

Algorithm-architecture co-design for CNN acceleration exploiting linear association of weights

Xing Lidong (✉), Wang Yuan, Feng Zhenfu

School of Electronic Engineering, Xi'an University of Posts and Telecommunications, Xi'an 710121, China

Abstract

To deploy convolutional neural networks (CNNs) on resource-constrained edge devices and accelerate inference, a hardware-aware CNN optimization algorithm named linear association of weights (LAW) was proposed in this paper. The algorithm reconstructs integer dot-product computations in quantized models, significantly reducing multiplication operations while maintaining accuracy. The implementation works at two granularities. At the single-kernel level, it transforms multiplications into efficient shift-and-add operations by exploiting linear relationships among weights. At the multi-channel level, a cross-channel weight association computation model ensures algorithm generality across different convolutional layers. Based on LAW, a configurable CNN hardware accelerator is designed, featuring an acceleration core that decodes weight association patterns and replaces conventional multipliers with shift-and-add combinations for hardware-level optimization. Experimental results demonstrate that compared with traditional 8-bit integer (int8) quantization, the LAW algorithm reduces 54% of multiplications for 7×7 convolution kernels and achieves 63% multiplication reduction for multi-channel 3×3 convolutional layers. When evaluated with residual network-18 (ResNet-18) on the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) dataset, the proposed accelerator achieves 40.21 ms inference latency per image with 70.16% top-1 accuracy and 90.12 giga operations per second (GOPS) peak computation amount performance. As an innovative algorithm specifically designed for vision tasks, LAW provides effective hardware acceleration support for vision-oriented deep learning networks, making it particularly suitable for unmanned systems and industrial inspection.

Keywords accelerated inference, hardware-aware, linear association, shift-and-add

1 Introduction

In recent years, CNNs have achieved substantial progress in various computer vision and natural language processing tasks^[1]. However, the computational and memory demands of these networks

limit their deployment on edge devices, such as autonomous vehicles, intelligent robots, and unmanned aerial vehicles. Consequently, researchers increasingly focus on deploying deep learning models with real-time performance and high accuracy on hardware platforms^[2–4].

Deploying neural networks on hardware primarily involves optimizing computational resources and memory usage. During inference, CNNs perform extensive multiply-accumulate (MAC) operations^[5].

Complex CNN architectures typically require substantial computational resources, leading to increased energy consumption. Additionally, large models demand significant memory to store parameters and intermediate results, posing challenges for deployment on resource-constrained edge computing platforms, such as field-programmable gate arrays (FPGAs), and embedded systems. Model compression is a key technique for deploying deep learning models on hardware, and weight quantization is particularly favored as a hardware-friendly compression method^[6]. Weight quantization techniques are broadly categorized into two types: post-training quantization (PTQ) and quantization-aware training (QAT). This study focuses on PTQ, which avoids retraining networks by converting floating-point weights to integers through specialized algorithms, enabling lightweight and efficient quantization^[7]. Using PTQ to reduce data precision from 32-bit to 8-bit decreases memory storage by a factor of 4 and reduces computational costs of matrix multiplication by a factor of 16^[8]. Although low-bitwidth quantization introduces noise and may degrade accuracy, 8-bit precision suffices for inference accuracy in most models^[9]. Recent studies, such as hardware-aware automated quantization with mixed precision^[10] and towards mixed-precision quantization of neural networks via constrained optimization^[11], have demonstrated that int8 quantization achieves

accuracy levels comparable to floating-point models. Quantization primarily optimizes numerical representations of model parameters rather than directly addressing computational inefficiencies during inference, thereby limiting the full utilization of hardware resources. Specific limitations include two issues. First, current quantization methods focus on converting floating-point to integer data. While integer MAC operations offer significant energy efficiency advantages over floating-point operations in hardware implementations, they fail to adequately address the resource overhead caused by large-scale integer computations. For example, the residual network-50 (ResNet-50) architecture contains numerous convolutional layers, kernels, and channels, requiring massive integer multiplications during hardware inference, which consume substantial resources and hinder speed improvements. Second, during PTQ (int 8), weights are constrained to the range of $[-128, 127]$. As shown in Fig. 1, where the horizontal axis represents weight values, and the vertical axis represents the number of occurrences for each value, the experimental data reveal that after absolute value processing, convolutional layers exhibit high-frequency weight repetition or approximation. Despite such redundancy, conventional implementations still perform dot-product operations on all weights, leading to redundant computations.

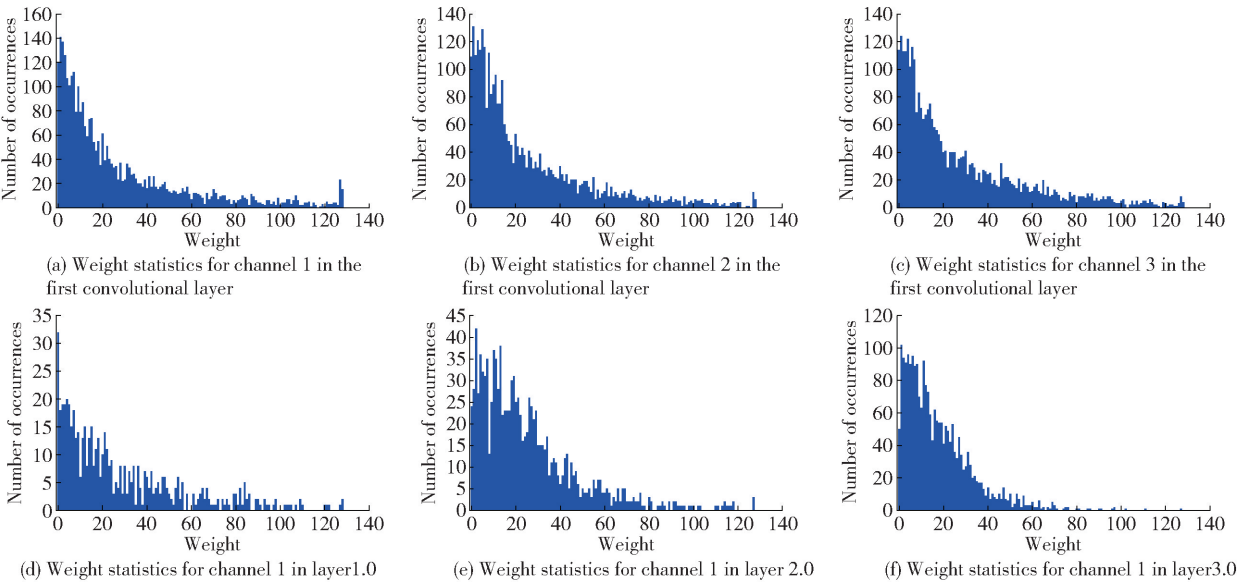


Fig. 1 Weight distribution after quantization of ResNet-50 network

A common hardware approach to accelerate neural network inference involves increasing the parallelism of convolutional operators in an FPGA^[5]. These operators typically comprise multiple processing engines, each containing parallel multipliers to compute multiple multiplications simultaneously. Although enhanced parallelism improves inference speed, it does not reduce the total number of multiplication operations in CNNs. To address these challenges, the LAW optimization algorithm was proposed in this paper, which effectively reduces multiplication counts. By analyzing weight associations during hardware deployment, LAW provides a novel pathway for accelerating neural network inference. The main contributions of this study are as follows.

1) A linear association analysis algorithm for weights is proposed. This algorithm performs linear association analysis on PTQ-quantized weights, integrates correlated weight terms, and replaces multiplications with shift-and-add operations, thereby accelerating inference.

2) A configurable hardware acceleration architecture based on the above optimization algorithm is designed. While retaining multiply-accumulate units, the architecture expands computational modules with shift-and-add operations, enhancing adaptability and optimization capabilities for diverse deep learning networks.

2 Linear association analysis of weight

To further enhance inference acceleration beyond quantization, the LAW algorithm within the PTQ framework is introduced with a focus on optimizing convolutional operations. By analyzing linear and proportional relationships among convolutional layer weights, LAW reduces the number of multiplications in convolution, thereby conserving hardware resources and improving inference speed without compromising quantization accuracy. This section investigates intra-kernel and cross-kernel weight associations, derives corresponding optimization algorithms capitalizing on these relational properties, and formulates the associated algorithmic model.

2.1 Intra-kernel association analysis

Mathematically, if the i th weight equals the j th weight (i. e. $w_i^p = w_j^p$), the convolution operation $w_i^p * x_i + w_j^p * x_j$, where x_i and x_j represent the activation values, can be rewritten as $w_i^p * (x_i + x_j)$, eliminating one multiplication in hardware. Similarly, when $2w_i^p = w_j^p$, a left-shift operation ($\ll 1$) can replace the multiplication by 2. Considering the characteristics of shift operations, weight distribution patterns, and hardware feasibility, the maximum shift count is limited to 3 in this work. Table 1 details the multiplicative relationships and their corresponding replacement operations.

Table 1 Analysis of proportional relationships in weight

Number of operations	Condition	Output
1 * ,1 +	$w_i^p = w_j^p$	$w_i^p (x_i + x_j)$
1 * ,1 +	$2w_i^p = w_j^p$	$w_i^p (x_i + (x_j \ll 1))$
1 * ,2 +	$3w_i^p = w_j^p$	$w_i^p (x_i + x_j + (x_j \ll 1))$
1 * ,1 +	$4w_i^p = w_j^p$	$w_i^p (x_i + (x_j \ll 2))$
1 * ,2 +	$5w_i^p = w_j^p$	$w_i^p (x_i + x_j + (x_j \ll 2))$
1 * ,2 +	$6w_i^p = w_j^p$	$w_i^p (x_i + (x_j \ll 1) + (x_j \ll 2))$
1 * ,2 +	$7w_i^p = w_j^p$	$w_i^p (x_i - x_j + (x_j \ll 3))$
1 * ,1 +	$8w_i^p = w_j^p$	$w_i^p (x_i + (x_j \ll 3))$
1 * ,2 +	$9w_i^p = w_j^p$	$w_i^p (x_i + x_j + (x_j \ll 3))$

For instance, if weights within a kernel exhibit relationships listed in Table 1, the “output” column specifies the shift-and-add operations to replace multiplications. These relationships are formalized into

$$w_i^p \left(x_i + \sum_{j \neq i} (x_j \ll m) \pm \sum_{w_j^p \in \{(2^m \pm 1) w_i^p\}} x_j \right) \quad (1)$$

where p denotes the convolution kernel index, w_i represents the weight value, x_i and x_j are the activation values corresponding to w_i^p and w_j^p , respectively, w_j^p is the weight satisfying a multiplicative relationship with w_i^p , and m indicates the number of shift bits determined by the multiplicative factor. In the LAW algorithm, the larger the number of w_j^p terms, the fewer the multiplication operations saved in hardware.

Inspired by multiplicative relationships, shift-and-add operations can also simplify computations when

weights exhibit numerical proximity. For instance, if $w_i^p + 8 = w_j^p$, the operation $w_i^p * x_i + w_j^p * x_j$ can be replaced with $w_i^p (x_i + x_j) + (w_j^p \ll 3)$, where $\ll 3$ denotes a 3-bit left shift. Again, with a maximum shift of 3 bit as the standard, the analysis of linear relationships is shown in Table 2.

Table 2 Analysis of linear relationships in weight

Number of operations	Condition	Output
$1 * , 2 +$	$w_j^p = w_i^p + 1$	$w_i^p (x_i + x_j) + x_j$
$1 * , 2 +$	$w_j^p = w_i^p + 2$	$w_i^p (x_i + x_j) + (x_j \ll 1)$
$1 * , 2 +$	$w_j^p = w_i^p + 4$	$w_i^p (x_i + x_j) + (x_j \ll 2)$
$1 * , 2 +$	$w_j^p = w_i^p + 8$	$w_i^p (x_i + x_j) + (x_j \ll 3)$

Based on Table 2, these relationships can be generalized into

$$w_i^p \left(x_i + \sum_{j \neq i} x_j \right) + \sum_{j \neq i} (x_j \ll m) \quad (2)$$

When weight values exhibit a clustered distribution, linear associations among weights become more pronounced, enabling effective hardware optimization through the LAW algorithm.

Further analysis of convolutional kernels reveals additional additive relationships beyond those listed in Table 2. Moreover, while the optimizations in Tables 1 and 2 focus on pairwise associations between weights w_i^p and w_j^p , extending the association analysis to triplets (w_i^p, w_j^p, w_k^p) , $i \neq j \neq k$, can further simplify computations. For example, $w_j^p = (w_i^p + 0) \&\& w_k^p = (w_j^p + 1)$, the original operation $w_i^p * x_i + w_j^p * x_j + w_k^p * x_k$ can be replaced with $w_i^p * (x_i + x_j + x_k) + x_k$. Under the constraints of a maximum shift count of 3 bit and a total multiply-add operation limit of 5, these extended relationships are systematically categorized in Table 3. The generalized optimization rules derived from Table 3 are formalized as

$$\left. \begin{aligned} &w_i^p (x_i + x_j + x_k) + \sum (x_k \ll m); \\ &\quad w_j^p = w_i^p \neq w_k^p \\ &w_i^p (x_i + x_j + x_k) + \sum (x_j \ll m); \\ &\quad w_k^p = w_j^p \neq w_i^p \\ &w_i^p (x_i + x_j + x_k) + \sum (x_j \ll m) + \\ &\quad \sum (x_k \ll m); \quad w_j^p \neq w_i^p \neq w_k^p \end{aligned} \right\} \quad (3)$$

Table 3 Analysis of multiple weight relationship

Number of operations	Condition	Output
$1 * , 3 +$	$w_j^p = (w_i^p + 0) \&\&$ $w_k^p = (w_j^p + 1)$	$w_i^p (x_i + x_j + x_k) + x_k$
$1 * , 3 +$	$w_j^p = (w_i^p + 0) \&\&$ $w_k^p = (w_j^p + 2)$	$w_i^p (x_i + x_j + x_k) + (x_k \ll 1)$
$1 * , 3 +$	$w_j^p = (w_i^p + 0) \&\&$ $w_k^p = (w_j^p + 4)$	$w_i^p (x_i + x_j + x_k) + (x_k \ll 2)$
$1 * , 3 +$	$w_j^p = (w_i^p + 0) \&\&$ $w_k^p = (w_j^p + 8)$	$w_i^p (x_i + x_j + x_k) + (x_k \ll 3)$
$1 * , 3 +$	$w_j^p = (w_i^p + 1) \&\&$ $w_k^p = (w_j^p + 0)$	$w_i^p (x_i + x_j + x_k) + x_j$
$1 * , 3 +$	$w_j^p = (w_i^p + 2) \&\&$ $w_k^p = (w_j^p + 0)$	$w_i^p (x_i + x_j + x_k) + (x_j \ll 1)$
$1 * , 3 +$	$w_j^p = (w_i^p + 4) \&\&$ $w_k^p = (w_j^p + 0)$	$w_i^p (x_i + x_j + x_k) + (x_j \ll 2)$
$1 * , 3 +$	$w_j^p = (w_i^p + 8) \&\&$ $w_k^p = (w_j^p + 0)$	$w_i^p (x_i + x_j + x_k) + (x_j \ll 3)$
$1 * , 4 +$	$w_j^p = (w_i^p + 1) \&\&$ $w_k^p = (w_j^p + 1)$	$w_i^p (x_i + x_j + x_k) + x_j + (x_k \ll 1)$
$1 * , 4 +$	$w_j^p = (w_i^p + 2) \&\&$ $w_k^p = (w_j^p + 2)$	$w_i^p (x_i + x_j + x_k) + (x_j \ll 1) + (x_k \ll 2)$
$1 * , 4 +$	$w_j^p = (w_i^p + 4) \&\&$ $w_k^p = (w_j^p + 4)$	$w_i^p (x_i + x_j + x_k) + (x_j \ll 2) + (x_k \ll 3)$
$1 * , 4 +$	$w_j^p = (w_i^p + 1) \&\&$ $w_k^p = (w_j^p + 3)$	$w_i^p (x_i + x_j + x_k) + x_j + (x_k \ll 2)$
$1 * , 4 +$	$w_j^p = (w_i^p + 1) \&\&$ $w_k^p = (w_j^p + 7)$	$w_i^p (x_i + x_j + x_k) + x_j + (x_k \ll 3)$

2.2 Cross-kernel association analysis

In CNNs, multi-channel convolution is widely adopted to process multidimensional feature data. Unlike single-kernel convolution, multi-channel convolution employs multiple kernels corresponding to the input feature map's channel count. First, a concise description of the multi-channel convolution workflow is provided. In this operation, input data typically has multiple channels (e. g., RGB images with red, green, and blue channels), and convolutional kernels are correspondingly multi-channel to match the input. As illustrated in Fig.2(a), the input is a three-dimensional (3D) tensor with dimensions $(2, 4, 4)$,

where 2 denotes the input channels, and 4 represents the spatial dimensions. The convolutional kernel, also a 3D tensor, has dimensions $(3, 2, 3)$. The first dimension 3 represents the number of output channels, the second dimension 2 corresponds to the number of input channels (which must match the number of channels in the input data), and the last dimension 3 indicates the kernel size.

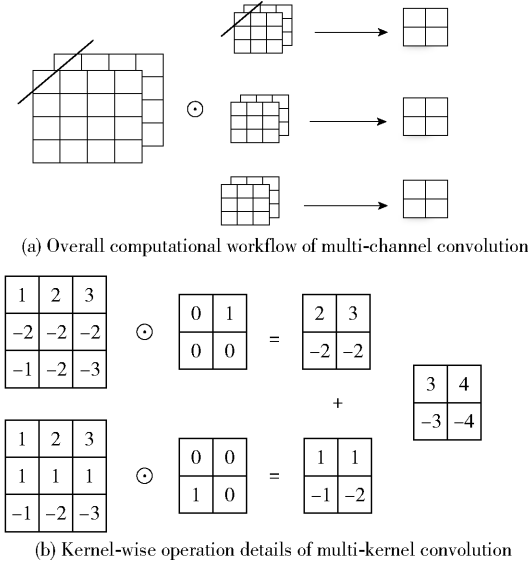


Fig. 2 Illustration of multi-channel convolution

The computation involves channel-wise convolution followed by summation, shown as

$$O = \sum_{n_i=0}^{N_i-1} \sum_{k_1=0}^{K-1} \sum_{k_2=0}^{K-1} I_{n_i, k_1, k_2} w_{n_o, n_i, k_1, k_2} \quad (4)$$

where O denotes the output feature map, I_{n_i, k_1, k_2} denotes the input feature map, w_{n_o, n_i, k_1, k_2} denotes the convolutional kernel weight, n_o denotes the index of output channels, n_i denotes the index of input channels, and k_1, k_2 denote kernel spatial positions. Detailed computational steps are depicted in Fig. 2(b).

Building on this, this work investigates associations among weights across different kernels within the same output channel to further optimize computations. Let w_i^p represent the weight at position i of the p th kernel. If $w_i^1 = w_i^2$, the operation $w_i^1 * x_i^1 + x_i^2 * w_i^2$ can be simplified to $w_i^1 * (x_i^1 + x_i^2)$, consistent with the intra-kernel optimization principles in Tables 1 and 2. However, cross-kernel associations are generally weaker than intra-kernel associations. To address this,

while retaining the rules in Table 1, Table 2 is extended to incorporate cross-kernel relationships, as summarized in Table 4. For practicality, the shift count remains capped at 3 bit, and cross-kernel analysis is restricted to pairwise associations to minimize complexity.

Table 4 Analysis of inter-kernel weight relationships

Number of operations	Condition	Output
$1 * , 1 +$	$w_i^2 = w_i^1$	$w_i^1 (x_i^1 + x_i^2)$
$1 * , 1 +$	$2w_i^2 = w_i^1$	$w_i^1 (x_i^1 + (x_i^2 \ll 1))$
$1 * , 2 +$	$3w_i^2 = w_i^1$	$w_i^1 (x_i^1 + x_i^2 + (x_i^2 \ll 1))$
$1 * , 1 +$	$4w_i^2 = w_i^1$	$w_i^1 (x_i^1 + (x_i^2 \ll 2))$
$1 * , 2 +$	$5w_i^2 = w_i^1$	$w_i^1 (x_i^1 + x_i^2 + (x_i^2 \ll 2))$
$1 * , 2 +$	$6w_i^2 = w_i^1$	$w_i^1 (x_i^1 + (x_i^2 \ll 1) + (x_i^2 \ll 2))$
$1 * , 2 +$	$7w_i^2 = w_i^1$	$w_i^1 (x_i^1 - x_i^2 + (x_i^2 \ll 3))$
$1 * , 1 +$	$8w_i^2 = w_i^1$	$w_i^1 (x_i^1 + (x_i^2 \ll 3))$
$1 * , 2 +$	$9w_i^2 = w_i^1$	$w_i^1 (x_i^1 + x_i^2 + (x_i^2 \ll 3))$
$1 * , 1 +$	$w_i^2 = (w_i^1 + 1)$	$w_i^1 (x_i^1 + x_i^2) + x_i^2$
$1 * , 1 +$	$w_i^2 = (w_i^1 + 2)$	$w_i^1 (x_i^1 + x_i^2) + (x_i^2 \ll 1)$
$1 * , 2 +$	$w_i^2 = (w_i^1 + 3)$	$w_i^1 (x_i^1 + x_i^2) + (x_i^2 \ll 1) + x_i^2$
$1 * , 1 +$	$w_i^2 = (w_i^1 + 4)$	$w_i^1 (x_i^1 + x_i^2) + (x_i^2 \ll 2)$
$1 * , 2 +$	$w_i^2 = (w_i^1 + 5)$	$w_i^1 (x_i^1 + x_i^2) + (x_i^2 \ll 2) + x_i^2$
$1 * , 2 +$	$w_i^2 = (w_i^1 + 7)$	$w_i^1 (x_i^1 + x_i^2) + (x_i^2 \ll 3) - x_i^2$
$1 * , 1 +$	$w_i^2 = (w_i^1 + 8)$	$w_i^1 (x_i^1 + x_i^2) + (x_i^2 \ll 3)$
$1 * , 2 +$	$w_i^2 = (w_i^1 + 9)$	$w_i^1 (x_i^1 + x_i^2) + (x_i^2 \ll 3) + x_i^2$

From the above analysis, cross-kernel linear association analysis necessitates data reorganization for both weights and input features. The weight rearrangement process is illustrated in Fig. 3(a), where K_{num} represents the number of convolution kernels, h and w represent the height and width of the weight feature map, respectively. Consider an output channel containing four convolutional kernels, each with a 3×3 spatial size. Weights at identical positions across the four kernels are extracted and grouped into one-dimensional (1D) vectors of length 4. Through this approach, the four 3×3 kernels are transformed into nine 1×4 vectors, each corresponding to a specific spatial position within the kernels. Input feature map reorganization follows the sliding window

mechanism, as shown by the orange matrix in Fig.3(b), where H_i denotes the height of the input feature map, W_i denotes the width of the input feature map, and N denotes the size of the output feature map. All sliding window data (matching the kernel size) are extracted simultaneously. These patches are then rearranged analogously to the weight reorganization method.

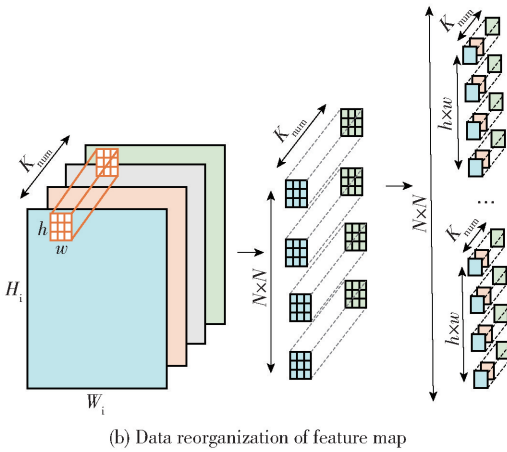
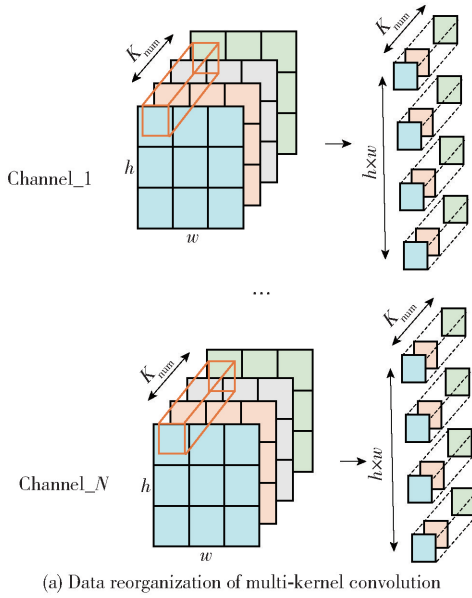


Fig.3 Schematic diagram of multi-channel convolution data reorganization

Fig. 4 demonstrates the computation process after reorganization. Feature values are extracted in sliding window units, rearranged as Fig. 3 (b), and paired with corresponding reorganized weights for multiply-add operations. The first value in the result matrix (highlighted in red) is computed by multiplying and

accumulating the first reorganized weight vector with the first input patch. This process iterates until all sliding windows are processed, emulating channel-wise convolution followed by summation while maintaining mathematical equivalence to traditional multi-channel convolution. Crucially, this reorganization only alters the data arrangement, not the computational outcome.

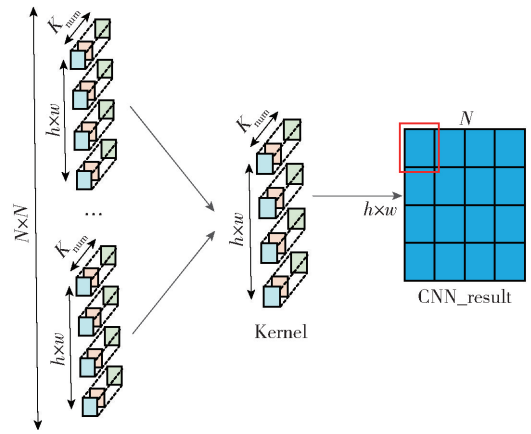


Fig.4 Schematic diagram of post-reorganization computation

2.3 Algorithm implementation

Based on the analyses in Sects. 2.1 and 2.2, the intra-kernel association algorithm operates according to the conditions defined in Tables 1 – 3, with detailed steps outlined in Algorithm 1. The algorithm takes all weights within a single convolutional kernel as input. These weights are stored in a list, and each weight is iteratively checked against the conditions in Tables 1 – 3. Weights satisfying any condition are flagged with their positions and association type, while the total count of matched weights is recorded. After evaluating all conditions, the algorithm selects the condition with the highest matched weight count, consolidates the flagged weights and their metadata into a data packet, and removes them from the list. This process repeats until no remaining weights meet any association condition. The output is a list of data packets, as illustrated in Fig. 5. Each packet contains three key components: base weight value (e.g., w_i^p); positions of correlated weights within the kernel (e.g., w_j^p); association coefficient (e.g., $2x_i$ for $w_j^p = 2w_i^p$).

Algorithm 1 Association statistic**Input:** Single convolutional kernel**Output:** Marked weights within the kernel and statistical metric C while each kernel p dostore the elements in the convolution kernel in order in a list (N_k is the number of elements in the convolution kernel).

Step 1: Clear all existing markers; Count 1 = 0;

for $i = 1$ to N_k dofor $j = i + 1$ to N_k doIf (w_i^p, w_j^p satisfy condition in Table 1 & (w_i^p, w_j^p)! = marked) thenMark w_j^p , coefficient, position;

Count 1 = Count 1 + 1;

end if

end for

end for

Step 2: Clear all checked;

Count 2 = 0, Count 3 = 0;

for $i = 1$ to N_k dofor $j = i + 1$ to N_k dofor $k = j + 1$ to N_k doIf (w_j^p, w_i^p, w_k^p satisfy condition in Table 3 & (w_i^p, w_j^p, w_k^p)! = marked) thenMark w_j^p, w_k^p , coefficient,

Count 3 = Count 3 + 1;

else if (w_j^p, w_i^p satisfy condition in Table 3 & (w_i^p, w_j^p)! = marked)Mark w_j^p , coefficient,

Count 2 = Count 2 + 1;

end if

end for

end for

end for

 $C = \max(\text{Count 1}, \text{Count 2}, \text{Count 3})$, removes the marked weight from the list. Exit the outermost loop when none of the elements in the list satisfy the condition.

done

at identical spatial positions across multiple kernels within the same output channel (as indicated by the dashed boxes in Fig. 3(a)). For instance, if an output channel contains 64 convolutional kernels of size 3×3 , the weights at each spatial position across all 64 kernels are grouped into a 64-weight set (equivalent to an 8×8 kernel matrix), yielding 9 independent groups (one group per spatial position in the 3×3 kernel). These groups are then sequentially analyzed for associations. The implementation details closely follow step 1 of Algorithm 1, with the key distinction being that the input is replaced by cross-kernel weight groups at identical spatial positions, rather than single-kernel weights.

Since the association algorithms alter the computational flow of convolution, weight data must be re-encoded to align with the optimization requirements. The specific re-encoding methodology and implementation will be thoroughly detailed in the subsequent chapter.

3 Hardware design of the LAW algorithm

This section details the hardware deployment strategy for the LAW algorithm and proposes LAW-core, a dedicated accelerator for convolutional neural networks.

3.1 Weight data structure design

The LAW algorithm modifies the computational paradigm of convolution, necessitating a redesigned input data structure. As shown in Fig. 6, a flag field is introduced during weight encoding to indicate the type of optimization algorithm selected. The basic_weight field stores the reference weight value used for association analysis with other weights. Notably, the width of the basic_weight field is adjustable based on the quantization precision. In this work, LAW is applied to 8-bit quantized networks, resulting in an 8-bit basic_weight. The encoding also includes a position field, which stores the spatial positions of weights correlated with the basic_weight. During optimized computation, these positions index corresponding activations in the feature map. Since LAW operates on

Association coefficient	w_i^p	Position
-------------------------	---------	----------

Fig. 5 Schematic diagram of data packet composition

The cross-kernel association algorithm, governed by the conditions in Table 4, focuses on analyzing weights

absolute weight values, the sign bit of each weight is stored in the most significant bit (MSB) of the position field, with the remaining bits encoding positional information.

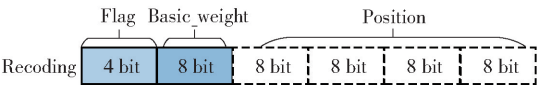


Fig. 6 Composition of re-encoded weight

The distribution patterns of quantized convolutional weights, which vary across different models, are primarily influenced by kernel size, stride, and network architecture. For instance, larger kernels (e.g., 11×11 in Alex convolutional neural network (AlexNet)) often exhibit stronger spatial associations. Consequently, a fixed encoding scheme may fail to fully exploit the LAW algorithm’s potential across diverse models. To address this, an adaptive encoding strategy is adopted: for networks with large kernels (e.g., AlexNet), more position bits are allocated to accommodate extensive spatial indexing (see dashed box in Fig. 6). For networks with small kernels (e.g., Visul Geometry Group 16 (VGG16)), fewer position bits suffice, reducing encoding overhead.

3.2 LAW-core hardware architecture design

The overall architecture of LAW-core is illustrated in Fig. 7, which is mainly composed of a control module, a storage management module, an on-chip storage module, and a computation module.

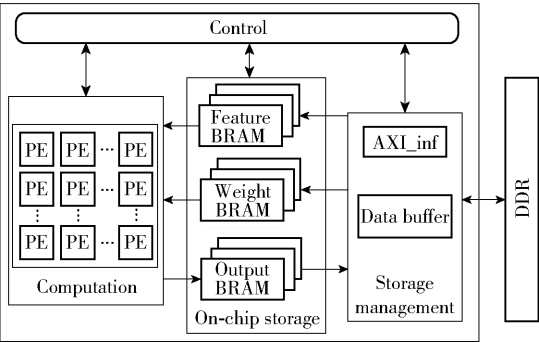


Fig. 7 Overall architecture of LAW-core

The main responsibility of the control module is to control the flow of data and ensure the normal operation of the computing module. Through the advanced extensible interface (AXI) bus, the storage

management module reads the test image data set in the off-chip memory (double data rate (DDR) memory) and the parameters (weight, bias) after the convolutional network quantization into the data buffer, and then writes the intermediate results generated by the operation back to the DDR. The on-chip storage module is responsible for temporarily storing the data in the storage management module into the on-chip memory buffer random access memory (BRAM), and then successively sending the data in the BRAM into the arithmetic module for inference calculation, and finally receiving the execution results of the arithmetic module and writing them back to the BRAM. The computation module mainly performs convolution operation, activation operation, pooling operation, and fully connected operation. In order to reduce the memory access cost, the activation function follows the convolution calculation.

3.2.1 Control module

The control module serves as the central control unit of the accelerator. When weight data and input image data are prepared in the DDR memory, the module is triggered by an enable signal to initiate its workflow. Fig. 8 shows the running logic of the finite state machine (FSM) in the control module.

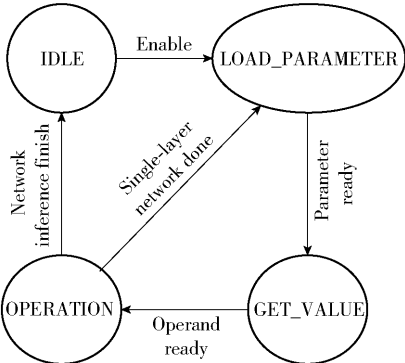


Fig. 8 Controller state transition diagram

In Fig. 8, after circuit reset, it enters the IDLE state. Upon receiving the enable signal, it transitions to the LOAD_PARAMETER state, where it reads reconfigurable parameters from the read-only memory (ROM). These parameters include the initial read/write addresses, padding enable flags (sent to the storage management module), and operational

configurations (e. g., convolution kernel dimensions, number of kernels, computation type) transmitted to the on-chip storage and computation modules. The FSM then proceeds to the GET_VALUE state, awaiting the on-chip buffering of input activations and weights fetched from DDR by the storage management module. Once data is ready, it advances to the OPERATION state. In OPERATION state, the system performs calculations and writes the calculation results back to the storage management module. If inference for the current layer completes, the FSM re-enters the LOAD_PARAMETER state to configure the next layer. If all layers are processed, it returns to the IDLE state.

3.2.2 Storage management module

The storage management module is responsible for data exchange between on-chip and off-chip storage. The on-chip storage is partitioned into feature BRAM, weight BRAM, and output BRAM, while the off-chip storage utilizes DDR memory. Data read/write operations between on-chip BRAM and DDR are coordinated by an arbiter that prioritizes request handling. The implementation of the storage management module is shown in Fig. 9, which mainly consists of an arbiter module, AXI_inf module, data buffer module, and padding module.

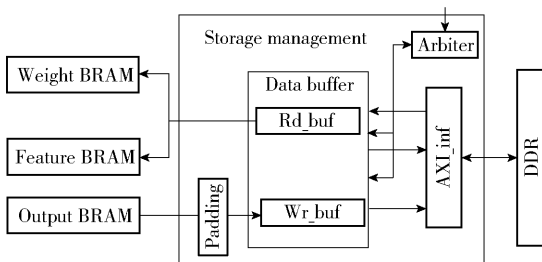


Fig. 9 Storage management module

The AXI_inf module handles bidirectional data transfers between the LAW-core and the DDR, including reading weights/features from DDR and writing processed results back to DDR. The data buffer module manages data caching and synchronization across clock domains, consisting of two sub-modules: a read data buffer (rd_buf) and a write data buffer (wr_buf).

The arbiter module follows a defined priority scheme: upon receiving a read/write request from the

control module, the arbiter responds to the read request preferentially. When the reread request and the eigenvalue read request arrive at the same time, the arbiter responds to the reread request preferentially. The arbiter sends read requests to the rd_buf and write requests to the wr_buf.

Considering the padding operations in convolution and pooling operations, a padding module is added to the storage management module to deal with this requirement.

3.2.3 On-chip storage module

The on-chip storage module handles data exchange between the computation unit and BRAM. It incorporates three dedicated BRAM components: input feature BRAM, weight BRAM, and output BRAM, and implements three independent data paths for feature, weight, and output data transmission, respectively.

3.2.4 Computation module

The computation module mainly consists of convolution unit, rectified linear unit (ReLU), max-pooling unit, and fully-connected unit. As convolutional computation forms the computational backbone of CNN inference, this subsection focuses on the LAW-optimized convolution unit, which replaces conventional multipliers with shift-and-add operations based on weight linear associations.

1) Convolution unit

The convolution unit based on the LAW algorithm, as depicted in Fig. 10, centers around a systolic array architecture. The array consists of multiple processing elements (PEs) configured to execute multiply-accumulate operations, with the PE count adjustable to match network requirements. The unit takes the weight and activation value (act) as inputs, and produces the convolution result (CNN_result) as output. The input weights are split into multiple segments of equal bit-width and propagated vertically among the PEs, while the activations are similarly divided into equal bit-width segments and transmitted horizontally across the PEs. Upon receiving the data, each PE performs computations and passes the results downward to

adjacent PEs. The final convolved results are sequentially output by the bottom-row PEs.

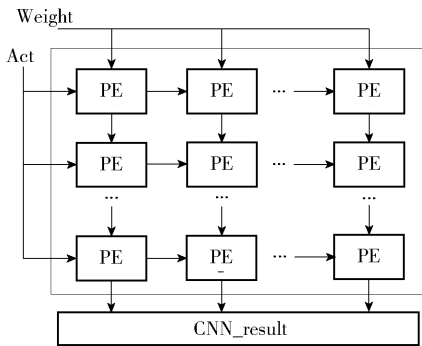


Fig. 10 Convolution operand structure based on LAW

The workflow of the LAW-optimized convolution unit is illustrated in Fig. 11, where x_{ge} denotes the activation value, w_{ab}^r denotes the re-encoded weight value. The blue-shaded regions represent re-encoded weight packets, where each column corresponds to the encoded data of a convolutional kernel. The yellow-shaded regions denote input activation values, with each row containing im2col-transformed feature map patches. This transformation unfolds each local patch into a column vector to enable efficient convolution via matrix multiplication. The computational workflow operates as follows: the activation data, organized row-wise, flows left to right into each PE at one-clock cycle intervals. The re-encoded weights are loaded into the corresponding PEs in a vertically pipelined manner (top to bottom). Upon receiving both the activation and weight data, a PE immediately initiates computation. The computed results propagate downward in the same direction as the weight flow but utilize independent output channels to avoid resource contention.

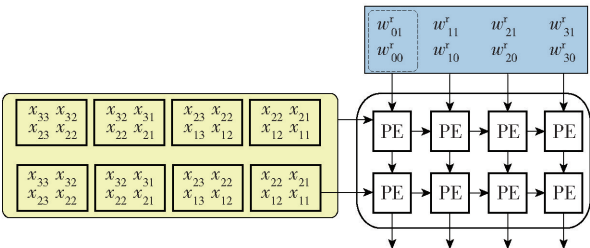


Fig. 11 Flowchart of convolutional unit operation based on LAW

The structure of the PE is illustrated in Fig. 12, with

input signals rw (re-encoded weight), act (activation value), and pre_sum (previous partial sum), and output signals rw_out , act_reg , and sum . The rw signal enters the PE before the act signal. When the rw signal is received, it is stored in the $weight_reg$ register, and the data in $weight_reg$ is then output via rw_out , enabling vertical propagation of weights to the next PE. Once all rw data is loaded, each PE retains a corresponding weight. When act data flows into the PE, it is stored in the act_reg register and transmitted to the next PE. The act value and the locally stored weight in the PE are then processed through computation ($temp_sum$) in the operation unit. The result ($temp_sum$) is added to the partial sum from the previous PE (pre_sum), which is stored in the sum_reg register upon arrival. The accumulated sum is then output via the sum signal to the PE below, completing the vertical cascade of partial results.

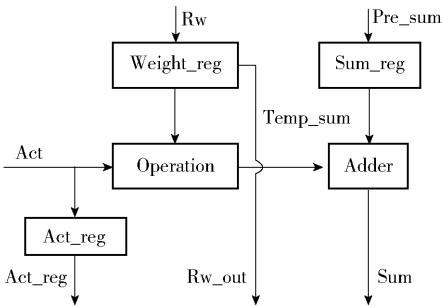


Fig. 12 PE structure based on LAW convolution unit

The operation unit receives input signals of sliding window-sized activation values and re-encoded weights. Its workflow is as follows. First, the module decodes the input rw to extract the position field ($Wposition$), flag field, and $basic_w$ field. The flag field selects the corresponding computational circuit, while the $Wposition$ field indexes the corresponding activation value in the act matrix, which is then computed with the $basic_w$ to generate results. The operation module incorporates five independent computational channels, shown as Fig. 13, to accommodate diverse weight association types, Channel 0 handles non-correlated weights, Channel 1 processes equal weights, Channel 2 addresses proportional relationships, Channel 3 manages linear relationships,

and Channel 4 handles multi-linear relationships. Considering that there are many weights that are not correlated and channel 0 will be used frequently, multiple channel 0 circuits are added in the design. In addition, the coding method of uncorrelated weights is also different, and multiple weights are spliced together for calculation at the same time. This design avoids the reduction of the calculation efficiency of the circuit when dealing with a large number of uncorrelated weights.

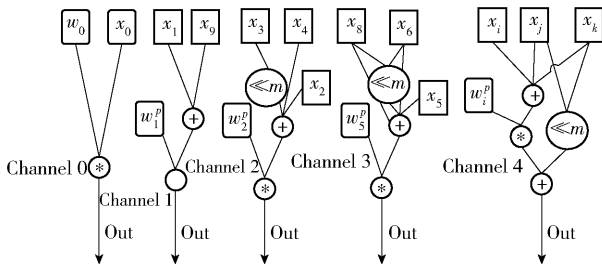


Fig. 13 Channel structure of operation unit

2) ReLU

The ReLU determines its output by detecting the MSB of the input data, if the MSB is 0 (indicating a positive value), the output equals the input; if the MSB is 1 (indicating a negative value), the output is clamped to 0. The proposed ReLU unit adopts a configurable architecture that supports both standard ReLU and ReLU6 functionalities.

3) Max-pooling unit

The max-pooling operation usually slides a fixed-size window (e. g. $2 \times 2, 3 \times 3$) over the input feature map, and then selects the maximum value within the window as the output. In hardware implementation, the max-pooling unit can be divided into two steps, one is to slide the value, the other is to use the comparator to find the maximum value.

4) Fully-connected unit

In CNNs, the input to a fully-connected layer is a 1D vector formed by flattening the output feature maps from the preceding layer. The weight matrix of the fully-connected layer is a 2D structure, where the number of rows corresponds to the output layer size, and the number of columns matches the length of the input vector. Given that the output dimensions are

significantly smaller than the input data, the design of the fully-connected unit adopts a 1D array architecture.

4 Experiment

This section evaluates the performance of the LAW optimization algorithm and hardware architecture.

4.1 Experiment setup

The experiments aim to investigate the applicability and performance of the LAW algorithm across different CNN models. The workflow begins with model quantization, followed by LAW-based analysis of the quantized weights to validate algorithmic effectiveness. Finally, the optimized models are deployed on hardware for performance benchmarking.

To comprehensively assess LAW's efficacy, experiments are conducted on multiple PyTorch-provided pretrained models. Since LAW relies on weight distribution patterns, densely connected convolutional networks 121 (DenseNet121) with high weight dispersion and residual network (ResNet) with low weight dispersion are selected to study the impact of weight distribution characteristics. Additionally, VGG16 with its exclusive use of 3×3 kernels and AlexNet with large 11×11 kernels are included to analyze kernel size effects, covering kernel sizes of $3 \times 3, 5 \times 5, 7 \times 7$, and 11×11 . The experimental environment utilizes Ubuntu 20.04 OS with an NVIDIA GeForce RTX 4060 Ti graphics processing unit (GPU), CUDA 11.6, and PyTorch 1.13.1.

For hardware-level analysis of the LAW algorithm, experiments are conducted on both convolutional computing units and the entire LAW architecture. The evaluation platform is implemented on the ALINX AX7Z020 development board (Xilinx Zynq-7000 SoC, chip model XC7Z020-2CLG400). The hardware performance metrics included are as follows.

1) Flip-flop (FF) indicates the number of flip-flops utilized.

2) Top-1 accuracy reflects the prediction accuracy of

the network model.

3) Look-up table (LUT) reflects the utilization of logic resources.

4) BRAM reflects the utilization of on-chip memory resources.

5) Digital signal processing (DSP) slice reflects the utilization of computational resources.

6) Bit-width (unit: bit) indicates the bit width allocated for storing each image or weight data point.

7) Clock frequency (unit: MHz) reflects the clock toggle rate, where higher frequencies proportionally increase computational operations per second.

8) Inference (unit: ms) measures the end-to-end inference time required by the LAW-core to process an entire neural network model.

9) Computation amount performance (unit: GOPS) quantifies the computational operations per second executed by the LAW-core, calculated as the total operations divided by inference latency. Higher values indicate enhanced throughput capacity of the accelerator.

4.2 Algorithm evaluation

To validate the effectiveness of the LAW algorithm, quantized network weights are first analyzed using three intra-kernel optimization strategies, and statistical results are derived through a greedy algorithm integrating these strategies. Fig. 14 illustrates the multiplications saved ratio by the LAW algorithm, where the horizontal axis represents the kernel index, and the vertical axis denotes the multiplication saved ratio reduced per kernel. Fig. 14(a) shows the results for 20 randomly selected 7×7 kernels in the first convolutional (conv1) layer of ResNet-18, while Fig. 14(b) corresponds to the 3×3 kernels of the first convolutional layer within the initial residual block of stage 1 (layer1.0.conv1). The multiplications saved ratio approaching 1 indicates that all elements in the kernel are zeros. Proportional analysis, linear analysis, and inter-weight correlation analysis are applied to each kernel. As shown in Fig. 14, all three strategies reduced multiplication counts to varying degrees. The LAW algorithm identifies the optimal

solution by dynamically combining these strategies. As shown in Fig. 14(b), for the fifth convolution kernel, the multiplication optimization effect of the LAW algorithm is the same as that of the linear analysis strategy. According to the workflow described in Algorithm 1, when processing this kernel, the linear correlation condition consistently covers the largest number of optimizable weights during each iteration of the LAW algorithm. The algorithm first simultaneously checks three correlation conditions, then selects weights satisfying the most conditions for optimization marking, and repeats this process for the remaining weights until no optimizable items remain. Therefore, when the linear condition maintains continuous dominance throughout the optimization sequence, the execution path of the LAW algorithm naturally coincides with that of the standalone linear analysis strategy.

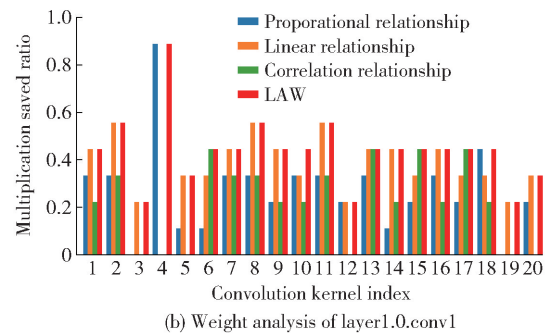
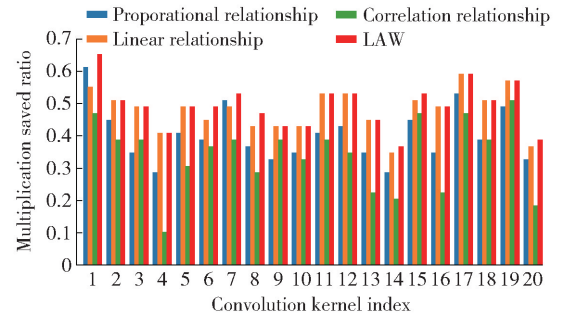


Fig. 14 Performance analysis of the intra-kernel in ResNet-18

Fig. 15 shows the weights statistics of the 5th convolutional layers (features.5) and the 10th convolutional layer (features.10) in VGG16, with a kernel size of 3×3 . It is found that the multiplication saved ratio by the single convolutional calculation method for the 3×3 convolutional kernel in the VGG16

network is roughly 20% – 35% .

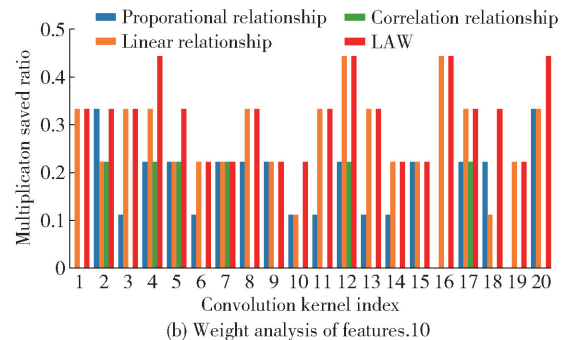
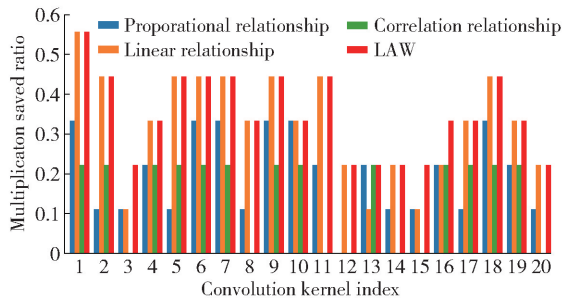


Fig. 15 Performance analysis of the intra-kernel in VGG16

Fig. 16 shows the weights statistics of the first convolutional layer (features.0, kernel_size is 11×11) and the 4th convolution layer (features.3, kernel_size is 5×5) in the AlexNet network.

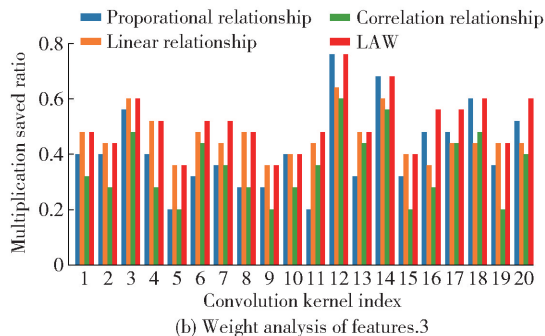
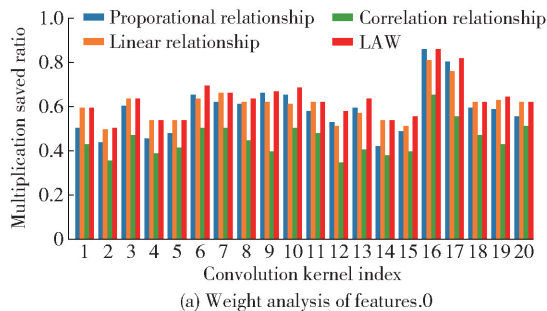


Fig. 16 Performance analysis of the intra-kernel in AlexNet

It can be seen from Fig. 16 that the optimization

effect of LAW algorithm for 11×11 convolution kernel can reach 50% – 70% , and the average optimization effect for 5×5 convolution kernel is in the range of 40% – 55% . The single convolution calculation method has better results for larger convolutions.

The experimental results are detailed in Table 5.

Table 5 Multiplication reduction of intra-kernel algorithm across different networks

Net	Kernel size	Multiplication reduction ratio/%
ResNet-18	7×7	54.0
ResNet-18	3×3	34.0
ResNet-50	7×7	50.4
ResNet-50	3×3	37.0
VGG16	3×3	30.0
DenseNet121	3×3	31.0
AlexNet	11×11	61.0
AlexNet	5×5	49.0

From Table 5, for large convolutional kernels, the LAW algorithm achieves an average reduction of nearly 50.0% in multiplications. However, the optimization effect is less pronounced for small kernels (e. g. , 3×3), primarily due to their fewer elements. Notably, 3×3 kernels in ResNet exhibit better optimization outcomes than those in DenseNet121, indicating that weight variance (data dispersion) influences LAW’s effectiveness. In general, the convolutional layer corresponding to a 3×3 convolution kernel has a large number of input and output channels, such as the 2nd convolutional layer in the 1st bottleneck block of stage 3 (layer3.0.conv2) in the ResNet-50, which has 256 input and output channels. Therefore, for a 3×3 convolution kernel, multiple convolution calculation methods in LAW can be used.

To validate the multi-kernel linear association algorithm, experiments are conducted on ResNet-18, VGG16, and DenseNet121 networks. For these networks, the algorithm is applied only to 3×3 convolutional layers with high input/output channel counts. The results for ResNet-18 are shown in Fig. 17, where the horizontal axis represents the positional indices of the weight matrix, and the vertical axis denotes the multiplication saved ratio across all

channels at each index. Figs. 17(a) – 17(e) presents the results obtained from the convolutional weight parameters of the 2nd convolutional layer in the 1st residual block of stage 1 (layer1.0.conv2) in ResNet-18. This layer uses 3×3 kernels with 64 input and output channels. Due to the large number of channels, only the results of the first five channels (channel 1 – 5) are shown, each containing 9 indices

corresponding to the 3×3 kernel positions. Each channel contains 9 sequence numbers corresponding to the 3×3 dimensions of the convolution kernel. As can be seen from Fig. 17, the most significant multiplication saving effect is the 4th position in kernel in channel 1, which saves nearly 70% of the multiplication operations, and the lowest saving rate is close to 50%.

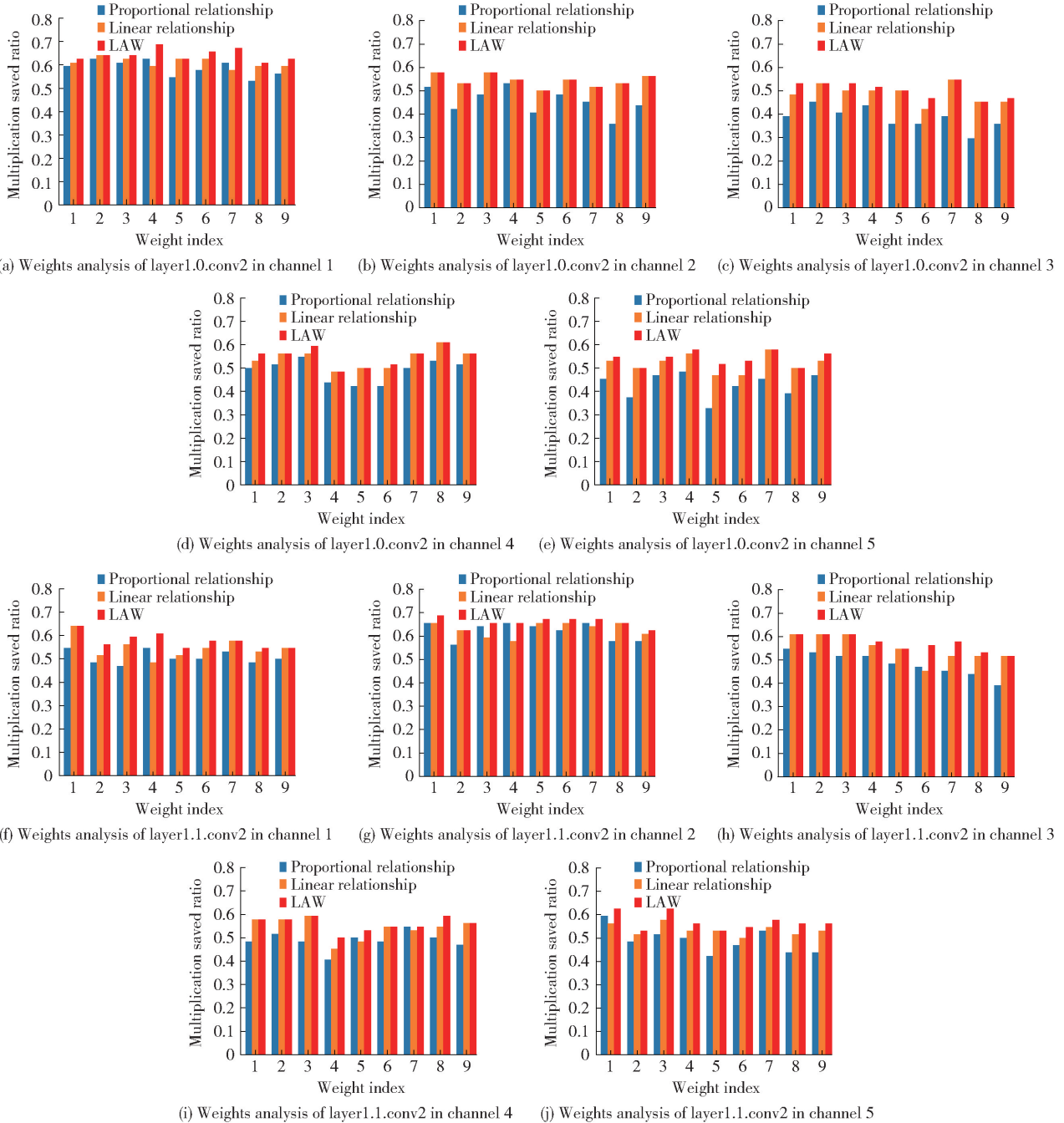


Fig. 17 Performance analysis of cross-kernel in ResNet-18

Fig. 18 shows the effect of multiple convolutional accounting in a VGG16 network. Figs. 18(a) – 18(e) show the result of the first convolutional layer of the second convolutional block (features.5), whose input channel is 64. It can be seen that the multiplications

saved rate is between 50% and 65%. Figs. 18(f) – 18(j) shows the result of the second convolutional layer of the third convolutional block (features.10), which has 128 output channels and multiplications saved rate of 55% to 70%.

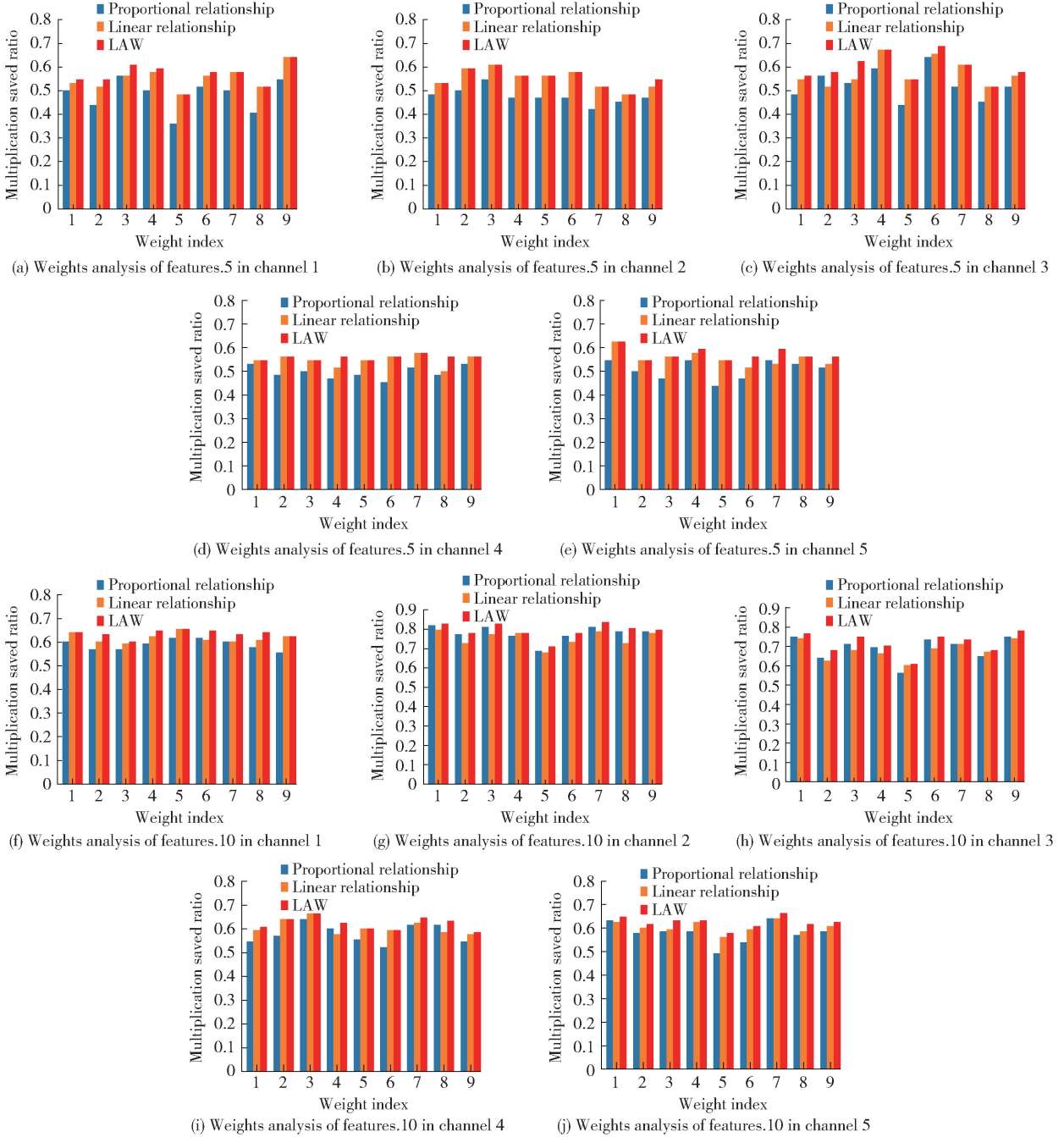


Fig. 18 Performance analysis of cross-kernel in VGG16

Fig. 19 illustrates the experimental results for the DenseNet121 network. Due to the deep architecture of DenseNet121, the analysis focuses on the first two

convolutional layers, Figs. 19(a) – 19(e) show the results from the convolutional layer within the first dense layer of the initial dense block (features.

denseblock1.denselayer1.conv2), while Figs. 19 (f) – 19(j) corresponds to the convolutional layer in the second dense layer of the same block (features. denseblock1.denselayer2.conv2). Both layers have

128 input channels, and the results demonstrate multiplication saved ratio of 60% – 70% across kernel positions.

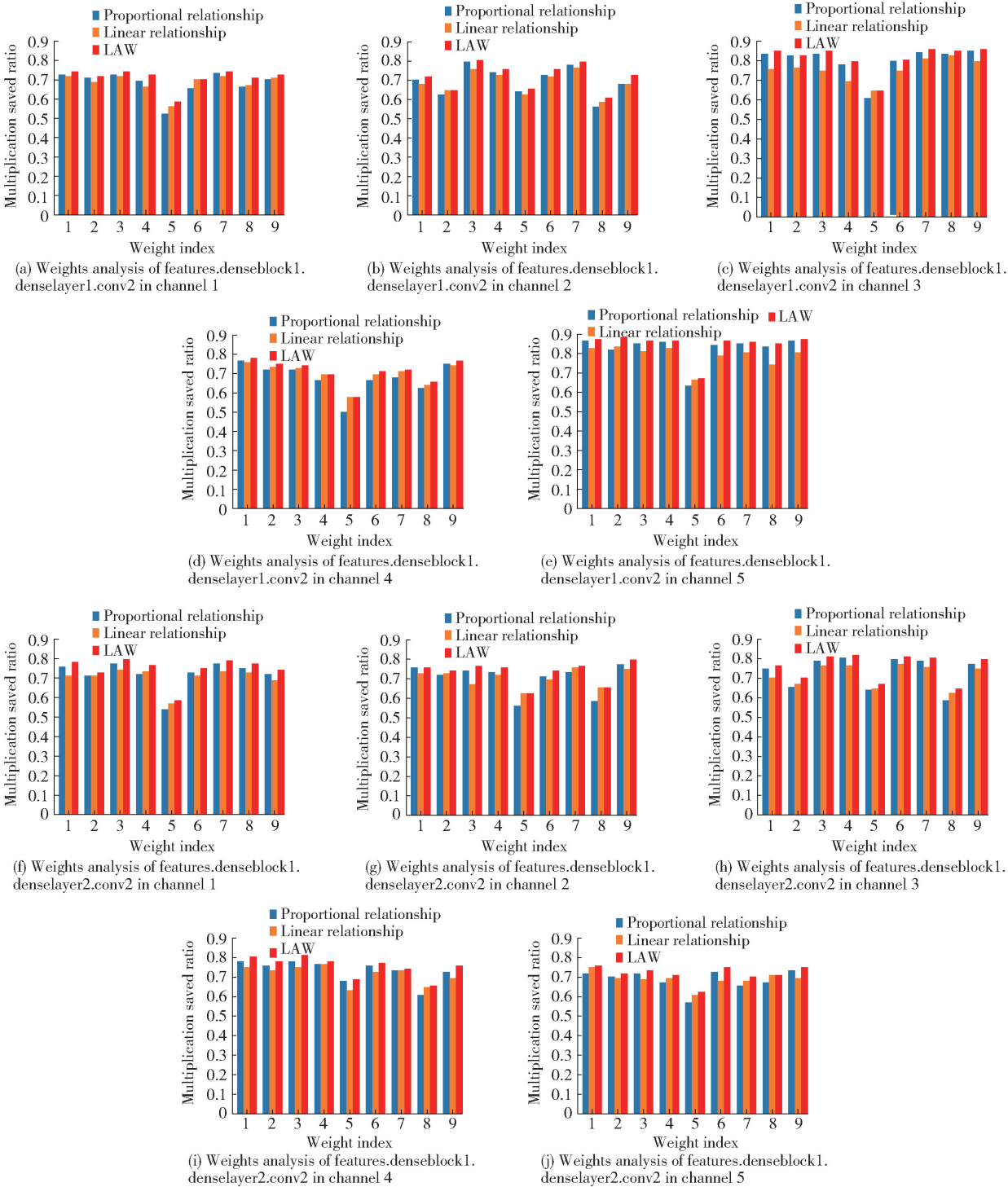


Fig. 19 Performance analysis of cross-kernel in DenseNet121

4.3 Hardware evaluation

1) Evaluation of the convolution unit

To assess the performance of the LAW-optimized convolution unit, experiments are conducted across two metrics: hardware resource utilization and clock cycles required to complete a convolutional layer. The test utilizes the first convolutional layer of ResNet-18 with the following parameters: 3 input channels, 64 output channels, 7×7 kernel size, 3 paddings, 2 strides, and 32×32 input feature maps. Two configurations are compared, a baseline systolic array with 49×3 dimensions using 8-bit multipliers, and the LAW-optimized convolution unit with 24×3 dimensions. As shown in Table 6, under a 100 MHz clock frequency, the LAW-optimized convolution unit increases hardware resource usage by 18% compared to the baseline but reduces clock cycle consumption by 35%. The experimental results vary depending on the network, because the core of the LAW algorithm is to reduce multiplication by using the association between weights.

Table 6 Comparison of systolic array resource utilization

Method	Bit-width/bit	LUT	FF	Clock count
Multiplication	8×8	19 658	6 104	4 288
LAW	8×8	19 440	11 749	2 752

Additional experiments are conducted from the perspectives of the intra-kernel association algorithm and multi-kernel algorithm. In this study, convolutional kernels of varying sizes (3×3 , 5×5 , 7×7 , 11×11) and layers with different channel configurations (64, 256, 512 channels, all using 3×3 kernels) are selected for comparative analysis of hardware resource utilization and computational latency. The fair evaluation of these metrics is based on the criterion defined as

$$P_{\text{erf}} = \frac{\frac{H_b}{H_1}}{\frac{C_1}{C_b}} \quad (5)$$

where H_b represents hardware resource consumption of the baseline (traditional multiplier array), H_1 represents hardware resource consumption of the LAW-

based design. C_b represents the clock cycles required to complete one convolutional layer in a traditional multiplier array, and C_1 represents the clock cycles required to complete one convolutional layer in LAW-based design. Performance value, P_{erf} , greater than 1 indicates that the speed improvement outweighs the resource overhead. The weights used in these experiments align with the data from algorithm evaluation, ensuring consistency across evaluations. The experimental results are shown in Fig. 20.

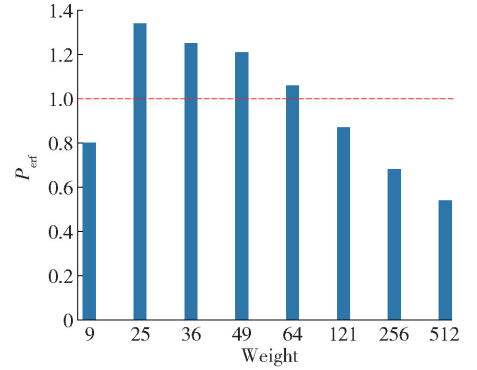


Fig. 20 Performance analysis

From Fig. 20, when the convolution kernel is 3×3 , the performance value is less than 1. This is mainly because the association between the nine weights is weak and does not increase the speed much compared to the convolution operation. For 5×5 and 7×7 convolutional kernels, the weight association is strong, so it is reasonable to exchange a small amount of hardware resources for a large increase in computing speed. When the convolution kernel size is 11×11 , although the weight relevance is significantly enhanced and the number of multiplications is theoretically reduced by 60%, the circuit complexity and hardware resource consumption increase disproportionately. This is primarily due to the expanded bit-width of input eigenvalues and the additional position information embedded in the encoded weights. Fig. 20 also shows that the same problem exists with a large number of input channels. Although the weight association in multi-channel is strong, the increase in hardware resource consumption is too large, resulting in a performance value lower than 1. Therefore, in deploying the LAW algorithm, the network's multi-

channel inputs are partitioned into smaller channels to strike an optimal balance between hardware resource utilization and operational efficiency.

2) Evaluation of the LAW-core

In this section, ResNet-18 model is selected to evaluate the overall performance of LAW-core. In order to deeply understand the ResNet-18 model, the calculation and data volume of ResNet-18 are calculated using Python, and the real-time inference performance is tested on the single-core Intel®Core™ i5-10400F processor. The parameter number of the network is 11.69 M, the computation amount is 3.65 GOPS, and the inference time on the central processing unit (CPU) is 67.10 ms.

Table 7 shows the performance comparison between LAW-core and several prior FPGA accelerators, including SP2^[9], Zac^[12], and the logarithmic general matrix multiplication (GEMM)^[13] method. In Table 7, dash (-) indicates that the value is not explicitly provided in the corresponding reference. SP2^[9] introduces a hardware-friendly quantization scheme for weights that follow an approximately Gaussian distribution. Its key innovation is transforming multiplications into logical shift and addition operations, enabling efficient implementation that primarily utilizes the LUT resources of an FPGA. Zac^[12] is an automated framework for deploying quantized deep neural networks (DNNs). It can automatically generate or invoke corresponding

hardware-optimized operators based on the selected data types, thereby producing a complete and deployable hardware design. GEMM^[13] proposes and validates an efficient logarithmic quantization method specifically designed for FPGA hardware. By pairing a novel low-precision quantization scheme with a co-designed hardware architecture that replaces multiplications with shift-and-add operations, it significantly improves computational efficiency and resource utilization on embedded FPGA platforms while maintaining model accuracy. In these studies, the employed network architecture is consistently ResNet-18. Through comparison, it can be found that Zac^[12] realizes the computation amount performance of 124.9 GOPS with a clock frequency of 200 MHz and more abundant computing resources. GEMM^[13] uses a 6-bit quantization bit-width, which used the least hardware resources for the accelerator. SP2^[9] achieves high computation amount performance through low precision quantization, and its accelerator adopts a general architecture. Compared to SP2^[9] and GEMM^[13], this accelerator consumes significantly more hardware resources, mainly because the convolutional systolic array employs a large number of PEs. In terms of performance, the reasoning delay of the LAW accelerator is 40.21 ms, and the computation amount performance reaches 90.12 GOPS, which is 1.67 times higher than that of CPU, and also reduces the reasoning delay by 6.89 ms compared with SP2^[9].

Table 7 Performance comparison of different accelerators on ResNet-18

Accelerator	Device	Bit-width of weight and activation/bit	Top-1 accuracy/%	Clock frequency/MHz	LUT	DSP	BRAM	Inference/ms	Computation amount performance/GOPS
SP2 ^[9]	XC7Z020	4,4	70.27	100	28 288	220	56	47.10	77.00
Zac ^[12]	ZC706	8,8	-	200	100 200	818	708	29.03	124.90
GEMM ^[13]	XC7Z020	6,6	67.41	150	12 600	8	82	52.03	69.70
LAW	XC7Z020	8,8	70.16	100	41 513	60	17	40.21	90.12

The experimental results combined with the optimization effects of the LAW algorithm on ResNet-18 demonstrate that LAW’s core advantage lies in algorithm-hardware co-acceleration. At the algorithmic level, LAW reduces the number of multiplications in convolution operations. At the hardware level, while

introducing increased hardware resource utilization, it achieves significantly greater acceleration benefits and simultaneously reduces reliance on DSP blocks.

5 Conclusions

The LAW algorithm in this study optimizes

conventional convolution operations by establishing linear equivalent models among weights through quantization techniques and data distribution analysis. By performing linear association analysis on both individual convolution kernels and multi-channel convolutions, the algorithm replaces multiplication operations with shift-add operations to improve inference speed. To enable hardware-algorithm co-acceleration, a dedicated LAW-core architecture is developed whose workflow first performs weight analysis and re-encoding via the LAW algorithm, with the processed weights pre-stored in DDR memory. Subsequently, configurable computing units decode the encoded weights and execute targeted operations guided by flag bits. Compared to conventional convolution based on per-channel MAC operations, the LAW algorithm transcends traditional computational limitations, providing a novel hardware-aware approach to neural network acceleration.

Acknowledgements

This work was supported by the Scientific Research Program Funded by Shaanxi Provincial Education Department (20JY058), and the Technology Talent Program of Xi'an City of Shaanxi Province (23GXFW0087).

References

- [1] LECUN Y, BENGIO Y, HINTON G. Deep learning. *Nature*, 2015, 521(7553): 436 – 444.
- [2] HUANG A, CAO Z, WANG C R, et al. An FPGA-based on-chip neural network for TDLAS tomography in dynamic flames. *IEEE Transactions on Instrumentation and Measurement*, 2021, 70: Article 4506911.
- [3] SAHANI M, DASH P K. Deep convolutional stack autoencoder of process adaptive VMD data with robust multikernel RVFLN for power quality events recognition. *IEEE Transactions on Instrumentation and Measurement*, 2021, 70: Article 9001912.
- [4] ABBASNEZHAD M, ALIZADEH B. FPGA-based implementation of an artificial neural network for measurement acceleration in BOTDA sensors. *IEEE Transactions on Instrumentation and Measurement*, 2019, 68(11): 4326 – 4334.
- [5] ZHOU X D, DU Z D, ZHANG S J, et al. Addressing sparsity in deep neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019, 38(10): 1858 – 1871.
- [6] DENG L, LI G Q, HAN S, et al. Model compression and hardware acceleration for neural networks: a comprehensive survey. *Proceedings of the IEEE*, 2020, 108(4): 485 – 532.
- [7] LEE J H, KIM J, KWON S J, et al. FlexRound: learnable rounding based on element-wise division for post-training quantization. *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*, 2023, Jul 23 – 29, Honolulu, HI, USA. 2023: 27p.
- [8] NAGEL M, FOURNARAKIS M, AMJAD R A, et al. A white paper on neural network quantization. *arXiv Preprint, ArXiv: 2106.08295*, 2021.
- [9] CHANG S E, LI Y Y, SUN M S, et al. Mix and match: a novel FPGA-centric deep neural network quantization framework. *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*, 2021, Feb 27 – Mar 3, Seoul, Republic of Korea. Piscataway, NJ, USA: IEEE, 2021: 208 – 220.
- [10] WANG K, LIU Z J, LIN Y J, et al. HAQ: hardware-aware automated quantization with mixed precision. *Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'19)*, 2019, Jun 15 – 20, Long Beach, CA, USA. Piscataway, NJ, USA: IEEE, 2019: 8612 – 8620.
- [11] CHEN W H, WANG P S, CHENG J. Towards mixed-precision quantization of neural networks via constrained optimization. *Proceedings of the 2021 IEEE/CVF International Conference on Computer Vision (ICCV'21)*, 2021, Oct 10 – 17, Montreal, Canada. Piscataway, NJ, USA: IEEE, 2021: 5350 – 5359.
- [12] XIAO Q C, LIANG Y. Zac: towards automatic optimization and deployment of quantized deep neural networks on embedded devices. *Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'19)*, 2019, Nov 4 – 7, Westminster, CO, USA. Piscataway, NJ, USA: IEEE, 2019: 6p.
- [13] JIANG T, XING L G, YU J M, et al. A hardware-friendly logarithmic quantization method for CNNs and FPGA implementation. *Journal of Real-Time Image Processing*, 2024, 21(4): Article 108.

(Editor: Wang Xuying)