

# FeatherCNN: Fast Inference Computation with TensorGEMM on ARM Architectures

Haidong Lan, Jintao Meng, Christian Hundt, Bertil Schmidt Minwen Deng, Weiguo Liu, Yanjie Wei and Shengzhong Feng

## ABSTRACT

In comparison to time-consuming training of convolutional neural networks (CNNs), inference is a relatively lightweight operation making it amenable to execution on mobile devices. Nevertheless, compute and energy 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 tries to improve the efficiency of inference computation on ARM-based multi-core and many-core architectures using a highly efficient TensorGEMM (generalized matrix multiplication) routine is applied to accelerate Winograd convolution on ARM CPUs. Experimental results reveal that, forward computation for VGG-16 on a server with 64 ARM A72 cores, FeatherCNN can scale up to 64 cores with an parallel efficiency of 27%, and achieve 48x, 14x and 12x speedup over Caffe+OpenBlas, Caffe2+Eigen and Caffe2+NNPACK, respectively. The source code of FeatherCNN library is publicly available at <https://github.com/tencent/feathercnn>.

## 1 Convolutional Neural Networks and Winograd Algorithm

We first introduce the symbol definitions of Convolutional Neural Networks(CNN) and refresh the basics for the Winograd algorithm, followed by three methods on optimizing fast inference computation on ARM architecture. Finally, FeatherCNN is developed with these technologies and experimental result confirms its superiorities to current state-of-the-art libraries.

A convolutional network layer receives a batch of  $N$  image data with  $C$  channels of size  $H \times W$ , a bank of  $K$  kernels with  $C$  channels of size  $R \times S$ , and outputs  $K$  channels of size  $E \times F$ . The computation of CNN is defined as:

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

Where  $0 \leq i < N$ ,  $0 \leq k < K$ ,  $0 \leq x < H$ ,  $0 \leq y < W$ .

Winograd algorithm [1] is proposed by Andrew Lavin in 2016. There are several 2D algorithms, without loss of generality, we specifically address the optimizing problem for  $F(2 \times 2, 3 \times 3)$ . The general 2D formula of  $F(2 \times 2, 3 \times 3)$  is as follows:

$$S = A^T [GgG^T] \odot [B^T dB] = A^T [U \odot V] A \quad (1)$$

Where  $B, G, A$  are matrixes with fixed values,  $g$  is kernel, and  $d$  is the  $4 \times 4$  blocks extracted from input images. Figure 1 (a) illustrates the basic workflow of convolutional Winograd algorithm.

The algorithm first slide over the image to compute the input transform, and then perform element-wise multiplication with the

transformed kernel. The results goes through the output transform to produce final  $2 \times 2$  output. The  $H \times W$  input image requires  $(H-2)/2 \times (W-2)/2$  transforms. Denote  $d$  as the index of each  $4 \times 4$  block, where  $0 \leq d < \left(\frac{H}{2} - 1\right) \times \left(\frac{W}{2} - 1\right)$ . Then formula 1 can be rewritten as

$$S_{k,d} = \sum_{c=0}^{C-1} A^T [U_{k,c} \odot V_{c,d}] A = A^T \left[ \sum_{c=0}^{C-1} U_{k,c} \odot V_{c,d} \right] A$$

Let  $M_{k,d} = \sum_{c=0}^{C-1} U_{k,c} \odot V_{c,d}$ , and  $U_{k,c}, V_{c,d}$  are  $4 \times 4$  matrixes.

Then  $M_{k,d}^i = \sum_{c=0}^{C-1} U_{k,c}^i \odot V_{c,d}^i$ ,  $0 \leq i < 16$

$$\text{or } M_{k,d}^i = U_k^i \times V_d^i \quad (2)$$

That means,  $M_{k,d}$  can be computed with 16 matrix multiplications.

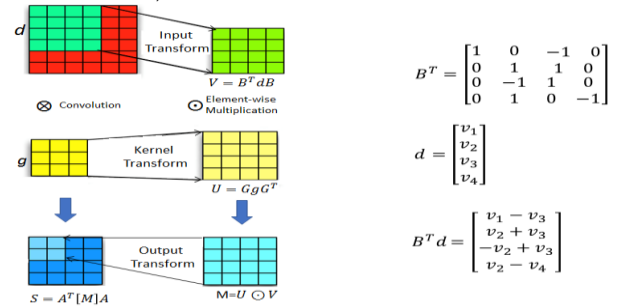


Figure 1. (a) describes the equality of  $F(2 \times 2, 3 \times 3)$  and convolutional neural networks on a  $4 \times 4$  block extracted from the input image. (b) is an example shows the input transform vectorization.

## 2 Optimization methods on ARM architecture

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:** 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 transforming to  $\theta$  matrices. This approach reduces the data movement stride by a factor of 16 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.

### 2.1 Embedding Winograd Convolutions with 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  $\mathbf{A}$  and  $\mathbf{B}$  with compatible shape can be interpreted as their pointwise Hadamard product  $\mathbf{A} \odot \mathbf{B}$  and subsequent sum-reduction over a set of shared rank identifiers.

TensorGEMM shall be defined as inner product of two rank three tensors  $\mathbf{A}_{k,c,i}$  and  $\mathbf{B}_{c,d,i}$ . Point-wise multiplication of entries can be accomplished after (virtually) broadcasting the tensors to their corresponding rank four embeddings  $\mathbf{A}'_{k,c,d,i} = \mathbf{A}_{k,c,i}$  for all  $d$  and  $\mathbf{B}'_{k,c,d,i} = \mathbf{B}_{c,d,i}$  for all  $k$  in order to guarantee compatible shapes. As shown before, TensorGEMM can rewrite formula 2 as a cascade of rank two inner products (batched ordinary matrix multiplications):

$$\begin{aligned} C_{k,d,i} &= \sum_c (\mathbf{A}' \odot \mathbf{B}')_{k,c,d,i} = \sum_c \mathbf{A}'_{k,c,d,i} \mathbf{B}'_{k,c,d,i} = \sum_c \mathbf{A}'_{k,c,i} \mathbf{B}'_{c,d,i} \\ &= \sum_c (\mathbf{A}_{k,c} \mathbf{B}_{c,d})_i \end{aligned} \quad (3)$$

The winograd index is identified as  $0 \leq i < 16$  in case of  $F(2 \times 2, 3 \times 3)$  with  $L$  lanes in vector registers. Hence all additive contributions over  $C$  input color channels can be accumulated in an inner loop over  $c$  in a vector register:  $C_{k,d,i*} = C_{k,d,i*} + A_{k,c,i*}$ , where  $i^* = (0, \dots, L-1)$  is a sequence of lane identifiers. If  $L < 16$  we have to perform multiple no-warm-up passes.

### 2.2 Reduction of Memory Movement

As mentioned before, we perform  $p=4$  no-warm-up passes to the fixed-length subroutine on the 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 Figure 2. We use 4 vectors to store a  $4 \times 4$  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.

In this progress, input transform and TensorGEMM can be conjunctively optimized with memory layout packing strategies to avoid data scattering and minimize interleaved distance in Tensors. These two memory layout packing strategies are denoted as internal packing and 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 causes significantly more cache misses as it writes a row-major 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.

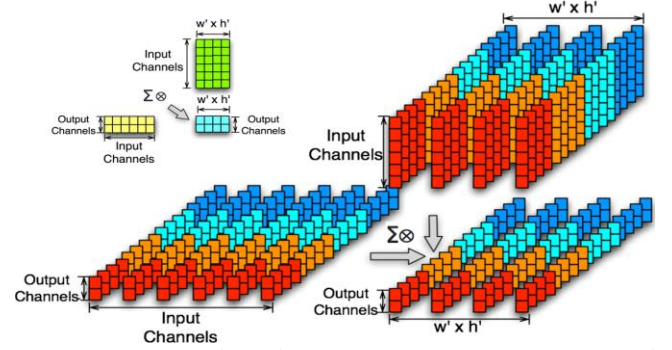


Figure 2. The up-left subfigure is a 2-dimension illustration of formula 4, where each element is a vector of length 16. The bottom-right subfigure in figure 2 is a 3-dimension description of formula 4. Note that  $w'=W/2-1$ ,  $h'=H/2-1$ .

### 2.3. Improvement of TensorGEMM Efficiency

Formula 2 has proved that  $M_{k,d}$  can be computed with 16 matrix multiplications, after reformulated with TensorGEMM, formula 3 confirms its equivalence with several passes of TensorGEMM. In figure 2, the up-left subfigure is a 2D form of formula 4, where each element is a vector of length 16. The bottom-right subfigure in figure 2 is a 3D illustration of formula 4, as ARM architecture supports 128 bit NEON instructions to hold 4 single floats, each  $4 \times 4$  tile is divided into four vectors (painted with different colors). Cubic GEMM is used to implement vector-as-element matrix multiplication, thus  $M_{k,d}$  can be computed with 4 cubic GEMMs. The computing efficiency of cubic GEMM is critical, we have adopted three optimization techniques to further improve its performance:

- 1). Pack the results of input/output transforms to enable sequential access in cubic GEMM, which minimizes cache miss for loading data.
- 2) Implement efficient kernel functions to achieve high compute / memory access ratio, which is 2 for armv8 and 1.67 for armv7.

## 3 Experimental Results

FeatherCNN is a fast inference computation library developed with the above optimization methods. An experiment is conducted to evaluate its performance with state-of-art libraries on VGG-16, the test platform is equipped with dual Huawei 32-core Cortex-A72 processor and 256GB memory. The 64-thread results in Figure 4(a) reveal that FeatherCNN achieves 48x, 14x and 12x speedup over Caffe+OpenBlas, Caffe2+Eigen and Caffe2+NNPACK respectively. Strong scale test evaluates thread scalability as well as efficiency for FeatherCNN on the VGG-16 neural network. Figure 4 (b) illustrated that the percentage of time usage on PoolingLayer and FCLayer after 8 cores increase dramatically because the two layers are pretty memory-intensive, and thus less scalable. To conclude, FeatherCNN cant scale to 64 cores at the efficiency of 27%.

## REFERENCES

- [1] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. CVPR 2016