Towards Effective Depthwise Convolutions on ARMv8 Architecture

Ruochen Hao  
National University of Defense Technology  
Changsha, China  
hrc@nudt.edu.cn

Qinglin Wang  
National University of Defense Technology  
Changsha, China  
wangqinglin_thu@163.com

Shangfei Yin  
National University of Defense Technology  
Changsha, China  
yin_shangfei@163.com

Tianyang Zhou  
National University of Defense Technology  
Changsha, China  
ztyemail@126.com

Siqi Shen  
Xiamen University  
Xiamen, China  
siqishen@xmu.edu.cn

Songzhu Mei  
National University of Defense Technology  
Changsha, China  
meisongzhu@yeah.net

Jie Liu  
National University of Defense Technology  
Changsha, China  
liuje@nudt.edu.cn

Abstract

Depthwise convolutions are widely used in lightweight convolutional neural networks (CNNs). The performance of depthwise convolutions is mainly bounded by the memory access rather than the arithmetic operations for classic convolutions so that direct algorithms are often more efficient than indirect ones (matrix multiplication-, Winograd-, and FFT-based convolutions) with additional memory accesses. However, the existing direct implementations of depthwise convolutions on ARMv8 architectures feature a bad trade-off between register-level reuse of different tensors, which usually leads to sub-optimal performance. In this paper, we propose new direct implementations of depthwise convolutions by means of implicit padding, register tiling, etc., which contain forward propagation, backward propagation and weight gradient update procedures. Compared to the existing ones, our new implementations can incur much less communication overhead between registers and cache. Experimental results on two ARMv8 CPUs show that our implementations can averagely deliver 4.88x and 16.4x performance improvement over the existing direct ones in open source libraries and matrix multiplications-based ones in Pytorch, respectively.

CCS Concepts

• General and reference → Performance; • Computing methodologies → Parallel algorithms; Machine learning.

Keywords

CNNs, Depthwise Convolution, ARMv8, Direct Convolution, Parallel Algorithm

ACM Reference Format:
Ruochen Hao, Qinglin Wang, Shangfei Yin, Tianyang Zhou, Siqi Shen, Songzhu Mei, and Jie Liu. 2022. Towards Effective Depthwise Convolutions on ARMv8 Architecture. In Proceedings of ACM Conference (Conference’17). ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnn.nnnnnn

1 Introduction

Convolution Neural Networks (CNNs), a class of artificial neural networks, have achieved amazing success in various machine learning tasks, such as image classification [8], object detection [14], and medical image diagnostics [25]. The building blocks of CNNs mainly involve convolutional, pooling, normalization, and fully connected layers. In general, the training and inference of CNNs require a large quantity of computation and memory resource, which are primarily consumed by convolutional layers. The optimization of convolutional layers plays a vital role in improving the performance of CNNs.

Now, many lightweight models have been proposed for mobile computing systems, such as MobileNetV1 [9], MobileNetV2 [24] and MnasNet-A1 [26]. These models often consist of a type of convolutions that adopt a single filter for each channel of input feature maps, named depthwise convolutions. In comparison with typical convolutions, depthwise convolutions have much less arithmetic operations and fewer parameters for filters. The sharp reduction of the arithmetic complexity makes the performance of depthwise convolutions is basically bounded by the hierarchical memory access...
bandwidth rather than the peak performance on most platforms [36].

There are four common methods to perform convolutions, including matrix multiplication [12, 33], Winograd [13], Fast Fourier Transform (FFT) [10] and direct algorithms [35]. All the three indirect algorithms introduce the additional transformations, which increase the total overhead of memory access. As a result, the direct algorithm has become a good choice for high-performance depthwise convolutions due to its relatively less memory access.

Although GPUs are the main hardware platforms in deep learning fields, there are many factors to motivate CNNs running on resource-constrained systems including mobile devices (computational and energy constraints) and CPU-based servers (computational constraints relative to popular GPUs) [18]. In mobile computing systems, CPUs maybe perform better than GPU in terms of performance and power consumption. Among all the mobile CPUs, the ones based on the ARMv8 architecture have got the largest market share. Moreover, ARMv8 CPUs are rapidly appearing in high performance computing systems, e.g. Mont-Blanc prototype [22], Tianhe-3 prototype [34], and Fugaku supercomputer [17]. Therefore, it’s of great significance to optimize direct depthwise convolutions on ARMv8 CPUs.

In deep learning, the two most common data layouts on multicore CPUs are NHWC (mini-batch, height, weight, channel) and NCHW (mini-batch, channel, height, weight). The latter exhibits better data locality for convolutions so that it is the default layout for Caffe, Mxnet, and Pytorch frameworks [4, 12, 20]. But the depthwise convolutions with NCHW layout feature much more irregular memory access under the vectorized optimization, which largely increase the difficulty of optimization. Existing open-source direct implementations with NCHW layout on ARMv8 architecture are not able to achieve a good balance between the register-level reuse of input and output feature maps tensors shown in Section 2, and often get sub-optimal performance. Meanwhile, existing researches only involve the forward propagation of depthwise convolutions. This paper focuses on effective direct implementations of depthwise convolutions with NCHW layout, and covers all the procedures for the training and inference.

In order to optimize direct depthwise convolutions, many common techniques like register tiling [23], vectorization, and multi-threading are collaboratively adopted in our work. In comparison with open-source algorithms, the most critical part of our work is how register tiling and implicit padding are applied in the micro-kernel design because it greatly reduces the communication between register and cache under the vectorized optimization with complex access patterns. We believe the application can also inspire similar bandwidth limited algorithms with complex memory access patterns. To the best of our knowledge, this is also the first work which studies direct algorithms for all the three procedures of depthwise convolutions on ARMv8 architecture. The main contributions of this paper can be concluded as follows.

- We propose new algorithms with good balanced register-level reuse for depthwise convolutions by means of implicit padding, register tiling, etc., which have less communication overhead between registers and cache, and cover forward propagation, backward propagation and weight gradient update kernels. And, the arithmetic intensity of the new algorithms and the existing ones are introduced to compare their theory performances, with the forward propagation as an example.
- All the new algorithms are benchmarked with all different depthwise convolutional layers from MobileNetV1 [9] and MobileNetV2 [24] on two ARMv8-based processors: Phytium FT1500A 16-core CPUs and Marvell ThunderX 48-core CPUs. The forward propagation implementations for depthwise convolutions are compared with the direct implementations in open source libraries Tengine, FeatherCNN, ncnn, and ARM Compute Library, while the left two passes are in comparison with the matrix multiplication-based algorithms in the popular deep learning framework Pytorch. The experimental results show that our new algorithms can achieve speedups of up to 2.73x, 6.15x, 7.61x, and 36.38x against Tengine, FeatherCNN, ncnn, and ARM Compute Library, respectively, and outperform the matrix multiplication algorithms in all the tests. The optimizations are further confirmed by the training and inference speedup for MobileNetV1 and MobileNetV2, after all the new implementations are used to replace the corresponding kernels in Pytorch.

The structure of this paper is as follows. Section 2 describes the definition of three procedures for depthwise convolutions, and discusses the relevant existing implementations in detail. Our new implementations of all three procedures are presented in Section 3, and the arithmetic intensities are also analyzed. Section 4 shows the benchmark results on two ARMv8-based CPUs. In Section 5, we give the prior studies on the optimization of direct convolutions. The conclusion of this paper and the future work can be found in the final section.

2 Analysis of Existing Implementations

2.1 Forward Propagation of Depthwise Convolutions

The forward propagation of depthwise convolutions takes the input feature maps ($T$) and filters ($F$), and produces the output feature maps ($O$). In NCHW layout, these tensors are expressed as $T[N][C][H][W]$, $F[C][H][W]$ and $O[N][C][H][W]$. Thus, depthwise convolution is defined by

$$O_{n,c,h_0,w_0} = \sum_{h=0}^{H-1} \sum_{w=0}^{W-1} (T_{n,c,h,h+p_1,w,w+s+p_2}) \otimes F_{c,h_0,w_0}$$

where $0 \leq n < N$, $0 \leq c < C$, $0 \leq h_0 < H_n$, $0 \leq w_0 < W_n$, $N$ is the mini-batch size, $C$ is the number of channels, $H_n/o/f$ and $W_n/o/f$ denote the spatial height and width, $p_1/p_2$ refers to the padding size in the spatial dimension, and $s$ is the stride size. In this paper, we mainly focus on depthwise convolutions with $H \times W = 3 \times 3$ and $s \in \{1, 2\}$, which are the most common cases in lightweight
models. From the equation 1, it can be found that the forward propagation is actually performing $N$ batched matrix-vector multiplications. During the computation, the filters are the shared tensors and often small enough to be kept in the on-chip memory all the time. The input feature maps are streamed into the on-chip memory, and then the produced output feature maps are streamed back into the main memory. In other words, there is little space to optimize the access to the main memory, and we will mainly study the communication between cache and register in the following.

![Figure 1: Data reuse in Tengine where $H_f \times W_f = 3 \times 3$ and the stride size is 1.](image)

We tested existing implementations for forward propagation of depthwise convolution in Tengine, FeatherCNN, ncnn and ARM Compute Library on two ARMv8 CPUs, and found that the best performance was achieved in most cases by Tengine. Tengine’s implementation is illustrated in Fig. 1. In ARMv8 architecture, when the width of vector units is 128-bit, each vector unit can deal with 4 single precision numbers simultaneously. In Tengine, each row of $I$ is only loaded one time from cache to vector registers while each row of $O$ is repeatedly loaded from cache to vector registers. For each sample, the elements of $F$ are also loaded into vector registers one time. In other words, in height direction, the elements of $I$ in registers are reused $H_f$ times while there is no reuse for the ones of $O$. For example, $I_r$ are reused three times to get $O_{0,2}$ after loaded into registers, shown in Fig. 1. The elements of $O_{0,2}$ are loaded into vector registers twice and stored back into cache three times. Therefore, we can find the philosophy in Tengine is reusing the elements of $I$ in registers as much as possible so that the communication overhead of $O$ between registers and cache are largely increased. The total communication between registers and cache is about $TC_{tg} = (N \times C \times H_I \times W_I + N \times C \times H_f \times W_f + 5 \times N \times C \times H_o \times W_o) \times 4$ Bytes.

### 2.2 Backward Propagation of Depthwise Convolutions

The backward propagation of depthwise convolutions takes the output gradient tensor ($dO$) and the filters ($F$), and produces the input gradient tensor ($dI$). The computation process can be described by the following equation:

$$dI_{n,c,h_o,x+y,p_x,p_y} = dO_{n,c,h_o,w_y} \times F_{c,h_f,w_y}$$

where $0 \leq n < N, 0 \leq c < C, 0 \leq h_o < H_o, 0 \leq w_y < W_o$.

From the equation 2 we can see that the backward propagation is similar to the forward propagation, so the optimizing strategy is still to reduce the communication between cache and register.

Most deep learning libraries and frameworks on ARMv8 architectures usually implement the backward propagation and weight gradient update of depthwise convolutions through the matrix multiplications based algorithms, which complete the passes by executing a batch of matrix multiplications. In the backward propagation, the matrix multiplication routine is carried out on the $C \times M_m \times K_m$ matrix $\mathcal{F}'$ and the $C \times N_m \times K_m$ matrix $dO'$ to get the $C \times M_m \times N_m$ matrix $dI'$, where $M_m = H_f \times W_f$, $N_m = N \times H_o \times W_o$ and $K_m = 1$. Finally, $dI$ can be obtained by performing the column-to-image transformation on matrix $dI'$, which incurs huge communication overhead between different levels in the memory hierarchy. Therefore, the matrix multiplications based backward propagation often can’t achieve the optimal performance.

### 2.3 Weight Gradient Update of Depthwise Convolutions

The weight gradient update of depthwise convolutions takes the input tensor ($I$) and the output gradient tensor ($dO$), and produces the weight gradient tensor ($dF$). The computation process can be described by the following equation:

$$dF_{c,h_f,w_f} = \sum_{n=0}^{N-1} \sum_{h_o=0}^{H_o-1} \sum_{w_y=0}^{W_o-1} (I_{n,c,h_o,x+y,p_x,p_y} \times dO_{n,c,h_o,w_y})$$

where $0 \leq n < N, 0 \leq c < C, 0 \leq h_o < H_o, 0 \leq w_y < W_o$.

In the matrix multiplication based algorithm of weight gradient update, the output gradient tensor is reshaped into $C \times N_m \times K_m$ matrix $dO'$, the input tensor ($I$) is lowered into a $C \times M_m \times N_m$ Toeplitz matrix $I'$ by image-to-column operations, and the generated $C \times M_m \times K_m$ matrix is finally reshaped back into $dF$, where $M_m = H_f \times W_f$, $N_m = N \times H_o \times W_o$ and $K_m = 1$. Therefore, the algorithm also brings about the huge redundant memory access for the weight gradient update, which incurs performance penalties.

### 3 Our Approach

In this section, we will illustrate how the optimizing techniques are applied to the forward propagation, backward propagation and weight gradient update procedures of direct depthwise convolutions, and analyze the arithmetic intensity.

#### 3.1 Forward Propagation Implementation

In depthwise convolutions, the elements of both $I$ and $O$ can be reused up to $H_f \times W_f$ times in registers. Each core in ARMv8 CPUs has only 32 vector registers so that we may need to load the same elements multiple times from cache to registers. As analyzed in Section 2.1, Tengine loads the elements of $O$ multiple times while maximizing the reuse of the elements of $I$. However, the repeated loading of $O$ is always accompanied by the repeated reading and storing of $O$ from registers to cache due to the accumulation. Therefore, our approach chooses to maximize the reuse of $O$ in registers, and uses implicit padding, vectorization and register tiling techniques to maximize the reuse of $I$ in registers, so that the total communication between registers and cache can be largely reduced. Our direct implementation for the forward propagation is shown in Algorithm 1.
As the vectorization is carried out in the width dimension, the elements of \( F \) are mainly limited by memory access latency. Our algorithm for the propagation of depthwise convolutions, which we call our Direct Implementation, is shown in Algorithm 1.

3.1.1 Implicit Padding

Compared to depthwise convolutions with NCHW layout, the ones with NCHW layout features irregular memory access, especially when dealing with padding. Fig. 2 shows the way how the padding is carried out into the input feature maps with NCHW and NHWC layout. The vectorization performs padding implicitly in this step. The red blocks in Fig. 2 indicate that a vector may consist of input and padding elements simultaneously in NCHW layout, while a vector includes only padding elements or input elements from VL(vector length) channels in NHWC layout. Therefore, it’s more difficult and expensive to deal with the padding under the vectorization optimization with NCHW layout. There are two common methods for padding. One method is explicitly padding input feature maps into a temporary space before computation, adopted by ncnn[28] and FeatureCNN[27]. The other is that the padding is implicitly done through data movement between registers during computation. In comparison with the former method, the latter brings the overhead of data movement in registers. However, the latter has only a half of the communication overhead between cache and registers in the former when only padding is considering. As the performance of depthwise convolution is mainly limited by memory access latency, our approach adopts the implicit padding to minimize the overhead of cache access in padding, shown in lines 13 - 14 of Algorithm 1.

3.1.2 Vectorization and Register Tiling

We further utilize vectorization and register tiling techniques to increase register-level data reuse. In convolutional operations, the filter firstly slides along the width dimension, so that the adjacent convolutions in the same row involve two overlapped regions from the input feature. To reduce the redundant loads, we vectorize the convolution operations in width direction, which divides the convolution operations in a row into groups of VL(vector length). Next, we divide the elements of \( O \) into blocks of \( H_r \times W_r \) size in the height and width dimensions to fix the data used in the computation of the basic block in register. As the vectorization is carried out in the width dimension, \( W_r \) must be a multiple of VL. At the same time, \( H_r \) and \( W_r \) are also limited by the total number of vector registers in ARMv8 CPUs. The kernel function with \( H_r \times W_r \) tiling is shown in lines 8 - 22 of Algorithm 1. The elements of \( F \) are loaded into registers \( v_{f0:H_f-1} \) in advance in lines 4. There are \( H_r \times W_r \times V_L \) registers for the \( H_r \times W_r \) block of \( O \), namely, \( v_{0:H_r \times W_r-1} \). And \( v_{0:H_r \times W_r-1} \) are reused \( H_f \times W_f \) times in the kernel function, and are only stored back into cache once. \( H_f \times (H_r - 1) \times s \) rows of \( I \) are involved to compute the \( H_r \times W_r \) block of \( O \), and each row will be extracted into \( W_f \times W_r \times V_L \) vectors, namely \( v_{0:H_r \times W_r-1} \) through register manipulations, and the padding operation is performed implicitly in this step.

For a more intuitive description of the computation procedure, we will go through the examples depicted in Fig. 3 and Fig. 4. Fig. 3 illustrates the case of unity stride. Without loss of generality, \( H_r \) and \( W_r \) are set to 2 and 8 in Fig. 3. We first load a row \( r_1 \) of input and extract it into \( v_{i0:5} \) along with padding elements. Then we multiply the corresponding elements of \( v_{f0} \) and \( v_{f1} \) to \( v_{i0:5} \) as indicated by the red and blue arrows, and the generated results are accumulated to the output vectors \( v_{o0:1} \) and \( v_{o2:3} \). The elements in \( v_{o0:3} \) are stored back until the final results of depthwise convolutions are acquired. Thus, when the stride is 1, the vectorization and register tiling strategy allow the elements of \( r_1 \) to be reused almost three times in width dimension and to be reused twice in height dimension, and the elements in \( v_{o0:3} \) are reused 9 times. The case of stride 2 is shown in Fig. 4. The elements of the loaded row \( r_1 \) are extracted into \( v_{i0:2} \) with a stride of 2 as indicated by the different colors and numbers. The multiplication and accumulation
operations follow the same process as the former case. The strategies play the same role in the case of stride 2, but exhibit much less reuse times on account of larger stride size. For example, in a kernel’s computation, only the row \( r_2 \) is reused twice and the other rows \( r_{0:1} \) and \( r_{3:4} \) are used only once. In total, the register-level data locality is determined by the tiling size \( H_r \times W_r \) and the stride size.

Additionally, register tiling also can increase the number of operations which can be processed in parallel and filled into the pipeline of ARMv8 CPUs, so that the latency of instructions can be efficiently hidden. It is worth noting that the function Kernel\( H_l \times W_l \) is implemented in assembly language, and all loops are unrolled.

3.1.3 Parallelism Apart from improving micro-kernel performance through aforementioned techniques, our approach can also increase the thread-level parallelism. From Algorithm 1 we can see that the outer-most two loops (Lines 1 - 2) are parallelized by default, which provides \( N \times C \) work items. The blocking parameter \( C_b \) can be adjusted according to the size of input tensor and thread number. However, the default parallelism strategy is not sufficient when the number of threads is greater than \( N \times C \). Since the computing of output\( (O) \) blocks are independent, we further parallelize loop \( h_o \) (Line 5) to fully utilize processor’s parallel processing capacity.

3.2 Backward Propagation Implementation

In this part, we will introduce our implementation of backward propagation. Backward propagation is described by equation 2 mentioned in section 2.2. By setting \( h_l = h_o + h_f - p_t \), \( w_t = w_o + w_f - p_t \), we can rewrite the equation as \( dI[l][n][c][h_l][w_t] = dO[n][c][h_l][w_t]/s \times F[c][h_f][w_f] + dO[n][c][h_l][w_t]/s \times \). In the case of unity stride, the equation is equivalent to \( dI[l][n][c][h_l][w_t] = dO[n][c][h_l][w_t]/s \). However, the default parallelism strategy is not sufficient when the number of threads is greater than \( N \times C \). Since the computing of output\( (O) \) blocks are independent, we further parallelize loop \( h_o \) (Line 5) to fully utilize processor’s parallel processing capacity.
of $\mathcal{F}$ involved in the computation of $dF_{h,c,w_0}$ are decided by the parity of $h_0$ and $w_0$. Therefore, there are 4 cases of the computation formula for $dI$, which depends on the parity of $h$ and $w$. Besides, every other elements($dI$) in a row have the same parity, thus they share the same formula and can be vectorized. Through register tiling strategy, we divide elements of $dI$ into blocks of $H_T \times W_T$ size in the height and width dimensions. As the computations of every other elements ($dI$) are vectorized in width dimension, $W_T$ must be a multiple of $2 \times V_L$. Fig. 5 shows an example of our approach with $2 \times 8$ tiling size, where $p_1$ and $p_2$ are set to 1. For brevity, we omit $h_i$, and derive the following equation of $dI$:

$$dF_{h,c,w_0} = \begin{cases} 
\text{dO}_{h,c,[w_0/2]} \times F_{c,0} & w_0 = 2k \\
\text{dO}_{h,c,[w_0/2]} \times F_{c,1} + \text{dO}_{h,c,[w_0/2]} \times F_{c,0} & w_0 = 2k + 1 
\end{cases}$$

(4)

From the equation 4, we can deduce that every $x$ elements in a row of $dI$ are related to $x/2+1$ elements in a row of $dO$. The equation 4 also applies for height dimension. The relations between rows of $dO$ and $dI$ are indicated by the arrows in the top of Fig. 3. The detailed computation procedures are described in the following. The elements of $dO$ are loaded into registers $v_{f0,2}$ in advance. We load a row($o_0$) of $dO$ and extract it into $v_{00,1}$ by implicitly padding and register manipulations. As illustrated by the arrows, the results are accumulated into $v_{02,1}$ and $v_{22,3}$ respectively. The temporary results in $v_{02,3}$ are kept in the registers until the final results are obtained. Since the computation of $dI_0$ is finished in Fig. 5, the $v_{01,1}$ are written back to memory with a stride of 2, which can be indicated by the letters labeled on the elements. We tried some common tiling sizes and adopted the $6 \times 8$ for its higher performances in most layers.

3.3 Weight Gradient Update Implementation

![Figure 6: A working example of our weight gradient update propagation implementation of stride-1 depthwise convolutions on ARMv8 CPUs, where $H_f \times W_f = 3 \times 3$, and $H_T \times W_T = 2 \times 2$](image)

In this sub-section, we will present our implementation of weight gradient update. Our implementation is described in Algorithm 2. As shown in equation 3, the gradient weight feature maps in the same channel $c$ will be added up to $dF_{c,0,0}$, so the loop $c$ is parallelizable(Line 2). The elements of $dO$ are divided into blocks of $H_T \times W_T$ size in the height and width dimensions(Lines 4-6) and

![Algorithm 2: Our Implementation of Weight Gradient Update](image)
Towards Effective Depthwise Convolutions on ARMv8 Architecture

With the tiling size 4, the AI of our implementation is larger than that of Tengine.

Therefore, the AI of our implementation is larger than that of Tengine.

3.4 Arithmetic Intensity

In this part, we will take the forward propagation as an example to analyze the Arithmetic Intensity (AI) [7] of our optimized implementations and Tengine, which has the largest AI among the existing implementations. The total number of arithmetic operations is \( T \cdot A = 2 \times N \times C \times H_r \times W_r \). When the tiling size \( H_r \times W_r \) is used, the total communication between registers and cache in our approach involves:

1. Loading \( \mathcal{F} \) once for each batch (Line 4). So the \( \mathcal{F} \) incurs \( T \mathcal{F} \).
2. Storing \( \mathcal{O} \) once (Line 22). Thus the traffic of \( \mathcal{O} \) is \( T \mathcal{O} = 4 \times N \times C \times H_r \times W_r \) bytes.
3. Loading \( (W_r - 1) \times s + W_f \) elements of \( I \) in loop r (Line 13).

So the traffic of \( I \) incurred by one complete execution of \( \text{Kernel}H_r \times W_r \times C \times T \mathcal{I} \) is \( ((W_r - 1) \times s + W_f) \times (h_f - 1) \times s + H_f \). The \( \text{Kernel}H_r \times W_r \times C \times T \mathcal{I} \) is called \( N \times C \times H_r \times W_r \)/times. Hence the total traffic of \( I \) is \( T \mathcal{I} = 4 \times N \times C \times H_r \times W_r \times T \mathcal{I} \).

With the tiling size 4 × 4, the AI of our implementation is

\[
A_{ours} = \frac{T \mathcal{A}}{T \mathcal{F} + T \mathcal{O} + T \mathcal{I}}
\]

\[
= \begin{cases} 
1/(0.13+2/(H_r \times W_r))(\text{Ops/Byte}) & s = 1 \\
1/(0.31+2/(H_r \times W_r))(\text{Ops/Byte}) & s = 2 
\end{cases}
\]

The AI of Tengine is:

\[
A_{tg} = \frac{T \mathcal{A}}{T \mathcal{F} + T \mathcal{O} + T \mathcal{I}}
\]

\[
= \begin{cases} 
1/(1.53+2/(H_r \times W_r))(\text{Ops/Byte}) & s = 1 \\
1/(2.12+2/(H_r \times W_r))(\text{Ops/Byte}) & s = 2 
\end{cases}
\]

Therefore, the AI of our implementation is larger than that of Tengine.

4 Experimental Evaluation

In this section, we firstly compare our forward propagation implementation to the existing ones in Tengine [19], FeatherCNN [27], ncnn [28] and ARM Compute Library (ACL) [2]. Secondly, we compare our backward propagation and weight gradient update implementations to the matrix multiplication-based ones in Pytorch. Thirdly, we evaluate the full topology speedup of MobileNetV1 and MobileNetV2 based on our implementations.

4.1 Experimental Setup

We run our experiments on the following two ARMv8 processors:

- **Phytium FT1500A**: 1.5GHz ARMv8 processor with 2 core groups each with 4 cores. Each core has 32KB L1 instruction cache and 32KB L1 data cache. 4 cores of a core group share 2MB L2 cache.
- **Marvell ThunderX**: 2.0 GHz ARMv8 processor with 48 cores. Each core has 78KB L1 instruction cache and 32KB L1 data cache. All 48 cores share 16MB L2 cache.

In the compilation of Pytorch, we use the OpenBLAS version 0.3.15 library to provide GEMM function. The depthwise convolutional layers from lightweight networks MobileNetV1 and MobileNetV2 are used in our tests. In the following tables and figures, MobileNetV1 and MobileNetV2 are labeled as v1 and v2. All the tests are iterated 10 times and the median runtime is reported as the result of each test.

4.2 Forward Propagation Performance

We compare our forward propagation implementation against the existing ones in Tengine [19], FeatherCNN [27], ncnn [28] and ARM Compute Library (ACL) [2] using different cores. Tengine works best in most cases among four open implementations, so we normalize the performance to Tengine.

The relative single-core performance of five implementations on FT1500A and ThunderX is shown in Fig. 8(a) and Fig. 8(b), respectively. The x-axis indicates the different depthwise convolutional layers from MobileNetV1 and MobileNetV2, while the y-axis shows the speedup of five implementations over Tengine. In the test of the single-core performance, the mini-batch size of all layers is set to 1. The results show that our implementation is better than all four open implementations in all cases on these two ARMv8 processors. Compared to Tengine, FeatherCNN, ncnn and ACL, our implementation obtains speedups up to 2.43x, 4.75x, 4.30x, 36.38x on FT1500A and 1.54x, 2.72x, 2.74x, 9.80x on ThunderX. When the same input size is adopted, the layers with stride 1 get much bigger speedup than the ones with stride 2. The main reason is that our approach reduces the access of output feature maps as much as possible and the larger output size let our implementation get higher performance improvements.

Fig. 9(a) and Fig. 9(b) show the relative multi-core performances of our implementation on FT1500A and ThunderX. We set the mini-batch size to 256 since 256 is frequently used in network training. The results demonstrate that our approach surpasses FeatherCNN, ncnn, ACL on all tested layers and obtains speedups range from 1.36x to 5.67x, 1.23x to 5.70x, 3.83x to 14.66x on FT1500A and 1.86x to 6.15x, 2.48x to 7.53x, 9.1x to 28.62x on ThunderX. When compared to Tengine, our implementation exhibits higher performance in most tested layers and yields average speedups of 1.55x on FT1500A and 1.57x on ThunderX, respectively.

In summary, our approach can effectively accelerate the forward propagation for depthwise convolutions.
Figure 8: Single-core performance of forward propagation implementations on FT1500A and ThunderX, where mini-batch size is 1 and performance is normalized to Tengine.

Figure 9: Multi-core performance of forward propagation implementations on all 16 cores of FT1500A and 48 cores of ThunderX, where mini-batch size is 256 and performance is normalized to Tengine.

4.3 Backward Propagation Performance

The relative performance of our backward propagation implementation against the matrix multiplication-based one in Pytorch are shown in Fig. 10(a) and Fig. 10(b), respectively. In the figures, different colors represent the tests using different mini-batch sizes. Among all the tests, the speedups of our implementation against Pytorch are greater than 3.96x on FT1500A and 1.76x on ThunderX. For the cases on FT1500A shown in Fig. 10(a), the speedup increases significantly with the decrease of input size. When input size is small, our tests show that the overhead of matrix multiplication in Pytorch accounts for more than 80% of the total cost on FT1500A. For the cases on ThunderX shown in Fig. 10(b), the layers with bigger input size show higher speedup. The matrix multiplication’s percentage in Pytorch ranges from 31.8% to 55.7% on ThunderX. In addition, we can observe that our implementation on FT1500A gets higher average improvement than that on ThunderX. The main reason is that the matrix multiplication-based algorithm is heavily dependent on the performance of functions in BLAS library, and the OpenBLas v0.3.15 library maybe has not been well optimized on FT1500A. In short, our direct implementation for backward propagation of depthwise convolutions are better than the matrix multiplication-based one in Pytorch, and doesn’t rely on any external computing libraries.

4.4 Weight Gradient Update Performance

Fig. 11(a) and Fig. 11(b) show the relative performance of our weight update propagation implementation against the matrix multiplication-based one in Pytorch. As described in the previous sub-section, this sub-section carries out the tests using different mini-batch sizes as well. The results show that our implementation acquires speedups of 6.83x - 89.44x on FT1500A and 2.11x - 8.50x on ThunderX against the one in Pytorch. And the performance trend on two CPUs is similar to that for backward propagation.

4.5 Full Topology Performance

Finally, we integrated our direct implementations into Pytorch and evaluated the end-to-end inference and training speedup of MobileNetV1 and MobileNetV2 over original Pytorch. The experimental results under different number of threads and mini-batch sizes are provided in Table1 and Table2 respectively. For the inference of MobileNetV1 and MobileNetV2, our work achieves an average speedup of 4.23x and 3.67x against original Pytorch on FT1500A, and an average speedup of 9.24x and 7.36x against original pytorch on ThunderX. When comparing the training of MobileNetV1 and
Towards Effective Depthwise Convolutions on ARMv8 Architecture
Conference ’17, July 2017, Washington, DC, USA

MobileNetV2 to original Pytorch, an average speedup of 13.72x and 12.96x is observed on FT1500A, and the corresponding one is 23.58x and 21.85x on ThunderX.

Table 1: Inference speedup for MobileNetV1/MobileNetV2 of our work against original pytorch. tx denotes that x threads is used and tx/y means x threads is used on FT1500A and y threads is used on ThunderX. bx indicates that mini-batch size is x.

| FT1500A | ThunderX |
|----------|-----------|
| MobileNetV1 | MobileNetV2 | MobileNetV1 | MobileNetV2 |
| t1, b1 | 3.94x | 2.62x | 4.26x | 2.38x |
| t16/48, b16 | 7.00x | 5.97x | 19.76x | 15.89x |
| t16/48, b32 | 5.28x | 4.54x | 13.21x | 10.94x |
| t16/48, b64 | 3.74x | 3.66x | 8.39x | 6.90x |
| t16/48, b128 | 2.85x | 2.65x | 5.81x | 4.53x |
| t16/48, b256 | 3.44x | 2.60x | 4.03x | 3.51x |

| FT1500A | ThunderX |
|----------|-----------|
| MobileNetV1 | MobileNetV2 | MobileNetV1 | MobileNetV2 |
| t16/48, b48 | 15.33x | 15.21x | 18.88x | 19.55x |
| t16/48, b32 | 14.03x | 12.73x | 26.89x | 23.26x |
| t16/48, b64 | 13.05x | 12.78x | 22.43x | 20.83x |
| t16/48, b128 | 12.97x | 12.00x | 25.19x | 21.86x |
| t16/48, b256 | 13.20x | 12.09x | 24.53x | 23.73x |

Table 2: Training speedup for MobileNetV1/MobileNetV2 of our work against pytorch.

| FT1500A | ThunderX |
|----------|-----------|
| MobileNetV1 | MobileNetV2 | MobileNetV1 | MobileNetV2 |
| t16/48, b48 | 15.33x | 15.21x | 18.88x | 19.55x |
| t16/48, b32 | 14.03x | 12.73x | 26.89x | 23.26x |
| t16/48, b64 | 13.05x | 12.78x | 22.43x | 20.83x |
| t16/48, b128 | 12.97x | 12.00x | 25.19x | 21.86x |
| t16/48, b256 | 13.20x | 12.09x | 24.53x | 23.73x |

5 Related Work
A great amount of work has been made to optimize the implementations of convolution operations. As previously mentioned, the widely-used methods to compute the convolutions contain matrix multiplication-based, Winograd-based, FFT-based and direct algorithms. The first three algorithms can be collectively referred to as the indirect algorithms.

The matrix multiplication-based algorithms, first proposed by Chellapilla [3] et al, cast convolutions into matrix-matrix multiplication operations, and can be adopted by convolutions with arbitrary parameters. As a result, the matrix multiplications-based algorithms can be found in all popular deep learning frameworks, such as Caffe [12], Mxnet [4], Pytorch [20] and TensorFlow [1]. The Winograd- and FFT-based algorithms can reduce the arithmetic
complexity of convolutions with specific parameters through Winograd and FFT transformations, and thus they are also called fast convolution algorithms [13, 29]. All the three indirect algorithms have been optimized on ARMv8 architecture [10, 11, 31–33]. However, all indirect algorithms generates non-trivial overhead of memory access. Although several methods [30, 33] have been proposed to optimize the memory access in the matrix multiplications-based algorithms, the overhead is still inevitable. Meanwhile, the fast algorithms maybe increase the computational complexity for depth-wise convolutions because their transformations also introduce some computation operations [13]. Thus, all indirect algorithms are not well-suited for depthwise convolutions.

As the direct convolution algorithm often can eliminate all memory overhead, it also attracts a lot of attention [6, 15, 35, 36]. Zhang et al. [35] and Georganas et al. [6] optimized the direct implementation of conventional convolutions based on specialized layouts, which avoids complex data movement for the vectorization. Lu et al. [15] first proposed an CUDA-based direct implementation of depthwise convolutions on NVIDIA GPUs. Zhang et al. [36] described an optimized forward propagation implementation for depthwise convolutions with NHWC layout on ARM-based mobile devices, which utilized register tiling and loop rescheduling techniques. Unlike [36], our focus is all the three procedures for depth-wise convolutions with NCHW layout on ARMv8 CPUs, which is the default for Caffe, Mxnet, and Pytorch. In theory, depthwise convolutions with NCHW layout can have better data locality than ones with NHWC layout. However, the NCHW layout also increases the memory access overhead accompanied by data alignment in the vectorization on ARMv8 CPUs, which is largely reduced by implicit padding and register tiling in this paper.

6 Conclusion

In this paper, we propose new direct implementations of forward propagation, backward propagation, weight gradient update for depthwise convolutions on ARMv8 architectures. Our implementations improve the register-level data locality through implicit padding, vectorization, register tiling and multi-threading techniques so that the communication between cache and registers is optimized. And the arithmetic intensities are analyzed as well. Through the experiments on two ARMv8 CPUs, we show that the new implementations can get better performance than existing implementations and reduce the overhead of end-to-end lightweight CNNs training and inference on ARMv8 CPUs. In the future, we will study how to generate optimized direct implementations for depthwise convolutions based on just-in-time compilation.

Acknowledgments

This research work was supported by the National Natural Science Foundation of China under Grant No. 6202365

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jefrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Jianjun Jiang, Andy Barbour, Paul Buehler, Mike Chen, Min-Che Hsu, Xiaoqiang Jiang, Zhiyi Jiang, David Jong, Monday Kam, Gregory Kandrot, Vicente Latorre, Pyo Lee, Haoyan Lin, Minjie Lin, Paul Liu, Wei Lu, Min McCormick, Yuan Qi, Yanliang Shi, Márcio de Souza, Louis Steiniger, Francisco Sun, Christian Szegedy, Brian Talbot, Lucy Tang, Meng-Wei Tsai, Meng-Mei Tu, Tonya Yeung, and Wang Ying. 2015. TensorFlow: A System for Large-Scale Machine Learning. In 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283.

[2] ARM. 2021. Compute Library. https://github.com/ARM-software/ComputeLibrary. Online, accessed 3-Sep-2021.

[3] K. Chellapilla, S. Purk, and P. Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. Tenth International Workshop on Frontiers in Handwriting Recognition (2006).

[4] Tianq Chen, Mu Li, Yutian Li, Min Lin, Naiyang Wang, Minjie Wang, Tianxiao Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:eCs.DC.1512.0174.

[5] Xinhai Chen, Peizhen Xie, Lihua Chi, Jie Liu, and Chunye Gong. 2018. An efficient SIMD compression format for sparse matrix-vector multiplication. Concurrency and Computation: Practice and Experience, 30 (2018), e4800.

[6] Evangelos Georganas, Sasikant Avancha, Kunal Banerjee, Dhiraj D. Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of high-performance deep learning convolutions on SIMD architectures. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018. IEEE / ACM, 66:1–66:12.

[7] Mark Harris. 2005. Mapping Computational Concepts to GPUs. In ACM SIGGRAPH 2005 Courses (SIGGRAPH ’05). Association for Computing Machinery, New York, NY, USA, 50–es. https://doi.org/10.1145/1958555.1958678

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE international conference on computer vision. 1026–1034.

[9] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR (2017).

[10] Xiandong Huang, Qinglin Wang, Shuyu Lu, Ruochen Hao, Songzhu Mei, and Jie Liu. 2021. Evaluating FFT-based algorithms for strided convolutions on ARMv8 architectures. Performance Evaluation (2021). 102248. https://doi.org/10.1016/j.peva.2021.102248.

[11] Xiandong Huang, Qinglin Wang, Shuyu Lu, Ruochen Hao, Songzhu Mei, and Jie Liu. 2021. NUMA-aware FFT-based Convolution on ARMv8 Many-core CPUs. arXiv:cs.DC.2109.12259

[12] Yangqiang Jia, Evan Shelhammer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In Proceedings of the 22nd ACM International Conference on Multimedia (MM’14). Association for Computing Machinery, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[13] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 4013–4021. https://doi.org/10.1109/CVPR.2016.435

[14] Shujie Li, Yong Dou, Xin Niu, Qi Lv, and Qiang Wang. 2017. A fast and memory saved GPU convolutional algorithm of convolutional neural networks for target detection. Neurocomputing 230 (2017), 48 – 59.

[15] Gangzhao Lu, Weizhe Zhang, and Zheng Wang. 2022. Optimizing Depthwise Separable Convolution Operations on GPUs. IEEE Transactions on Parallel and Distributed Systems 33, 1 (2022), 70–87.

[16] Marvell. 2022. ThunderX_CoProcessor. https://www.marvell.com/server-processors/thunders-arm-processors. Online, accessed 1-Jan-2022.

[17] Satoshi Matsuoaka. 2021. Fagaku and A64FX. the First Exascale Supercomputer and its Innovative ARM CPU. In 2021 Symposium on VLSI Circuits. 1–3. https://doi.org/10.23919/VLSICircuits52068.2021.9492415

[18] Sparsh Mittal, Poonam Rupput, and Sreevani Subramoney. 2021. A Survey of Deep Learning on CPUs: Opportunities and Co-Optimizations. IEEE Transactions on Neural Networks and Learning Systems (2021), 1–21. https://doi.org/10.1109/TNNLS.2021.3071762

[19] OPEN AI LAB. 2021. Tengine. https://github.com/OAID/Tengine. Online, accessed 3-Sep-2021.

[20] Adam Paszke, Sam Gross, Francisco Massa, and et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:cs.LG/1912.01703.

[21] Adam Paszke, Sam Gross, Francisco Massa, and et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:cs.LG/1912.01703

[22] N. Rajovic, A Ramírez Bellido, A. Rico, F. Mantovani, D. Ruiz, O. Villarubi, C. Gómez, L. Backes, D. Nieto, and H. Servat. 2016. The Mont-Banc prototype: an Alternative for High-Performance Computing Applications. Frontiers in Handwriting Recognition (2016), 1–21. https://doi.org/10.1016/j.peva.2021.102248

[23] Lakshminarayanan Renganarayana, Uday Bondhugula, Salem Derisavi, Alexandre E. Ieichenberg, and Kevin O’Brien. 2009. Compact Multi-Dimensional Kernel Extraction for Register Tiling. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC ’09). Association for Computing Machinery, New York, NY, USA, Article 45, 12 pages. https://doi.org/10.1109/SC.2009.4013

[24] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks.
In 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[25] Rajeev Kumar Singh and Rohan Gorantla. 2020. DMENet: Diabetic Macular Edema diagnosis using Hierarchical Ensemble of CNNs. *PLOS ONE* 15, 2 (2020).

[26] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). 2815–2823. https://doi.org/10.1109/CVPR.2019.00293

[27] Tencent. 2021. FeatherCNN. https://github.com/Tencent/FeatherCNN. Online, accessed 3-Sep-2021.

[28] Tencent. 2021. ncnn. https://github.com/Tencent/ncnn. Online, accessed 3-Sep-2021.

[29] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2015. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. In 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, Conference Track Proceedings, Yoshua Bengio and Yann LeCun (Eds.).

[30] Aravind Vasudevan, Andrew Anderson, and David Gregg. 2017. Parallel multi channel convolution using general matrix multiplication. In Application-specific Systems, Architectures and Processors (ASAP). 2017 IEEE 28th International Conference on. IEEE, 19–24.

[31] Qinglin Wang, Dongsheng Li, Xiadong Huang, Siqi Shen, Songzhu Mei, and Jie Liu. 2020. Optimizing FFT-Based Convolution on ARMv8 Multi-core CPUs. In European Conference on Parallel Processing. 248–262. https://doi.org/10.1007/978-3-030-57675-2_16

[32] Qinglin Wang, Dongsheng Li, Songzhu Mei, Zhiquan Lai, and Yong Dou. 2020. Optimizing Winograd-Based Fast Convolution Algorithm on Phytip Multi-Core CPUs (In Chinese). *Journal of Computer Research and Development* 57, 6 (2020), 1140 – 1151. https://doi.org/10.7544/issn1000-1239.2020.20200107

[33] Qinglin Wang, Mei Songzhu, Jie Liu, and Chunye Gong. 2019. Parallel convolution algorithm using implicit matrix multiplication on multi-core CPUs. In 2019 International Joint Conference on Neural Networks (IJCNN). 1–7. https://doi.org/10.1109/IJCNN.2019.8852012

[34] Xin You, Hailong Yang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2019. Performance Evaluation and Analysis of Linear Algebra Kernels in the Prototype Tianhe-3 Cluster. In Supercomputing Frontiers, David Abramson and Bronis R. de Supinski (Eds.). Springer International Publishing, Cham, 86–105. https://doi.org/10.1007/978-3-030-18645-6_6

[35] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. 2018. High Performance Zero-Memory Overhead Direct Convolutions. In International Conference on Machine Learning. 5771–5780.

[36] Pengfei Zhang, Eric Lo, and Baotong Lu. 2020. High Performance Depthwise and Pointwise Convolutions on Mobile Devices. In Proceedings of the AAAI Conference on Artificial Intelligence. AAAI Press, 6795–6802.