mathematics

MDPI

Article

Optimization of Direct Convolution Algorithms on ARM Processors for Deep Learning Inference

Shang Li [1], Fei Yu [1,*], Shankou Zhang [1], Huige Yin [1] and Hairong Lin [2]

[1] School of Computer Science and Technology, Changsha University of Science and Technology, Changsha 410076, China; lishangls@stu.csust.edu.cn (S.L.); 22408050290@stu.csust.edu.cn (S.Z.); 003662@csust.edu.cn (H.Y.)
[2] School of Electronic Information, Central South University, Changsha 410083, China; haironglin@csu.edu.cn
* Correspondence: yufeiyfyf@csust.edu.cn

Abstract: In deep learning, convolutional layers typically bear the majority of the computational workload and are often the primary contributors to performance bottlenecks. The widely used convolution algorithm is based on the IM2COL transform to take advantage of the highly optimized GEMM (General Matrix Multiplication) kernel acceleration, using the highly optimized BLAS (Basic Linear Algebra Subroutine) library, which tends to incur additional memory overhead. Recent studies have indicated that direct convolution approaches can outperform traditional convolution implementations without additional memory overhead. In this paper, we propose a high-performance implementation of the direct convolution algorithm for inference that preserves the channel-first data layout of the convolutional layer inputs/outputs. We evaluate the performance of our proposed algorithm on a multi-core ARM CPU platform and compare it with state-of-the-art convolution optimization techniques. Experimental results demonstrate that our new algorithm performs better across the evaluated scenarios and platforms.

Keywords: deep learning; convolution; direct algorithm; ARMv8 architecture; performance optimization

MSC: 65K10; 65Y20; 68W40; 68T07

check for updates

Academic Editor: Ioannis Tsoulos

Received: 10 January 2025
Revised: 20 February 2025
Accepted: 26 February 2025
Published: 27 February 2025

Citation: Li, S.; Yu, F.; Zhang, S.; Yin, H.; Lin, H. Optimization of Direct Convolution Algorithms on ARM Processors for Deep Learning Inference. *Mathematics* **2025**, *13*, 787. https://doi.org/10.3390/math13050787

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Since the birth of computer technology, great progress has been made in many research directions. For example, significant progress has been made in recent years in data center network optimization [1–4], sensor and neural network optimization [5–9], the Internet of Things [10–13], and the rapidly emerging field of deep learning algorithms [14–17]. These studies not only promoted technological innovation in related fields but also provided important support for the development of computer technology. Artificial intelligence is a key area of modern scientific and technological progress. It uses computer science and engineering technology to give machines the ability to perform and imitate human intelligent behaviors. Its performance is comparable to or even better than that of humans in many application scenarios. In this context, convolutional neural networks (CNNs) have become an indispensable part of artificial intelligence due to their high accuracy. They have shown a wide range of application values in computer vision fields such as image recognition [18–21], image classification [22,23], object detection [24–27], and image enhancement [28]. Since Yann Lecun et al. first proposed LeNet-5 [29] in 1998, CNN technology has experienced rapid development. From AlexNet [30] developed by Hinton

et al., to ZFNet [31] by Huang et al., to VGG [32] proposed by the Visual Geometry Group of the University of Oxford, and the subsequent GoogleNet [33], ResNet [34], and DenseNet [31], the performance of CNNs has been continuously improved, and the problems they solve have become increasingly complex.

With the popularity of mobile device processors (mostly based on ARM architecture) and the increasing number of deep learning applications on real-time devices, optimizing the computational efficiency of convolution algorithms based on ARM CPU has become particularly important and urgent. The types of ARM CPUs are extremely diverse, covering embedded/mobile SoCs, personal computer processors, and server-level processors in data centers. This diversity poses significant challenges to the performance optimization of convolution algorithms. With the increase in machine learning applications based on deep neural networks, it is time to rethink how convolutional layers are computed. According to some current work, the optimization of convolution operators has made great progress in format conversion, memory usage, and performance. However, a careful analysis of some recent work does not make it difficult to see the problem. The IM2COL (Image to Column) [35] technique combined with the GEMM (General Matrix to Matrix Multiplication) algorithm is a widely adopted method. This strategy achieves acceleration by converting the convolution operation into a matrix multiplication problem, and then utilizing the highly optimized GEMM function in the Basic Linear Algebra Subroutines (BLAS) library. However, the IM2COL method has certain drawbacks, mainly that it requires additional memory space to store the rearranged matrix, which may be a problem when processing large-scale datasets or on memory-constrained devices. In addition, the IM2COL operation itself also introduces a certain amount of computational overhead. LIBXSMM (Accelerating Small Matrix Multiplications by Runtime Code Generation) [36,37] is a new library-based solution for implementing direct convolution on CPU. The library uses a specific data layout and shows better performance than the traditional IM2COL GEMM method on x86 and ARM CPU platforms. LIBXSMM requires format conversion, which requires code refactoring, and Im2col+ GEMM [35] may increase memory pressure. Other works have also attempted to introduce specialized data formats that allow continuous memory access and direct use of SIMD instructions and FMA (Fused Multiply-Accumulate) units to optimize the convolution operation [38–41]. All previous approaches leave much room for performance improvements.

In this study, we designed an efficient convolution operator that is seamlessly compatible with mainstream deep learning frameworks and does not require refactoring of the existing underlying convolution operator code (refactoring, modifying the code structure or data interface to adapt to new algorithms). This design aims to provide a plug-and-play solution that allows developers to easily use this algorithm in existing deep learning projects to achieve performance improvements without having to go into the underlying implementation details.

Our convolution operator is designed with full consideration of the diverse ARM CPU architectures and significantly improves the execution efficiency of the algorithm by optimizing data access patterns and computational processes. In addition, the operator uses advanced optimization techniques such as loop unrolling, memory prefetching, and vectorized calculation to further accelerate the processing speed of convolution operations.

To verify the performance of the convolution operator we designed, we compared it with two state-of-the-art and mainstream convolution algorithms. The test results show that our operator exhibits excellent performance on different data scales. Specifically, when our operator performs convolution computation tasks, our algorithm provides 1.43× and 1.1× geometric mean speedups over IM2COL+OpenBLAS and LIBXSMM, respectively. This result demonstrates the practicality and efficiency of our operator, providing a powerful

tool for deep learning applications based on ARM CPUs. The main contributions of this paper can be summarized as follows.

First, we propose a novel and efficient direct convolution algorithm for inference based on ARM architecture processors. The core innovation of the algorithm is that it can be directly applied to standard data layouts without any rearrangement or transformation of the data. This feature significantly simplifies the operational steps of convolution operations, reduces computational complexity and execution time, and enables hardware devices to execute complex deep learning models more efficiently.

Second, in many existing convolution implementations, in order to improve computational efficiency, additional memory space is required to store intermediate computation results or optimized data structures. Our algorithm effectively utilizes existing memory resources by optimizing the computational process and memory access patterns, thus avoiding the need for additional memory.

In summary, the algorithm in this paper not only improves the computational efficiency of convolutional networks based on ARM CPU but also enhances the practicality and applicability of the algorithm by reducing the demand for resources. These innovations make our approach of significant promise and value in the field of deep learning, especially in resource-constrained mobile and embedded devices.

2. Background and Related Work

In the structure of a convolutional neural network (CNN), different layers have their own unique functions and tasks, such as convolutional layers, ReLU layers, and pooling layers. In order to accelerate the reasoning process of the entire network, the most effective strategy is to fine-tune the calculation of each layer. Among these layers, the convolutional layer is particularly critical because it occupies most of the CPU resources and running time in the operation of the neural network. Specifically, the convolutional layer accounts for about 90% of the total reasoning time of CNN [42], which reflects its importance and resource consumption in the entire network operation.

The computational efficiency of the convolution layer determines the efficiency of inference. Therefore, many studies are looking for more effective ways to accelerate convolution on different platforms. Efficient convolution calculations are implemented according to the capabilities of the platform. The method of the commonly used convolution algorithm IM2COL is to convert the convolution kernel into multiple columns to construct a matrix. Then, the convolution operation is converted into the multiplication of two matrices. The multiplication between matrices is calculated in The Basic Linear Algebra Subroutine library, such as General Matrix Multiplication (GEMM) [43,44]. For the CPU, the Winograd algorithm is more suitable. Winograd is an approximate algorithm [45] that replaces multiplication with addition. It is well known that addition takes fewer cycles on the CPU. Therefore, this method is often used by the DNN inference library of the CPU. Direct convolution is also called the sliding window method. This sliding window method is easy to implement on different platforms. No transformation is required before and after the calculation. The sliding window method is the most commonly used and performs well in most cases.

2.1. Related Work

In deep learning applications, especially in the implementation of convolutional neural networks (CNNs), convolution operations are usually one of the most computationally intensive parts. To improve the computational efficiency of convolution operations, researchers have developed a variety of optimization algorithms. Among them, IM2COL (Image to Column) technology combined with the GEMM (General Matrix to Matrix Multi-

plication) algorithm is a widely used method [35]. This strategy achieves acceleration by converting convolution operations into matrix multiplication problems and then utilizing the GEMM function in the highly optimized Basic Linear Algebra Subroutines (BLAS) library [43]. IM2COL is a data rearrangement technique used to convert the input image in the convolution operation into a column vector, where each column vector corresponds to an area on the input image covered by the convolution kernel. This allows the convolution operation, which was originally based on element-by-element multiplication, to be transformed into a matrix multiplication problem. Specifically, for a specific convolution kernel and input image, IM2COL first generates all possible image patches and rearranges these patches into a matrix. Subsequently, this matrix is multiplied with the matrix corresponding to the convolution kernel to produce the final output feature map. However, the IM2COL method has certain defects, mainly that it requires additional memory space to store the rearranged matrix, which may be a problem when processing large-scale datasets or on memory-constrained devices. In addition, the IM2COL operation itself also introduces a certain computational overhead.

The Winograd algorithm is an efficient convolution calculation method that reduces the number of multiplication and addition operations required for convolution operations through mathematical transformations [45]. The algorithm effectively exploits the inherent sparsity of the convolution operation, thereby reducing the number of floating-point operations required to compute the output feature map. The core idea of the Winograd algorithm is to transform the convolution kernel and the input feature map into a new mathematical domain, in which the convolution calculation is accelerated by performing fewer multiplication operations. The advantage of the Winograd algorithm is that it can identify the optimal transformation matrix for a specific convolution kernel size. These matrices mainly convert most multiplication operations in the convolution operation into addition operations. The algorithm is particularly efficient when processing small convolution kernels (such as 3×3 or 5×5) commonly used in deep learning, and can significantly reduce the number of multiplication operations required, often to only a fraction of the original number. This reduction is achieved by precomputing the transformation matrix between the filter and the input and performing the dot product operation in the transformed domain. In practical applications, it is necessary to find the optimal transformation matrix for different convolution kernel sizes and input feature map sizes. Furthermore, the implementation of the algorithm needs to be carefully designed to ensure that the transformation operation does not introduce too much additional computational burden. It should be noted that the Winograd algorithm may introduce a certain loss of accuracy.

LIBXSMM is a recent library-based solution designed to implement direct convolution on the CPU [46]. The library adopts a specific data layout, and LIBXSMM shows better performance than the traditional IM2COL GEMM method on x86 and ARM CPU platforms. Despite its potential, LIBXSMM suffers from two major limitations. First, the design of its data layout is incompatible with the data layout commonly used in mainstream deep learning frameworks (such as NCHW (Batch Size, Input Channels, Input Height, Input Width) or NHWC (Batch Size, Input Height, Input Width, Input Channels)) [47,48]. Therefore, integrating the BRGEMM routine into existing deep learning frameworks requires either refactoring the underlying framework code or adding a format conversion stage in the user code at each call and end of a convolution operation. The latter approach not only requires modifying the standard user model code but also introduces additional computational overhead. Second, although LIBXSMM still uses the traditional GEMM-based microkernel, it does not fully exploit the data reuse potential in convolution operations to improve performance.

2.2. Direct Convolution

Before determining the loop strategy and vectorization method of direct convolution, let us first introduce the basic concept of convolution, with special attention to the implementation based on direct convolution. In the CNNs, the convolution operator involves an input tensor (I), and a convolution kernel (or filter) tensor (F), which produces an output tensor (O). In each convolution layer, the number of convolution kernels is the same as the depth (i.e., the number of channels) of the output tensor (K = Co), and the depth of each convolution kernel is equal to the depth of the input tensor.

We can represent the convolution operator as:

$$O = Conv(I, F) \tag{1}$$

where I is a 4D input tensor consisting of N inputs of size $H \times W \times C$, where $H \times W$ represents the height×width of the image and C represents the number of its input channels. In addition, the value of the F variable is a 4D input tensor consisting of K filters of $height \times width \times channels = C \times R \times S$. The operator in (1), then, produces a 4D output tensor O, each of size $Ho \times Wo \times Co$, where $Ho \times Wo$ represents the output $height \times width$ and K represents the number of output channels (equal to the number of filters). To achieve this, the computation is performed by repeatedly applying the filter to the entire input tensor in a sliding window manner, and the convolution operator produces a single output complete entry. In the convolution calculation, the relationship between the height and width of the output and the input and filter follows Formula (2).

$$\begin{cases} H_0 = \frac{H-R+2\times padding}{stride} + 1 \\ W_0 = \frac{W-S+2\times padding}{stride} + 1 \end{cases} \tag{2}$$

Algorithm 1 shows a basic algorithm for computing convolution. In general, the algorithm is executed by traversing the dimensions of the convolution operator in order from the outermost to the innermost. We assume that the operands are arranged in row-major order of the multidimensional tensor, which means that the dimensions are arranged in increasing order from right to left in memory (i.e., the nearest dimension to the farthest dimension). In addition, we use the notation I[N][H][W][C], O[N][Ho][Wo][Co] to specify the layout of the 4D input/output tensors in memory. In order to better demonstrate and illustrate the methods and arguments of this paper, we use a unified parameter representation method. For specific parameter representation, please see Table 1. Note that this layout corresponds to the standard NHWC format for 4D tensors in deep learning. The 4D filter tensor is stored in the standard RSCK format, i.e., F[R][S][C][K]. The seven loops in the algorithm are independent of each other and have nothing to do with the memory layout of the input, output, and filter tensors. Therefore, these algorithm loops can be reorganized in any order while producing correct results.

Table 1. Symbolic representation of tensors used in the paper.

Tensor Type	Height	Width	Channels	Batch
Input (I)	H	W	C	N
Filter (F)	R	S	C	K
Output (O)	H_0	W_0	C_0/K	N

Algorithm 1 Naive direct convolution algorithm

Input: I[N][H][W][C]; F[R][S][C][K], stride str;
Output: O[N][Ho][Wo][Co]

for $n = 0$ to $N - 1$ **do**
  for $c = 0$ to $C - 1$ **do**
    for $k = 0$ to $K - 1$ **do**
      for $h = 0$ to $Ho - 1$ **do**
        for $w = 0$ to $Wo - 1$ **do**
          for $r = 0$ to $R - 1$ **do**
            for $s = 0$ to $S - 1$ **do**
              $O_{n,h,w,k} += I_{n,h*str+r,w*str+s,c} \times F_{r,s,c,k}$;

2.3. Position of Our Work

The convolution operator bears a large part of the algorithmic cost in deep neural networks (DNNs). Therefore, it is worthwhile to research how to perform this operation effectively. Direct convolution algorithms are usually implemented as six or seven layers of nested loops, which are mainly centered around multiplication and accumulation operations. The strategies for optimizing convolution operations on CPUs differ significantly from the approaches on GPUs. Especially on embedded processors, it is particularly important to avoid data redundancy and repeated calculations. Our proposed optimization technique fully exploits SIMD instructions to achieve instruction-level parallelism. Considering that most modern CPUs are equipped with efficient pipelines, optimizing convolution operations should fully consider the efficiency of the pipeline. While the compiler can vectorize the data and take advantage of SIMD instructions, the nested loop structure of convolutions limits the potential for compile-time optimizations. Therefore, there is still room for further improvement in the efficiency of convolution calculations on ARM-based CPUs.

Unlike existing methods, our algorithm does not require rearrangement or format conversion of input data. This is particularly important because data conversion not only consumes computational and memory resources but may also introduce additional latency. By operating directly on the original data layout (such as NHWC or NCHW), our method simplifies the entire computation process while avoiding the overhead and complexity associated with data conversion.

In our work, special attention is paid to improving memory access efficiency and enhancing data reuse. Through carefully designed memory access strategies, we optimize the data reading and writing process and reduce the latency and bandwidth requirements of memory access. In addition, we increase the reuse rate of data in the computation process through algorithm optimization and reduce unnecessary data loading and storage operations.

3. Algorithm Design

To implement a truly high-performance direct convolution algorithm, two key points must be considered: (1) finding the optimal parameter values for the convolution loop based on the hardware characteristics and (2) writing a convolution loop that can fully utilize the underlying hardware characteristics and functions. To calculate the code, we developed a microkernel architecture specifically optimized for convolutional computations, with the main goal of maximizing the efficiency of data access and significantly reducing latency. The microkernel has been carefully designed to ensure that cache misses are minimized and memory bandwidth usage is optimized. In addition, the microkernel adopts an optimized data prefetching strategy, which can schedule data to the processor in advance according to the specific needs of convolution operations to reduce waiting time and improve the

overall computing speed. To clearly illustrate our optimization strategy, this article uses FP32 tensor data type and 3 × 3 convolution kernel as examples.

3.1. Algorithm Overview

Our algorithm is mainly divided into two parts, the microkernel for convolution computation and the outer loop of the kernel. We need to design these two parts separately, and we will explain each part below.

Loop outside the microkernel: Our goal for the outer loop of the microkernel is to partition all computational dimensions so that the kernel can be called most efficiently. The computational loop at this level must promote optimal memory movement in different memory hierarchies. To achieve this, we carefully analyze and decide at this level which data should be placed at which level of the memory hierarchy to minimize memory movement between levels. Based on this, we divide and sort the loops for each computational dimension accordingly. The individual dimensions of the convolutional computation can be divided into smaller blocks, i.e., tiles, to fit the size of the required memory level. This allows the computational blocks to be processed without further access to lower levels of memory, thereby maximizing data reuse in a given memory hierarchy. The benefits of optimizing the external loop are obvious. By dividing the batches, output channels, and spaces, and using the cache hierarchy of the ARM processor, the external loop division can fully utilize memory resources and improve cache efficiency.

Convolutional computation microkernel: Our algorithm sets up a microkernel dedicated to computation, using assembly instructions to carefully prefetch data into the processor vector registers to overlap communication and computation. For the kernel to compute the output, the input is accessed sequentially during the microkernel computation. During the computation, since the input and convolution kernel matrices can be accessed row by row, the cache prefetch mechanism can largely hide the access latency of the main memory. The goal of the computation kernel is to provide a highly optimized computational block to repeatedly complete our algorithm. In this case, it is a convolution computation with a fixed set of computational dimensions. In the microkernel, the NEON SIMD extension of the ARMv8 architecture provides strong support for vectorized computing. The instruction-level optimization and data prefetching strategy for SIMD can significantly improve computing efficiency and reduce memory access latency.

3.2. Loops Outside the Microkernel

Loop tiling: The outermost loop traverses the input batch, i.e., N. The results of the batch do not affect the calculation of small blocks of data and can be prepared for subsequent parallelization. The second outermost loop traverses the output channels, and the number of channels processed each time is controlled by the parameter Kb. The purpose of this design is to help ensure that data can be cached in a faster cache when processing small batches of channels. The outermost loops Ho and Wo loops process the height and width of the output feature map. In particular, in the width loop, by processing 7 width units of data each time, this consideration will help keep the necessary input and output data in the cache. Next is the inner C loop. The input data are laid out in NHWC format. According to the principle of spatial locality, we place the loop on the channel close to the innermost loop. To fit the cache at all levels, the Cb parameter is used to control when traversing the input channels. The key to choosing a suitable Cb value is to balance the amount of data loaded into the cache at each iteration to avoid cache overflow and maximize cache utilization. Based on the above discussion, we can clearly understand the loop structure outside the microkernel, and its specific implementation process is clearly shown in Algorithm 2.

Algorithm 2 New direct convolution

Input: I[N][H][W][C]; F[R][S][C][K];
Output: O[N][Ho][Wo][Co]

```
for (n = 0; n < N; N++) do
  for (k = 0; k < Kb; k+ = Kb) do
    for (h = 0; h < Ho; h++) do
      for (w = 0; w < Wo; w+ = mr) do
        for (c = 0; c < C; c+ = Cb) do
          for (kk = 0; kk < Kb; kk+ = nr) do
            Micro_kernel;
```

Cache strategy: In the inner deepest loop, the microkernel is responsible for performing the actual convolution operation. The microkernel processes relatively small blocks of data (related to the fixed size of the Cb value and the output width), which helps keep active data in the L1 cache. This is because the L1 cache is small but has extremely fast access speed, which is suitable for small datasets that are frequently accessed. Since the elements of the filter are converted to a continuous memory space, the processor accesses the filter data in a pipelined manner, which has a fast access speed. The input data I can be kept in the L1 level cache and the iteration size of the filter data in the L1 cache can be determined by adjusting the Kb size. In the L2 level cache, the size of the filter data in the L2 cache can be determined by adjusting the size of Kb and Cb. By controlling the size of Kb and Cb, the size of the input data I and the filter F processed in each iteration can be guaranteed to fit in the L2 cache. Since the L3 cache is usually a shared cache, it can store larger datasets, such as the entire input or a large part of the output feature map. Properly dividing the block sizes of Kb Ho and Wo in the outer loop helps the overall data scheduling and management, ensuring that the data are already in the L3 cache before entering the processing, reducing the number of main memory accesses.

3.3. Microkernel Design

Our study targets the ARMv8-A [49] architecture, which is the standard for most mobile device platforms, and the optimization approach is also applicable to other hardware platforms. In the ARMv8-A architecture, the key feature that is closely related to the performance of our algorithm is its vector processing capability, especially the NEON SIMD extension. NEON technology provides 32 128-bit vector registers, supporting 8-bit, 16-bit, 32-bit, and 64-bit data modes. In particular, when operating on 32-bit floating-point numbers, these registers can store 32 vectors containing 4 consecutive elements, that is, a total of 128 elements. NEON technology also supports floating-point multiply-add (FMA) operations, which is an optimized hardware implementation of multiply-accumulate (MAC) operations. To fully utilize the computational pipeline of ARMv8-A, a sufficient number of independent FMA instructions are required. The specific number of instructions required in the computation process depends on the specific microarchitecture, but for optimal performance, the output O can be set to $Co \times Wo \geq 32$. Such a configuration not only provides enough registers for FMA instructions, but also allows load and store instructions to be interleaved with FMA instructions, achieving parallel processing of memory operations and calculations, and maximizing the utilization of instruction-level parallelism (ILP) of the ARMv8-A architecture.

Vectorization: Our algorithm is committed to improving data reuse in the direct convolution process and enhancing instruction-level parallel processing capabilities by adopting the ARM NEON SIMD extension. The core of the algorithm is to make full use of the 32 128-bit wide vector registers and the arithmetic fused multiply-accumulate (FMA) unit provided by the ARMv8 CPU. The main challenge is to choose the appropriate vector

parameter configuration to maximize register reuse. We can use analytical methods to determine the appropriate parameters. In convolution calculations, all vector registers can be used to store single-precision floating-point elements of input, filter, and output tensors. To ensure that the required data can be loaded into the available vector registers, the following conditions must be met:

$$\begin{cases} \frac{Co \times Wo}{4} + \frac{Kb}{4} + \frac{Wo \times Ci}{4} \leq 32 \\ Kb\%4 = 0 \\ Ci\%4 = 0 \end{cases} \tag{3}$$

Since each vector register can hold four FP32 elements and the data layout of input I is NHWC, the length Ci is set to a multiple of four to make full use of the vector unit. Figure 1 shows the input and output forms of the data layout. Data are arranged according to the channel dimension first, and different channel data at the same position are stored first, and then the data at subsequent spatial positions are processed in sequence. In order to make full use of the continuity of memory and improve data access efficiency, we set Ci to 4 during the calculation process of the microkernel. Then, according to Formula (3), we can set the algorithm's calculation kernel parameters to 7×12, specifically: 21 vector registers are used to store the output data. Seven vector registers are used to store the seven columns of the input, each column containing four elements of C. Three vector registers are used to store the twelve elements of the filter F.

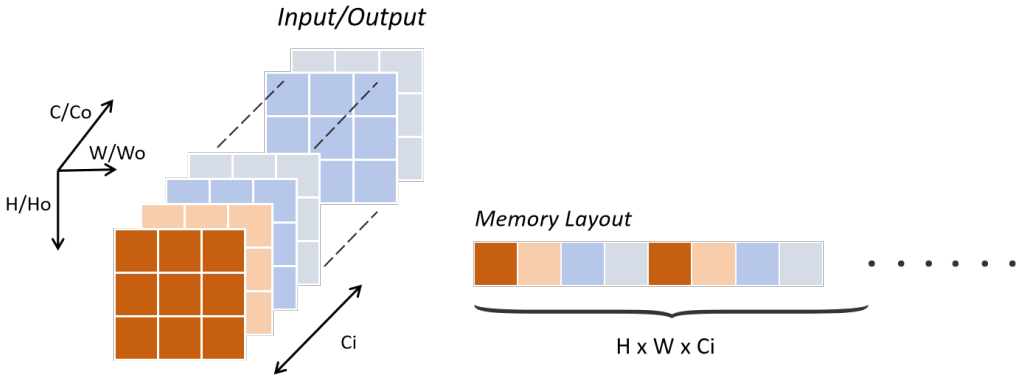


Figure 1. Data layout diagram.

Optimizing cache constraints: We have determined the structure and dimensions of the loop hierarchy in Algorithms 2 and 3. Based on the principle of data locality, we use static analysis methods to determine the block dimensions of each loop level to achieve co-optimization of the ARM architecture memory system. Our design aims to take advantage of the vector FMA unit while taking advantage of the memory hierarchy of caches and vector registers. To optimize L1 data cache utilization, during each iteration of loop 6 of Algorithm 2, each $mr \times nr \times Ci$ of input I should be kept in the L1 cache. Additionally, the L1 cache should also hold $Kb \times Ci \times R \times S$ of the filter variable at this loop level. Therefore, the variable C_{L1} representing the size of cache L1 satisfies the following constraints:

$$Kb \times Ci \times R \times S + mr \times nr \times Ci \leq C_{L1}. \tag{4}$$

Algorithm 3 Micro_kernel

Input: I; Filter: F;

```
for (r = 0; r < R; r++) do
  for (s = 0; s < S; s++) do
    for (cc = 0; cc < Cb; cc+ = 4) do
    V22 − V28 ← I
    V29 − V31 ← F[F : F + 12]
    (V1 − V21) ← FMA(V22 − 28[0], V29 − 31)
    V29 − V31 ← F[F + 12 : F + 24]
    (V1 − V21) ← FMA(V22 − 28[1], V29 − 31)
    V29 − V31 ← F[F + 24 : F + 36]
    (V1 − V21) ← FMA(V22 − 28[2], V29 − 31)
    V29 − V31 ← F[F + 36 : F + 48]
    (V1 − V21) ← FMA(V22 − 28[3], V29 − 31)
    Store to Output
```

Similarly, during each iteration of loop 5 in Algorithm 2, we keep the L2 cache holding a block of size $Kb \times R \times S \times Cb$ in the filter variable and a block of size $mr \times Kb \times Cb$ in the input I. Because the L2 cache on ARM CPUs often stores both data and instructions, the cache needs to reserve some space for the instructions being executed and the data elements of O, and we introduce a reservation factor α. Therefore, the variable C_{L2} representing the cache L2 size satisfies the constraint:

$$Kb \times Cb \times R \times S + mr \times Kb \times Cb \leq \alpha \times C_{L2} \tag{5}$$

The optimal parameters of Kb and Cb can be derived according to Equations (4) and (5). This analysis method is cross-platform scalable and can complete parameter adaptation at the compilation stage without introducing runtime overhead by precalculating the cache capacity and number of vector registers of the target platform.

Computation process: The microkernel design implements a complete convolution calculation process, including data loading, weight application, and result storage. This design approach allows the microkernel to be used as an independent computing unit and can be reused in different contexts, improving the modularity and maintainability of the code. In the calculation of the microkernel, the outer product calculation method is used. The calculation contains three nested loops, corresponding to the rows R, columns S, and channels Ci of the convolution kernel, which are used to control the execution of the loop. Corresponding to the input format of NHWC, when iterating along the R loop, the input data row stride is four times C. When iterating along the S loop, the input data row stride is four times $W \times C$. The innermost loop is performed along the channel direction, and each loop iteration completes the calculation of four channels. Figure 2 shows the calculation process for four channels. In the iteration, the complete output is stored using 21 vector registers. During the execution of the microkernel, the filter tensor is converted to a continuous memory space. The benefits of this are obvious. The processor accesses the filter data in a pipelined manner. Such a conversion only needs to be performed once because the filter of the DNN model does not change during the inference process. The general structure of the microkernel is shown in Algorithm 3, which shows the loop process. To ensure compatibility between outputs and inputs of consecutive layers in convolutional DNNs, the input tensors adopt the same layout; that is, the output data are organized as $N \times Ho \times Wo \times K$.

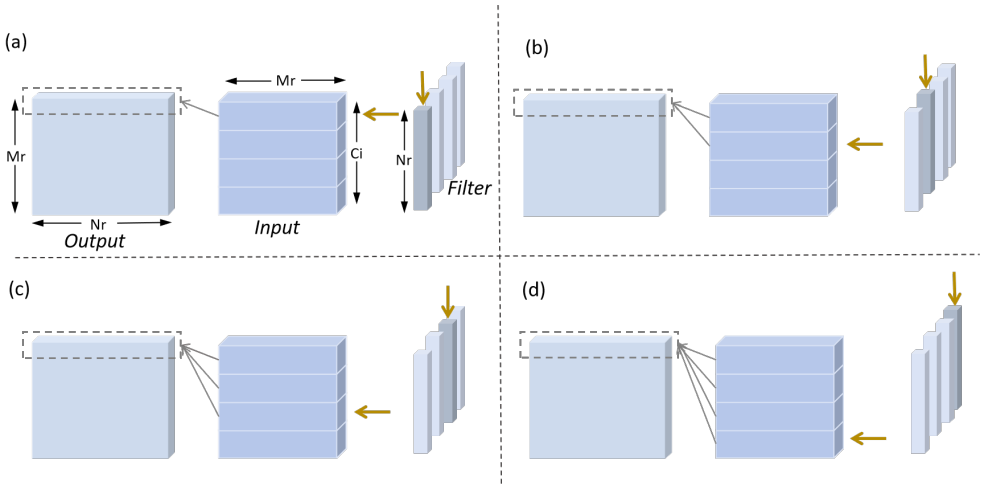


Figure 2. Four stages of the inner loop of the microkernel. From (**a**–**d**), calculations are performed on different channels. The input data are arranged along the channel direction, and the convolution kernel is iterated in a sliding manner. The convolution kernel and the input data are multiplied and added element-wise in the channel direction. Each convolution kernel produces a local output value, which will be combined to form the final output feature map.

Pipeline optimization: Pipeline optimization is one of the key technologies to improve the performance of microkernels. Under the ARMv8 architecture, pipeline optimization can alternate calculation and data fetch operations according to the characteristics of instructions, thereby reducing data dependence and improving instruction-level parallelism. The key to pipeline optimization is to arrange the execution order of instructions reasonably so that calculation and data fetch operations can be executed in parallel. We carefully optimize instructions so that when one instruction is performing calculations, the next instruction can start fetching operations. This optimization can reduce waiting time and improve processor utilization. Data prefetching is a common optimization technique that loads data into the cache in advance by predicting the future memory access pattern of the program. Cache optimization can improve cache efficiency by adjusting the cache size and replacement strategy. In the microkernel, data prefetching is performed on input data and convolution kernels to optimize data access in memory and improve cache hit rate.

Parallelization strategy: To achieve efficient parallel computing on multi-core architectures, we can use OpenMP with static work partitioning to parallelize CONV operations. To exploit hardware parallelism, we use all available CPU cores. Ideally, all cores will start and finish work at the same time, so no core is idle at any point in time. However, this is not always possible due to memory access latency and application workload. Therefore, we need to carefully determine how to parallelize the dimensions that can be parallelized. By analyzing the computational characteristics of Algorithm 2, the parallelizable dimensions include the batch dimension N and the output feature map spatial dimensions H and W. Parallelization of these three dimensions maintains the spatial locality of data access between threads and is conducive to the effective use of L2 cache lines. We will not parallelize the reduced dimensions because doing so will cause write conflicts because all participating threads will write to the same location in the output O.

4. Experimental Design and Evaluation

There are various convolution solutions in ARM, and we include widely used solutions and competitive solutions in our tests for comparison. Our experiments will be conducted on a high-performance computing system.

4.1. Test Data and Benchmarks

Our evaluation is conducted on the Kunpeng 920 platform [50]. Figure 3 shows the architectural design of the KunPeng 920 CPU in detail. The processor contains multiple cores, each with a main frequency of 2.6 GHz and equipped with independent L1 and L2 caches. The L1 cache is divided into instruction cache and data cache, each with a capacity of 64 KB, which is used to accelerate the instruction processing and data access of the core. The capacity of the L2 cache is 512 KB, which further improves the computing efficiency of a single core. In addition, all cores share a unified L3 cache (L3 Cache) with a capacity of 128 MB. This design helps to reduce data transmission delays between cores and improve overall performance. The structure in the figure reflects the characteristics of KunPeng 920 in performance optimization and cache hierarchy design, which is suitable for application scenarios of high-performance computing and large-scale data processing. In order to fully verify the performance of our algorithm in practical applications and analyze its performance changes under different data scales, we will use these features of the KP920 to conduct a performance evaluation.

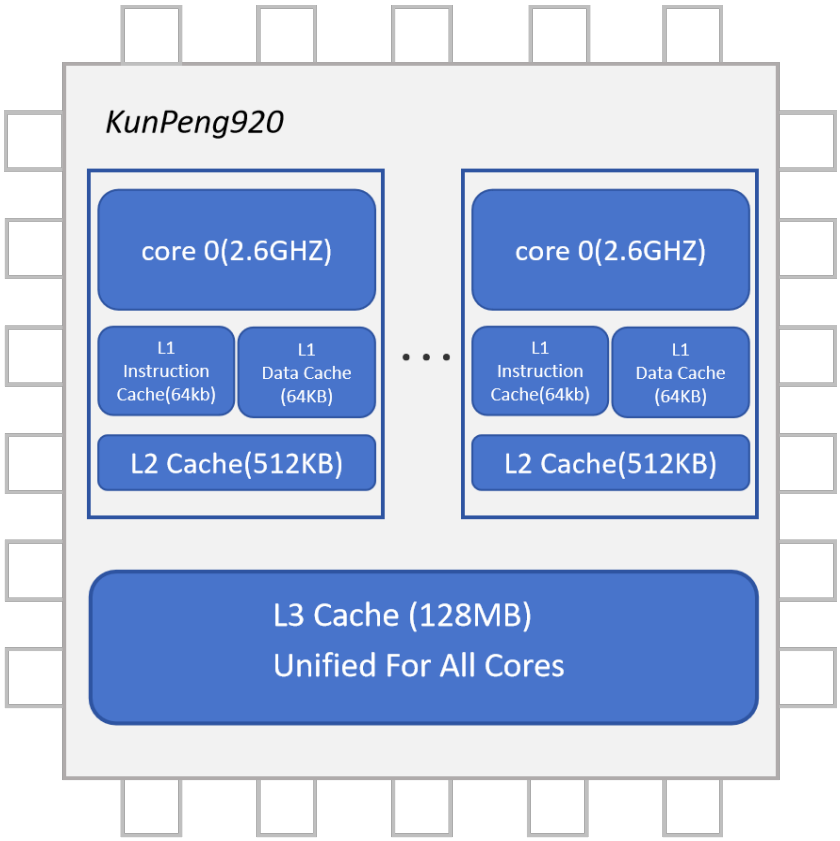


Figure 3. KunPeng 920 CPU architecture diagram.

4.2. Hardware and Software Environment

We tested our algorithm against various existing GEMM-based convolutions. IM2COL +OpenBLAS is our original IM2COL code based on MXNet [51]. We implemented the IM2COL routine in C language and piped the result into the high-performance GEMM routine provided by the OpenBLAS [43] library. This approach of combining OpenBLAS with IM2COL routine has been used as a reference solution in many recent studies [47,48,52–54]. It provides a low-effort and good-performance implementation of GEMM-based convolutions. The direct convolution provided by LIBXSMM is a small GEMM-based microkernel generated by JIT. It requires converting the input tensor to a specified format. We used LIBXSMM version 1.17. In our algorithm, C language is used

as the main implementation, the design of the microkernel is implemented through arm inline assembly, and the compilation tool chain uses GCC 12.2.0 compiler and enables -O2 optimization level.

GFLOPS is calculated as the number of single floating-point operations, given by $2 \times N \times Ho \times Wo \times C \times R \times S \times K$, divided by the clock time of the respective convolution procedure on the entire image batch. The results given for each experiment are the average of 20 runs. In the experiments, we selected data of different sizes to ensure that we can fully evaluate the performance of our algorithm and its performance under different computational requirements. Table 2 lists the specific parameter settings of the experiment. Among them, the convolution step size is set to 1, and the convolution kernel size is 3×3.

Table 2. Configuration parameters of the convolution operator.

ID	C	K	H/W	ID	C	K	H/W
1	1024	2400	16	18	256	240	16
2	1024	1200	58	19	64	240	9
3	512	1200	58	20	64	240	58
4	256	1200	30	21	64	240	9
5	512	1200	30	22	1024	120	58
6	256	1200	30	23	512	120	30
7	512	1200	16	24	1024	120	16
8	256	1200	16	25	128	120	30
9	128	1200	58	26	256	120	9
10	1024	480	58	27	64	120	9
11	512	480	58	28	1024	48	58
12	512	480	30	29	1024	48	30
13	256	480	30	30	512	48	58
14	128	480	9	31	256	48	30
15	64	480	58	32	128	48	9
16	1024	240	58	33	64	48	58
17	512	240	30	34	32	48	226

4.3. Performance Analysis

Figure 4 shows the performance comparison of our algorithm, IM2COL+OpenBLAS, and LIBXSMM on different test cases, using GFLOPS as the unit to measure the efficiency of each algorithm in processing floating-point operations. It can be seen from the experimental data that our algorithm shows good performance under different data scales. In particular, our algorithm has significant performance advantages when processing large-scale data, especially in scenarios requiring high-throughput computation. When processing smaller data, some experimental data are not as good as LIBXSMM, but in most cases, it is worse than LIBXSMM. Under different data scales, our algorithm can maintain a high GFLOPS value, which shows that the algorithm has good scalability. Especially when the data scale is large, our algorithm can reach 28.97 GFLOPS, while IM2COL+OpenBLAS is 26.83 and LIBXSMM is 26.25. This result demonstrates the advantage of our algorithm in processing large-scale data, especially in applications such as deep learning that require massively parallel computing. In our evaluation system, our algorithm provides 1.43 times and 1.1 times geometric mean speedup compared with IM2COL+OpenBLAS and LIBXSMM, respectively, confirming the effectiveness of our optimization.

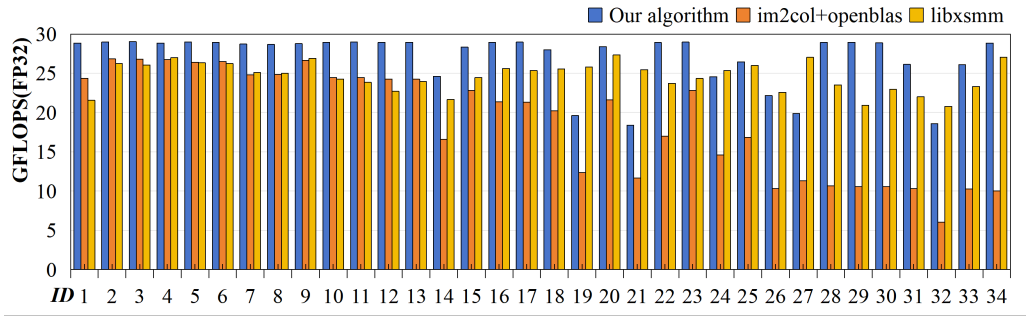


Figure 4. Algorithm performance comparison (GFLOPS).

Figure 5 shows the speedup of our algorithm and the IM2COL+OpenBLAS on different test cases. The speedup is normalized relative to the performance of the IM2COL+OpenBLAS to measure the efficiency advantage of our algorithm. It can be seen from the data that our algorithm achieves acceleration of IM2COL+OpenBLAS under multiple data scales. The improvement of the speedup ratio means that the execution efficiency of the algorithm has been improved, which is usually related to the optimization strategy of the algorithm. Our algorithm achieves a speedup over the baseline algorithm by reducing unnecessary memory accesses, improving data locality and parallelism, and leveraging the SIMD instruction set of modern processors. Under different data scales, the change in speedup ratio also reflects the adaptability of the algorithm to different computing loads. For example, the speedup ratio is getting increasingly larger when the data scale is small, which shows that our algorithm can provide significant performance improvement when processing small-scale data. This result proves that our algorithm is effective in practical applications.

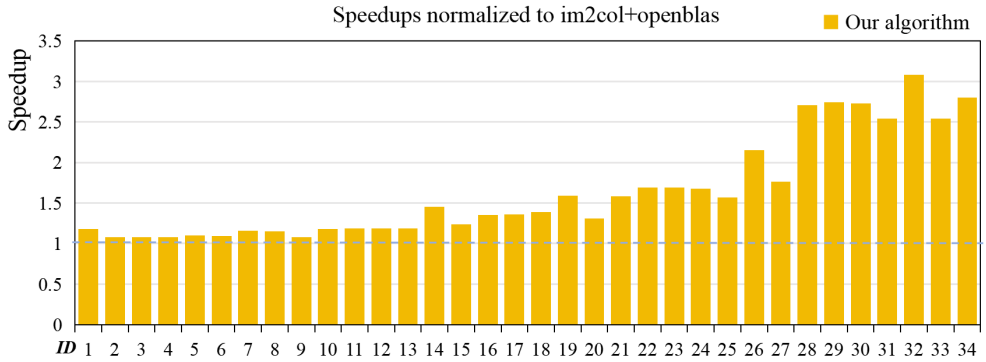


Figure 5. Comparison of algorithm speedup with IM2COL+OpenBLAS.

Figure 6 shows the percentage of CPU peak performance achieved by our algorithm, IM2COL+OpenBLAS, and LIBXSMM on different test cases. These percentages are used to measure the utilization of each algorithm relative to the theoretical peak performance of the CPU. The peak performance percentage refers to the ratio of the actual performance of an algorithm to its theoretical peak performance and is usually used to measure the efficiency of the algorithm's use of hardware resources. As can be seen from the data, our algorithm is close to or exceeds the peak performance percentage of IM2COL+OpenBLAS at multiple data scales. Our algorithm can more effectively utilize the computing resources of the hardware platform and improve data processing speed by reducing memory access latency and making reasonable use of the third-level cache. It can be seen from the experimental data that our algorithm shows good performance under multiple data scales. Especially on large-scale datasets, our algorithm can better utilize hardware resources, thus achieving high-performance computing. This result demonstrates that our algorithm

optimization strategies, such as loop unrolling, data prefetching, and utilization of SIMD instruction sets, can effectively improve computational efficiency, reduce memory access latency, and increase data processing speed. Experiments show that our algorithm also has good scalability. As the data size increases, the performance of the algorithm improves more significantly. This shows that our algorithm can adapt to different computing needs, especially in scenarios where large-scale data need to be processed. This feature is particularly important for applications such as deep learning that require processing large-scale data. Although our algorithm performs well in most cases, its performance improvement is not obvious on some small-scale datasets. Future work can explore how to optimize the processing of small-scale datasets to further improve the performance of the algorithm.

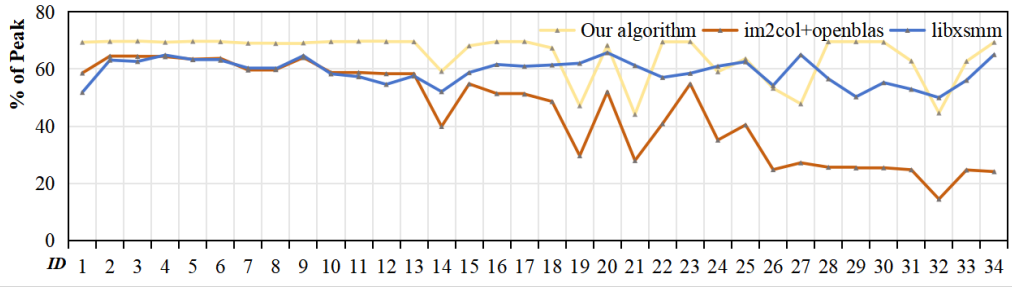


Figure 6. Comparison of algorithm performance as a percentage of CPU peak performance.

In summary, our algorithm shows excellent performance on the KP920 hardware platform, especially on large-scale datasets. The experimental results prove that our algorithm can effectively achieve high-performance computing. Although there are some limitations on small-scale datasets, overall, our algorithm provides an effective solution for convolution operations in deep learning. Future work can further optimize the algorithm to improve its performance on small-scale datasets and explore the application of the algorithm on other hardware platforms.

5. Discussion

The direct convolution optimization method proposed in this study for the ARMv8 architecture shows significant performance improvement in the forward propagation of convolutional neural networks. However, its potential application areas go far beyond this, and we will explore the possible application of this method in the backward pass stage of convolution computation, as well as its potential for extension in terms of different data layouts and multi-threaded parallelization.

In deep learning, the backpropagation algorithm is an integral part of training neural networks, which is responsible for calculating the gradient of the loss function concerning the network parameters. The convolutional layer in this process also requires a large number of convolution calculations, especially gradient calculations. The optimization method proposed in this study has been initially verified in the forward propagation stage by improving the memory access mode and calculation strategy of convolution operations. Future work will explore applying this optimization technique to the backward pass phase, which is expected to similarly improve the efficiency of gradient computation. This will not only help speed up the network training process but also potentially further reduce energy consumption, especially on mobile and embedded devices using the ARMv8 architecture.

Current optimization methods are mainly designed for the standard NHWC data layout. Different data layouts affect the memory access pattern of data, which has a significant impact on computing performance. In order to make this optimization method more universal and flexible, future research will consider supporting more types of data

layouts. This will enable this optimization method to be more widely applied to different frameworks and hardware platforms, further enhancing its practical value.

Although the ARMv8 architecture provides powerful multi-core processing capabilities, the current implementation of this study is mainly single-threaded. In order to make full use of hardware resources and achieve higher computing efficiency, future research will focus on the development of multi-threaded parallelization technology. The performance of parallel processing can be significantly improved by designing thread-safe data access mechanisms and optimizing workload distribution among threads. In addition, combined with the unique heterogeneous multi-core processing characteristics of the ARMv8 architecture, we can further explore the collaborative working mode between different cores to achieve the optimal energy efficiency ratio. Through these future works, we hope to extend the optimization techniques developed in this study to a wider range of application scenarios, not only limited to improving computing performance, but also including comprehensive improvements in energy efficiency, scalability, and versatility. This will provide more powerful and flexible support for ARMv8-based systems in deep learning applications.

6. Conclusions

In this study, we proposed a direct convolution optimization method for the ARMv8 architecture, which improves the efficiency and performance of convolution operations at multiple levels. Our method first tilings the loop of a naive direct convolution. Tiling technology reduces function call overhead and improves cache hit rates by breaking large loops into smaller, more manageable loop bodies. These optimizations significantly reduce the latency between loop iterations and improve the continuity of instruction execution, thereby improving overall computing performance. Based on loop optimization, we developed an efficient computing microkernel. This microkernel is designed specifically for the ARMv8 architecture, taking full advantage of its advanced SIMD instruction set and multi-core processing capabilities. The ideas and methods of designing microkernels can also be applied to other hardware architectures.

By comparing the speedup of our algorithm on different test cases, it can be shown that the speedup is achieved at multiple data scales, up to three times or more. The improvement of the speedup ratio means that the execution efficiency of the algorithm is improved, which is usually related to the optimization strategy of the algorithm. Our algorithm improves data locality and modern processors that utilize SIMD instructions to achieve speedup over the baseline algorithm. The change in speedup ratio at different data scales also reflects the adaptability of the algorithm to different computational loads. The results show that the algorithm is effective in practical applications. Our microkernel significantly improves the throughput of convolution operations while maintaining computational accuracy. Therefore, our microkernel significantly improves the throughput of convolution operations while maintaining computational accuracy. Through in-depth analysis and careful design of convolution algorithms, our microkernel demonstrates excellent performance when processing large-scale data, especially when dealing with high-dimensional convolution kernels and feature maps. During the optimization process, we paid special attention to preserving the standard data format. This means that our optimization method not only improves performance but also maintains compatibility with existing deep learning frameworks and libraries. This compatibility ensures that our optimization technology can be seamlessly integrated into existing systems without the need for additional conversion or adaptation of data formats. This not only simplifies the deployment process but also facilitates future expansion and maintenance. Our research results not only improve the performance of direct convolution but also take into account compatibility and ease of use

in practical applications. Our approach provides an efficient and flexible solution for the deployment of deep learning on the ARMv8 architecture, which is of great significance for promoting the application of deep learning technology on mobile and embedded devices.

Author Contributions: Conceptualization, S.L. and F.Y.; methodology, S.L. and F.Y.; software, H.L.; validation, S.L., F.Y., and H.Y.; formal analysis, H.L.; investigation, H.Y.; resources, F.Y. and S.Z.; data curation, S.L.; writing—original draft preparation, S.L. and F.Y.; writing—review and editing, S.L. and F.Y.; visualization, H.L. and S.Z.; supervision, H.Y. and H.L.; project administration, S.L.; funding acquisition, H.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Key Research and Development Program "National Quality Infrastructure System" Special Project under grant no. 2024YFF0617900 and by the Hefei Minglong Electronic Technology Co., Ltd. under Grants 2024ZKHX293, 2024ZKHX294, and 2024ZKHX295.

Data Availability Statement: The data are available from the corresponding author on reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Hu, J.; He, Y.; Luo, W.; Huang, J.; Wang, J. Enhancing Load Balancing with In-Network Recirculation to Prevent Packet Reordering in Lossless Data Centers. *IEEE/ACM Trans. Netw.* **2024**, *32*, 4114–4127. [CrossRef]
2. Iqbal, M.S.; Chen, C. Instant queue occupancy used for automatic traffic scheduling in data center networks. *Comput. Netw.* **2024**, *244*, 110346. [CrossRef]
3. Hu, J.; Zhou, Z.; Zhang, J. Lightweight Automatic ECN Tuning Based on Deep Reinforcement Learning with Ultra-Low Overhead in Datacenter Networks. *IEEE Trans. Netw. Serv. Manag.* **2024**, *21*, 6398–6408. [CrossRef]
4. Lu, Y.; Ma, X.; Cui, C. DCCS: A dual congestion control signals based TCP for datacenter networks. *Comput. Netw.* **2024**, *247*, 110457. [CrossRef]
5. Yu, F.; Zhang, Z.; Su, D.; Wu, Y.; Gracia, Y.M.; Yin, H. Dynamic Analysis and Implementation of FPGA for a New 4D Fractional-Order Memristive Hopfield Neural Network. *Fractal Fract.* **2025**, *9*, 115. [CrossRef]
6. Wang, J.; Liu, Y.; Rao, S.; Zhou, X.; Hu, J. A novel self-adaptive multi-strategy artificial bee colony algorithm for coverage optimization in wireless sensor networks. *Ad Hoc Netw.* **2023**, *150*, 103284. [CrossRef]
7. Ramalingam, S.; Dhanasekaran, S.; Sinnasamy, S.S.; Salau, A.O.; Alagarsamy, M. Performance enhancement of efficient clustering and routing protocol for wireless sensor networks using improved elephant herd optimization algorithm. *Wirel. Netw.* **2024**, *30*, 1773–1789. [CrossRef]
8. Yu, X.; Fu, Y.; Li, J.; Mao, J.; Hoang, T.; Wang, H. Recent advances in wireless sensor networks for structural health monitoring of civil infrastructure. *J. Infrastruct. Intell. Resil.* **2024**, *3*, 100066. [CrossRef]
9. Zhong, R.; Peng, F.; Yu, J.; Munetomo, M. Q-learning based vegetation evolution for numerical optimization and wireless sensor network coverage optimization. *Alex. Eng. J.* **2024**, *87*, 148–163. [CrossRef]
10. Yu, F.; Lin, Y.; Yao, W.; Cai, S.; Lin, H.; Li, Y. Multiscroll hopfield neural network with extreme multistability and its application in video encryption for IIoT. *Neural Netw.* **2025**, *182*, 106904. [CrossRef]
11. He, S.; Li, G.; Yi, T.; Alfarraj, O.; Tolba, A.; Kumar Sangaiah, A.; Simon Sherratt, R. Graph Structure Learning-Based Multivariate Time Series Anomaly Detection in Internet of Things for Human-Centric Consumer Applications. *IEEE Trans. Consum. Electron.* **2024**, *70*, 5419–5431. [CrossRef]
12. Lin, H.; Deng, X.; Yu, F.; Sun, Y. Diversified Butterfly Attractors of Memristive HNN with Two Memristive Systems and Application in IoMT for Privacy Protection. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2025**, *44*, 304–316. [CrossRef]
13. Yu, F.; Wu, C.; Xu, S.; Yao, W.; Xu, C.; Cai, S.; Wang, C. Color video encryption transmission in IoT based on memristive hopfield neural network. *Signal Image Video Process.* **2025**, *19*, 77. [CrossRef]
14. Amin, A.A.; Iqbal, M.S.; Shahbaz, M.H. Development of intelligent fault-tolerant control systems with machine learning, deep learning, and transfer learning algorithms: A review. *Expert Syst. Appl.* **2024**, *238*, 121956. [CrossRef]
15. Zhang, J.; Qin, Y.; Fan, S.; Xiao, Z.; Zhang, J. SiamTFA: Siamese Triple-Stream Feature Aggregation Network for Efficient RGBT Tracking. *IEEE Trans. Intell. Transp. Syst.* **2024**, *26*, 1900–1913. [CrossRef]
16. Aldhaheri, A.; Alwahedi, F.; Ferrag, M.A.; Battah, A. Deep learning for cyber threat detection in IoT networks: A review. *Internet Things Cyber-Phys. Syst.* **2024**, *4*, 110–128. [CrossRef]
17. Wu, H.; Huang, P.; Zhang, M.; Tang, W.; Yu, X. CMTFNet: CNN and Multiscale Transformer Fusion Network for Remote-Sensing Image Semantic Segmentation. *IEEE Trans. Geosci. Remote Sens.* **2023**, *61*, 1–12. [CrossRef]

18. Horng, S.J.; Supardi, J.; Zhou, W.; Lin, C.T.; Jiang, B. Recognizing very small face images using convolution neural networks. *IEEE Trans. Intell. Transp. Syst.* **2020**, *23*, 2103–2115. [CrossRef]
19. Qiao, S.; Ma, J. Fpga implementation of face recognition system based on convolution neural network. In Proceedings of the 2018 Chinese Automation Congress (CAC), Xi'an, China, 30 November–2 December 2018; pp. 2430–2434.
20. Wang, S.; Sun, J.; Mehmood, I.; Pan, C.; Chen, Y.; Zhang, Y.D. Cerebral micro-bleeding identification based on a nine-layer convolutional neural network with stochastic pooling. *Concurr. Comput. Pract. Exp.* **2020**, *32*, e5130. [CrossRef]
21. Wang, H.; Wang, Y.; Zhou, Z.; Ji, X.; Gong, D.; Zhou, J.; Li, Z.; Liu, W. Cosface: Large margin cosine loss for deep face recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 5265–5274.
22. Ullah, F.; Long, Y.; Ullah, I.; Khan, R.U.; Khan, S.; Khan, K.; Khan, M.; Pau, G. Deep hyperspectral shots: Deep snap smooth wavelet convolutional neural network shots ensemble for hyperspectral image classification. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2023**, *17*, 14–34. [CrossRef]
23. Esteva, A.; Kuprel, B.; Novoa, R.A.; Ko, J.; Swetter, S.M.; Blau, H.M.; Thrun, S. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* **2017**, *542*, 115–118. [CrossRef] [PubMed]
24. Arkin, E.; Yadikar, N.; Xu, X.; Aysa, A.; Ubul, K. A survey: Object detection methods from CNN to transformer. *Multimed. Tools Appl.* **2023**, *82*, 21353–21383. [CrossRef]
25. Cao, Y.H.; Wu, J. A random cnn sees objects: One inductive bias of cnn and its applications. In Proceedings of the AAAI Conference on Artificial Intelligence, Virtual, 22 February–1 March 2022; Volume 36, pp. 194–202.
26. Shih, K.H.; Chiu, C.T.; Lin, J.A.; Bu, Y.Y. Real-time object detection with reduced region proposal network via multi-feature concatenation. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *31*, 2164–2173. [CrossRef]
27. Zhao, R.; Niu, X.; Wu, Y.; Luk, W.; Liu, Q. Optimizing CNN-based object detection algorithms on embedded FPGA platforms. In Proceedings of the Applied Reconfigurable Computing: 13th International Symposium, ARC 2017, Delft, The Netherlands, 3–7 April 2017; Proceedings 13; Springer: Berlin/Heidelberg, Germany, 2017; pp. 255–267.
28. Jung, C.; Han, Q.; Zhou, K.; Xu, Y. Multispectral fusion of rgb and nir images using weighted least squares and convolution neural networks. *IEEE Open J. Signal Process.* **2021**, *2*, 559–570. [CrossRef]
29. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [CrossRef]
30. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet classification with deep convolutional neural networks. *Commun. ACM* **2017**, *60*, 84–90. [CrossRef]
31. Huang, G.; Liu, Z.; Van Der Maaten, L.; Weinberger, K.Q. Densely connected convolutional networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017; pp. 4700–4708.
32. Simonyan, K. Very deep convolutional networks for large-scale image recognition. *arXiv* **2014**, arXiv:1409.1556.
33. Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 1–9.
34. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
35. Chellapilla, K.; Puri, S.; Simard, P. High performance convolutional neural networks for document processing. In Proceedings of the Tenth International Workshop on Frontiers in Handwriting Recognition, La Baule, France, 1 October 2006.
36. Libxsmm. Available online: https://github.com/libxsmm/libxsmm (accessed on 4 January 2025).
37. libxsmm-dnn. Available online: https://github.com/libxsmm/libxsmm-dnn (accessed on 4 January 2025).
38. Santana, A.d.L.; Armejach, A.; Casas, M. Efficient direct convolution using long SIMD instructions. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, Montreal, QC, Canada, 25 February 2023; pp. 342–353.
39. Georganas, E.; Avancha, S.; Banerjee, K.; Kalamkar, D.; Henry, G.; Pabst, H.; Heinecke, A. Anatomy of high-performance deep learning convolutions on simd architectures. In Proceedings of the SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, 11–16 November 2018; pp. 830–841.
40. Sridhar, U.; Tukanov, N.; Binder, E.; Low, T.M.; McMillan, S.; Schatz, M.D. Small: A software framework for portable machine learning libraries. *arXiv* **2023**, arXiv:2303.04769.
41. Zhao, T.; Hu, Q.; He, X.; Xu, W.; Wang, J.; Leng, C.; Cheng, J. ECBC: Efficient convolution via blocked columnizing. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *34*, 433–445. [CrossRef] [PubMed]
42. Fang, J.; Fu, H.; Zhao, W.; Chen, B.; Zheng, W.; Yang, G. swdnn: A library for accelerating deep learning applications on sunway taihulight. In Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, USA, 29 May–2 June 2017; pp. 615–624.
43. Zhang, X. Openblas: An Optimized Blas Library. 2021. Available online: https://www.openblas.net (accessed on 4 January 2025).

44. Yang, W.; Fang, J.; Dong, D.; Su, X.; Wang, Z. LIBSHALOM: Optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, MO, USA, 14–19 November 2021; pp. 1–14.
45. Lavin, A.; Gray, S. Fast algorithms for convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 4013–4021.
46. Heinecke, A.; Henry, G.; Hutchinson, M.; Pabst, H. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In Proceedings of the SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 13–18 November 2016; pp. 981–991.
47. Wang, P.; Yang, W.; Fang, J.; Dong, D.; Huang, C.; Zhang, P.; Tang, T.; Wang, Z. Optimizing Direct Convolutions on ARM Multi-Cores. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 12–17 November 2023; pp. 1–13.
48. Zhang, J.; Franchetti, F.; Low, T.M. High performance zero-memory overhead direct convolutions. In Proceedings of the International Conference on Machine Learning, Stockholm, Sweden, 10–15 July 2018; PMLR 80, pp. 5776–5785.
49. Flur, S.; Gray, K.E.; Pulte, C.; Sarkar, S.; Sezgin, A.; Maranget, L.; Deacon, W.; Sewell, P. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, FL, USA, 20–22 January 2016; pp. 608–621.
50. Huawei. Kunpeng-920. Available online: https://www.hisilicon.com/cn/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920 (accessed on 4 January 2025).
51. Mxnet. Available online: https://github.com/apache/mxnet (accessed on 4 January 2025).
52. Anderson, A.; Vasudevan, A.; Keane, C.; Gregg, D. Low-memory gemm-based convolution algorithms for deep neural networks. *arXiv* **2017**, arXiv:1709.03395.
53. Huang, X.; Wang, Q.; Lu, S.; Hao, R.; Mei, S.; Liu, J. Evaluating fft-based algorithms for strided convolutions on armv8 architectures? *ACM Sigmetrics Perform. Eval. Rev.* **2022**, *49*, 28–29. [CrossRef]
54. Zhang, P.; Lo, E.; Lu, B. High performance depthwise and pointwise convolutions on mobile devices. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 6795–6802.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.