Parallel convolution algorithm using implicit matrix multiplication on multi-core CPUs

Qinglin Wang, Songzhu Mei✉, Jie Liu, Chunye Gong
Science and Technology on Parallel and Distributed Processing Laboratory
National University of Defense Technology, Changsha 410073, China
School of Computer Science, National University of Defense Technology, Changsha 410073, China
Email: wangqinglin.thu@gmail.com, meisongzhu@yeah.net

***Abstract*—Convolution neural networks (CNNs) have been extensively used in machine learning applications. The most time-consuming part of CNNs are convolution operations. A common approach to implementing convolution operations is to recast them as general matrix multiplication, known as the im2col+GEMM approach. There are two main drawbacks of this approach. One is that large additional memory space is required. The other is the packing on the input elements of convolution operations are not memory-efficient enough. In this paper, we present a new parallel convolution algorithm using implicit matrix multiplication on multi-core CPUs. In comparison with Im2col+GEMM, our new algorithm can reduce the memory footprints and improve the packing efficiency. The experiment results on two ARV8-based multi-core CPUs demonstrate that our new algorithm gives much better performance and scalability than the im2col+GEMM method in most cases.**

I. Introduction

Convolutional Neural Networks (CNNs), a class of Deep Neural Networks (DNNs), have been widely used in various AI applications such as natural language processing, speech recognition and computer vision [1]–[3]. CNNs often contains convolution layers, pooling layers, activation layers, and fully-connected layers. In the training and inference of CNNs, a huge fraction of computation is spent on the convolution layers, so it is important to improve the performance of convolution operations on parallel hardware resources.

There are a number of common approaches used to implement convolution operations. One of the most well-known approaches is to transform a convolution operation into general matrix multiplication (GEMM), called as the im2col+GEMM approach [4]–[6]. In the approach, the input data is firstly lowered into a Toeplitz matrix (i.e. im2col, image-to-column). Then, a fast GEMM routine is carried out on the Toeplitz matrix, by means of available highly optimized Basic Linear Algebra Subprograms (BLAS) libraries targeted at parallel processors. At last, the output result of GEMM is reshaped into the output of the convolution operation (i.e. col2im, column-to-image). Until now, this approach has been employed in mainstream deep learning frameworks e.g. Mxnet [7], Caffe [5] and Tensorflow [8].

This research work is supported by the National Key Research and Development Program of China (No. 2017YFB0202104), and the National Natural Science Foundation of China under grant nos. 61502507, 61602500, 91530324 and 91430218.

Unfortunately, there are mainly two disadvantages of the im2col+GEMM approach. One disadvantage is the explosion of memory space when constructing the Toeplitz matrix. For example, the kernel size of a convolution operation is $k \times k$, the stride size is 1, and the size of the Toeplitz matrix is k^2 times that of the original input data. The other is that the memory packings on the convolution input are not memory-efficient enough. There are totally two packings on the convolution input elements. The first one is replicating the input data to construct the Toeplitz matrix. The second one is packing the Toeplitz matrix into a temporary array in order to make the consecutive operations act on the consecutive memory in a GEMM routine. Both the packings above move the convolution input elements from off-chip memory to on-chip caches, and the total memory access overhead of packing the convolution input is non-trivial in Im2col+GEMM.

Some efficient algorithms have been proposed to solve the issues above [9]–[12]. Cho and Brand [9] convert a convolution operation into multiple overlapping matrix multiplication operations relying on some BLAS GEMM interface, and the memory consumption is significantly reduced. Vasudevan et al. [10] express a convolution as the sum of a few separate small convolutions. In that way, the data locality are drastically improved, but the output size of the GEMM is increased with a factor of k^2. Anderson et al. [11] continue on the reexpression method, and make the additional memory overhead decrease dramatically. Chetlur et al. [12] create the sub-tiles of the Toeplitz matrix according to the tiling method of the underlying GEMM implementation on GPUs. However, these algorithms are not perfectly suitable for the optimization with BCHW (batch, channel, height, weight) data layout on multi-core CPUs, which is the default setting of Mxnet, Caffe, and PyTorch [13].

In this paper, we propose a new parallel convolution algorithm using implicit matrix multiplication, targeted at the convolution optimization with BCHW data layout on multi-core CPUs. Our new algorithm fuses im2col conversion and packing operations in matrix multiplication algorithms together. In this way, the whole Toeplitz matrix is no longer required, and the temporary array in matrix multiplication is created directly from the convolution input. Therefore, memory footprints are reduced and the memory efficiency of packing on the input data is improved. The new algorithm

978-1-7281-2009-6/$31.00 paper N-20120.pdf

is verified on two ARMV8-based multi-core CPUs (Phytium FT1500A and Marvell ThunderX_CP). The results show our new algorithm works better in both performance and scalability, than the im2col+GEMM approach on a great majority of tested convolution operations. Based on Im2col+GEMM, our new algorithm achieves speedups ranging from a factor of 1.0 to 5.17.

The structure of this paper is as follows. Section II presents the relative background and details about the convolution algorithm based on im2col and matrix multiplication operations. Section III describes our proposed parallel convolution algorithm with implicit matrix multiplication on multi-core CPUs. The performance results are analyzed in Section IV. Finally, Section V concludes this paper.

II. BACKGROUND

A convolution operation takes two source tensors ($Input$ and $Filters$) as input and produces a tensor ($Output$). Only tensors with BCHW data layout are involved in the following algorithms. A matrix multiplication routine operates on two two-dimensional tensors (X and Y) and gets a two-dimensional tensor (Z). In C code, these tensors above can be written as:

$$\begin{aligned} &float \quad Input[B][C_i][H_i][W_i];\\ &float \quad Filters[C_o][C_i][H_f][W_f];\\ &float \quad Output[B][C_o][H_o][W_o];\\ &float \quad X[M][K];\\ &float \quad Y[K][N];\\ &float \quad Z[M][N]; \end{aligned}$$

A convolution operation based on $Input$ and $Filers$ tensors is defined by

$$Conv_{(b,co,ho,wo)}(Input, Filters) = \sum_{ci=0}^{C_i-1}\sum_{hf=0}^{H_f-1}\sum_{wf=0}^{W_f-1} [Input_{(b,ci,ho\times s+hf,wo\times s+wf)} \times Filters_{(co,ci,hf,wf)}], \quad (1)$$

where $0 \le b < B$, $0 \le co < C_o$, $0 \le ho < H_o$, $0 \le wo < W_o$ and s stands the stride size. In fact, there are seven nests loops in the convolution operation, and it's not easy to get good performance on multi-core CPUs.

The convolution implementation through the im2col+GEMM approach is shown in Algorithm 1. The two most important steps of the im2col+GEMM approach are creating X tensor with the im2col function and getting the result of $X \times Y$ with the help of BLAS libraries. In the following subsections, we give details of the im2col function as well as a common parallel matrix multiplication implementation on Multi-core CPUs.

A. Im2col

The im2col function converts a multi-dimensional source tensor ($Input$) of convolution operations to a two-dimensional tensor (X). The first dimension (M) of X is determined by $B \times H_o \times W_o$, and the second one (K) depends on the size

Algorithm 1: Im2col+GEMM Convolution Algorithm

input : $Input$, $Filters$, $Bias$, Stride s
output: $Output$

1 Allocate X, Y and Z with MK, KN and MN elements ($M = BH_oW_o$, $K = C_iH_fW_f$, $N = C_o$)
2 Interpret X, Y and Z as $M \times K$, $K \times N$ and $M \times N$ tensors
3 Reshape $Filters$ to Y
4 Copy $Input$ to contruct X `// im2col`
5 $Z = X \times Y$ `// matrix multiplication`
6 Reshape Z to $Output$ `// col2im`

($C_i \times H_f \times W_f$) of single filter. A transforming examples with the im2col function is shown in Fig. 1. When a filter is applied to some location of $Input$, the corresponding blocks are unfolded into a row of X. When the height and width of the filter are bigger than the stride size (s), the applying locations of the filter may overlap each other. As a result, elements of $Input$ may be copied up to $H_f \times W_f$ times, and X is $H_f \times W_f$ times the size of $Input$.

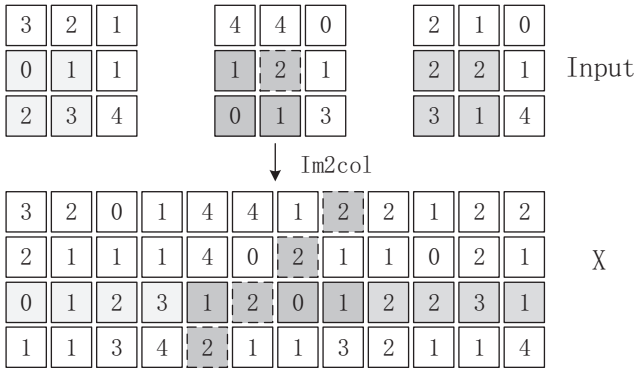

Fig. 1. Transforming examples with the im2col function, where $B = 1$, $C_i = 3$, $H_i = W_i = 3$, $H_f = W_f = 2$, and $H_o = W_o = 2$.

B. Parallel Matrix Multiplication Algorithm on Multi-Core CPUs

Parallel matrix multiplication algorithms on multi-core CPUs have been well studied [14], [15]. Algorithm 2 shows a common parallel matrix multiplication algorithm on multi-core CPUs. The input tensors X and Y are partitioned into subblocks. The block size M_b depends on how many cores can be used to deal with matrix multiplication in parallel. The other block sizes m_r, n_r, K_b and N_b are mainly affected by the number of the available registers, the L1 data cache size, the loading bandwidth from the L2 Cache to the registers, and the L2 Cache size in cores. The detailed discussions on the choice of these block sizes can be found in the references [14], [15].

In order to get the maximum efficiency of loading data to the registers, it's essential to pack Y and X into the special formats in Lines 1 and 4, which consecutive operations can executed on consecutive data in memory. The packing example

of X is shown in Fig. 2. In the packing, the elements of X are carefully arranged according to the order of operations in the algorithm. In other words, the data from $Input$ is moved in the memory once again. Thus, the arrangement on the data of $Input$ is not efficient in the im2col+GEMM approach.

Algorithm 2: A Common Parallel Matrix Multiplication Algorithm on Multi-Core CPUs

input : $X[M][K]$, $Y[K][N]$
output: $Z[M][N]$

```
1  Pack Y into Ỹ in parallel
2  for m = 0: M_b: M do in parallel
3      for k = 0: K_b: K do
4          Pack X_{m,k} [M_b] [K_b] into X̃
5          for n = 0: N_b: N do
6              for cm = 0: m_r: M_b do
7                  for cn = 0: n_r: N_b do
                       // X̃_cm [m_r] [K_b], Ỹ_cn [K_b] [n_r]
8                      Z_aux = X̃_cm × Ỹ_cn
9                      Unpack Z_aux into Z
```

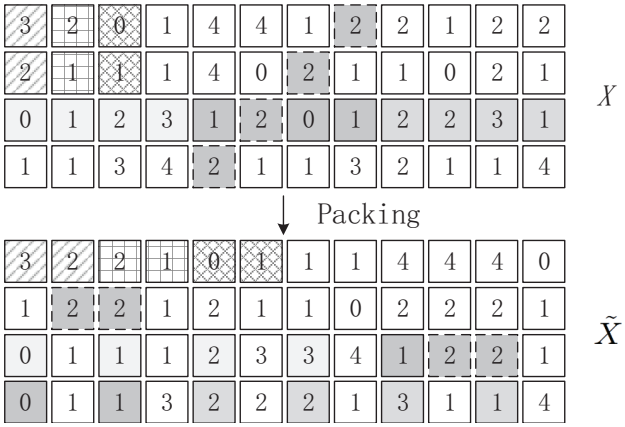

Fig. 2. Packing Examples of $X[M_b][K_b]$ tensor in Matrix Multiplication, where $M_b = 4$, $m_r = 2$ and $K_b = 12$.

III. A NEW APPROACH USING IMPLICIT MATRIX MULTIPLICATION

The im2col+GEMM approach can make use of the available BLAS libraries, but it requires a tensor X with a $H_f \times W_f$ increase in the size of $Input$, and inefficient memory packing on the input data. In this section, a new approach using implicit matrix multiplication is proposed to solve these two problems above.

A. Parallel Convolution Algorithm Based On Implicit Matrix Multiplication

A new parallel convolution algorithm based on implicit matrix multiplication is shown in Algorithm 3. In the algorithm, im2col and matrix multiplication operations are fused together. Instead of the tensor X with $M \times K$ elements, an $M \times K_b$ tensor $\tilde{X}$ in matrix multiplication is required to be allocated. In Lines 3 and 6, $\tilde{Y}$ and $\tilde{X}$ are constructed directly from the original data of $Filters$ and $Input$, respectively. For example, $3 \times 3 \times 3$ $Input$ in Fig. 1 are arranged directly into 4×12 $\tilde{X}$ in Fig. 2. Therefore, our method can address the memory footprint issue and improve the packing efficiency of the input data. In the parallelization, the block size M_b is set to a multiple of W_o for efficient packing of $Input$. Simultaneously, M_b must be a multiple of the number of available data in one cacheline so that the overhead of the cache coherence maintenance among all cores can be minimized. We also require K_b to be a multiple of W_f for getting the better locality in the packing.

Algorithm 3: Parallel Implicit Matrix Multiplication Convolution Algorithm

input : $Input$, $Filters$, $Bias$, Stride s
output: $Output$

```
1  Allocate X̃, Ỹ with MK_b, KNelements
     (M = BH_oW_o, K = C_iH_fW_f, N = C_o)
2  Interpret X̃ and Ỹ as M × K_b and K × N tensors
3  Pack Filters into Ỹ in parallel
4  for m = 0: M_b: M do in parallel
5      for k = 0: K_b: K do
6          Pack Input into X̃_m // im2bcol
7          for n = 0: N_b: N do
8              for cm = 0: m_r: M_b do
9                  for cn = 0: n_r: N_b do
                       // X̃_cm [m_r] [K_b], Ỹ_cn [K_b] [n_r]
10                     Z_aux = X̃_cm × Ỹ_cn
11                     Unpack Z_aux into Z
12 Reshape Z to Output // col2im
```

B. Im2bcol

The direct packing algorithm from $Input$ to $\tilde{X}$ is called as image-to-blocked-column (im2bcol), shown in Algorithm 4. A tensor $Arow$ with W_i elements is required to cache a row data of $Input$, and the source of packing in Lines 8-10 is $Arow$ rather than $Input$. In general, input width W_i of convolution layers in CNNs is not great. The use of the cache tensor $Arow$ can greatly improve the data locality and increase the hit ratios of caches.

The computing of the indexs ci and hf requires integer division and modulus operations, which are not low-cost. The kb loop is set to be the outermost loop so that the number of indexing computing is significantly reduced. At the same time, the algorithm proposed in [16] is also applied to transforming integer division and modulus into multiplies and shifts. Thus, the total indexing overhead is further decreased.

Algorithm 4: Im2bcol Algorithm

input : $Input[B][C_i][H_i][W_i]$, $Arow[W_i]$, m, M_b, H_f, W_f, W_o, Stride s,
output: $\tilde{X}[M_b][K_b]$

Calculate B_s, B_e, H_{os}, $H_{oe}$with m and M_b
for $kb = 0 : W_f : K_b$ **do**
  $ci = \left\lfloor \frac{kb}{H_f W_f} \right\rfloor$, $cr = kb\%(H_f W_f)$
  $hf = \left\lfloor \frac{cr}{W_f} \right\rfloor$
  for $b = B_s \rightarrow B_e$ **do**
    for $ho = H_{os} \rightarrow H_{oe}$ **do**
      Load a row ($Input_{b,ci,ho\times s+hf}$) of $Input$ into $Arow$
      for $wo = 0 \rightarrow W_o$ **do**
        for $wf = 0 \rightarrow W_f$ **do**
          Copy $Arow_{wo\times s+wf}$ into $\tilde{X}$

TABLE I
SPECIFICATIONS OF THE EXPERIMENT PLATFORMS

	Phytium FT1500A	Marvell ThunderX_CP
Architecture	ARMv8	ARMv8
Frequency	1.5 GHz	2.0 GHz
Cores	16	48
L1 Data Cache	32 KB/core	32 KB/core
L2 Cache	2 MB/4 cores	16 MB/48 cores

TABLE II
EXTRA MEMORY REQUIRED FOR PACKING INPUT DATA

Method	Extra Memory
Im2col+GEMM	$N \times H_o \times W_o \times (C_i \times H_f \times W_f)$
Im2bcol+IMM	$N \times H_o \times W_o \times min(C_i \times H_f \times W_f, K_b)$

IV. EXPERIMENTAL RESULTS

This section introduces experimental results of our new approach against the im2col+GEMM approach on two ARMV8-based multi-core CPUs.

A. Experimental Setup

Our experiments are carried out on Phytium FT1500A [17], [18] and Marvell ThunderX_CP [19] architectures. The details of these platforms are described in Table I. We compare the performance of our implicit matrix multiplication convolution algorithm, against that of the im2col+GEMM approach provided by ARM Compute Library (ACL) [20] on these platforms. The version of ACL is v18.08. The im2col+GEMM approach in ACL has been fully parallelized, including the parallel im2col and GEMM implementations.

We use the convolution layers from three popular CNNs in our tests: Alexnet [21], VGG16 [22], and InceptionV1 [23]. Except for the batch size, the specifications of these convolution layers in our tests are the same as the configuration of ACL benchmarks. The batch size for Alexnet and InceptionV1 is 128 while that for VGG16 is set to 16. The main reason is that the ACL v18.08 is not working correctly for some convolution layers of VGG16 when the batch size is bigger than 16. All algorithms are iterated 10 times and the median runtime is reported. In the following, our new approach is labeled as Im2bcol+IMM.

B. Memory Usage

The extra memory for packing the data of $input$ is listed in Table II. For Im2col+GEMM, only the memory space of the converted input data (X) is calculated, and the temporary storage in a GEMM routine is exclusive. The extra memory of Im2col+GEMM grows with single filter size ($C_i \times H_f \times W_f$). The temporary memory of our new method is chiefly determined by the block size K_b in a matrix multiplication routine, when K_b is less than the single filter size. The block size K_b means how much of the packed tensors $\tilde{X}$ and $\tilde{Y}$ can be loaded into the L1 data cache of one core on multi-cores CPUs. In most cases of our tests, the single filter size is much greater than K_b. This demonstrates our new method is an effective way of addressing the memory footprint issue.

C. Performance

The relative performance of our implicit matrix multiplication convolution implementations based on the im2col+GEMM implementations of ACL on Phytium FT1500A and Marvell ThunderX CP is shown in Fig. 3 and Fig. 4, respectively. In the tests, all cores of multi-core CPUs are utilized. The column bars from left to right shows the results of the convolution layers from top to bottom in each network.

On Phytium FT1500A, our new approach is better than ACL Im2col+GEMM in most convolution layers. For Alexnet, our new method outperforms the im2col+GEMM approach by at least 19% and up to 76%. For the first convolution layer of VGG16, our Im2bCol+IMM is as good as ACL Im2col+GEMM. For the left convolution layers of VGG16, the minimum and maximum speedup of Im2bcol+IMM against Im2col+GEMM is 1.13x and 1.32x, respectively. For InceptionV1, a minimum of 1.10 times performance gain is realized. Among all the performance tests on Phytium FT1500A, our new method achieves the maximum speedup of 3.66 times.

On Marvell ThunderX_CP, our new implementation also shows good performance. Expect for the first convolution layer of VGG16, our new approach surpasses the im2col+GEMM method of ACL. For three CNNs, our implicit matrix multiplication method can get the maximum speedups of 1.95x, 2.57x and 5.17x against the im2col+GEMM method, respectively.

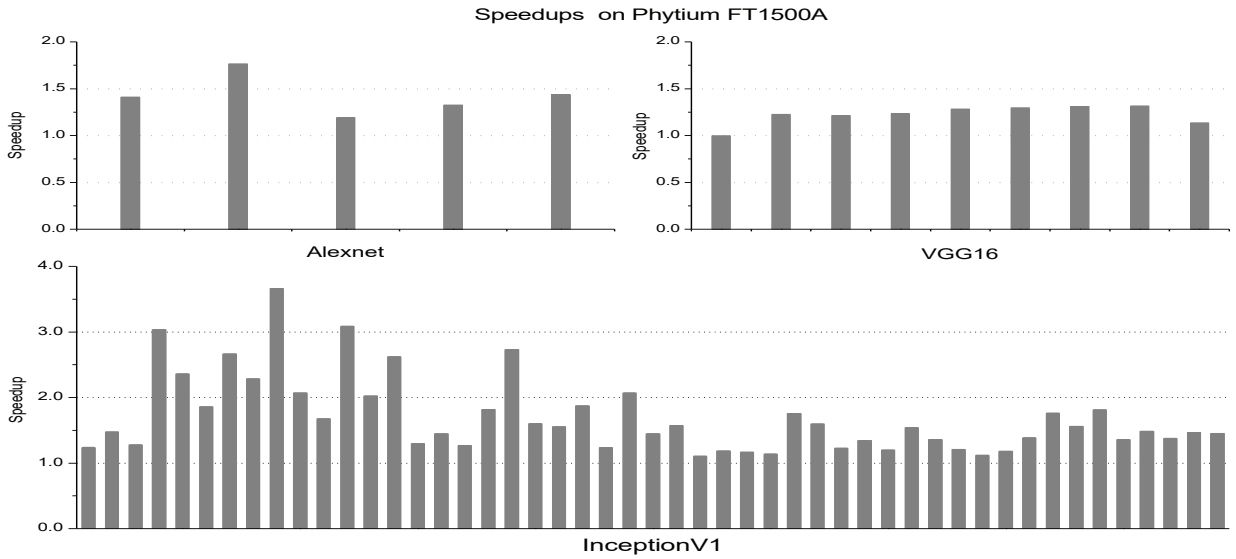

Fig. 3. Speedups of our Im2bcol+IMM algorithm based on ACL Im2col+GEMM algorithm on Phytium FT1500A. All 16 cores of Phytium FT1500A are utilized. The column bars from left to right indicate convolution layers from top to bottom in CNNs.

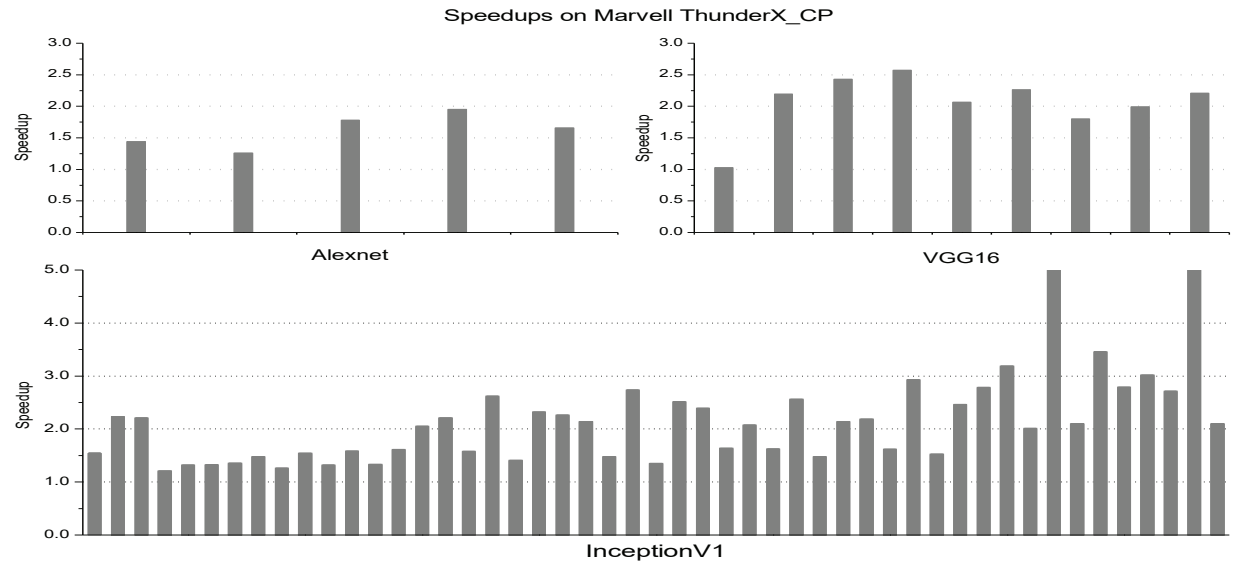

Fig. 4. Speedups of our Im2bcol+IMM algorithm based on ACL Im2col+GEMM algorithm on Marvell ThunderX_CP. All 48 cores of Marvell ThunderX_CP are utilized. The column bars from left to right indicate convolution layers from top to bottom in CNNs.

D. Scalability

The scalability comparation of our Im2bcol+IMM and ACL im2col+GEMM on Phytium FT1500A and Marvell ThunderX CP are shown in Fig. 5 and Fig. 6, respectively. The scalability here refers to the efficiency of strong scaling, which signifies the ratio of the speedups between multi-core and serial implementations to the number of cores used in multi-core implementations. In the figures, only the scalability from 1 to the maximum number of cores in multi-core CPUs has been shown due to the limited space of figures. Among all the scalability tests with the use of all 16 cores on Phytium FT1500A, the minimum scaling efficiency of Im2col+GEMM is only 20% , whereas the efficiency of our new method can reach 40% for the same convolution layer. When all 48 cores are applied on Marvell ThunderX CP, the minimum scaling efficiency of ACL Im2col+GEMM is 15%, and our Im2bcol+GEMM gets the efficiency of up to 55% on the corresponding convolution layer. For all the convolution layers tests, the scalability of our new method and the im2col+GEMM approach decreases with the number of cores. However, on most convolution layers, our new approach gets greater scaling efficiency than Im2col+GEMM with the utilization of the same number of cores, just like the situation shown in Fig. 5 and Fig. 6.

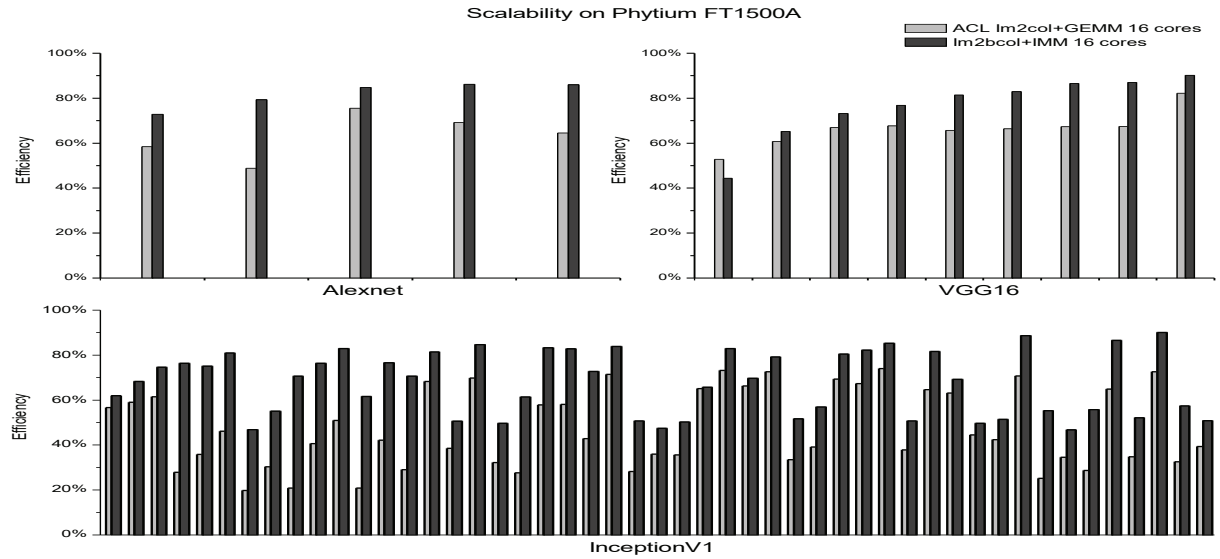

Fig. 5. Scalability comparation between our Im2bcol+IMM and ACL Im2col+GEMM implementations on Phytium FT1500A. The column bars from left to right indicate convolution layers from top to bottom in CNNs.

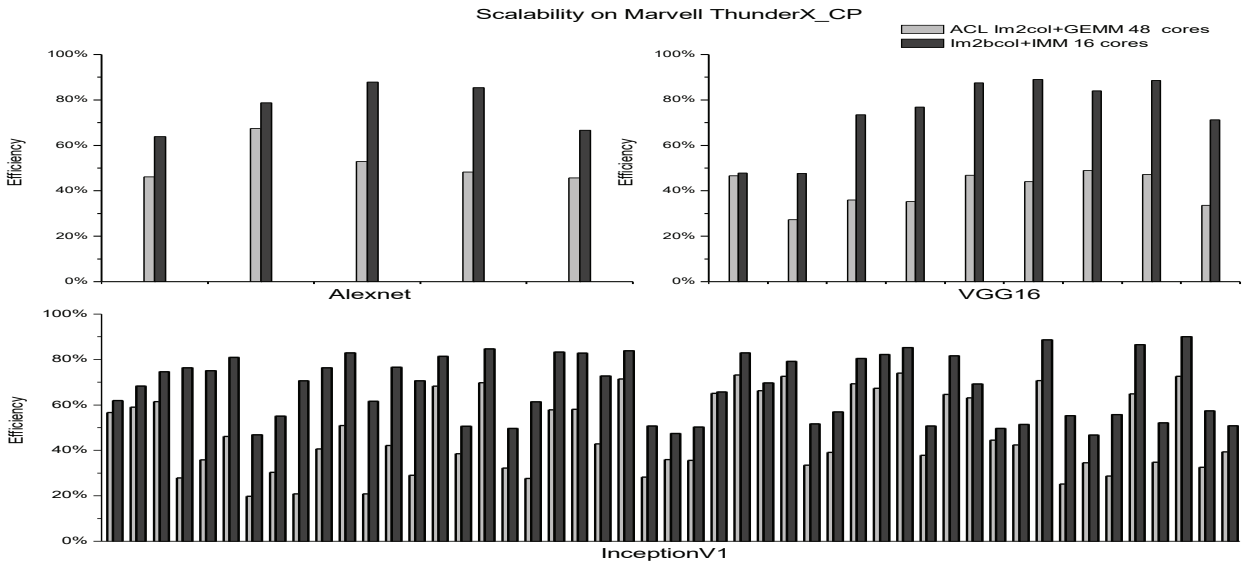

Fig. 6. Scalability comparation between our Im2bcol+IMM and ACL Im2col+GEMM implementations on Marvell ThunderX_CP. The column bars from left to right indicate convolution layers from top to bottom in CNNs.

V. Conclusion

In this paper, we have presented a new parallel convolution algorithm using implicit matrix multiplication, which is fit for the convolution optimization with BCHW data layout on multi-core CPUs. Based on the detail analysis of Im2col+GEMM, our new algorithm integrates im2col and matrix multiplication operations as a whole. The im2bcol algorithm is used to pack the input data of convolution operations directly into a temporary tensor with specific formats, which is required in matrix multiplication. Compared with Im2col+GEMM, our new algorithm can reduce the requirement of temporary memory, and improve the packing efficiency on the input data. The proposed new algorithm is tested on Phytium FT1500A and Marvell ThunderX_CP processors. On Phytium FT1500A, our new algorithm can achieve speedups of a factor of about 1.0 to 3.66 against ACL Im2col+GEMM. On Marvell ThunderX_CP, the maximum speedup can reach up to 5.17 times. In addition, our new algorithm have better scalability than ACL Im2col+GEMM in most cases.

References

[1] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[2] A. L. Jia, S. Shen, D. Li, and S. Chen, "Predicting the implicit and the explicit video popularity in a user generated content site with enhanced social features," *Computer Networks*, vol. 140, pp. 112–125, 2018.

[3] D. Li, Y. Zhang, J. Wang, and K.-L. Tan, "Topox: Topology refactorization for efficient graph partitioning and processing," in *Proceedings of 45th International Conference on Very Large Data Bases*, 2019.

[4] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[5] 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 *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[6] J. Gu, Y. Liu, Y. Gao, and M. Zhu, "Opencl caffe: Accelerating and enabling a cross platform machine learning framework," in *Proceedings of the 4th International Workshop on OpenCL*. ACM, 2016, p. 8.

[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[9] M. Cho and D. Brand, "Mec: memory-efficient convolution for deep neural network," *arXiv preprint arXiv:1706.06873*, 2017.

[10] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International Conference on*. IEEE, 2017, pp. 19–24.

[11] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "Low-memory gemm-based convolution algorithms for deep neural networks," *arXiv preprint arXiv:1709.03395*, 2017.

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

[13] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch," 2017.

[14] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

[15] T. M. Smith, R. Van De Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1049–1059.

[16] H. Warren, "Hacker's delight, 2002."

[17] Phytium, "FT-1500A/16," http://www.phytium.com.cn/Product/detail?language=1&product_id=9, 2019, online, accessed 10-Jan-2019.

[18] X. Chen, P. Xie, L. Chi, J. Liu, and C. Gong, "An efficient simd compression format for sparse matrix-vector multiplication," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, p. e4800, 2018.

[19] Marvell, "Thunderx_CP family," https://www.marvell.com/server-processors/thunderx-arm-processors/thunderx-cp, 2019, online, accessed 10-Jan-2019.

[20] ARM, "Compute Library," https://developer.arm.com/technologies/compute-library, 2019, online, accessed 10-Jan-2019.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[22] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[23] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.