High Performance Depthwise and Pointwise Convolutions on Mobile Devices

Pengfei Zhang, Eric Lo, Baotong Lu
The Chinese University of Hong Kong
{pfzhang, ericlo, btlu}@cse.cuhk.edu.hk

Abstract

Lightweight convolutional neural networks (e.g., MobileNets) are specifically designed to carry out inference directly on mobile devices. Among the various lightweight models, depthwise convolution (DWConv) and pointwise convolution (PWConv) are their key operations. In this paper, we observe that the existing implementations of DWConv and PWConv are not well utilizing the ARM processors in the mobile devices, and exhibit lots of cache misses under multi-core and poor data reuse at register level. We propose techniques to re-optimize the implementations of DWConv and PWConv based on ARM architecture. Experimental results show that our implementation can respectively achieve a speedup of up to 5.5× and 2.1× against TVM (Chen et al. 2018) on DWConv and PWConv.

Introduction

Recently, there is an increasing trend to carry out convolutional neural network (CNN) inference on mobile devices directly because of both privacy and real-time latency (user experience) requirements. (Loc, Lee, and Balan 2017; Han et al. 2016; Howard et al. 2017; Sandler et al. 2018). However, since mobile devices are subjected to both computational and energy constraints, recent research therefore puts effort on designing more lightweight "mobile models" that are composed of fewer layers and/or using less computational expensive operations.

In terms of CNN, examples of such lightweight mobile models include Xception (Chollet 2016), MobileNetV1 (Howard et al. 2017), MobileNetV2 (Sandler et al. 2018), MnasNet (Tan et al. 2018), EfficientNet (Tan and Le 2019), to name a few.

When optimizing the performance of a program with respect to a type of processors, developers often use the *roofline model* (Williams, Waterman, and Patterson 2009) to guide their implementation. Figure 1 shows the roofline model of quad-core ARM Cortex-A57. The roofline (the dashed line) indicates the maximum achievable performance of any program under that processor.

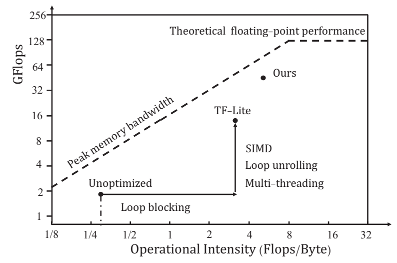


Figure 1: Roofline model for ARM Cortex-A57 with respect to MobileNetV1 inference

Given the roofline model of a processor, one can check whether her implementation has fully utilized that processor or not. In Figure 1, the point 'Unoptimized' represents a naive C implementation of MobileNetV1 written by us. The point 'TF-Lite' represents the popular TensorFlow Lite binary compiled with math optimization, auto vectorization and linking to Eigen (Guennebaud, Jacob, and others 2010) BLAS library. Since TF-Lite is open source, it is known that it has already optimized using all the optimization tricks suggested in the roofline article (e.g., using SIMD intrinsics). Unfortunately, even the popular TensorFlow Lite (TF-Lite) is not fully utilizing the processor. So, what is missing?

ARM processors get the lion's share of the mobile device processor industry (SoftBank Group 2017); and DWConv and PWConv are the two most dominating operations in state-of-the-art mobile models and they take up 90+% of total inference time (Howard et al. 2017; Sandler et al. 2018; Tan and Le 2019). Therefore, the goal of this paper is to optimize *depthwise convolution* (DWConv) and *pointwise convolution* (PWConv) on ARM processors. We observe there are two major issues that hurt the performance of DWConv and PWConv on ARM processors.

First, we point out that the **existing DWConv and PWConv implementations are poor in core scalability**, which is against the trend of getting more cores in ARM processors (e.g., Huawei's latest mobile phone SoC chipset, Kirin 980, has eight ARM cores). Second, we point out that the **optimization tricks suggested in the roofline article are necessary but insufficient for ARM processors**. Specifically, while both ARM and x86 processors can carry out 2 FMA (fused-multiply-add) instructions per cycle, ARM processors can only load 1 register (from the cache) per cycle whereas x86 processors can load 4 registers per cycle. In other words, while optimizing the cache miss and increasing parallelism could eliminate the major bottleneck on x86 processors, on ARM processors those tricks could only shift the bottleneck to the traffic between the register and the cache. Based on the above observations, we therefore develop high performance version of DWConv and PWConv for mobile devices. Using techniques like *loop rescheduling* (Markatos and LeBlanc 1992) and *register tiling* (Jiménez, Llabería, and Fernández 2002), our implementations are able to reduce the traffic between the cache and the memory **as well as** the traffic between the register and the cache. Experimental results show that our implementation can respectively achieve a speedup of up to 5.5× and 2.1× against TVM (Chen et al. 2018) on DWConv and PWConv, which leads to a 46GFlops on ARM Cortex-A57 in terms of overall MobileNetV1 inference.

Preliminaries

ARM processors dominate the mobile device market. Latest ARM processors all support a 64-bit architecture, named "AArch64". AArch64 is a load-store architecture where data has to be loaded into the registers before the operations take place. AArch64 supports SIMD instruction and each core has 32 SIMD registers. Each SIMD register is 128-bit, which means each SIMD instruction can operate on 4 single precision numbers simultaneously. The predominate instruction used in model inference is the FMA (fused-multiply-add) SIMD instruction. An FMA instruction requires 3 SIMD registers to fully operate. Each FMA instruction carries out a 4-way SIMD multiplication, followed by a 4-way SIMD addition.

Depthwise Convolution

Depthwise convolution (DWConv) is a key operation in mobile models. It takes three inputs: (i) a 3d array $\mathcal{I}$ (the input feature map) of size $H_i \times W_i \times C$, (ii) a 3d array $\mathcal{F}$ (the filter) of size $H_f \times W_f \times C$, (iii) the stride s. It produces a 3d array (the output feature map) $\mathcal{O}$ of size $H_o \times W_o \times C$. In the above, H and W are the spatial height and width, C is the number of channels. The subscripts i, f and o refers to the input feature map, the filter, and the output feature map respectively.

Algorithm 1: Unoptimized Depthwise Convolution

```
   Input: Input feature map $\mathcal{I}$, Filter $\mathcal{F}$, stride $s$;
   Output: Output feature map $\mathcal{O}$;
1  for $l = 0$ to $H_o - 1$ do
2      for $k = 0$ to $W_o - 1$ do
3          for $i = 0$ to $C_i - 1$ do
4              for $n = 0$ to $H_f - 1$ do
5                  for $m = 0$ to $W_f - 1$ do
6                      $\mathcal{O}_{l,k,i}$ += $\mathcal{I}_{l \times s+n, k \times s+m, i} \times \mathcal{F}_{n,m,i}$
```

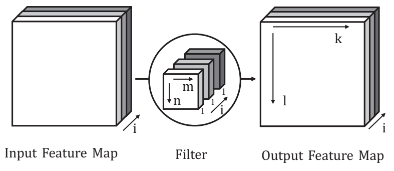


Figure 2: Depthwise convolution

Figure 2 illustrates the concept of depthwise convolution. Algorithm 1 is its plain implementation, which consists of 5 tightly-nested loops around a multiply-accumulate (MAC) statement (Line 6). Referring to Figure 2, the implementation iteratively applies the filter (lines 4 and 5) per channel (Line 3), and then repeats the task by moving the filter from left to right (Line 2) and then from top to bottom (Line 1).

Algorithm 2: Depthwise Convolution (TF-Lite)

```
    Input: Input feature map $\mathcal{I}$, Filter $\mathcal{F}$, stride $s$;
    Output: Output feature map $\mathcal{O}$;
1   for $l = 0$ to $H_o - 1$ in parallel do
2       for $k' = 0$ to $W_o/W_{o,b} - 1$ do
3           for $n = 0$ to $H_f - 1$ do
4               for $kk = 0$ to $W_{o,b} - 1$ do
5                   for $m = 0$ to $W_f - 1$ do
6                       for $i' = 0$ to $C/4$ do
                            // Loop unrolling
                               here.
7                           $k = k' \times W_{o,b} + kk$
8                           $V_I$=SIMD_Load(
                             $\mathcal{I}_{l \times s+n, k \times s+m, i' \times 4 \sim i' \times 4+3}$)
9                           $V_F$=SIMD_Load(
                             $\mathcal{F}_{n,m,i' \times 4 \sim i' \times 4+3}$)
10                          $V_O$=SIMD_Load(
                             $\mathcal{O}_{l,k,i' \times 4 \sim i' \times 4+3}$)
11                          $V_O$=SIMD_FMA($V_I$, $V_F$, $V_O$)
12                          SIMD_Store($\mathcal{O}_{l,k,i' \times 4 \sim i' \times 4+3}$,
                             $V_O$)
```

Algorithm 2 shows the implementation of DWConv in TF-Lite. It mainly applies 4 tricks to improve its efficiency.

1. Loop rescheduling and SIMD. Any permutation of the ordering (scheduling) of the loops would yield the same correct result but with different efficiency. Furthermore, each channel of the filter can apply to corresponding channel of the input independently and thus *in parallel*. Consequently, Algorithm 2 reschedules the innermost loop to process the MAC across 4 channels using SIMD (lines 6–12).
2. Loop Unrolling. The innermost loop actually possesses loop independence, meaning one iteration does not depend on its previous iteration. In other words, the loop

can be run in parallel. Consequently, the actual implementation of the innermost loop is *unrolled* (or called as *flattened*) (Dongarra and Hinds 1979). Loop unrolling not only improves ILP (instruction-level parallelism), but also reduces branch mis-prediction incurred by the test condition of each iteration. Algorithm 2 however does not explicitly show the unrolled loop for brevity.

3. Loop Blocking. When involving matrix/tensor, loop blocking is often used to reduce cache misses (Xue 2000). In TF-Lite, loop blocking is applied to the k loop (Algorithm 1; Line 2) and it becomes the k' loop in (Algorithm 2; Line 2) and the kk loop in (Algorithm 2; Line 4). By doing so, the data loaded in the k' loop (Algorithm 2; Line 2) could stay in the cache and get re-used again and again by the inner n loop.
4. Multi-threading. As real-time inference is getting more important, TF-Lite also uses multiple cores to parallel the outermost loop (Line 1). In other words, the blocks across the l direction in Figure 2 are generated by multiple cores.

Pointwise Convolution

Algorithm 3: PWConv Implementation by MM

```
  Input: Input feature map $\mathcal{I}$, Filter $\mathcal{F}$;
  Output: Output feature map $\mathcal{O}$;
1 Mat A = $\mathcal{I}$.reshape([$G$, $C_i$])
2 Mat B = $\mathcal{F}$.reshape([$C_i$, $C_o$])
3 Mat D = $A \times B$
4 $\mathcal{O}$ = D.reshape([$H_o$, $W_o$, $C_o$])
```

Another key component in mobile models is the pointwise convolution (PWConv). PWConv is a simple 1×1 convolution. It takes as inputs: (1) a 3d input feature map $\mathcal{I}$ of size ($H_i \times W_i \times C_i$), and (2) a 4d filter $\mathcal{F}$ of size ($1 \times 1 \times C_i \times C_o$), and produces a 3d output feature map $\mathcal{O}$ of size ($H_o \times W_o \times C_o$), where $H_o = H_i$ and $W_o = W_i$.

Algorithm 3 shows the implementation of PWConv in TF-Lite. It essentially transforms the problem into a matrix-matrix (MM) multiplication problem $D = A \times B$, where the 2d matrix A is flatten from the 3d input $\mathcal{I}$, so that A is a $G \times C_i$ matrix, where $G = H_i \times W_i$ (Line 1); and B is a matrix of size $C_i \times C_o$ flatten from $\mathcal{F}$ (Line 2) since the first two dimensions are of size 1.

Since MM multiplication is a classic problem that has been well studied, TF-Lite simply calls the high performance MM routine in a BLAS library (Dongarra et al. 1990). MM multiplication implementations in BLAS are highly optimized with all the tricks (e.g., SIMD, loop rescheduling) mentioned above. Recently, Google released an experimental matrix multiplication library named Ruy (Google 2019). Ruy achieves good performance on small matrices (e.g., 100×100) but its the performance on large matrices is poorer than BLAS. Since Ruy's code is still immature and flux, we do not analyze it here but include that in our experiments.

High Performance DWConv and PWConv

In this section, we present techniques to optimize the implementations of DWConv and PWConv on ARM processors. We will explain in detail why the existing "well-optimized" implementations are not efficient on ARM processors and propose our solutions. One of the key elements there is about the notions of *operational intensity* in the roofline model (Williams, Waterman, and Patterson 2009) and the notion of *arithmetic intensity* (Harris 2005).

Algorithm 4: High Performance Depthwise Convolution

```
   Input: Input feature map $\mathcal{I}$, Filter $\mathcal{F}$, stride $s$;
   Output: Output feature map $\mathcal{O}$;
1  for $i' = 0$ to $C/4 - 1$ in parallel do
2      for $l' = 0$ to $H_o/H_{o,b} - 1$ do
3          for $k' = 0$ to $W_o/W_{o,b} - 1$ do
4              Kernel($i'$, $l'$, $k'$, $s$)
5  Function Kernel($i'$, $l'$, $k'$, $s$):
6      $\alpha = 0$
7      if $l' == 0$ && $k' == 0$ then
8          for $n = 0$ to $H_f - 1$ do
9              for $m = 0$ to $W_f - 1$ do
10                 $V[\alpha]$ = SIMD_Load($\mathcal{F}_{n,m,i'\times 4\sim i'\times 4+3}$)
11                 $\alpha$ += 1
12     else
13         $\alpha = W_f \times H_f$
14     for $ll = 0$ to $H_{o,b} - 1$ do
15         for $kk = 0$ to $W_{o,b} - 1$ do
16             $l = l' \times H_{o,b} + ll$
17             $k = k' \times W_{o,b} + kk$
18             $V[\alpha]$ = SIMD_Load($\mathcal{O}_{l,k,i'\times 4\sim i'\times 4+3}$)
19             $\alpha$ += 1
20     for $ll = 0$ to $H_{o,b} - 1$ do
21         for $kk = 0$ to $W_{o,b} - 1$ do
22             $l = l' \times H_{o,b} + ll$
23             $k = k' \times W_{o,b} + kk$
24             for $n = 0$ to $H_f - 1$ do
25                 for $m = 0$ to $W_f - 1$ do
26                     $V[\alpha]$ = SIMD_Load(
                        $\mathcal{I}_{l\times s+n,k\times s+m,i'\times 4\sim i'\times 4+3}$)
27                     V[$H_f \times W_f + ll \times W_{o,b} + kk$] =
                        SIMD_FMA($V[\alpha]$,$V[n\times W_f+m]$,
                        $V[H_f \times W_f + ll \times W_{o,b} + kk]$)
28     $\alpha = W_f \times H_f$
29     for $ll = 0$ to $H_{o,b} - 1$ do
30         for $kk = 0$ to $W_{o,b} - 1$ do
31             $l = l' \times H_{o,b} + ll$
32             $k = k' \times W_{o,b} + kk$
33             SIMD_Store($\mathcal{O}_{l,k,i'\times 4\sim i'\times 4+3}$, $V[\alpha]$)
34             $\alpha$ += 1
```

Roofline Model The roofline model (Williams, Waterman, and Patterson 2009) is often used to understand the estimated performance of a given compute kernel running on a type of processor by showing the inherent hardware limitations, and potential benefit and priority of optimizations (e.g., locality, bandwidth, and different parallelization tricks). The roofline model, however, focuses on cache misses. In other words, it focuses on the traffic between the cache and the memory and assumes if the program is well optimized with little cache miss, the program could fully utilize the hardware. The key metric inside the roofline model is "operational intensity" (OI), which measures the average number of floating-point operations that can be carried out per byte of memory loaded from the memory.

Arithmetic Intensity "Arithmetic Intensity" (AI) (Harris 2005) measures the average number of floating-point operations that can be carried out per byte of memory loaded from the cache to the register.[1] This is exactly what we want to go after if the memory bottleneck can be removed. Let W be the number of arithmetic operations carried out, β be the number of bytes transferred between cache and registers, the arithmetic intensity T is $\frac{W}{\beta}$.

Given a particular layer of convolution (e.g., DWConv), W is a constant as it is dedicated by the problem definition and algorithm, a larger T means the implementation is more efficient because there are fewer data transferred between the cache and the registers, which implies the implementation is doing a good job in keeping the data in the register as long as it is necessary.

Depthwise Convolution

Core Inscalability Existing implementations of DWConv have poor scalability on the number of cores. Take TF-Lite implementation as an example (Algorithm 2), it picks the H_o dimension as the outer-most loop to apply thread parallelism (Line 1). In other words, given p cores, each core is assigned with a chunk of output feature map in size of $H_o/p \times W_o \times C$ to compute.

Since the chunk spans over all the output channels, each core has to copy the whole filter $\mathcal{F}$ of size $H_f \times W_f \times C$ into its tiny L1 cache. In other words, when the input feature map, the filter, and the output feature map cannot all fit into the L1 cache, the number of L1 cache misses will fly high. Furthermore, the situation exacerbates with the number of layers because the filters are getting larger when they appear deeper in the model.

Poor AI Although the implementation of DWConv in TF-Lite has good performance from the perspective of OI (and thus in terms of cache misses when we do **not** use more cores), its performance is next limited by its poor arithmetic intensity. This is not an issue on x86 processors. However, this is a big issue on ARM processor because ARM processors can only load 1 register per cycle while it can process 2 SIMD FMA instructions per cycle. In other words, if we do not optimize the pipeline well, the FMA instructions are always waiting for data to be loaded to the registers.

To be specific, we first analyze the AI of TF-Lite implementation (Algorithm 2). Its inner-most loop is able to process 4 output elements in parallel by SIMD (Line 10). In order to do so, however, it has to carry out 3 SIMD load instructions (Lines 7–9) to retrieve the filter, input and output respectively from cache to registers, and 1 SIMD store instruction to write back the updated output elements to L1 cache (Line 11). Thus, the arithmetic intensity of this implementation is $T_{tf}^{DW} = \frac{1\times2\times4 \texttt{ ops}}{4\times16 \texttt{ bytes}} = \frac{1}{8}$. If the width W_f of the filter and the number of channels C are small, compilers may keep $W_f \times C$ elements of the filter in the register for the kk loop (Line 4). To give TF-Lite such benefit of doubt, we assume this happens and thus its arithmetic intensity can become $T_{tf}^{DW} = \frac{1\times2\times4 \texttt{ ops}}{(3+\frac{1}{W_{o,b}})\times16 \texttt{ bytes}} = \frac{1}{(3+\frac{1}{W_{o,b}})\times2} < \frac{1}{6}$. Nonetheless, it is still a very poor number.

Our implementation Algorithm 4 is our proposed implementation. To address the core inscalability problem, we reschedule the loop order and picks the C dimension as the outer-most loop to apply thread parallelism (Line 1). This way, each core is assigned with a chunk of output feature map in size $H_o \times W_o \times \frac{C}{p}$ to compute. Under such parallelism, since a chunk only spans C/p output channels, each core needs to retrieve $H_f \times W_f \times C/p$ elements of the filter $\mathcal{F}$ to its L1 cache. Compared with TF-Lite implementation that retrieves $H_f \times W_f \times C$ elements of the filter $\mathcal{F}$ to the L1 cache, we fetch only $1/p$ of those in cache, which significantly reduce the cache misses and improve the core scalability.

To improve the arithmetic intensity, we exploit different techniques to increase the reuse of the data in the register as much as we can. The first technique we applied is *register tiling* (Jiménez, Llabería, and Fernández 2002) (Lines 2 and 3). It splits the filter $\mathcal{F}$ into tiles of size $H_f \times W_f \times 4$. By doing so, a tile can be kept in the registers as long as possible. The kernel is used to compute the convolution results of a small output block of size $H_{o,b} \times W_{o,b} \times 4$. $H_{o,b}$ and $W_{o,b}$ are set to ensure the output block stay in the registers across the `Kernel`. The kernel is skillfully tuned to increase its AI by reducing the traffic between the registers and the cache. Specifically, lines 7 to 11 in the kernel aim to load the filter into the registers. However, this load process is only done when $l' = 0$ and $k' = 0$ (Line 7), meaning for the nested loops in lines 2 and 3, the filter is only loaded **once** and stays in the registers for long. Lines 14 to 19 in the kernel aim to load a specific output block of size $H_{o,b} \times W_{o,b} \times 4$ into the registers. Notice that this specific output block is only loaded **once** and would never get re-loaded again. Similarly, Lines 29 to 34 in the kernel aim to store the updated output block back to the cache. Again this specific output block is only stored **once**, as it would never get re-loaded for any further processing after it carries out the FMA in lines 20-27.

We now analyze the AI of our implementation. That

[1]We remark that there is a misconception online (e.g., Wikipedia) that OI is equivalent to AI. That misconception comes from the fact that cache miss is the major bottleneck on x86 processors and thus the traffic between the registers and the cache is immaterial after the cache bottleneck is resolved. However, for ARM processors, it is not the case.

would help us to see why it outperforms the existing implementations. It is easy to know the arithmetic operations are all inlined within the `Kernel`. In the `Kernel`, the number of arithmetic FMA operations all lies in lines 18–25, which has 4 `for` loops. So, the FMA operation is carried out $\mathcal{W} = H_{o,b} \times W_{o,b} \times H_f \times W_f$ times. Thus, the number of floating-point operations is $8 \times \mathcal{W}$, which will be the numerator in the AI.

The denominator of AI captures the number of bytes transferred between cache and registers. For our implementation, it involves:

1. Loading the filter block once (Lines 7-11) across the nested two loops l' and k' (Lines 2 and 3) and reused $H_o/H_{o,b} \times W_o/W_{o,b}$ times. Thus, `Kernel` incurs an average of $\frac{H_f \times W_f}{W_o/W_{o,b} \times H_o/H_{o,b}} \times 16$ bytes traffic between the registers and cache.
2. Loading the output block once (Lines 14-19) and storing once (Lines 29-34) in the kernel. So, the traffic for output block in `Kernel` is $H_{o,b} \times W_{o,b} \times 2 \times 16$ bytes.
3. Loading one SIMD register data of $\mathcal{I}$ in the inner-most loop (Lines 20-27). Thus, the traffic for $\mathcal{I}$ is $16 \times H_{o,b} \times W_{o,b} \times H_f \times W_f = 16 \times \mathcal{W}$ bytes.

Putting it all together, the AI of our implementation is:

$$T^{DW} = \frac{8 \cdot \mathcal{W}}{16(\frac{H_f \cdot W_f}{W_o/W_{o,b} \cdot H_o/H_{o,b}} + H_{o,b} \cdot W_{o,b} \cdot 2 + \mathcal{W})} \quad (1)$$

Since the size of the filter is either 3×3 or 5×5, and the block sizes $H_{o,b}$ and $W_{o,b}$ are empirically set as 1 or 2 (they are set with the objective of saving some registers because we indeed apply loop unrolling to the 4 tightly-nested loops in Lines 18-25). So, the term $\frac{H_f \times W_f}{W_o/W_{o,b} \times H_o/H_{o,b}}$ is negligible. Therefore, we rewrite equation (1) as $T^{DW} = \frac{(H_f \times W_f)}{(2+H_f \times W_f) \times 2} \geq \frac{9}{22}$, which is obviously way larger than T^{DW}_{tf}.

Pointwise Convolution Implementation

Core Inscalability TF-Lite's PWConv implementation by default calls the MM multiplication rountine in Eigen (Guennebaud, Jacob, and others 2010). However, it is known that OpenBLAS (OpenBLAS 2015) has the best performance and thus we set TF-Lite to use OpenBLAS instead. Nonetheless, it is known that current matrix-multiplication implementations including OpenBLAS cannot scale well on multiple cores for deep learning workload (Zhang, Franchetti, and Low 2018; Zhang et al. 2018; Rajbhandari et al. 2017).

Poor AI Algorithm 5 is the implementation of a BLAS MM routine (e.g., `SGEMM` in OpenBLAS). It has applied *loop blocking* to increase data reuse in the memory hierarchy. Its kernel is the function `RTRA` (Line 4), which stands for Register Tiling Reuse block A. The logical view of `RTRA` is depicted in Figure 3 (left). It first SIMD loads a block of matrix A, which is represented as $\boxed{A}$, into the registers

Algorithm 5: Matrix Multiplication in BLAS Libraries

```
Input: Matrix A of size $(G \times C_i)$, Matrix B of size
       $(C_i \times C_o)$;
Output: Matrix D of size $(G \times C_o)$;
1 for $i' = 0$ to $C_i/C_{i,b}$ do
2     for $g' = 0$ to $G/G_b$ in parallel do
3         for $j' = 0$ to $C_o/C_{o,b}$ do
4             RTRA$(i', g', j')$
```

(Line 2). $\boxed{A}$ is of size $G_b \times C_{i,b}$. The elements of $\boxed{A}$ stay in the registers across the *j' loop* (Line 3 in Algorithm 5) and are reused $C_o/C_{o,b}$ times.

Inside the function `RTRA` (Figure 3), Line 3 aims to stream a block of matrix $\boxed{B}$ and a block of matrix $\boxed{D}$ into the registers. $\boxed{D}$ is of size $(G_b, C_{o,b})$ and $\boxed{B}$ is of size $(C_{i,b}, C_{o,b})$. A matrix multiplication between $\boxed{A}$ and $\boxed{B}$ is performed to update $\boxed{D}$ (Line 4), and it costs $\frac{G_b \times C_{i,b} \times C_{o,b}}{4}$ FMA operations and the number of floating-point operations is $2 \times G_b \times C_{i,b} \times C_{o,b}$. Finally, the updated $\boxed{D}$ has to be stored to the cache.

The AI of BLAS MM implementation is as follows. The arithmetic operations are all inlined in the kernel `RTRA`. In routine `RTRA`, its AI is:

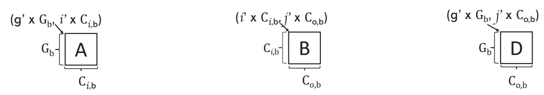
$$T^{PW}_{RTRA} = \frac{2 \times G_b \times C_{i,b} \times C_{o,b} \text{ ops}}{(G_b \times C_{o,b} \times 2 + C_{i,b} \times C_{o,b} + \frac{G_b \times C_{i,b}}{C_o/C_{o,b}}) \times 4 \text{ bytes}}$$

Since AArch64 has 32 128-bit SIMD registers, in order to fully allocate the registers, G_b, $C_{i,b}$ and $C_{o,b}$ are usually set as 8, 8 and 4 in the BLAS Libraries (e.g., OpenBLAS). Then, we can get $T^{PW}_{RTRA} = \frac{4}{3+\frac{8}{C_o}}$. Note that the `RTRA` kernel has a poor AI because $\boxed{D}$ has to be transferred twice between the cache and the registers (one load and one store).

Algorithm 6: High Performance Matrix Multiplication

```
Input: Matrix A of size $(G \times C_i)$, Matrix B of size
       $(C_i \times C_o)$;
Output: Matrix D of size $(G \times C_o)$;
1 for $g' = 0$ to $G/G_b$ in parallel do
2     for $j' = 0$ to $C_o/C_{o,b}$ do
3         for $i' = 0$ to $C_i/C_{i,b}$ do
4             RTRD$(i', g', j')$
```

Our Implementation We propose another loop blocking method with better AI (Algorithm 6). It calls another kernel `RTRD` (Register Tiling Reuse block D), whose concept is listed in Figure 3 (right). `RTRD` first loads block $\boxed{D}$ into the registers. The elements of $\boxed{D}$ stay in the registers across the *i' loop* (Line 3; Algorithm 6) and are reused. After that, it streams blocks $\boxed{A}$ and $\boxed{B}$ into the registers and then evaluates a small matrix multiplication to update $\boxed{D}$ (Line 4).



```
Algorithm RTRA( i', g', j' )
if j' == 0:
    Load $\boxed{A}$ into registers
Load $\boxed{D}$ and $\boxed{B}$ into registers
$\boxed{D}$ += $\boxed{A}$ × $\boxed{B}$
Store $\boxed{D}$ to cache
```

```
Algorithm RTRD( i', g', j' )
if i' == 0:
    Load $\boxed{D}$ into registers
Load $\boxed{A}$ and $\boxed{B}$ into registers
$\boxed{D}$ += $\boxed{A}$ × $\boxed{B}$
if i' >= $C_i/C_{i,b}$:
    Store $\boxed{D}$ to cache
```

Figure 3: RTRA vs RTRD

Differ from RTRA, RTRD only stores the block $\boxed{D}$ to the cache in the last iteration of loop i'. Though this way is inefficient on x86 processor (Smith et al. 2014), it is very efficient for ARM processors because ARM processors is sensitive to AI. The arithmetic intensity of RTRD for MM multiplication is:

$$T^{PW}_{RTRD} = \frac{2 \times G_b \times C_{i,b} \times C_{o,b} \texttt{ ops}}{(G_b \times C_{i,b} + C_{i,b} \times C_{o,b} + \frac{G_b \times C_{o,b} \times 2}{C_i/C_{i,b}}) \times 4 \textbf{ bytes}}$$

To fully allocate the registers, we can set $G_b = 8$, $C_{o,b} = 8$ and $C_{i,b} = 4$. Thus, $T^{PW}_{RTRD} = \frac{2}{1+\frac{8}{C_i}}$ it is about 1.5× larger than T^{PW}_{RTRA}, since C_o and C_i are often much larger than 8. Of course, our actual implementation also includes all the optimization tricks such as software prefetching, loop rolling etc. But we do not repeat them here.

Experimental Evaluation

In this section, we present performance results of our high performance depthwise and pointwise convolution on mobile devices. We run our experiments on a 2.0GHz quad-core ARM Cortex-A57. Each core has 48KB L1 instruction cache and 32KB L1 data cache. All cores share 2MB unified L2 cache. We compare performance of our DWConv and PWConv implementations with two versions of TF-Lite, one links to OpenBLAS (OpenBLAS 2015) and the other one links to Ruy (Google 2019). In addition, we compare the performance with TVM (Chen et al. 2018). TVM implementations suppose to deliver performance as good as the performance offered by manually optimizing the implementation for a specific hardware. The DWConvs and PWConvs operations in this study are extracted from MobileNetV1(Howard et al. 2017), MobileNetV2(Sandler et al. 2018) and MnasNet (Tan et al. 2018). They are different in input size, output size and filter size.

Performance Figure 4 to Figure 6 show the speedup of our implementations (and TVM) with respect to TF-Lite, on different DWConv and PWConv extracted from MobileNetV1, MobileNetV2, and MnasNet-A1, respectively. For example, in Figure 4, $D1$ to $D9$ refer to nine different DWConvs found in MobileNetV1. Results show that our DWConv implementation outperforms TF-Lite at least by 2.9× and up to 9.0×. In addition, our DWConv implementation outperforms TVM generated binaries by at least 1.4× and up to 5.5×, showing that TVM is not able to reach the level of optimizations that we can achieve.

Our PWConv implementation achieves 1.3× to 5.1× speedup over TF-Lite(OpenBLAS), which is essentially calling the OpenBLAS library for MM multiplication. Our PWConv implementation also achieves up to 2.1× speedup over TF-Lite(Ruy), which uses the aggressively tuned library Ruy to implement PWConv. In addition, our PWConv implementation achieves 1.05× to 2.11× speedup over TVM, which once again shows TVM is not able to reach the level of optimizations that we can achieve.

Scalability In Figure 7, we compare the scalability of our DWConv and PWConv performances with respect to the number of cores. We include TF-Lite (which uses OpenBLAS to implement PWConv) there for comparisons.[2] For space reasons, we only include the results from MobileNetV1 as results from MobileNetV2 and Mnasnet-A1 are largely similar.

From Figure 7, we see that our implementations scale better than TF-Lite. We almost achieve perfect speedup when using 2 threads, which is very promising because every parallel program has its serial part based on Amdahl's law. When using 4 threads, the core instability of TF-Lite immediately manifest – TF-Lite has only around 2× speedup on DWConv and 1.8× to 2.7× on PWConv. In contrast, our implementations achieve 2.2× to 3.9× speedup on DWConv and 3.2× to 3.9× speedup on PWConv.

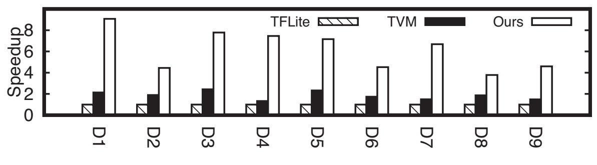
Related Work

Most works on optimizing deep learning operations focus only on conventional convolutions (Zhang, Franchetti, and Low 2018; Cho and Brand 2017; Georganas et al. 2018; Rajbhandari et al. 2017) but not depthwise and pointwise convolutions appeared in mobile models. To our best knowledge, this paper is the first to discuss the optimization of depthwise and pointwise convolutions on mobile processors. In (Qin et al. 2018), there are treatments to improve the performance of DWConv, but they focus on training and GPU, whereas our focus is on inference and ARM. TVM (Chen et al. 2018) is a compiler stack for generating highly efficient binaries for deep network. It supports CPU, GPU, ARM, and specialised accelerators. Our experimental results show that binaries optimized by TVM not yet fully utilize the power of mobile processors. BLAS libraries (Smith et al. 2014; OpenBLAS 2015; Guennebaud, Jacob, and others 2010) offer highly efficient implementations for PWConv. However, we are able to show that they are still lacking on mobile devices.

Conclusions and Future Work

In this paper, we show that existing implementations of depthwise convolution and pointwise convolution are not

[2]We do not include TVM here because TVM generates different binaries for different number of threads, making things incomparable.


(a) DWConv: Speedup over TFLite

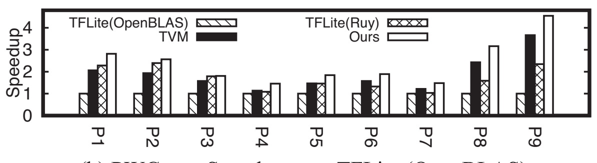
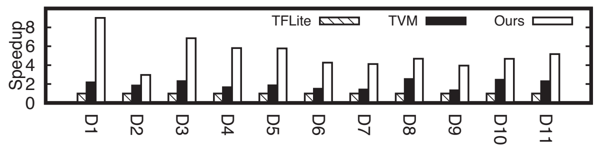

(b) PWConv: Speedup over TFLite (OpenBLAS)

Figure 4: MobileNetV1 (4/4 cores/threads)


(a) DWConv: Speedup over TFLite

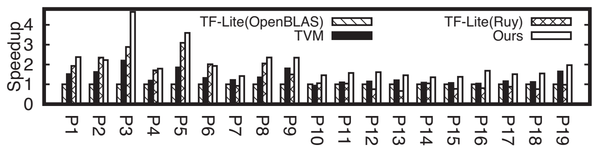
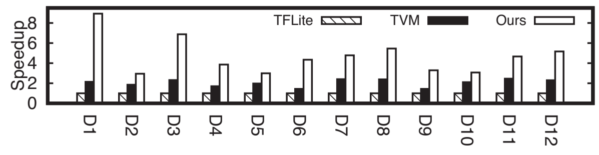

(b) PWConv: Speedup over TFLite (OpenBLAS)

Figure 5: MobileNetV2 (4/4 cores/threads)


(a) DWConv: Speedup over TFLite

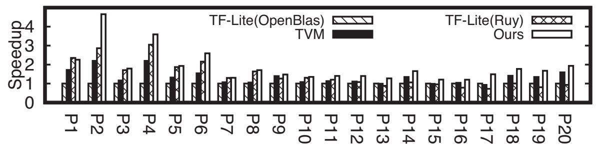
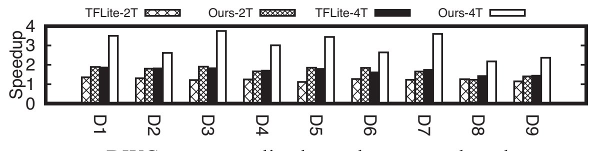

(b) PWConv: Speedup over TFLite (OpenBLAS)

Figure 6: MnasNet-A1 (4/4 cores/threads)


DWConv: normalized speedup to one thread

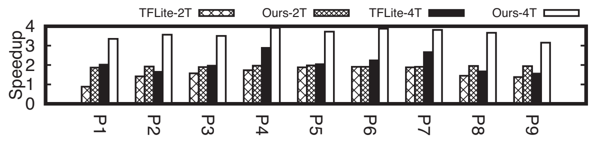

PWConv: normalized speedup to one thread

Figure 7: Scaling behavior with increasing number of threads

efficient enough on mobile devices. The major reason is that those implementations have not considered the fact that ARM processors are getting more cores as well as the latency gap between the load and FMA instructions in ARM processors.

To this end, we re-optimize the implementations of DW-Conv and PWConv specifically for ARM. That is because ARM processors are dominating the mobile device market and there is an increasing demand to carry out inference directly on the mobile devices. Experimental results show that our implementations can outperform industry-strength implementations from TF-Lite as well as optimized binaries generated from TVM. Using MobileNetV1 as an example, our optimized implementation can carry out inference at 46GFlops, a performance that is almost hitting the roofline of ARM processors. The encouraging result also reveals one important future work for us. Since TVM is a compiler framework for deep learning models, our results indicate that we incorporate our techniques (e.g., register tiling) into TVM so to make it generate highly efficient binaries for mobile models on mobile devices.

Acknowledgment

This work is supported by Hong Kong General Research Fund (14200817, 15200715, 15204116), Hong Kong AoE/P-404/18, Innovation and Technology Fund ITS/310/18.

References

Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; Guestrin, C.; and Krishnamurthy, A. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*.

Cho, M., and Brand, D. 2017. MEC: memory-efficient convolution for deep neural network. In *ICML*.

Chollet, F. 2016. Xception: Deep learning with depthwise separable convolutions. *CoRR*.

Dongarra, J. J., and Hinds, A. R. 1979. Unrolling loops in FORTRAN. *Softw., Pract. Exper.*

Dongarra, J. J.; Croz, J. D.; Hammarling, S.; and Duff, I. S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*

Georganas, E.; Avancha, S.; Banerjee, K.; Kalamkar, D.; Henry, G.; Pabst, H.; and Heinecke, A. 2018. Anatomy of high-performance deep learning convolutions on simd architectures. In *SC*.

Google. 2019. Ruy. https://github.com/tensorflow/tensorflow.

Guennebaud, G.; Jacob, B.; et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

Han, S.; Shen, H.; Philipose, M.; Agarwal, S.; Wolman, A.; and Krishnamurthy, A. 2016. MCDNN: an approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*.

Harris, M. 2005. Mapping computational concepts to gpus. In *ACM SIGGRAPH Courses*.

Howard, A. G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; and Adam, H. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*.

Jiménez, M.; Llabería, J. M.; and Fernández, A. 2002. Register tiling in nonrectangular iteration spaces. *TOPLAS*.

Loc, H. N.; Lee, Y.; and Balan, R. K. 2017. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *MobiSys*.

Markatos, E. P., and LeBlanc, T. J. 1992. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *SC*.

OpenBLAS. 2015. http://www.openblas.net.

Qin, Z.; Zhang, Z.; Li, D.; Zhang, Y.; and Peng, Y. 2018. Diagonalwise refactorization: An efficient training method for depthwise convolutions. In *IJCNN*.

Rajbhandari, S.; He, Y.; Ruwase, O.; Carbin, M.; and Chilimbi, T. M. 2017. Optimizing cnns on multicores for scalability, performance and goodput. In *ASPLOS*.

Sandler, M.; Howard, A. G.; Zhu, M.; Zhmoginov, A.; and Chen, L. 2018. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*.

Smith, T. M.; van de Geijn, R. A.; Smelyanskiy, M.; Hammond, J. R.; and Zee, F. G. V. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *IPDPS*.

SoftBank Group. 2017. Arm business strategy. https://group.softbank/en/corp/d/annual-reports/2017/future-forward/segars-interview/.

Tan, M., and Le, Q. V. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*.

Tan, M.; Chen, B.; Pang, R.; Vasudevan, V.; and Le, Q. V. 2018. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*.

Williams, S.; Waterman, A.; and Patterson, D. A. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*.

Xue, J. 2000. *Loop Tiling for Parallelism*. Norwell, MA, USA: Kluwer Academic Publishers.

Zhang, M.; Rajbhandari, S.; Wang, W.; and He, Y. 2018. Deepcpu: Serving rnn-based deep learning models 10x faster. In *ATC*.

Zhang, J.; Franchetti, F.; and Low, T. M. 2018. High performance zero-memory overhead direct convolutions. In *ICML*.