FeatherCNN: Fast Inference Computation with TensorGEMM on ARM Architectures

Haidong Lan, Jintao Meng, Christian Hundt, Bertil Schmidt, *Senior Member, IEEE*, Minwen Deng, Xiaoning Wang, Weiguo Liu, Yu Qiao, and Shengzhong Feng

Abstract—Deep Learning is ubiquitous in a wide field of applications ranging from research to industry. In comparison to time-consuming iterative training of convolutional neural networks (CNNs), inference is a relatively lightweight operation making it amenable to execution on mobile devices. Nevertheless, lower latency and higher computation efficiency are crucial to allow for complex models and prolonged battery life. Addressing the aforementioned challenges, we propose *FeatherCNN* – a fast inference library for ARM CPUs – targeting the performance ceiling of mobile devices. *FeatherCNN* employs three key techniques: 1) A highly efficient TensorGEMM (generalized matrix multiplication) routine is applied to accelerate Winograd convolution on ARM CPUs, 2) General layer optimization based on custom high performance kernels improves both the computational efficiency and locality of memory access patterns for non-Winograd layers. 3) The framework design emphasizes joint layer-wise optimization using layer fusion to remove redundant calculations and memory movements. Performance evaluation reveals that *FeatherCNN* significantly outperforms state-of-the-art libraries. A forward propagation pass of VGG-16 on a 64-core ARM server is 48, 14, and 12 times faster than Caffe using OpenBLAS, Caffe2 using Eigen, and NNPACK, respectively. In addition, *FeatherCNN* is 3.19 times faster than the recently released TensorFlow Lite library on an iPhone 7 plus. In terms of GEMM performance, *FeatherCNN* achieves 14.8 and 39.0 percent higher performance than Apple's Accelerate framework on an iPhone 7 plus and Eigen on a Samsung Galaxy S8, respectively. The source code of *FeatherCNN* library is publicly available at https://github.com/tencent/feathercnn.

Index Terms—Convolutional neural networks, ARM architecture, inference computation, tensorGEMM

1 Introduction

The historic shrinking of the manufacturing process of semiconductors over three orders-of-magnitude has powered 50 years of advances in High Performance Computing. During the last decade, single-core performance of modern CPUs has been stagnating due to hard architectural limitations of silicon-based processors. The free lunch is over, so to speak. While server CPUs struggle to compete with Moore's law, the accumulated compute capacity of mobile devices continues to grow at Moore's rate. First, from 2014 to 2017, the peak performance (see Table 1) of *A series* chips built in Apple devices has improved by a factor of 3.1, while Snapdragon performance has improved 21.8× over 4 years. Second, new architectures such as ASIC [1], Risc-v [2], and DSP invigorate the mobile market, thus causing a decline in price and shortening of tape-out cycles. Third, in 2017 alone, 1.5 billion mobile phones have been sold. Assuming 50 GFlops average performance per device, their overall theoretical peak performance outmatches the world's fastest supercomputer – *Summit* – located in Oak Ridge National Laboratory by a factor of 397 [3].

In the recent past, a plethora of Deep Learning applications have dispersed from an exclusively scientific domain into the consumer market. The traditional design of deep classification and regression networks incorporates both the time-consuming training of model parameters and lightweight inference on heavyweight processors such as high-end CUDA-enabled accelerator boards or even GPU clusters. In the context of mobile computing, centralized processing of real time input data is considered intractable, as these devices are expected to generate unprecedented amounts (petabytes) of data which would have to be transferred between cloud and edge devices. Hence, inference operations performing forward-pass evaluations of already trained neural networks should be delegated to edge devices.

The existence of powerful mobile processors in combination with on-device inference establishes the need for highly efficient inference solutions in order to support complex models and sustaining battery life. Inference on edge devices is

- *H. Lan, M. Deng, and X. Wang are with the Tencent AI Lab, Shenzhen 518000, China. E-mail: turbo0628@163.com, {danierdeng, xningwang}@tencent.com.*
- *J. Meng is with the Tencent AI Lab, Shenzhen, 518000, China, and also with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China. E-mail: jintaomeng@tencent.com.*
- *C. Hundt and B. Schmidt are with the Parallel and Distributed Architectures Group, Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz 55122, Germany. E-mail: christian@metalabs.de, bertil.schmidt@uni-mainz.de.*
- *W. Liu is with the Shandong University, Jinan 250100, China. E-mail: weiguo.liu@sdu.edu.cn.*
- *Y. Qiao and S. Feng are with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China. E-mail: {yu.qiao, sz.feng}@siat.ac.cn.*

Manuscript received 7 Aug. 2018; revised 18 Aug. 2019; accepted 1 Sept. 2019. Date of publication 6 Sept. 2019; date of current version 10 Jan. 2020. (Corresponding authors: Jintao Meng and Bertil Schmidt.) Recommended for acceptance by B. He. Digital Object Identifier no. 10.1109/TPDS.2019.2939785

TABLE 1
The Theoretical Peak Single-Precision Performance in GFlops and Memory Bandwidth in GB/s of Selected Chips used in Edge and Server Devices During 2014 to 2017

SOC Name	Year	BW	Performance in GFlops		
		(GB/s)	*CPU*	*GPU*	Overall
Apple A8	2014	12.8	44.8	115.2	160
Apple A9	2015	25.6	88.8	172.8	261.6
Apple A10	2016	25.6	112.3	384.6	495
Snapdragon 410	2014	8.5	19.2	10.8	30
Snapdragon 616	2015	12.8	35.2	29.7	129.8
Snapdragon 821	2016	29.9	37.4	260.0	297.4
Snapdragon 835	2017	29.9	78.4	576.0	654.4

TABLE 2
Computational Footprint of Various Layer Types Measured in Terms of MFlops for Four CNN Architectures

Network	Convolution layer			FC	Pool	Other
	Wino	General	DW			
VGG-16	29,271	0	0	236	6	13
GoogLeNet	1,836	1,180	0	2	12	166
ResNet-50	3,528	3,827	0	4	2	407
MobileNet–V1	0	1,052	33	0	0	73
Inception–V3	4,684	6,209	0	2	25	27
Inception–V4	7,459	15,911	0	2	45	46

Here Wino, General, DW, FC, Pool and Other denotes winograd convolution, general convolution, depth-wise convolution, fully-connected pooling and other types of layers respectively.

challenging since: (1) There is a diverse set of chips and software environments. Edge devices may contain ARM CPUs, GPUs, FPGAs, or other specialized hardware such as neural processing units (NPUs). In addition, we have to deal with different software environments, namely operating systems covering iOS, Android, embedded Linux, and programming models including Neon/Vulkan/OpenCL/OpenGL. Among them, ARM CPUs are prominent owing to their broad deployment on mobile devices and the consistent programming model across various software platforms. (2) Basic primitives in Convolutional Neural Networks (CNNs) exhibit vastly different behavior in terms of compute intensity, memory bandwidth, and latency which may be interleaved in a non-trivial manner. Their individual and joint optimization is a crucial task. (3) Traditional computation patterns of aforementioned primitives may exhibit redundant calculations wasting valuable resources. Hence, detecting and eliminating superfluous re-computation of quantities can increase performance. (4) Inference computation exhibits inherently less parallelism than data parallel training since entities are processed one by another which demands for orthogonal parallelization techniques.

Our contributions are three-fold:

- *Optimizing Winograd convolutions using small kernels:* We reformulate the constituent transformations of the Winograd algorithm to support efficient SIMD vectorization. Furthermore, a novel TensorGEMM subroutine based on *Internal Packing* and *External Packing* techniques is introduced allowing for two times higher performance compared to established GEMM routines for matrix multiplication.
- *Non-Winograd layer optimization:* We accelerate general, depth-wise convolution and pooling layer computation using offline kernel packing techniques and cache block adaptation to further increase efficiency.
- *Lightweight framework design and layer fusion:* A slim and self-contained inference framework is developed. We further increase inference throughput using layer fusion. This is achieved by reusing intermediate results stored in registers in order to mitigate superfluous memory accesses.

Performance evaluation of *FeatherCNN* reveals that our implementation is orders-of-magnitude faster than state-of-the-art solutions. Using *FeatherCNN* on a 64-core ARM server, a forward-pass of VGG-16 is 48, 14 and 12 times faster than Caffe [4] with OpenBLAS [5] backend, Caffe2 [6] with Eigen [7] backend, and NNPACK [8], respectively. More specifically, the forward-pass for VGG-16 operates with 10 frames per second (FPS) on the ARM server, and 3.52 FPS on an iPhone 7 plus. In addition, *FeatherCNN* outperforms the recently released TensorFlow Lite [9] library by a factor of 3.19 on an iPhone 7 plus. In terms of GEMM performance, *FeatherCNN* achieves 14.8 and 39 percent higher performance than Apple's Accelerate framework [10] on an Apple iPhone 7 plus and Eigen [7] on a Samsung Galaxy S8, respectively.

The rest of the paper is organized as follows. Mathematical foundations of inference computation are discussed in Section 2. Three optimization approaches on TensorGEMM accelerated Winograd convolution, general layer optimization and layer fusion are described in Section 3. Performance evaluation is carried out in Section 4. Section 5 discusses previous work on inference optimization. Section 6 concludes the paper.

2 Inference Computation using CNNs

2.1 Mathematical Foundations

Convolutional Neural Networks (CNNs) approximate differentiable functions as a contiguous cascade of affine maps that are interleaved with non-linearities such as activation functions, extreme value and mean value projections (max/min/average-pooling), as well as normalization and dropout layers. The main computational load during inference can be attributed to matrix-vector and matrix-matrix products. Table 2 shows that convolutional layers occupy more than 95 percent of the computational load in four commonly used networks. Note that we abbreviate Winograd-style convolutions as *Winograd*, general convolutions as *General*, depth-wise convolutions as *DW*, fully-connected layers as *FC*, and pooling layers as *Pool*. In particular, we distinguish between three classes of convolutional layers:

- *Winograd-style convolution layers:* Apply several filter masks of shape 3×3 to an input image using unit stride and subsequently agglomerate input color channel contributions.
- *Depth-wise convolution layers:* Treat contributions of color channels independently without subsequent agglomeration.
- *General convolution layers:* Do not fit any of the two definitions of aforementioned convolutional layer types.

In the following, we briefly discuss the mathematical foundations of those aforementioned layers.

A convolutional layer maps a batch consisting of N *input images* of shape $H \times W$ and C color channels onto N *output images* of shape $E \times F$ and K color channels. We assign the corresponding tensor ranks in the order [batch, channel, height, width] resulting in an "NCHW" signature for the input and "NKEF" for the output. In general, the convolutional layer combines contributions of all ranks except rank zero – the batch identifier. Hence, distinct images within a batch can be processed independently. Moreover, images are usually processed one by another during inference such that data parallel batch processing is infeasible. Consequently, we set $N = 1$ for better readability.

The *general convolutional* layer computes the cross-correlation between fixed-sized filters of shape $R \times S$ for all combinations of K channels of the output image $S_{k,x,y}$ and C channels of the input image $D_{c,x,y}$. The corresponding parametrization $G_{k,c,u,v}$ is a 4th degree tensor with signature "KCRS" which stores the $R \times S$ filter weights for every channel combination (k, c). Contributions of input channels are accumulated in order to reduce $K \times C$ finite-impulse-responses to exactly K output channels:

$$S_{k,x,y} = \sum_{c=0}^{C-1} \sum_{u=0}^{R-1} \sum_{v=0}^{S-1} D_{c,x+u,y+v} \cdot G_{k,c,u,v} \quad , \tag{1}$$

where $0 \leq k < K,\ 0 \leq c < C,\ 0 \leq x < H - R,\ 0 \leq y < W - S$. When using non-unit stride, the sums over x and y would increment with step size $stride > 1$. The naïve evaluation of Equation (1) results in $\Theta\big((K \times C) \cdot (H \times W) \cdot (R \times S)\big)$ operations. Note that it is possible to reduce the theoretical time complexity for filter shapes $R \times S \gg \log(H \times W)$ to a log-linear dependency in the number of pixels by employing fast convolution using point-wise Hadamard products in Fourier space [11]. However, mainstream CNN architectures rely on the iterative evaluation of small filters (usually 3×3) over a relatively large feature map (from 12×12 to 224×224 and even larger) which rules out spectral approaches from the very beginning. A third – less known – approach is the Winograd algorithm [12] which reduces the number of evaluations in Equation (1) by exploiting the algebraic structure of the underlying ring $(\mathbb{R}, +, \cdot)$.

The *Winograd convolution* algorithm introduces several 2-dimensional schemes. Without loss of generality, we focus on the optimization of the 3×3-tap finite-impulse-response (FIR) filter $F(2 \times 2, 3 \times 3)$ which produces 2×2 outputs. We further validate the techniques on a larger configuration, namely $F(6 \times 6, 3 \times 3)$. Note that Winograd schemes exhibit a slightly higher memory footprint than traditional general convolution schemes. In general, the memory usage for kernel elements is $(K \times C) \cdot (R \times S)$. The Winograd algorithm, however, demands more memory to store the offline transformed filter elements. The memory requirement for $F(m \times m, r \times r)$ equals $(K \times C) \cdot (m + r - 1)^2$ for a specialized filter with $R = r$ and $S = r$. Therefore the expansion factor is $\frac{1}{r^2}(m + r - 1)^2$. The general 2-dimensional formula of the $F(2 \times 2, 3 \times 3)$ FIR filter can be written as

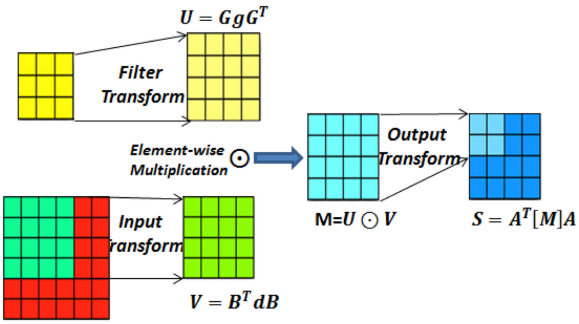


Fig. 1. Illustration of the 2-D convolutional Winograd algorithm with $F(2 \times 2, 3 \times 3)$.

$$S = A^T\big(\big[GgG^T\big] \odot \big[B^T dB\big]\big)A = A^T\big(U \odot V\big)A. \tag{2}$$

Here B, G, A are constant matrices with fixed values defined in [12], g is a 3×3 matrix embedding the filter entries, and d is a 4×4 sliding block extracted from the input images. Fig. 1 illustrates the basic workflow of the 2-dimensional convolutional Winograd algorithm. Computation is partitioned into four stages.

1) *Filter transform:* $U = GgG^T$
2) *Input transform:* $V = B^T dB$
3) *Element-wise Multiplication:* $M = U \odot V$
4) *Output transform:* $S = A^T MA$

An arithmetic complexity reduction by a factor of 2.25 in comparison to general convolution in Equation (1) has been proven in [12]. Nevertheless, a straightforward implementation of Lavin's Winograd scheme on ARM processors such as provided by NNPACK [8] may result in sub-optimal utilization of resources. The computation of $F(2 \times 2, 3 \times 3)$ can be further accelerated by a factor of up to 2 as shown in Section 3.1.

Depth-wise convolutions are designed to significantly reduce computational complexity and the amount of parameters in recent CNN architectures [13], [14]. In contrast to general convolutions, they compute $K = C \cdot \alpha$ output channels from C input channels without subsequent color channel reduction where $\alpha \geq 1$ is a channel multiplier. In the following, we discuss the special case for $\alpha = 1$ (i.e., $K = C$), its corresponding parametrization $G_{c,u,v}$ is a 3rd degree tensor with signature "CRS" which stores the filter weights for every input channel. The computation scheme is shown in

$$S_{c,x,y} = \sum_{u=0}^{R-1} \sum_{v=0}^{S-1} D_{c,x+u,y+v} \cdot G_{c,u,v} \quad , \tag{3}$$

where $0 \leq c < C,\ 0 \leq x < H - R,\ 0 \leq y < W - S$. Note that the spatial coordinates x and y can be sampled using potentially non-unit $stride \geq 1$.

The transformations in Equation (2) accept 4×4 tiles from the input image and subsequently compute 2×2 finite-impulse-responses that are stored in the output image. The evaluation of Equation (2) involves linear conversions of the factors U and V and their element-wise multiplication. Depending on the platform, this might be inefficient if performed in a naïve manner. In the following, we discuss the computational pattern step by step.

TABLE 3
Shape and Computational Load of VGG-16 Conventional Layers

Layer Name	C	K	H, W	$H' \times W'$ F(2x2,3x3)	$H' \times W'$ F(6x6,3x3)	GFLOP
conv1.1	3	64	224	12544	1444	0.17
conv1.2	64	64	224	12544	1444	3.70
conv2.1	64	128	112	3136	361	1.85
conv2.2	128	128	112	3136	361	3.70
conv3.1	128	256	56	784	100	1.85
conv3.2-3.3	256	256	56	784	100	3.70
conv4.1	256	512	28	196	25	1.85
conv4.2-4.3	512	512	28	196	25	3.70
conv5.1-5.3	512	512	14	49	9	0.92

2.2 Implementation Details of Winograd Convolutions

In the following, we briefly discuss the workflow of Lavin's proposed Winograd algorithm [12] and subsequently discuss improvements over the original implementation to enhance performance.

1) During input transform, the input image is processed in multiple passes. A sliding window of t consecutive tiles is transformed simultaneously in one of those passes. Here t equals 3 for x86_64 CPUs and 32 for CUDA-enabled GPUs.
2) In each pass, the elements generated by the input transform of each tile must be packed into an interleaved layout using a stride of $C \times W' \times H'$ to support traditional GEMM routines.
3) A batch $\big((U^0, V^0), \ldots, (U^i, V^i), \ldots, (U^{\theta-1}, V^{\theta-1})\big)$ consisting of θ matrix pairs U^i and V^i of fixed shapes $K \times C$ and $C \times W' \times H'$ is processed using `cblas_sgemm` routines.
4) During the output transformation phase, the processed results of the GEMM calls remain in the aforementioned interleaved data layout which may have to be repacked for later usage. This involves extensive data movement.

The proposed strategy allows for a straightforward implementation by means of existing and highly-optimized GEMM routines. Nevertheless, it introduces a number of issues related to the achievable performance on common CPU architectures.

a) Both input and the output transforms are memory-bound resulting in low computational intensity. This diminishes the benefits of replacing scalar arithmetic by SIMD instructions.
b) Scattering the data block generated by the input transform into θ matrix pairs using interleaved indexing and packing the layout back into continuous order after GEMM involves enormous memory movement on both input and output transforms thus significantly reducing the overall performance.
c) Right factor matrices V^i exhibit a long rectangular shape. See Table 3 for a comprehensive list of used shapes of 13 convolutional layers in VGG–16. Unfortunately, existing GEMM routines are typically not optimized for this case. Recent work on optimizing batched processing of small GEMMs [15] is also not applicable due to large matrix shapes occurring in the Winograd algorithm.

To address the aforementioned concerns, we propose a reformulated Winograd algorithm by embedding a specialized variant of inner tensor product computation – namely TensorGEMM – at its core. Our approach is based on the following three key points:

- *Embedding of TensorGEMM:* The Winograd algorithm is reformulated by embedding TensorGEMM at its core – a memory-aware linear algebra primitive for the efficient computation of single precision tensor-valued inner products.
- *Reduction of Memory Movement:* The tensors are agglomerate before TensorGEMM is performed instead of scattering the data block generated by the input transform into θ matrices. This approach reduces the data movement stride by a factor of θ regardless of the specific processor instructions. Note that the same strategy can be applied to data reorganization before the output transform.
- *Improvement of TensorGEMM Efficiency:* Our TensorGEMM routine is optimized with register blocking to achieve the maximal compute-to-memory access ratio. Internal and external packing is further used to minimize the overhead of data placement incurred during TensorGEMM.

3 Optimization Techniques

3.1 Expressing Winograd Convolutions in Terms of TensorGEMM Primitives

Initially, we rewrite Equation (1) in Winograd fashion according to the coordinate representation of Equation (2):

$$\begin{aligned} S_{k,d} &= \sum_{c=0}^{C-1} A^T \left[U_{k,c} \odot V_{c,d}\right] A \\ &= A^T \left[\sum_{c=0}^{C-1} U_{k,c} \odot V_{c,d}\right] A \quad , \end{aligned} \tag{4}$$

where $0 \leq k < K$, and $d \in [0, H') \times [0, W')$ is a spatial index enumerating tiles in the output image. For $F(m \times m, r \times r)$, the number of $m \times m$ output tiles along each dimension is $H' = \lceil \frac{H-r+1}{m} \rceil$ and $W' = \lceil \frac{W-r+1}{m} \rceil$. The matrix A is constant and thus can be safely moved outside the sum to avoid redundant recomputation of matrix products. As a result, the final *output transformation* $S = A^T MA$ is only computed once.

The two linear conversions resulting in U and V are the *filter transform* and the *input transform*. The *input transform* iterates over output channels K while the *filter transform* is independent of the image tiles. Hence, the remaining *element-wise multiplication* takes the major share. Let $M_{k,d} = \sum_{c=0}^{C-1} U_{k,c} \odot V_{c,d}$ be the agglomerated pointwise products along input color channels then $M^i_{k,d} = (U^i \circ V^i)_{k,d}$ denotes the ith entry of a Winograd tile where $0 \leq i < \theta$ and $\circ$ is the traditional matrix product. Note that $F(m \times m, r \times r)$ produces tiles with $\theta = (m + r - 1)^2$ elements resulting in $\theta = 16$ entries in case of $F(2 \times 2, 3 \times 3)$. The coordinate representation $M^i_{k,d}$ can be reinterpreted

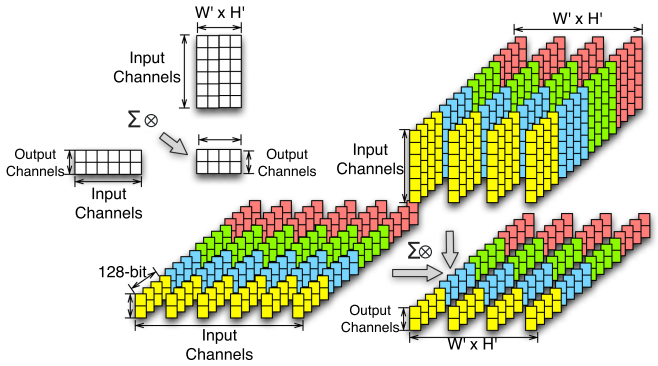


Fig. 2. Upper left: 2-dimensional illustration of Equation (5) where each element is a vector of length 16. Bottom-right: A corresponding 3-dimensional illustration of $F(2 \times 2, 3 \times 3)$.

as plain matrix multiplication over a batch of θ factors U^i and V^i :

$$M^i_{k,d} = \sum_{c=0}^{C-1} U^i_{k,c} \cdot V^i_{c,d} \quad \forall i,k,d \quad , \tag{5}$$

whereby $0 \leq k < K$, $0 \leq d < H' \times W'$ and $0 \leq i < \theta$. As a result, element-wise multiplication illustrated in Fig. 2 could be performed using θ consecutive calls to GEMM using matrices of dimensions $K \times C$ and $C \times (H' \times W')$. While this is pleasing in terms of code complexity performance suffers as explained in the issues a), b), and c) in Section 2.2.

In order to relieve data movement introduced by interleaved data storage before and after θ GEMMs, TensorGEMM accommodates a distinct data layout incorporating θ-length tensors which inherently match the Winograd transformations. Consequently, many matrix multiplications are expressed in terms of a single tensor-valued inner product.

Definition of TensorGEMM. Tensors are higher-order rank generalizations of linear maps storing data over a regular grid of arbitrary dimension. As an example, rank zero tensors are scalars, rank one denotes vectors, rank two corresponds to matrices etc. Inner products of two tensors $\mathcal{A}$ and $\mathcal{B}$ with compatible shape can be interpreted as their pointwise Hadamard product $\mathcal{A} \odot \mathcal{B}$ and subsequent sum-reduction over a set of shared rank identifiers. In this paper, *TensorGEMM* shall be defined as inner product of two rank three tensors $\mathcal{A}_{k,c,i}$ and $\mathcal{B}_{c,d,i}$. Point-wise multiplication of entries can be accomplished after (virtually) broadcasting the tensors to their corresponding rank by embeddings $\hat{\mathcal{A}}_{k,c,d,i} = \mathcal{A}_{k,c,i}$ for all d and $\hat{\mathcal{B}}_{k,c,d,i} = \mathcal{B}_{c,d,i}$ for all k in order to guarantee compatible shapes. As shown before, TensorGEMM can be rewritten as a cascade of rank two inner products (batched ordinary matrix multiplications):

$$\begin{aligned} \mathcal{C}_{k,d,i} &= \sum_c \left(\hat{\mathcal{A}} \odot \hat{\mathcal{B}}\right)_{k,c,d,i} &&= \sum_c \hat{\mathcal{A}}_{k,c,d,i} \cdot \hat{\mathcal{B}}_{k,c,d,i} \\ &= \sum_c \mathcal{A}_{k,c,i} \cdot \mathcal{B}_{c,d,i} &&= \sum_c \left(\mathcal{A}_{k,c} \cdot \mathcal{B}_{c,d}\right)_i = \left(\mathcal{C}_{k,d}\right)_i \end{aligned} \tag{6}$$

In the following, we identify the Winograd index $0 \leq i < \theta$ with L lanes in vector registers, here $\theta = 16$ in case of $F(2 \times 2, 3 \times 3)$. Hence all additive contributions over C input color channels can be accumulated in an inner loop

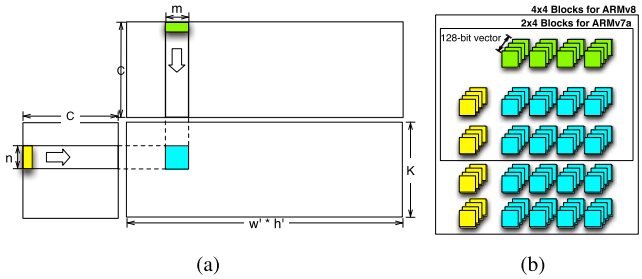


Fig. 3. Illustration of the processing order in TensorGEMM on ARM architectures. Note that, each element in the matrix is a tensor, which maps to a 128-bit vector register containing 4 floats on ARM-based architectures.

over c in a vector register $\mathcal{C}_{k,d,i^*} = \mathcal{C}_{k,d,i^*} + \mathcal{A}_{k,c,i^*} \cdot \mathcal{B}_{c,d,i^*}$ where $i^* = (0, \ldots, L-1)$ is a sequence of lane identifiers. If $L < \theta$ we have to perform multiple no-warm-up passes. In pass p where $0 \leq p < \frac{\theta}{L}$, we accumulate Winograd indices in the range $p \cdot L \leq i < (p+1) \cdot L$. Since current ARM architectures feature 128 bit vector registers storing $L = 4$ single precision floating-point values, we need $p = 4$ no-warm-up passes to compute a total of $\theta = 16$ independent contributions for $F(2 \times 2, 3 \times 3)$. The remaining loops over the output channel index k, the spatial coordinates d, and the pass identifier p are accelerated by means of multi-threading.

The discussed Winograd reformulation can be adapted to other architectures by adjusting the parameters θ calculated from $F(m \times m, r \times r)$, vector length L, and running passes p. As an example, $F(6 \times 6, 3 \times 3)$ can be realized on x86_64 CPUs with AVX512 support by setting $\theta = 64$, $L = 16$, $p = 4$. Further extensions to half precision (FP16) and INT8 support on modern hardware accelerators are conceivable. In this case, we have to multiply the parameter L by a factor of 2 or 4, respectively. However, we focus on current ARM architecture throughout the rest of this paper.

3.2 Optimization of TensorGEMM

3.2.1 Register Blocking

TensorGEMM partitions tensors into basic blocks and subsequently computes as many results as possible between every two memory accesses. Initially, we set up a group of accumulator registers and afterwards load several entries from a column in $\mathcal{A}$ and from a row in $\mathcal{B}$. In a subsequent phase, the tensors are multiplied and accumulated in registers. When computation progresses to the right border of $\mathcal{A}$ and the bottom of $\mathcal{B}$, we write back the accumulators. Fig. 3 shows a schematic view of the described computational pattern.

Let us carry out a compute-to-memory-access ratio (CMAR) analysis [16]. Let $m \cdot n$ be the number of tensors loaded each time, we perform $2 \cdot m \cdot n$ vectorized floating-point instructions during each iteration. The SIMD vector register usage is m for $\mathcal{A}$, n for $\mathcal{B}$, and $m \cdot n$ for the accumulators, respectively. Let R be the amount of available registers per core, then the block size should satisfy $(m + n + m \cdot n) \leq R$. Under this constraint, the CMAR can be maximized:

$$\text{CMAR} = \frac{2 \cdot m \cdot n}{m + n}.$$

Note that the CMAR tends to be higher for larger block sizes. On ARMv8-based architectures with 32 vector registers per

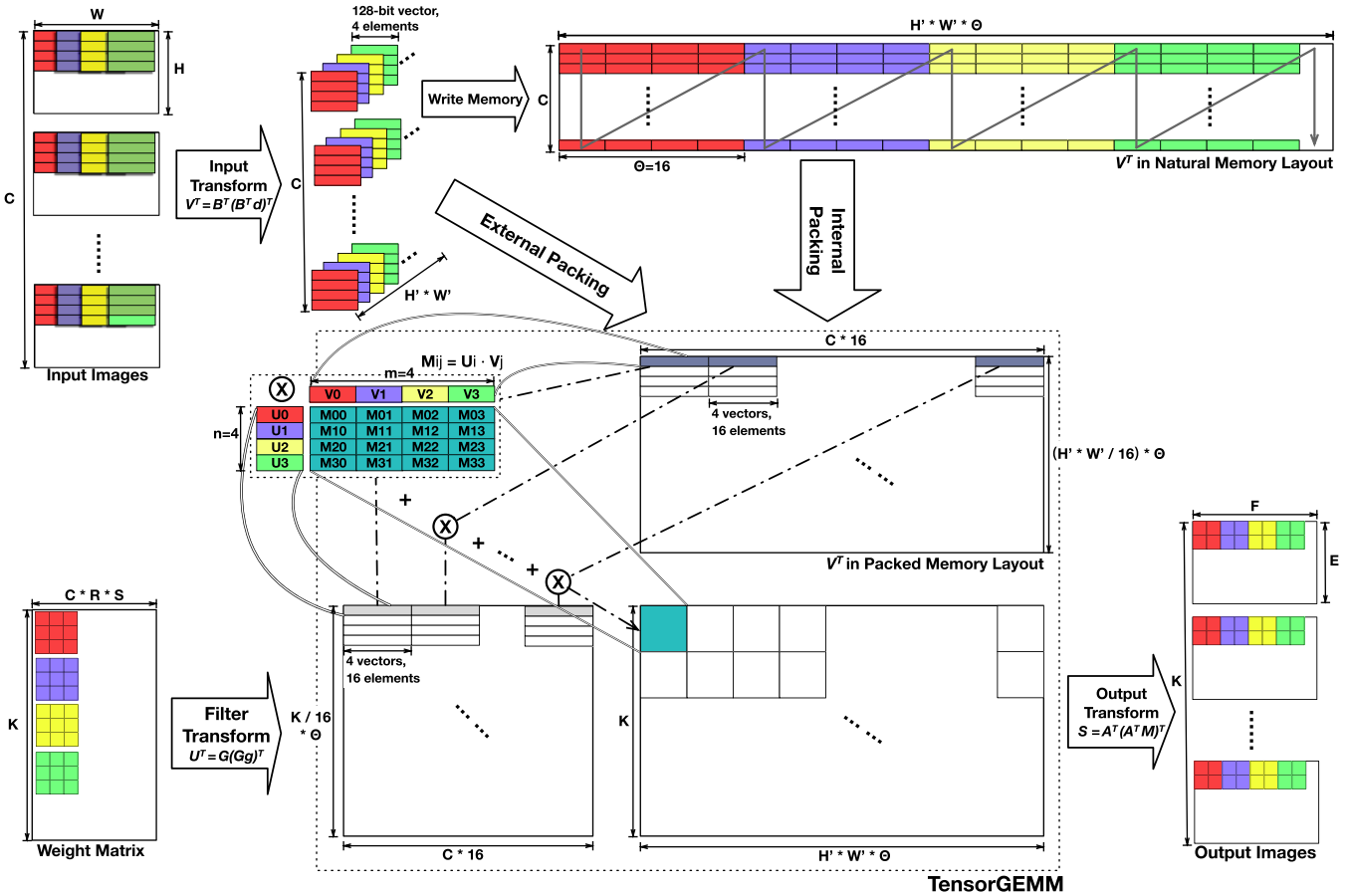


Fig. 4. Illustration of our memory packing strategy. The upper right matrix shows the natural memory layout of input transformation and the subsequent loading order in TensorGEMM. The packing strategy straightens the buffer for sequential access in TensorGEMM. Specifically, register blocks (see Fig. 3) require vectors from m tiles in V and n tiles in U to be bundled. Here we only illustrate the $m = 4, n = 4$ case. The desired packed memory layout interleaves adjacent m and n tiles for efficient vector loading, and arranges the vector bundles as they are loaded in TensorGEMM's innermost loop, say multiplication and accumulation. The internal packing strategy packs a chunk of data when it is loaded into last level cache by TensorGEMM. We also leverage cache to diminish packing overhead. The external packing strategy skips the original layout of input transformation by writing data directly in packed order, so as to eliminate the packing overhead in TensorGEMM. External packing is only applied on small images due to considerable memory writing overhead on large images.

core, we use 4×4 blocks which occupy 24 registers yielding $\mathrm{CMAR} = 4$. Larger blocks either need more registers than available or are hard to partition. In case of ARMv7a-based architectures providing only 16 vector registers per core, we use a smaller block size of 2×4, which employs 14 vector registers with a CMAR of 2.3.

Meanwhile, the CMAR for a plain GEMM subroutine can be larger than the CMAR for TensorGEMM, where the register blocks are usually 4×12 or 8×8 elements. Plain GEMM tends to have a higher CMARs owing to larger register blocks. However, the above analysis ignores the write-back overhead for the accumulators. Note that TensorGEMM performs L times more arithmetic operations in each iteration than plain GEMM calls. In the Winograd context where L is small, TensorGEMM writes back the accumulators significantly less often than plain GEMM subroutines. Hence, TensorGEMM performs better on stretched rectangular matrices which typically occur in Winograd schemes.

3.2.2 Memory Layout

As mentioned before, we perform $p = 4$ no-warm-up passes to the fixed-length subroutine on ARM architectures to propagate a convolutional layer of type $F(2 \times 2, 3 \times 3)$. Each pass computes a part of these tensors, which are denoted in same color in Fig. 2. We use 4 vectors to store a 4×4 tile each holding a row. The vectors are subsequently serialized into an intermediate buffer for repetitive use in the tensor-valued inner product computations. We have carefully designed its memory layout which agglomerates the tensors in order to minimize data movements, while also ensure high accessing efficiency in the subsequent TensorGEMM.

As register blocks progress across rows in the matrix, the access pattern is unsuitable for the processors' cache hierarchy. We have carefully applied packing strategies to the working arrays such that computation traverses memory in a contiguous manner (see Fig. 4). In particular, the arrays are rearranged according to TensorGEMM's loading order (see Fig. 3). The filter transformation of the left factor U is constant during inference. Hence it is packed before computation at the initialization stage. The input transformation V (right factor) varies during inference of distinct images. Hence, we apply a packing strategy in conjunction with cache blocking. We first load a part of the matrix which could fit in the last level of the cache, and subsequently

pack the array into cache. Loop orders are adjusted such that this array is re-used until no longer needed. This method is denoted as "*internal packing*"

The Winograd algorithm's multi-stage character allows for the moving of the packing procedure to the input transform phase. Consequently, the input transform instantly writes its results in packed order, such that TensorGEMM receives a packed working array from the input transform. This method shall be denoted as "*external packing*".

The efficiency of the two packing approaches depends on the size of the input images. For large images, the internal packing method based on cache blocking is superior. The external packing approach does re-ordering in the input transform stage prior to TensorGEMM. It is complex to jointly leverage cache blocking and external packing due to their occurrences in different Winograd stages. As a consequence, this approach causes significantly more cache misses as it writes a large row-major intermediate buffer by column. In case input images are small enough to fit into cache, the writing overhead of the external packing approach increases only marginally, and therefore achieves better performance.

3.2.3 Optimizing Sandwich Products

The input transformation, filter transformation, and output transformation exhibit a common mathematical form $\varphi_{\mathcal{A}} : \mathcal{B} \mapsto \mathcal{C} = \varphi_{\mathcal{A}}(\mathcal{B}) := \mathcal{A}^T \mathcal{B} \mathcal{A}$ where $\mathcal{B}$ is sandwiched between a fixed-value matrix $\mathcal{A}$ and its transposed variant. Multiplying a matrix from the left manipulates rows in contrast to right actions which alter columns. As our matrices are stored in row-major order, left operands can be applied efficiently. As a result, we have rewritten the transformations as $\mathcal{C} = \mathcal{A}^T(\mathcal{A}^T\mathcal{B}^T)^T = \mathcal{A}^T\mathcal{B}\mathcal{A}$ exploiting fast transposition in registers. In case of Winograd convolution, Equation (2) can be rewritten by exploiting the equality $(U \odot V)^T = U^T \odot V^T$:

$$S = A^T(U \odot V)A = A^T(A^T(U^T \odot V^T))^T \quad . \tag{7}$$

According to Equations (6) and (7), we can finally reformulate four stages of Winograd algorithm using TensorGEMM:

1) *Filter Transform:* $U^T = G(Gg)^T$
2) *Input Transform:* $V^T = B^T(B^T d)^T$
3) *TensorGEMM:* $(M^T)^i = (V^T)^i \times (U^T)^i, 0 \leq i < \theta$
4) *Output Transform:* $S = A^T(A^T M^T)^T$

Algorithm 1 illustrates the unfolded pseudo-code of the input transformation in case of $F(2 \times 2, 3 \times 3)$. The output transformation and filter transformation are implemented accordingly.

The 8 Neon vector instructions perform 32 floating-point operations, which is already the theoretical minimum. Minor overhead is caused by the transposition of 4 registers.

3.3 Non-Winograd Layer optimization based on Packing and Blocking

With Amdahl's law, after achieving orders of magnitude speedup based on fine-tuned optimization of Winograd convolutional layers, the time taken to process the remaining layer types will be non-negligible. In the following, the optimization of two other compute-intensive layers – namely general and depth-wise convolution will be discussed.

Algorithm 1. The Compact Vectorized Kernel to Compute the input Transformation of $F(2 \times 2, 3 \times 3)$

Data: A 4×4 block in the input image d
Result: A 4×4 block in the input transformation V^T
Load rows of d into vectors vD_0, vD_1, vD_2, vD_3
$vW_0 = vD_0 - vD_2$;
$vW_1 = vD_1 + vD_2$
$vW_2 = vD_2 - vD_1$;
$vW_3 = vD_3 - vD_1$
Transpose vW_0, vW_1, vW_2, vW_3
$vD_0 = vW_0 - vW_2$;
$vD_1 = vW_1 + vW_2$
$vD_2 = vW_2 - vW_1$
$vD_3 = vW_3 - vW_1$
Store vD_0, vD_1, vD_2, vD_3 to V^T

3.3.1 General Convolution

For general convolutional layers, we apply the *im2col* method [17] to stretch the convolution tiles into columns, and subsequently perform matrix multiplication using an optimized GEMM subroutine. The stretched input image exhibits a shape of $(C \times R \times S) \cdot [(W - R + 1) \times (H - S + 1)]$. This introduces an expansion factor of roughly $R \times S$ of the input size, which is a major drawback. Hence, the *im2col* subroutine produces noticeable overhead, which is however compensated by the high performance of the optimized GEMM routines. Furthermore, this method is flexible to accommodate a wide variety of convolutional layers by applying different data reshaping methods.

On the edge devices, the diverse runtime environments and operating systems complicate performance portability. Therefore, we develop our own SGEMM subroutine for ARM CPUs specifically tailored to usual matrix shapes during neural network inference. The major optimization techniques are similar to [16] with a minor adaption: the filter matrix is packed only once at initialization such that we can save the on-the-fly packing overhead. A CMAR analysis similar to the TensorGEMM (Section 3.2.1) yields efficient 8×8 and 4×8 register block shapes for ARMv8 and ARMv7-a based architectures, respectively. We also employ a home-grown thread pool implementation which is portable across iOS, Android and Linux in case of lacking OpenMP support.

3.3.2 Depth-Wise Convolution

In the following, we isolate optimization of depth-wise convolution from general convolution since color channels are computed independently. Hence the accumulation over C is skipped. For the Winograd approach, the TensorGEMM has to update its accumulators quite often, which dwarfs the advantages of aforementioned optimization approaches. Note that *im2col* generates images of shape $(R \times S) \cdot [C \times (H - R + 1) \times (W - S + 1)]$. Since R and S are usually small in modern CNN architectures, current GEMM routines are unlikely to be efficient considering the effort for reshaping. Therefore, we take a straightforward approach to vectorize convolutions directly. We group several convolutional tiles in lanes of registers, and accumulate multiplications thereby. Performance is sensitive to the group shape. In the case of

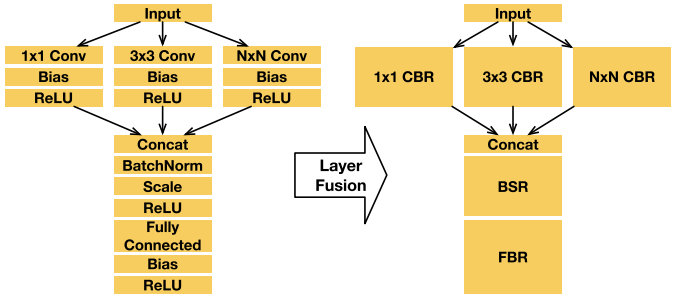


Fig. 5. Illustration of layer fusion with respect to CBR, FBR, and BSR patterns.

accelerating 3×3 depth-wise convolutions on ARMv8 ARM architectures, we agglomerate 2×8 tiles together. We provide several alternative implementations to cover most typical cases. In all other cases it falls back to a specialized general convolution routine.

3.4 Framework Design and Optimization

FeatherCNN's layer and parameter definitions are compliant to Caffe [4] in order to guarantee compatibility with pre-trained networks from the model zoo. In terms of performance, we emphasize execution latency. Edge devices are usually equipped with power-efficient compute units and memory modules operating at low frequency which results in limited bandwidth. Hence, *FeatherCNN* follows three major principles:

1) The framework should be capable to compactly invoke compute routines.
2) Memory movement should be minimized considering bandwidth and power limitations.
3) A self-contained code base which allows for easy deployment on a variety of platforms.

These principles are enforced by co-designing the framework with the compute-intensive primitives in mind. We provide our own optimized domain-specific library instead of using BLAS subroutines such that we can fuse different layer types at the function level. Layer fusion allows for the significant reduction of memory accesses especially in case of memory-bound activation layers that simply scan the image. For example, *Rectified Linear Unit* (ReLU) and *Bias* layers simply traverse the image, zero negative values, or add certain offsets. *Batch Normalization* layers often appear with subsequent *Scale* layers to compute affine transformations. The formulae of such layers do simple arithmetic with respect to every single element. Therefore, memory accesses are the major overhead. We agglomerate computation into a fused layer in order to perform a number of successive computations within a single memory traversal. In practice, three layer patterns are fused: Convolution-Bias-ReLU (CBR), Fully Connected-Bias-ReLU (FBR), and Batch Normalization-Scale-ReLU (BSR). Layer fusion on a commonly seen network substructure is illustrated in Fig. 5.

On the implementation level, different convolution types require distinct approaches to layer fusion. We activate the output tiles in the Winograd *output transformation* before writing them to memory. The accumulators in the GEMM subroutine of general convolution are activated at the last pass of accumulation. The depth-wise compute function can also fuse these computations in a single writing pass. Please note that layer fusion is hard to realize using third party BLAS calls since they are forced to write back intermediate results before termination.

Our framework scans the network for potentially fusible layers and combines them subsequently. Hereby, it yields a more compact network topology. As an example, we can reduce ResNet-50 from 229 layers to 127 layers and MobileNet–V1 from 111 layers to 57 layers by fusing all *ReLU* and *Scale* layers. Even though these layers have small computational load, layer fusion can significantly improve thread scalability.

4 Performance Evaluation

FeatherCNN is a lightweight self-contained framework implemented in C++, ARM NEON instructions, and OpenMP/POSIX Threads. FeatherCNN currently supports 15 layer types with carefully optimized and/or fused compute kernels. Most commonly used neural networks can be transformed and executed by *FeatherCNN* to provide real-time inference computation with high efficiency. It can be easily deployed on a wide range of edge devices, and is portable to a variety of operating systems and C++ compilers. In the following, *FeatherCNN's* performance is evaluated using a representative selection of neural networks and hardware devices.

4.1 Experimental Setup

FeatherCNN's performance has been assessed on two mobile devices, two ARM servers and an embedded development board. Regarding mobile devices, we have tested a Samsung Galaxy S8 and an Apple iPhone 7 plus. The Galaxy S8 is equipped with an octa-core Qualcomm Snapdragon 835 processor and 4GB memory. The iPhone 7 plus is powered by a quad-core A10 Fusion processor featuring two Apple-customized performance cores and two efficient cores. The memory capacity is limited to 2GB. The ARM server manufactured by Huawei exhibits two CPU sockets each housing a 32-core Cortex-A72 system on chip (SOC). Regarding the SOC, every eight cores are connected with a ring. The 4 core groups are interconnected with an on-chip network. Another platform, Phytium FT1500A, integrates only a single CPU socket powered by a 16-core FTC660 chip with support for ARMv8 instructions, and is running a 64-bit Kylin Linux. Finally, the developer board Firefly-RK3399 is equipped with 2 Cortex-A72 performance cores, 4 Cortex-A53 efficient cores and 4GB main memory.

The detailed hardware specifications are listed in Table 4. VGG-16, GoogLeNet, ResNet-50, and MobileNet–V1 are selected from Table 2 as candidates for subsequent performance evaluation.

4.2 Step-Wise Evaluation

In order to highlight the benefits of our optimization techniques, performance measures are evaluated in terms of three aspects:

1) Runtimes for Winograd convolutions are assessed for different layer configurations and decomposed into *input transform*, *TensorGEMM*, and *output transform* contributions.

TABLE 4
The Detailed Hardware Specifications of our Test Platforms

Device	*Processor*	*#CPUs@Clock Speed*	*CPU Arch.*	*Memory*	*OS*	*SOC Power*
Samsung Galaxy S8	Snapdragon 835	4@2.45+ 4@1.90GHz	Kryo	4 GB	Android 7.0	≈5W
Vivo IQOO	Snapdragon 855	1@2.84 + 3@2.42+ 4@1.80GHz	Kryo	8 GB	Android 9.0	≈5W
Xiaomi 8SE	Snapdragon 710	2@2.2 + 6@1.70Ghz	Kryo	4 GB	Android 8.0	≈5W
Huawei Mate 10	Kirin 970	4@2.4 + 4@1.80GHz	Cortex	4 GB	Android 7.0	≈5W
Apple iPhone 7 plus	A10 Fusion	2@2.34 + 2@1.05GHz	Hurricane	2 GB	iOS 11.1	≈5W
Huawei D05 Server	Hi1616	2 × 32@2.40GHz	Cortex	256 GB	Ubuntu 16.04	> 100W

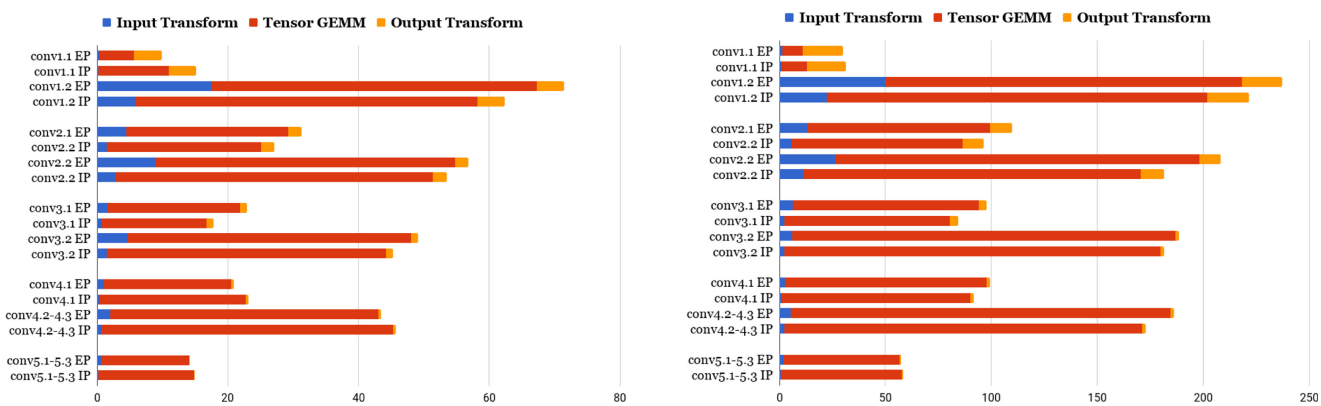
2) A side-by-side evaluation of achieved performance (in terms of GFLOPS) for 3×3 convolutional layers processed with *General Convolution* and the *Winograd* approach.
3) Measuring the impact of layer fusion.

Regarding the first two aspects, we have tested convolutional layers of *VGG–16* with respect to the shapes and computational loads listed in Table 3. The dimensions of TensorGEMM multipliers are $(K \cdot \theta) \times C$ and $C \times (H' \cdot W' \cdot \theta)$ (see Fig. 4). The *VGG–16*'s convolutional layers cover a wide variety of representative shapes generally composed of two typical patterns: large images with relatively few channels and small images with more channels. Similar shapes also appears in ResNet, DenseNet, SqueezeNet, and many other frequently used neural network architectures. Hence, we have decided to use *VGG–16* for testing representative Winograd convolution evaluation. In order to investigate the impact of layer fusion, we have tested *MobileNet–V1*, *GoogLeNet*, and *ResNet–50* covering a variety of neural network architectures.

We investigate the performance of our single-threaded Winograd implementation on an iPhone 7 plus and a Galaxy S8 by separately evaluating the *External Packing* and the *Internal Packing* strategy. Results are averaged over 20 runs and illustrated in Fig. 6. For large images (*conv1.2-conv2.2*), the input transformation consumes a considerable portion of time. The internal packing strategy outperforms external packing due to its more efficient memory access pattern. TensorGEMM exhibits similar performance due to the properly applied cache blocking approach. In case of small images (*conv4.1-5.3*), input/output transformations are performed in negligible time. Hence, the processing of TensorGEMM dominates the compute time. External packing performs better in case the image buffer fits into cache. Note that the two devices produce diametrical results for the *conv4.1-4.3* layers. This can be explained by the fact that the iPhone 7 plus is equipped with a larger cache which is able to accommodate the respective images. TensorGEMM consequently benefits from the reduced overhead by virtue of external packing.

Regarding absolute performance, we have evaluated the convolutional layers listed in Table 3 using GEMM and the Winograd approach on an iPhone 7 plus and a Galaxy S8. Performance is measured in terms of GFLOPS. To ensure a fair comparison to plain GEMM routines, we evaluate Winograd's performance in terms of "effective" performance to mirror both implementation and algorithmic speedups, which may yield a number even higher than the theoretical peak performance. The effective performance is calculated by dividing GFLOPs of the standard algorithm as listed in last column of Table 3 by the compute time. The input transformation, TensorGEMM, and the output transformation are included in the compute time, while the filter transformation is excluded as it is performed at initialization.

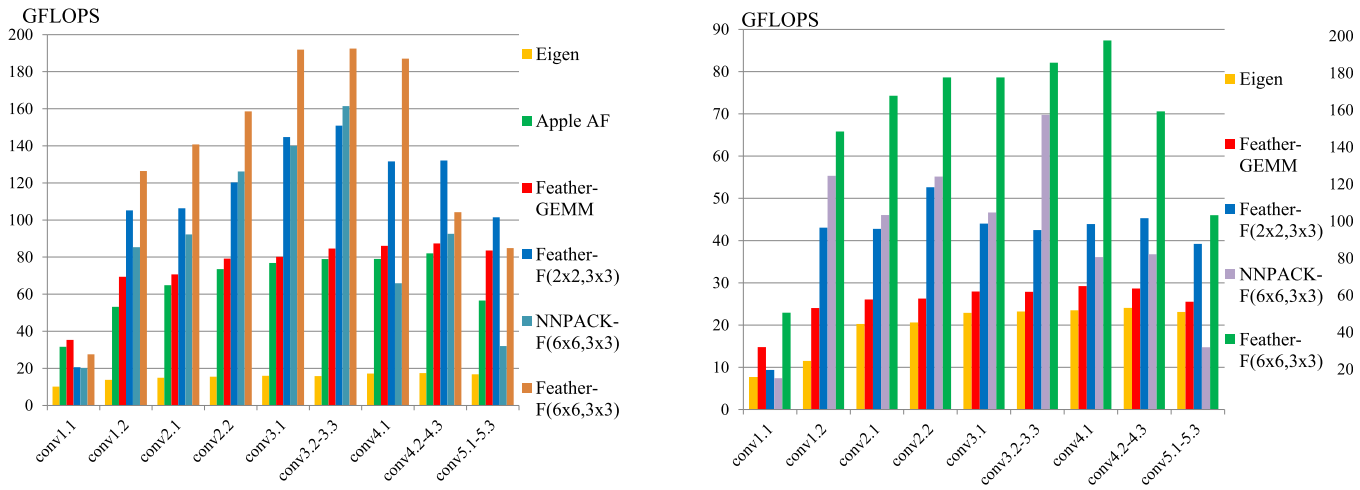
Regarding the GEMM approach, we exclude the *im2col* time to reflect bare-metal performance. Except for the *conv1.1* layer, the FeatherCNN Winograd $F(2 \times 2, 3 \times 3)$ scheme outperforms its own GEMM implementation between 21.5 and 80.6 percent on an iPhone 7 plus and between 50.4 and 100.3 percent on a Galaxy S8. The small input channel number of $C = 3$ in the *conv1.1* layer renders TensorGEMM less efficient in this case. Regarding the Winograd $F(6 \times 6, 3 \times 3)$ scheme, we compare to



(a) iPhone 7 plus, single thread.

(b) Galaxy S8, single thread.

Fig. 6. Decomposed Winograd F(2×2, 3×3) compute time (in milliseconds) for convolutional layers in VGG-16. IP denotes "Internal Packing" and EP refers to "External Packing".



(a) iPhone 7 plus, 2 threads. (b) Galaxy S8, 4 threads.

Fig. 7. VGG-16 Layerwise Performance on Mobile Phones.

NNPACK, which implements the approach proposed in [12] on ARM architectures. On an iPhone 7 plus and Galaxy S8, FeatherCNN outperforms NNPACK by 36 to 183 percent and 42 to 212 percent with respect to distinct layer configurations. FeatherCNN is the more stable performer across different layer settings. This can be explained by the high efficiency of TensorGEMM across various neural network dimensions, as well as unit-stride memory access pattern for the transformations.

We have further compared GEMM performance to Eigen and the Apple Accelerate Framework (AF). Our GEMM implementation achieves an average speedup of 4.85 over Eigen and is 15 percent faster than Apple AF on an iPhone 7 plus, and further outperforms Eigen by 39 percent on a Galaxy S8. The absence of OpenMP on iOS notably diminishes Eigen's performance. The remaining speedups are mainly attributed to our offline packing strategy and tuned cache block parameters for stretched rectangular matrices.

Compute efficiency is measured as the ratio of achieved average performance over nominal peak performance. The efficiency is limited by the stretched rectangular matrix shape. *FeatherCNN* achieves an average efficiency of 68.06 percent whereas Apple AF achieves 60.09 percent. Regarding the Galaxy S8, the nominal peak performance is $2.45\ \text{GHz} \times 4\ (\text{cores}) \times 4\ (\text{SIMD}) \times 2\ (\text{FMA})\ \text{ops} = 78.4$ GFLOPS, which yields an average efficiency of 34.3 and 24.9 percent for *FeatherCNN* and Eigen, respectively. However, a third party benchmark tool, the Geekbench4, reveals a multi-threaded SGEMM performance of 40.5 GFLOPS [18]. From this perspective, *FeatherCNN* and Eigen achieve 66.5 and 52.2 percent of the actual peak performance.

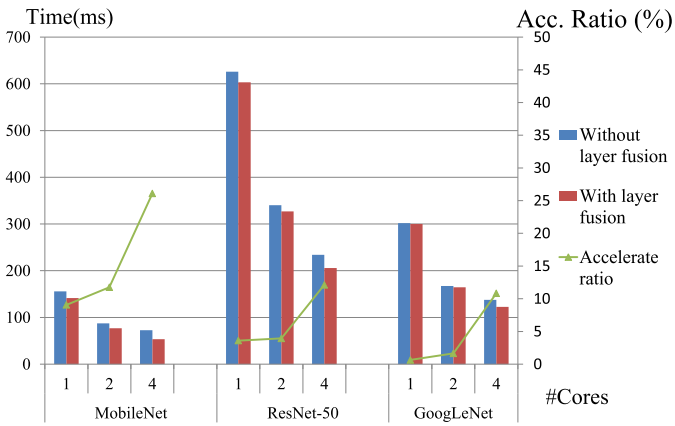


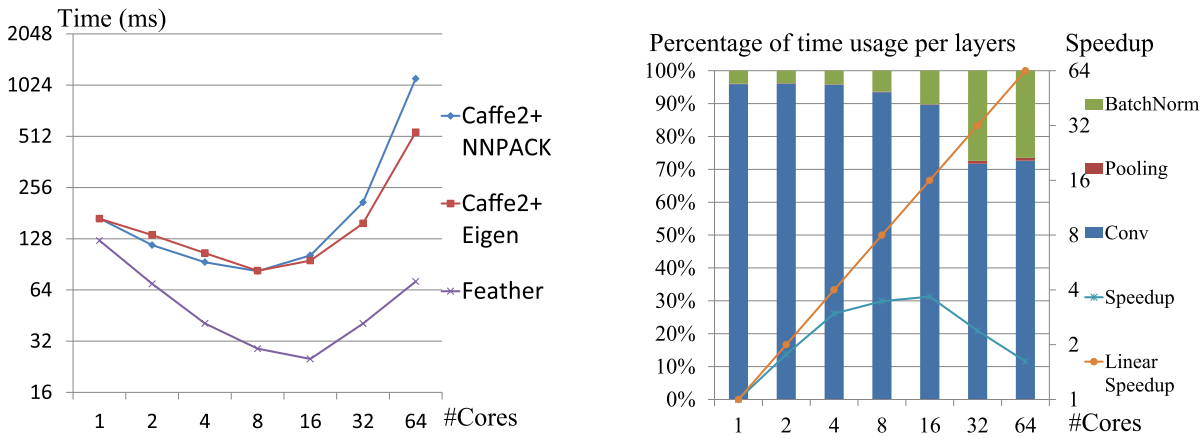
Fig. 8. Runtime comparison and percentage of runtime reduction when using layer fusion for different CNNs. Bars on runtime correspond to the left vertical axis. The lines on acceleration ratio correspond to the right vertical axis. FeatherCNN runtime with (red bar, denoted as β) and without layer fusion (blue bar, denoted as α) are tested on three networks. Acceleration ratio is the percentage of speedup delivered by layer fusion: $\frac{\alpha-\beta}{\alpha} \times 100\%$. The Winograd scheme is $F(2 \times 2, 3 \times 3)$.

Fig. 8 depicts the experimental results for a Galaxy S8 using layer fusion. The amount of layers have been reduced significantly: we fuse 111 layers into 57 layers for MobileNet–V1, 229 layers into 127 layers for ResNet-50, 143 layers into 133 layers for GoogLeNet, and 38 layers into 25 layers for VGG–16. The fused layer types are *ReLU*, *Batch Normalization* and *Scale*. In Fig. 8, the bars represents execution times with respect to amount of threads and neural network configurations. The lines denote the acceleration ratio after enabling layer fusion. Note that multi-threading may benefit more in case of activated layer fusion, due to the reduced memory bandwidth.

4.3 Integral Evaluation

Finally we assess FeatherCNN's performance on aforementioned network topologies and compare it to performance of competing state-of-the-art libraries for neural inference. Experiments are conducted on the Huawei ARM Server (see Table 4) as the major test platform. The ARM-based servers integrate many cores on a single chip to remedy peak performance. With this regard, we emphasis on scalability during evaluation.

We perform a strong scaling test using MobileNet–V1 and VGG–16, respectively. The former one has small model size by virtue of depth-wise convolutional layers while the latter one exhibits significantly higher computational complexity. All convolutional layers in VGG–16 can benefit from Winograd acceleration, while none of the layers in MobileNet–V1 employs 3×3 convolutions, and therefore relies on general convolution and depth-wise convolutional



(a) Scalability evaluation.

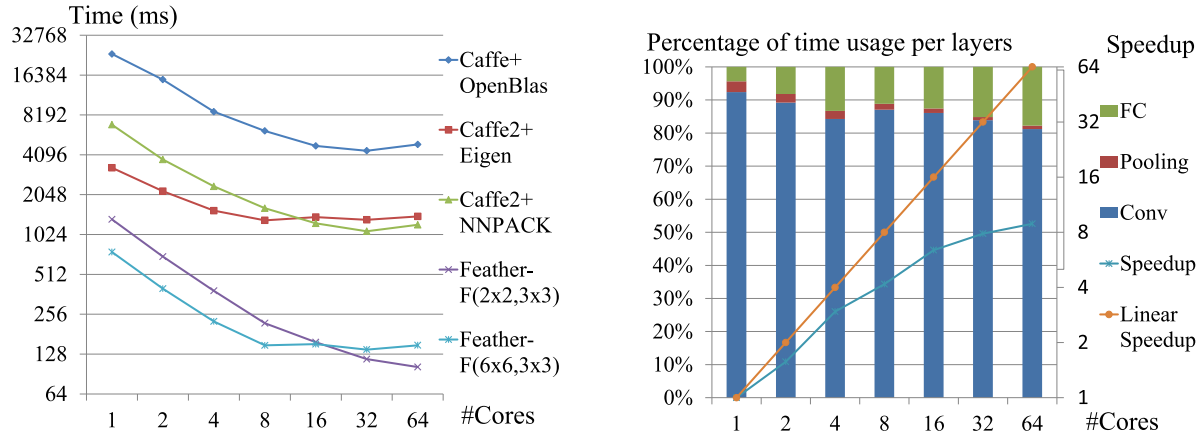
(b) Run time decomposition and speedup over single thread.

Fig. 9. Scalability evaluation on MobileNet against state of art libraries.

layer types. Figs. 9 and 10 show the results. In Fig. 9, FeatherCNN scales up to 16 threads for MobileNet–V1 before being memory bandwidth bound. The speedup over Caffe2 is 3.3 in the case of 16 threads. The minimum inference time is 25 ms. The performance improvements are mainly due to layer fusion and optimization of depth-wise convolutions. Fig. 9b reveals that the many-thread case incurs intensive memory contention in *Batch Normalization* layers causing a significant performance degradation. Performance can be further improved by assigning only a limited number of threads in case of memory intensive layers.

Regarding the VGG–16 neural network, Fig. 10 reveals that the $F(2 \times 2, 3 \times 3)$ scheme of FeatherCNN scales up to 64 cores, while the $F(6 \times 6, 3 \times 3)$ scheme saturates beyond 8 threads. This is induced by two matters. One is that $F(6 \times 6, 3 \times 3)$ has a memory expansion factor of 7.1 (1.8 for $F(2 \times 2, 3 \times 3)$) thus demanding higher memory bandwidth. The other is that $F(6 \times 6, 3 \times 3)$ generates much narrower matrix shapes on last three layers in VGG16 and therefore is incapable to scale over many cores. This result shows that scaling is dependent on different Winograd schemes. Fig. 10b also indicates that memory bounded fully-connected layers are in-negligible limiting factor in terms of scalablity.

Inference time for Inception–V3, VGG–16 and MobileNet–V1 on four Android mobile phones are listed in Table 5. For the evaluation on ARM CPU, Qualcomm SNPE, Tensorflow Lite, and FeatherCNN are tested with all high-performance cores. GPUs and DSP are tested with SNPE only. FeatherCNN is configured to enable $F(6 \times 6, 3 \times 3)$ schemes. It outperforms TensorFlow Lite by a factor of 2 to 4 on Inception–V3 and MobileNet–V1, and by a factor of 4 to 6 on VGG16 due to our Reformulated Winograd acceleration. SNPE can only run on Snapdragon series SOC chips, thus no data is collected for kirin 970. On ARM CPUs, SNPE performances better on models with winograd operations, but worse on models without Winograd operations, compared with TensorFlow Lite. SNPE has been integrated with approximately two times faster GPU performance, and over 5 times faster DSP performance (using "INT8" quantized model) on Inception–V3 compared with FeatherCNN. However SNPE performances worse on MobileNet–V1 with DSP, and incompatible with many models and devices including some Snapdragon SOC chips. To make a comparison with TVM [19], We cite TVM's latest performance results on Kirin 970 from [20] for comparison. It takes TVM 444ms, 41ms and 486ms to finish the inference computation on Inception–V3, MobileNet–V1 and VGG16. According to the figures in Table 5, FeatherCNN with



(a) Scalability evaluation.

(b) Run time decomposition and speedup over single thread with $F(2 \times 2, 3 \times 3)$ scheme.

Fig. 10. Scalability evaluation on VGG-16 against state-of-art libraries.

TABLE 5
Collected Experimental Data Comparing ARM CPUs, GPUs, and Co-Processors on Vivo IROO (sdm855), Vivo X20 (sdm660), Xiaomi 8SE (sdm710) and Huawei Mate 10 (Kirin970)

model	soc chip	CPU			GPU	DSP
		SNPE	TFlite	Feather	SNPE	SNPE
Inception–V3	sdm855	620.26	686.55	134.65	73.85	17.67
	sdm835	1023.92	681.08	290.16	115.84	62.03
	sdm710	764.67	1000.35	467.85	230.55	
	kirin970		683.634	312.26		
VGG16	sdm855	864.29	1088.29	261.74		36.68
	sdm835	1998.67	1815.69	454.82		145.85
	sdm710	1901.45	2525.11	634.47		
	kirin970		3034.60	512.65		
MobileNet–V1	sdm855	493.59	112.63	27.33		27.61
	sdm835	819.95	123.89	50.28		39.23
	sdm710	686.97	104.26	55.90		88.29
	kirin970		156.98	57.62		

The inference engines selected for comparison include Qualcomm SNPE, Google TFLite, and Tencent FeatherCNN. The evaluated network models are Inception–V3, MobileNet–V1, and VGG16.

Reformulated Winograd $F(6 \times 6, 3 \times 3)$ algorithm alone takes 312ms, 57ms and 512ms respectively. FeatherCNN and TVM have presented competitive performance on these three models but with totally different strategies. FeatherCNN explores Winograd optimization and its implementation on ARM CPU's memory hierarchy without considering other methods such as IM2COL, and Direct Convolution. TVM is pursing highest performance by searching a best configuration on existing algorithms with a tensor-like representation and ML-based scheduling. The two work have their own selling points and are complementary to each other.

We further want to highlight FeatherCNN's compact memory consumption compared to other frameworks. For the VGG–16 inference test, the weights alone demand over 500MB of memory space. FeatherCNN using GEMM needs 770MB, with Winograd F(2×2, 3×3) it needs 845MB, and using Winograd F(6×6, 3×3) it needs 1023MB. The Winograd scheme consumes more memory due to the larger kernel transformation matrix U^T, which is processed at initialization and then resides in memory throughout computation. Caffe with OpenBLAS backend using GEMM requires 955MB. Caffe2 with Eigen backend using GEMM consumes 756MB, and with NNPACK backend using Winograd-F(6×6, 3×3) occupies an excessive 1652MB. Our advantage can be explained carefully managed allocations and the reuse of intermediate buffers across different layers. The low memory footprint of FeatherCNN thus allows its deployment on many memory-limit embedded devices.

5 Related Work

Previous work on accelerating deep neural networks on mobile devices can be grouped into three classes: model compression, algorithm optimization, and acceleration of inference computation using specialized hardware.

Model compression is the primary way to reduce storage space and computational costs [21], [22], [23], [24], [25]. Deep compression [25] reports impressive compression rates on AlexNet [26] and VGGNet [27] by pruning weights and subsequent retraining without decreasing the overall accuracy. However, pruning parameters does not necessarily reduce the computation time since the majority of removed parameters stem from the fully connected layers where the computational cost is low. As an example, the fully connected layers of VGG–16 occupy 95 percent of the total parameters but only contribute less than 1 percent of the overall floating point operations (FLOP). They demonstrate that the convolutional layers can also be compressed and accelerated with sparse BLAS libraries or even specialized hardware. But 10 percent of sparsity can result in limited or no speedup with modern sparse libraries for CNN computation [28]. Instead of compressing existing models, researchers started to design new lightweight models such as SqueezeNet [29], Darknet [30] and MobileNets [13]. Those topologies inherently have less parameters at the cost of higher network complexity and minor accuracy loss.

Optimization of algorithms is another important strategy to accelerate CNNs. Convolutions can be transformed into matrix multiplication using GEMM [17], pointwise multiplication after applying FFT [11], and Winograd schemes [12]. However, GEMM introduces a larger memory footprint, FFT is beneficial for kernels larger than 7×7, and Winograd is fast but limited to 3×3 kernels. Efficient algorithms are currently implemented and optimized only for GPUs on servers with high memory bandwidth. Approximate algorithms without retraining include SVD [31], bit-compression or fixed-point implementation [32]. Network pruning [33], [34] can be used to pursue higher computational speed but may induce prediction loss.

Since only a limited number of target algorithms is available for inference of CNNs, hardware acceleration is used to pursue higher compute performance and better energy efficiency. These algorithms can be executed on multicore CPUs using SIMD [35], on GPUs [17], [36], on FPGAs [37], and on ASICs [1], [38]. Mobile devices, however, are rarely equipped with these specialized hardware devices. Those who feature dedicated hardware, however, tend to be significantly more expensive. Currently, mainstream libraries start to support ARM-CPUs. NNPACK [8] has integrated GEMM, FFT, and Winograd algorithms, they also implement these algorithms with NEON instructions but in a straightforward manner based on the naïve evaluation of the mathematical equations in [12]. Thus, its performance on Caffe2 is worse than Eigen [7] in some cases. Other libraries such as OpenBLAS [5], Eigen [7], or Atlas [39] can be used to support GEMM algorithms. However, there is still space for performance improvements on ARM-based multi-core and many-core architectures.

There are recent advances in batched GEMM calculations and tensor contractions [40], [41], [42], [43], [44], [45]. Batched GEMM mainly focus on the case of multiplying hundreds of small matrices of shapes in the range from 2×2 to 32×32 [40], [41]. Previous work aims to fully exploit vectorization and avoid launch overhead of multiple small GEMMs, batching larger matrix straightforwardly however will result in lower performance [43]. Regarding the optimization methods, they range from interleaved memory layout on Intel CPUs [40] to autotuning over

existing GEMM kernels on GPUs [42]. Interleaved memory layouts are effective for small matrices, but will degrade performance on large matrices when overloading the cache. Tensor contraction is a memory bounded kernel [44], [45]. Embedding it after (and before) the Winograd's input (and output) transforms will introduces extra overhead and drag down the computational intensity of the whole phase. Thus, an integrated optimization over all four stages of the Winograd algorithm – such as the one proposed in this paper – is crucial to boost its performance on target architectures.

6 Conclusion

The ever increasing prevalence of Deep Learning-driven applications create an enormous demand for efficient and energy-aware inference solutions on ubiquitous mobile devices. Common approaches for the efficient evaluation of convolutional layers such as Winograd convolution and depth-wise convolution have been ported to a wide variety of platforms including current ARM CPUs. Straightforward reference implementations such as NNPACK [8], however, do not fully exploit the compute capabilities of employed hardware.

In this paper, we present a fast and lightweight inference library called *FeatherCNN* specifically taylored to the characteristics of current ARM architectures. We propose a high performance linear algebra primitive – namely TensorGEMM – in order to significantly speed-up the processing of Winograd-style convolutional layers. The proposed optimizations include both algorithmic improvements and bare-metal tuning to allow for efficient SIMD processing and reduction of memory movements. Sophisticated memory blocking and packing approaches are applied to further improve performance. In addition, memory-intensive layers are specifically optimized by means of layer fusion and vectorization in order to improve thread scalability.

Driven by the rapid development of ARM-based architectures including many-core servers and high performance mobile processors, FeatherCNN may be even deployed in case of heavyweight full-precision network topologies, in the foreseeable future. Both the algorithmic optimization and tuning of vectorization is independent of the given instruction set, and therefore could be further extended to other architectures such as x86_64 CPUs and CUDA-enabled GPUs.

Acknowledgments

We would like to thank Prof. Tong Zhang from Tencent AI Lab for his support and suggestion on using TensorGEMM. Parts of the calculations were performed on next generation pilot supercomputer in National Supercomputer Center in Tianjin. This work was supported in part by the National Science Foundation of China under Grant No. 61702494 and U1813203, National High Technology Research and Development Program of China under grant No. 2015AA020109 and 2016YFB0201305, the Shenzhen Fundamental Research Fund under grant No. JCYJ20160331190123578 and GGFW2017073114031767, and Shenzhen Discipline Construction Project for Urban Computing and Data Intelligence, Youth Innovation Promotion Association, CAS to Yanjie Wei.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, and A. Borchers, et al., "In-datacenter performance analysis of a tensor processing unit," *2017 ACM/IEEE 44th Annu. Int. Symp. Comput. Architecture*, pp. 1–12, 2017.

[2] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, "A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *Proc. Eur. Solid State Circuits Conf.*, 2014, pp. 199–202.

[3] J. Dongarra, H. Simon, M. Meuer, H. Meuer, and E. Strohmaier, "Top500," 2018. [Online]. Available: https://www.top500.org/lists/2018/06/

[4] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.

[5] X. Zhang, "OpenBLAS library," 2013. [Online]. Available: https://github.com/xianyi/OpenBLAS

[6] Facebook, "Caffe2," 2017. [Online]. Available: https://caffe2.ai/

[7] G. Guennebaud, B. Jacob, et al., "Eigen v3," 2010. [Online]. Available: http://eigen.tuxfamily.org

[8] M. Dukhan, "The NNPACK library," 2015. [Online]. Available: https://github.com/Maratyszcza/NNPACK

[9] Google, "Tensorflow lite," 2017. [Online]. Available: https://www.tensorflow.org/mobile/tflite

[10] Apple, "Implementation of the basic linear algebra subprograms (blas)," 2017 [Online]. Available: https://developer.apple.com/documentation/accelerate/blas

[11] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," in *Proc. Int. Conf. Learn. Representations*, 2015.

[12] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4013–4021.

[13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[14] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 1251–1258.

[15] C. Cecka, 2017. [Online]. Available: https://devblogs.nvidia.com/cublas-strided-batched-matrix-multiply/

[16] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, 2008, Art. no. 12.

[17] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," arXiv:1410.0759, 2014.

[18] Geekbench4 SGEMM Benchmark for Samsung Galaxy S8, 2018. [Online]. Available: http://browser.geekbench.com/v4/cpu/7658873, Retrieve: Mar. 28, 2018.

[19] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimization stack for deep learning," *13th USENIX Symp. Operating Syst. Des. Implementation*, pp. 578–594, 2018.

[20] L. Zheng, "TVM's benchmark results for several popular image classification models," 2019. [Online]. Available: https://github.com/dmlc/tvm/wiki/Benchmark

[21] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Proc. Advances Neural Inf. Process. Syst.*, 1990, pp. 598–605.

[22] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Proc. Advances Neural Inf. Process. Syst.*, 1993, pp. 164–171.

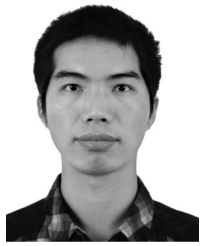
[23] S. Srinivas and R. V. Babu, "Data-free parameter pruning for deep neural networks," in *Proc. Brit. Mach. Vis. Conf.*, X. Xie, M. W. Jones, and G. K. L. Tam, Eds. BMVA Press, Sep. 2015, pp. 31.1–31.12. [Online]. Available: https://dx.doi.org/10.5244/C.29.31, doi: 10.5244/C.29.31.

[24] Z. Mariet and S. Sra, "Diversity networks: neural network compression using determinantal point processes," in *Proc. Int. Conf. Learn. Representations*, 2016.

[25] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proc. Int. Conf. Learn. Representations*, 2016.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Advances Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[27] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Representations*, 2015, pp. 730–734.
[28] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.
[29] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.
[30] J. Redmon, "Darknet: Open source neural networks in c," 2013–2016. [Online]. Available: http://pjreddie.com/darknet/
[31] J. Xue, J. Li, and Y. Gong, "Restructuring of deep neural network acoustic models with singular value decomposition," in *Proc. Interspeech*, 2013, pp. 2365–2369.
[32] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2849–2858.
[33] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.
[34] J. Jin, A. Dundar, and E. Culurciello, "Flattened convolutional neural networks for feedforward acceleration," *arXiv preprint arXiv:1412.5474*, 2014.
[35] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proc. Deep Learn. Unsupervised Feature Learn. NIPS Workshop*, 2011, vol. 1, Art. no. 4.
[36] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots HPC systems," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1337–1345.
[37] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, 2010.
[38] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops*, 2011, pp. 109–116.
[39] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proc. IEEE/ACM Conf. Supercomputing*, 1998, pp. 38–38.
[40] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, and M. Zounon, "Optimized batched linear algebra for modern architectures," in *Proc. Eur. Conf. Parallel Process.*, 2017, pp. 511–522.
[41] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra, "High-performance matrix-matrix multiplications of very small matrices," in *Proc. Eur. Conf. Parallel Process.*, 2016, pp. 659–671.
[42] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched GEMM for GPUs," in *Proc. Int. Conf. High Perform. Comput.*, 2016, pp. 21–38.
[43] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Batched matrix computations on hardware accelerators based on GPUs," *Int. J. High Perform. Comput. Appl.*, vol. 29, no. 2, pp. 193–208, 2015.
[44] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, et al."High-performance tensor contractions for gpus," *Procedia Comput. Sci.*, vol. 80, pp. 108–118, 2016.
[45] P. Springer and P. Bientinesi, "Design of a high-performance GEMM-like tensor–tensor multiplication," *ACM Trans. Math. Softw.*, vol. 44, no. 3, 2018, Art. no. 28.

Haidong Lan received the BS and PhD degrees in computer science from the Shandong University, Jinan, China, in 2013 and 2018 respectively. He is currently an engineer at Tencent AI Platform Department. His research interests include high performance computing, especially in the areas of bioinformatics and deep learning.



Jintao Meng received the BS and MS degrees in computer science from the Central China Normal University, Wuhan, in 2005 and 2008 respectively, and the PhD degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2016. From 2008 to 2016, he was an engineer with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. Since 2017, he has been a senior engineer with Tencent. He is the author of SWAP-Assembler, and published 20 articles, and 18 inventions. His research interests include high performance computing, bioinformatics, and graph computing.

Christian Hundt received the diploma degree in theoretical physics for the analysis of quantization maps on curved manifolds and the PhD degree in computer science for the efficient subsequence alignment of time series on CUDA-enabled accelerators from the University of Mainz, Germany, in 2010 and 2015. In his current position, as a postdoctoral researcher at the Parallel and Distributed Architectures group, he investigates the design and parallelization of algorithms in the field of bioinformatics.

Bertil Schmidt (M'04-SM'07) is a tenured full professor and a chair for Parallel and Distributed Architectures at the University of Mainz, Germany. Prior to that, he was a faculty member at Nanyang Technological University (Singapore) and at the University of New South Wales (UNSW). His research group has designed a variety of algorithms and tools for Bioinformatics (mainly focusing on the analysis of large-scale sequence and short read datasets) and Data Mining. For his research work, he has received a CUDA Research Center award, a CUDA Academic Partnership award, a CUDA Professor Partnership Award, and the Best Paper Award at IEEE ASAP 2009 and IEEE ASAP 2015. He is a senior member of the IEEE.

Minwen Deng received the BS and MS degrees in computer science from the Sun Yat-Sen university and Institute of Computing Technology, Chinese Academy of Sciences, in 2006 and 2009 respectively. From 2009 to 2010, He has worked as an engineer in Alibaba cloud. Since 2010, He is a senior engineer in Tencent leading the work on AI infrastructure construction.

Xiaoning Wang received the BS and MS degrees in computer science from the Shandong University, Jinan, China, in 2015 and 2018 respectively. She now is joined in Tencent AI Platform Department as an Engineer. Her research interests include high performance computing and deep learning.

Weiguo Liu received the bachelor's and master's degrees from the Xi'an JiaoTong University, China, in 1998 and 2002, and the PhD degree from the Nanyang Technological University (NTU), Singapore, in 2006. He is currently a professor and the director of the High-performance Computing & Big Data Processing Lab at Shandong University and published more than 50 articles. He was nominated as "Taishan Scholar" in Shandong province, and received numerous awards (e.g. ACM Gordon Bell Prize award in SC 2017, Fraunhofer IGD Best Paper award, and CCF HPC China Best Paper award). He is also the committee of the high performance computing of China Computer Federation. His research interests include high-performance computing, bioinformatics, and data mining. His research group has designed tools and algorithms for applications in data processing and computational science using parallel computing technologies such as CUDA-enabled GPUs, CPU/GPU/Xeon Phi clusters, and supercomputers.

Yu Qiao received the bachelor's eng. in automation and master eng. degrees in control theory and engineering from Chongqing University, in 2000 and 2003, respectively, and the PhD degree from the University of Electro-Communications, Tokyo, Japan. He is the director of the Institute of Advanced Computing and Digital Engineering in Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. After that, He had been a project assistant professor in the Graduate School of Information Science and Technology, The University of Tokyo, Japan. From 2010, Prof. Qiao is a professor with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, where I co-direct the Multimedia Research Center. He was a scholar supported by the "Hundred Talents Program" of the Chinese Academy of Sciences. He received the "Lu Jiaxi Young Researcher Award" from the Chinese Academy of Sciences in 2012. His research interests include Computer Vision, Deep Learning, Pattern Recognition, Speech Processing, Robotics etc. Recently, He focuses on develop novel deep learning techniques for action recognition, scene understanding, facial analysis and recognition, object detection with applications in surveillance, robotics and human-machine interface. His ultimate goal is to enable machine to perceive and understand complex visual scenes like human.

Sheng-Zhong Feng received the bachelor's degree from the University of Science and Technology of China, in 1991, and the PhD degree from the Beijing Institute of Technology, in 1997. He is the director of National Supercomputing Center in Shenzhen, and special director assistant of Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. Before joining SIAT in 2009, he has worked as an associate professor with the Institute of Computing Technology in Chinese Academy of Sciences and as visiting professor with the University of Toronto. His research interest includes high performance computing, cloud computing and bioinformatics, etc. He is a member of the general expert group on the National Key Research and Development Program of HPC, committee of the high performance computing of China Computer Federation. He, as the technology leader, is responsible for the establishment of the National Supercomputing Center in Shenzhen. He also participated in the development of Dawning series supercomputer and as the principle investigator of many national level projects, such as 863 program, NSFC projects, knowledge innovation project of the Chinese Academy of Sciences, etc. He has received many awards, such as Hundred-Talent Program from Chinese Academy of Sciences, Outstanding Science and Technology Progress Award from Chinese Academy of Sciences, second prize of the National Science and Technology Progress Award and so on. He has published more than 60 research papers which are indexed by SCI/EI, and applied more than 20 patents in resent 5 years.