. Pacifici, et al., "The Internet of Things for Intelligent Transportation Systems in Real Smart Cities Scenarios," *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, 2019.

[3] A. Munusamy et al., "Service Deployment Strategy for Predictive Analysis of FinTech IoT Applications in Edge Networks," in *IEEE Internet of Things Journal*, 2021.

[4] K. Wang, X. Liu, et al., "Voice-Transfer Attacking on Industrial Voice Control Systems in 5G-Aided IIoT Domain," in *IEEE Transactions on Industrial Informatics*, vol. 17, no. 10, pp. 7085-7092, Oct. 2021.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, May 2017.

[6] Y. Wu, el al., "Google's neural machine translation system: Bridging the gap between human and machine translation,", 2016.

[7] V. Bianchi, M. Bassoli, et al., "IoT Wearable Sensor and Deep Learning: An Integrated Approach for Personalized Human Activity Recognition in a Smart Home Environment," in *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8553-8562, Oct. 2019.

[8] P. J.Kang, et al., "Machine learning-based imaging system for surface defect inspection." *International Journal of Precision Engineering and Manufacturing-Green Technology*, 2016.

[9] J. A. J. A. Geetha, M. Aravindan, "Cnn based fall detection and health monitoring system using iot," *International Journal of Advanced Science and Technology*, vol. 29, no. 3, Apr. 2020.

[10] G. Hussain, M. K. Maheshwari, et al., "A cnn based automated activity and food recognition using wearable sensor for preventive healthcare," *Electronics*, vol. 8, no. 12, 2019.

[11] R. Taheri, M. Shojafar, et al., "Fed-IIoT: A Robust Federated Malware Detection Architecture in Industrial IoT," in *IEEE Transactions on Industrial Informatics*, vol. 17, no. 12, pp. 8442-8452, Dec. 2021.

[12] M. Abbasi, A. Shokrollahi, et al., "High-performance flow classification using hybrid clusters in software defined mobile edge computing," *Computer Communications*, vol. 160, 2020.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," *Commun. ACM*, vol. 58, no. 1, p. 105–115, Dec. 2014. [Online].

[14] "TensorFlow Lite: An open source deep learning framework for on-device inference", https://www.tensorflow.org/lite.

[15] "Pytorch Mobile: End-to-end workflow from Training to Deployment for iOS and Android mobile devices", https://pytorch.org/mobile/home/.

[16] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar and L. Fei-Fei, "Large-Scale Video Classification with Convolutional Neural Networks," *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725-1732.

[17] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *2nd International Conference on Learning Representations, ICLR 2014*, 2014.

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[21] T.-J. Yang, et al., "Designing energy-efficient convolutional neural networks using energy-aware pruning," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[22] R.H. Gong, X.L. Liu, S.H. Jiang, et al., "Differentiable soft quantization: Bridging full-precision and low-bit neural networks", *Proceedings of the IEEE International Conference on Computer Vision*, pages=4852–4861, year=2019.

[23] C. Leng, Z.S. Dou, H. Li, et al., "Extremely Low Bit Neural Network: Squeeze the Last Bit Out With ADMM", *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, 2018.

[24] P. Gysel, J. Pimentel,et al., "Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5784-5789, Nov. 2018.

[25] T. N. Sainath, B. Kingsbury, et al., "Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets," *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[26] J. Kim, S. Park and N. Kwak, "Paraphrasing Complex Network: Network Compression via Factor Transfer", *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018*, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. pages = 2765–2774, year = 2018.

[27] Y. Li, et al., "Revisiting Knowledge Distillation via Label Smoothing Regularization", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages=3903–3911, year=2020.

[28] Y.W. Guo, A.B. Yao, et al., "Dynamic network surgery for efficient dnns", *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016*, December 5-10, 2016, Barcelona, Spain, pages= 1379–1387, year=2016.

[29] M.B. Lin, R.H. Ji, et al., "HRank: Filter Pruning using High-Rank Feature Map", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages=1529–1538, year=2020.

[30] H. Mao, S. Han, et al., "Exploring the regularity of sparse structure in convolutional neural networks," in *NIPS*, 2017, pp. 1–10.

[31] Y. He, J. Lin, et al., "Amc: Automl for model compression and acceleration on mobile devices," *ECCV*, 2018.

[32] Z. Liu, M. Sun, et al., "Rethinking the value of network pruning," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[33] Z.C. Liu, H.Y. Mu, et al., "MetaPruning: Meta Learning for Automatic Neural Network Channel Pruning", 3295-3304.

[34] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *2015 Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, 2015.

[35] A. Ren, T. Zhang, et al., "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, ASPLOS*, 2019.

[36] H. Mostafa and X. Wang, "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization," in *Proceedings of the 36th International Conference on Machine Learning, ICML* 2019.

[37] X. Ding, G. Ding, et al., "Global sparse momentum SGD for pruning very deep neural networks," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2019.

[38] S. Han, W. Dally, et al., "Invited: Bandwidth-efficient deep learning," *DAC*, 2018.

[39] W. Wen, C. Wu, et al., "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems*, 2016.

[40] A. Parashar, M. Rhu, et al., "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA*, 2017.

[41] L. Lu, J. Xie, et al., "An efficient hardware accelerator for sparse convolutional neural networks on fpgas," in *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM*, 2019.

[42] D. Wang, K. Xu, et al., "ABM-SpConv: A novel approach to fpga-based acceleration of convolutional neural network inference," in *Proceedings of the 56th Annual Design Automation Conference*, 2019.

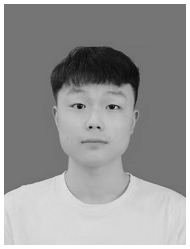
[43] C. Lomont, "Introduction to Intel Advanced Vector Extensions," *Intel White Paper*, 2011.

[44] ARM, "Architecture support for NEON and VFP", https://developer.arm.com/architectures/instruction-sets/simd-isas/neon.

[45] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016.

[46] J. Qiu, J. Wang, S. Yao and et al., "Going deeper with embedded FPGA platform for convolutional neural network," *in Proc.*

ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16), 2016.
[47] "The ARM Computer Vision and Machine Learning library", https://github.com/ARM-software/ComputeLibrary.
[48] "Tencent NCNN: a high-performance neural network inference computing framework optimized for mobile platforms", https://github.com/Tencent/ncnn.
[49] "Caffe2: A New Lightweight, Modular, and Scalable Deep Learning Framework", https://caffe2.ai/.
[50] A. Ignatov et al., "AI benchmark: Running deep neural networks on Android smartphones," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 288–314.
[51] Chunjie. Luo, Xiwen. He, et al., "Comparison and Benchmarking of AI Models and Frameworks on Mobile Devices," arXiv preprint arXiv:2005.05085, 2020.
[52] S. Wang, G. Ananthanarayanan, et al., "High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2254-2267, Oct. 2020.
[53] Y. Wang, K. Hong, et al., "A CNN-Based Visual Sorting System With Cloud-Edge Computing for Flexible Manufacturing Systems," in *IEEE Transactions on Industrial Informatics*, vol. 16, no. 7, pp. 4726-4735, 2020.
[54] F. Wang, R. Liu, et al., "Cascade Convolutional Neural Network With Progressive Optimization for Motor Fault Diagnosis Under Nonstationary Conditions," in IEEE Transactions on Industrial Informatics, vol. 17, no. 4, pp. 2511-2521, 2021.
[55] T. Hussain, K. Muhammad, et al., "Intelligent Embedded Vision for Summarization of Multiview Videos in IIoT," in *IEEE Transactions on Industrial Informatics*, vol. 16, no. 4, pp. 2592-2602, 2020.
[56] M. S. Haghighi, A. Sheikhjafari, et al., "Automation of Recording in Smart Classrooms via Deep Learning and Bayesian Maximum a Posteriori Estimation of Instructor's Pose," in *IEEE Transactions on Industrial Informatics*, vol. 17, no. 4, pp. 2813-2820, 2021.
[57] M. Zhang, R. Chu, et al., "Residual Learning Diagnosis Detection: An Advanced Residual Learning Diagnosis Detection System for COVID-19 in Industrial Internet of Things," in *IEEE Transactions on Industrial Informatics*, vol. 17, no. 9, pp. 6510-6518, 2021.
[58] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "Sparce: Sparsity aware general-purpose core extensions to accelerate deep neural networks," *IEEE Trans. Comput.*, vol. 68, no. 6, 2019.
[59] K. Hegde, J. Yu, et al., "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.
[60] Y. Liu, Y. Wang, et al., "Optimizing cnn model inference on cpus," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC, 2019.
[61] H. Lan et al., "FeatherCNN: Fast Inference Computation with TensorGEMM on ARM Architectures," in IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 3, pp. 580-594, 2020.
[62] N. D. Lane et al., "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices," *15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2016, pp. 1-12.
[63] Y. Kim, J. Kim, et al., "ulayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," *in Proceedings of the 14 EuroSys Conference*, 2019.
[64] L. N. Huynh, Y. Lee, et al., "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*, ser. MobiSys '17, 2017.
[65] Y. Chen, J. Emer and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367-379.
[66] Geoffrey E. Hinton and Oriol Vinyals and Jeffrey Dean, "Distilling the Knowledge in a Neural Network,", *arXiv preprint arXiv:1503.02531*, 2015.
[67] R. Tang, Y. Lu, et al.,"Distilling task-specific knowledge from bert into simple neural networks,", *arXiv preprint arXiv:1903.12136*, 2019.
[68] S. Mukherjee and A.H. Awadallah, "Distilling BERT into Simple Neural Networks with Unlabeled Transfer Data", *arXiv preprint arXiv:1910.01769*, 2019.
[69] Y. Liu, H. Xiong, et al., "End-to-End Speech Translation with Knowledge Distillation," in *Proc. Interspeech 2019*, 2019.
[70] X. Jiao, Y. Yin, et al. "Tinybert: Distilling BERT for natural language understanding, "*CoRR*, vol. abs/1909.10351, 2019.



Xianduo Li received the BE degree from the College of Computer Science and Technology, China University of Mining and Technology, Xuzhou, China, in 2020. He is currently working toward the MS degree at the College of Computer Science, Nankai University. His research interests include heterogeneous computing and machine learning algorithm optimization.

Xiaoli Gong (Member IEEE) received the PhD degree from Nankai University. He is currently an associate professor with Nankai University. He is a member of ACM and China Computer Federation. His main research interests include system virtualization, embedded system design, and optimization.

Dong Wang (Member, IEEE) received the M.S. and Ph.D. degrees in electronic engineering from Xi'an Jiaotong University, Xi'an, China, in 2006 and 2010, respectively. He is an Associate Professor with the Institute of Information Science, Beijing Jiaotong University, Beijing, China. He was a Visiting Scholar with the Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA, USA, from 2018 to 2019. His research interests include computer arithmetic for reconfigurable devices and high performance and energy efficient computing architectures for embedded and machine learning applications.

Huayou Su was born in 1985. He is a Ph.D. recipient, an associate professor at College of Computer, National University of Defense and Technology. He is a member of China Computer Federation. His research interests include high performance computing, parallel computing, and CPU-GPU hybrid computing.

Jin Zhang (Member, IEEE) received the PhD degree from Nankai University. He is an associate professor with Nankai University. He is a member of China Computer Federation. His main research interests include mobile cloud computing.

Thar Baker (Senior Member, IEEE) received the Ph.D. degree in autonomic cloud applications from Liverpool John Moores University (LJMU), U.K., in 2010 and became a Senior Fellow of Higher Education Academy in 2018. He is an Associate Professor with the Department of Computer Science, University of Sharjah (UoS), UAE. Before joining UoS, he was a Reader of Cloud Engineering and the Head of the Applied Computing Research Group, Faculty of Engineering and Technology, LJMU. Prior to that, he was a Lecturer of Computer Science with the Department of Computing and Mathematics, Manchester Metropolitan University, U.K. He has published numerous refereed research papers in multidisciplinary research areas, including parallel and distributed computing, algorithm design, green and sustainable computing, and energy routing protocols.