Pattern Recognition Letters 125 (2019) 303–309

Contents lists available at ScienceDirect

Pattern Recognition Letters

journal homepage: www.elsevier.com/locate/patrec

BitStream: An efficient framework for inference of binary neural networks on CPUs ☆

Yanshu Jiang[a,1], Tianli Zhao[a,b,1,*], Xiangyu He[b], Cong Leng[b], Jian Cheng[b]

[a] Harbin University of Science and Technology, Department of Automation, China
[b] National Laboratory of Pattern Recognition, Institute of Automation, Chinese Academy of Science, China

ARTICLE INFO

Article history:
Available online 18 April 2019

Keywords:
Convolutional neural networks
Binary neural networks
Image classification

ABSTRACT

Convolutional Neural Networks (CNN) has been well-studied and widely used in the field of pattern recognition. Many pattern recognition algorithms need features extracted from CNN models to adapt to complex tasks, such as image classification, object detection, natural language processing and so on. However, to deal with more and more complex tasks, modern CNN models are becoming larger and larger, contain large number of parameters and computation, leading to high consumption of memory, computational and power resources during inference. This makes it difficult to run CNN based applications in real time on mobile devices, where memory, computational and power resources are limited. Binarization of neural networks is proposed to reduce memory and computational complexity of CNN. However, traditional implementations of Binary Neural Networks (BNN) follow the conventional im2col-based convolution computation flow, which is widely used in floating-point networks but not friendly enough to cache when it comes to binarized neural networks. In this paper, we propose BitStream, a general architecture for efficient inference of BNN on CPUs. In BitStream, we propose a simple but novel computation flow for BNN. Unlike existing implementations of BNN, in BitStream, all the layers, including convolutional layers, binarization layers and pooling layers are all calculated in binary precision. Comprehensive analyses demonstrate that our proposed computation flow consumes less memory during inference of BNN, and it's friendly to cache because of its continuous memory access.

1. Introduction

Convolutional Neural Networks (CNN) has emerged as one of the most widely-used algorithms in the field of pattern recognition, for its superior performance on many tasks such as image classification [8,13,21], object detection [14,19,20], natural language processing [6,11] and so on, for its superior performance compared to traditional methods [28–31]. However, most modern CNN models contain a large number of parameters and floating-point multiply-accumulate operations. For example, the popular CNN model VGG-16 [21] contains 138.85*M* parameters and needs 30.76*G* floating-point operations during inference. This makes it difficult to run CNN-based applications in real time on low-ended devices where memory, computational and power resources are largely constrained [12,16]. Nowadays, on the one hand, to deal with more and more complex tasks, modern CNN models are becoming larger and larger, on the other hand, the needs to run CNN-based applications on mobile devices are increasing rapidly [5,9,15], this problem is thus becoming more and more critical.

Many efforts have been done to improve the computation efficiency of CNN. An important class of these methods are called Binary Neural Networks (BNN) [3,4,18]. These methods present the parameters and feature maps of CNN as $+1$ or -1. In this way, parameters and feature maps of CNN can be stored bit by bit in memory. The model size can be then compressed as the factor of $32\times$. What's more, in binary neural network, the floating-point convolution can be calculated through efficient bitwise operations, instead of expensive floating-point multiply-accumulate operations. Computation complexity can be then largely reduced. Recent results of BNN [18] have shown that BNN can get nearly state-of-the art performance on the task of image classification.

However, the efficient implementation of BNN on CPUs is still a challenging problem and only few of previous works focus on it. BMXNet [25] is an implementation of BNN on CPUs. It follows the traditional im2col-based convolution computation flow, which is widely used in floating-point convolution. However, when it comes to binary neural networks, this computation flow is

☆ **Conflict of interest**: There are no conflicts of interest.

* Corresponding author.
E-mail address: tianli.zhao@nlpr.ia.ac.cn (T. Zhao).

[1] These authors have contributed equally to this work.

https://doi.org/10.1016/j.patrec.2019.04.016
0167-8655/

inefficient because of its uncontinuous memory access. Moreover, in this computation flow, all the other layers, excluding convolutional layers, are calculated in floating-point precision. This can not only increase the memory consumption, but also reduce computation efficiency during the whole calculation. There are also other works focusing on acceleration of BNN on GPUs [17] and FPGAs [23,27].

In this paper, we propose BitStream, a general architecture for efficient inference of BNN on CPUs. In BitStream, all the kernels and feature maps are stored in channel-first order. We propose a simple but novel computation flow for BNN. Different from existing implementations of BNN, in BitStream, all the layers, including convolutional layers, pooling layers and binarization layers are computed in binary precision. Comprehensive analyses show that the memory consumption of BitStream is largely reduced compared to that of existing methods. The memory access in BitStream is more continuous, which is more friendly to cache, and furthure leading to its high computation efficiency. Extensive experimental results show that BitStream is not only memory but also computation efficient compared to other existing algorithms. For example, on dual core Coretex-A72 CPUs, BitStream is overall $10\times$ and somtimes more than $30\times$ faster than floating-point im2col-based convolution on some popular networks.

2. Preliminary

The algorithm introduced in this paper is closely related to im2col-based convolution algorithm. In this section, we first define the notations used in the whole paper, and then review this algorithm briefly. We will also introduce how calculation can be accelerated when parameters and feature maps are all binarized, that is to say, all the parameters and feature maps are $+1$ or -1. Finally, we will introduce traditional implementations of BNN and their problems.

2.1. Notations

To improve the clarity of description, in this section, we briefly introduce the notations used in this paper. Notations related to a convolutional layer are shown in Table 1. We use capital characters *A, B,…* to denote floating-point matrices or tensors, and we use lowercase unbolded characters *a, b,…* to denote a single number. Lowercase bolded characters **a, b,…** are used to denote a floating-point column vector. We use a sequence of lowercase characters included in a pair of bracket [*i, j*] to denote index. And a capital character followed by an index is used to denote the specific element of the tensor. For example, *A*[*i, j*] is used to denote the element at the $i'th$ row and $j'th$ column of the matrix (or tensor) *A*.

2.2. Im2col-based conovlution

It is well known that convolution can be done through im2col and GEMM, which can be calculated efficiently via highly optimized libraries [7,26], this is so called im2col-based convolution. Im2col-based convolution has been widely used in most modern deep learning frameworks [1,2,10]. Fig. 1 demonstrates the main idea of im2col-based convolution algorithm. The main idea

Table 1
Notations used in this paper.

i_c, i_h, i_w	channels, height and width of inputs
k_h, k_w	height and width of kernels
o_c, o_h, o_w	channels, height and width of results
s_h, s_w	stride of convolution on height/width dimensions
p_h, p_w	padding of convolution on height/width dimensions

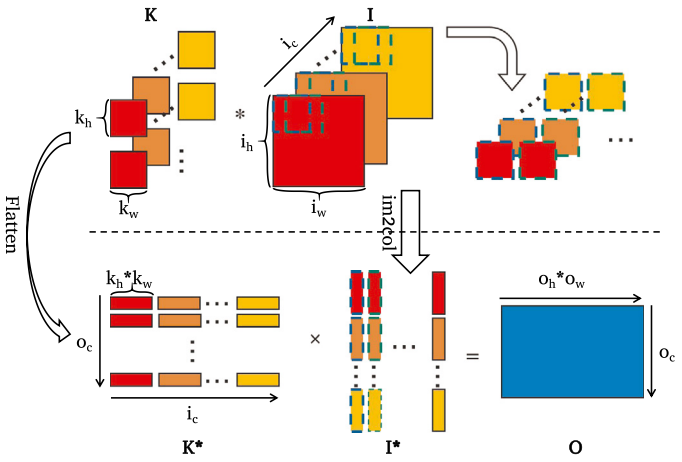

Fig. 1. Computation flow of im2col-based convolution.

of im2col based convolution is to first transform the input tensor *I* into a flat matrix I^*. As the kernel *K* sliding through both dimensions of *I* with strides s_h, s_w, the corresponding patch of *I* is vectorized and duplicated into a column of I^*. For example, the sub-patch with blue dashed line borders is vectorized and duplicated to the first column of I^*, the sub-patch with green dashed line borders is vectorized and duplicated to the second column of I^*, etc. The kernel tersor *K* is directly reinterpreted as a $o_c \times i_c k_h k_w$ matrix K^*. Then the results of convolution can be calculated with GEMM between K^* and I^*: $O = K^* \times I^*$. The output *O* is automatically stored in $o_c \times o_h \times o_w$ order after GEMM, so no reordering is needed.

2.3. Acceleration of binary innerproduct

Innerproduct is the center operation of convolution. When the inputs of innerproduct are all binarized, that is, when all the inputs of inner product are $+1$ or -1, it can be calculated efficiently via bitwise operation. Suppose that we have two $d-dimensional$ vectors **w, x**, each element of which is either $+1$ or -1. The innerproduct between **w** and **x** can be then calculated as follows

$$\boldsymbol{w}^T\boldsymbol{x} = \sum_{i=1}^{d} \boldsymbol{w}[i] * \boldsymbol{x}[i] = \sum_{i=1}^{d} \mathbb{I}(\boldsymbol{w}[i] * \boldsymbol{x}[i] = 1) - \sum_{i=1}^{d} \mathbb{I}(\boldsymbol{w}[i] * \boldsymbol{x}[i] = -1) \tag{1}$$

where $\mathbb{I}(\cdot)$ is a conditional function, defined as $\mathbb{I}(true) = 1$ and $\mathbb{I}(false) = 0$. On the other hand, it is obvious that:

$$\sum_{i=1}^{d} \mathbb{I}(\boldsymbol{w}[i] * \boldsymbol{x}[i] = 1) + \sum_{i=1}^{d} \mathbb{I}(\boldsymbol{w}[i] * \boldsymbol{x}[i] = -1) = d \tag{2}$$

Considering Eqs. (1), (2) can be then rewritten as:

$$\boldsymbol{w}^T\boldsymbol{x} = 2\sum_{i=1}^{d} \mathbb{I}(\boldsymbol{w}[i] * \boldsymbol{x}[i] = 1) - d \tag{3}$$

The first term of the right side of Eq. (3) can be accelerated via efficient bitwise operations, this can be done in two steps. First, each *N* elements of floating-point vectors **w** and **x** are packed into a single $N-bit$ number, where the number $+1$ is replaced by bit 1, and the number -1 is replaced by bit 0, generating two new vectors $\boldsymbol{w}_b$ and $\boldsymbol{x}_b$. The calculation can be then efficiently done as follows:

$$\sum_{i=1}^{d} \mathbb{I}(\boldsymbol{w}[i] * \boldsymbol{x}[i] = 1) = \sum_{i=1}^{\lceil \frac{d}{N} \rceil} popcnt(XNOR(\boldsymbol{w}_b[i], \boldsymbol{x}_b[i])) \tag{4}$$

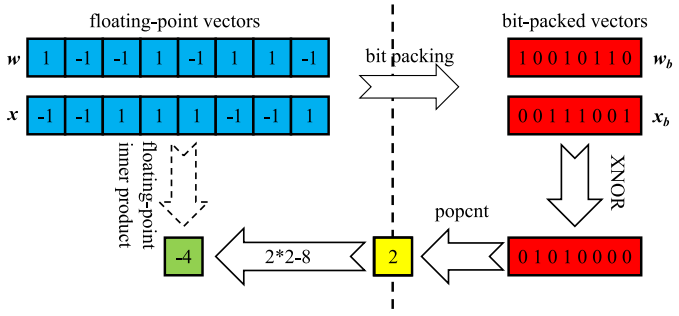

Fig. 2. Floating-point binarized inner product (left) and its acceleration via bitwise operations (right).

Where *popcnt* is an operation counting the number of bits set to 1 in a $N-bit$ number. Fig. 2 shows a simple example for $d=8$ and $N=8$. The binarized floating-point vectors $\boldsymbol{w}$ and $\boldsymbol{x}$ (the blue vectors at the top left of Fig. 2) are first bit-packed as two $8-bit$ numbers $\boldsymbol{w}_b$ and $\boldsymbol{x}_b$ (the red vectors at the top right of Fig. 2). Inner product between $\boldsymbol{w}$ and $\boldsymbol{x}$ can be then done through just one time of *XNOR* and *popcnt* between $\boldsymbol{w}_b$ and $\boldsymbol{x}_b$. In this way, the computation complexity of floating-point innerproduct can be largely reduced.

2.4. Traditional implementations of BNN and their limitations

We have introduced im2col-based convolution algorithm and how computation can be accelerated when inputs of convolution are binarized. In this section, we briefly introduce traditional implementations of BNN and their problems. The left side of Fig. 3 gives an overview of traditional implementations of BNN. To calculate the convolution between binarized kernel K and input I, im2col procedure is first done, (the top to middle part of the left side of Fig. 3). This procedure is exactly the same as what has been described in Section 2.2, generating a transformed matrix I^*. During this stage, the kernel K, stored in order $o_c \times i_c \times k_h \times k_w$, is directly unfolded to a matrix K^* of size $o_c \times i_c k_h k_w$. The output of convolution O can be then calculated with $O = K^* \times I^*$. Considering that K^* and I^* are all binarized, to calculate multiplication between K^* and I^* efficiently, bit-packing procedure is then done (the middle to bottom part of the left side of Fig. 3). During this stage, for K^*, each N elements at the same row and adjacent columns of K^* are bit-packed into a single $N-bit$ number, generating K^*_b. For I^*, each N elements at the same column and adjacent rows of I^* are bit-packed into a single $N-bit$ number, generating I^*_b. The output tensor O of the convolution can be then calculated via *XNOR* and *popcnt* operations between K^*_b and I^*_b.

There are mainly three problems for traditional implementations of BNN. First, the dimension of matrix I^* generated by im2col procedure is $i_c k_h k_w \times o_h o_w$, which is several times of the size of feature maps. This will consume too much extra memory. Second, the bit-packing of I^* needs too much *shift* and *or* operations. Third, during the bit-packing procedure, each N elements at the same row and adjacent columns of I^* are bit-packed into a single $N-bit$ number. Memory access will be uncontinuous, which is not cache-friendly. All these three problems will largely reduce the computation efficiency of BNN.

3. Algorithm

In this section, we present our proposed computation flow for BNN. The overall architecture of our algorithm is shown in the right side of Fig. 3, which consists of three stages, bit-packing, channel-first im2col and binarized matrix to matrix multiplication. All the kernels and feature maps in BitStream are stored in channel-first order, bit-packing procedure is done along the channel dimension, just before channel-first im2col is done. In this way, memory consumption can be largely reduced, and memory access of the whole computation can be more continuous, which is friendly to cache.

3.1. Data layout

Before introducing our algorithm, we first describe the data layout of BitStream. The data layout of BitStream is designed to improve the continuousness of memory access during the whole computation. Detailed analyses will be given in Section 3.5.

For kernels and feature maps of convolution, we store all of them in channel-first order. For i_c feature maps, each of size $i_h \times i_w$, are stored in memory as an array of size $i_h \times i_w \times i_c$. Kernels of a convolutional layer with i_c inputs, o_c outputs and kernel size of

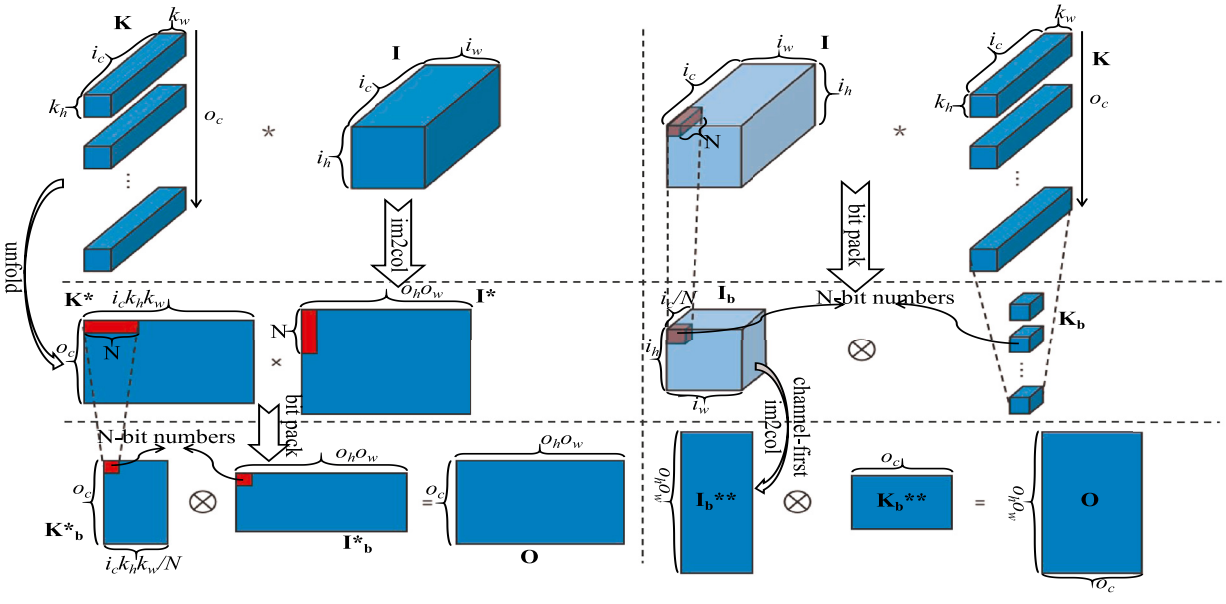

Fig. 3. Computation flow of traditional implementations of BNN (left) and optimized computation flow of BitStream (right).

$k_h \times k_w$, are stored in memory as an array of size $o_c \times k_h \times k_w \times i_c$. Bit-packed kernels ($K_b^{**}$ at the bottom part of the right side of Fig. 3) with height of $\frac{i_c k_h k_w}{N}$ and width of o_c, are stored in memory as an array of size $o_c \times \frac{i_c k_h k_w}{N}$, namely column-major order.

3.2. Bit-packing

In the first stage, bit-packing procedure is done (the top to middle part of the right side of Fig. 3). Different from existing implementations of BNN, in BitStream, all the linear operations, such as BatchNormalization and Scale operations, are all fused into bit-packing operation. Consider a stacked layers of $Convolution + BatchNorm + ReLU(Quantization)$, we have shown in Section 2.3 that:

$$\boldsymbol{w}^T \boldsymbol{x} = 2 \sum_{i=1}^{\lceil \frac{d}{N} \rceil} popcnt(XNOR(\boldsymbol{w}_b[i], \boldsymbol{x}_b[i])) - d \tag{5}$$

Now we define:

$$\boldsymbol{w} \odot \boldsymbol{x} = \sum_{i=1}^{\lceil \frac{d}{N} \rceil} popcnt(XNOR(\boldsymbol{w}_b[i], \boldsymbol{x}_b[i])) \tag{6}$$

The whole computation of $Convolution + BatchNorm + ReLU(Quantization)$ can be formulated as:

$$o = \begin{cases} 1, & \alpha \frac{\boldsymbol{w}^T \boldsymbol{x} + b - \mu}{\sqrt{\sigma + \epsilon}} + \beta > 0 \\ 0, & otherwise \end{cases} \tag{7}$$

From equations above, we can immediately get the equation as follows:

$$o = \begin{cases} 1, & \boldsymbol{w} \odot \boldsymbol{x} > \xi \\ 0, & otherwise \end{cases} \tag{8}$$

where:

$$\xi = \frac{-\frac{\beta \sqrt{\sigma + \epsilon}}{\alpha} + \mu + d - b}{2} \tag{9}$$

We calculate a proper threshold for each channel via Eq. (8) in advance, and perform bit-packing procedure with *shift* and *or* operations. During this stage, each N elements at the same location and adjacent channels of the input tensor I are bit-packed into a single $N-bit$ number, where numbers greater than the corresponding thresehold ξ are replaced by bit 1, and other numbers are replaced by bit 0. Kernels are fixed after training, so for inference only computation, they can be stored in arbitrary format without any extra memory and time. In BitStream, kernels are all directly stored in bit-packed format.

3.3. Binary convolution

After bit-packing procedure is done, binary convolution can be calculated in two steps. First, channel-first im2col prcedure is performed (the middle to bottom part of the right side of Fig. 3). Channel-first im2col procedure is the same as what is described in Section 2.2, except that as the kernel sliding through the bit-packed tensor I_b along height and width dimensions with stride s_h and s_w, the corresponding sub-pactch of I_b is unfolded and duplicated to a row of I_b^{**}, instead of a column of I_b^{**}. The convolution between binarized I and K can be then calculated efficiently via *XNOR* and *popcnt* operations between I_b^{**} and K_b^{**}. Note that in this way, the output O is automatically stored in channel-first order, so no reordering is needed after computation.

After channel-first im2col procedure is done, binary matrix to matrix multiplication between I_b^{**} and K_b^{**} is then calculated via *XNOR* and *popcnt* operations (the bottom part of the right side of Fig. 3). During this stage, each row of I_b^{**} and column of K_b^{**} are first combined with *XNOR* operation. *Popcnt* operations are then done to count the number of bits set to 1 of the combined vector, generating a specific element of the output O. As what we have described, the bit-packed kernel K_b^{**} is stored in column-major order in memory, so during this stage, the memory access of both matrices I_b^{**} and K_b^{**} are completely continuous, which is friendly to cache.

3.4. Binary pooling

Considering that for two single bits w_b and x_b, we have:

$$max(w_b, x_b) = OR(w_b, x_b) \tag{10}$$

so after bit-packing, max-pooling procedure can be done efficiently via *OR* operation. In this way, computation complexity and memory consumption can be further reduced.

3.5. Analyses

In this section, we analyse BitStream and traditional implementations of BNN in terms of memory consumption, memory access and computation complexity. Consider a common stacked layers $Conv + ReLU(Quantization) + Pooling$. To simplify our analyses, we assume that $i_c = o_c$, $i_h = i_w = o_h = o_w$, and the image size after pooling is the half of the size of pooling input, which is usually true in most layers. We use γ to denote the number of bytes of a floating-point precision number.

3.5.1. Memory consumption

We first analyse the memory consumption of traditional implementations of BNN. For convolutional layer, memory consumption of traditional implementations of BNN can be divided into 4 parts, the memory of floating-point inputs, $\gamma i_c \times i_h \times i_w$, the memory of I^* generated by im2col procedure, $\gamma i_c k_h k_w \times o_h o_w$, the memory of bit-packed input I_b^*, which is $\frac{i_c k_h k_w}{8} \times o_h o_w$ and the memory of floating-point output O, which is $\gamma o_c o_h o_w$, so for convolutional layer, memory consumption of traditional implementations of BNN is:

$$MEM_{trad}^{conv} = \gamma i_c i_h i_w + \gamma i_c k_h k_w o_h o_w + \frac{i_c k_h k_w}{8} \times o_h o_w + \gamma o_c o_h o_w \tag{11}$$

ReLU (Quantization) layer is fused into the bit-packing procedure, so in traditional implementations, ReLU (Quantization) layer doesn't consume any extra memory. For pooling layer, the memory consumption of traditional implementations mainly consists of the memory of floating-point outputs of pooling:

$$MEM_{trad}^{pool} = \gamma \frac{o_c o_h o_w}{4} \tag{12}$$

We then analyse the memory consumption of BitStream. In BitStream, inputs of binary convolutional layer are binarized, so the memory needed by inputs of convolutional layer is $\frac{i_c}{8} \times i_h i_w$. The memory for I_b^{**} generated by channel-first im2col procedure is $\frac{i_c k_h k_w}{8} \times o_h o_w$. The size of memory for result O of convolutional layer is $\gamma o_c o_h o_w$. So the memory consumption for convolutional layer of BitStream is:

$$MEM_{bs}^{conv} = \frac{i_c}{8} \times i_h i_w + \frac{i_c k_h k_w}{8} \times o_h o_w + \gamma o_c o_h o_w \tag{13}$$

For ReLU (Quantization) layer, the only needed memory is the bit-packed output:

$$MEM_{bs}^{relu} = \frac{o_c}{8} \times o_h o_w \tag{14}$$

For pooling layer, the memory consumption of BitStream is:

$$MEM_{bs}^{pool} = \frac{o_c}{8} \times \frac{o_h o_w}{4} \tag{15}$$

Under the assumption that $i_c = o_c$ and $i_h = i_w = o_h = o_w$, the total memory consumption of traditional implementations of BNN and BitStream are:

$$MEM_{trad}^{total} = MEM_{trad}^{conv} + MEM_{trad}^{pool} = i_c i_h^2 [2.25\gamma + \left(\gamma + \frac{1}{8}\right) k_h k_w] \tag{16}$$

$$MEM_{bs}^{total} = MEM_{bs}^{conv} + MEC_{bs}^{relu} + MEM_{bs}^{pool} = i_c i_h^2 \left(\frac{1}{8} k_h k_w + \frac{9}{32} + \gamma\right) \tag{17}$$

respectively. The ratio between memory consumption of traditional methods and BitStream is:

$$\frac{MEM_{trad}^{total}}{MEM_{bs}^{total}} = \frac{2.25\gamma + (\gamma + \frac{1}{8}) k_h k_w}{\frac{9}{32} + \gamma + \frac{1}{8} k_h k_w} > min\left(\frac{2.25\gamma}{\frac{9}{32} + \gamma}, \frac{\gamma + \frac{1}{8}}{\frac{1}{8}}\right) \tag{18}$$

From Eq. (18) we can see that the memory consumption of BitStream is always less than that of traditional methods. The larger the kernel size is, the more memory BitStream will save. For example, for a common convolutional layer with kernel size of 3×3, and single floating-point numbers are used in full-precision feature maps. In this situation, $\gamma = 4$, then $8.53\times$ of memory will be saved by BitStream.

3.5.2. Memory access

In this section, we analyse the memory access of BitStream and traditional implementations of BNN. The memory access of BitStream is improved mainly in the procedure of bit-packing.

For traditional implementations of BNN, during the procedure of bit-packing, each N elements at the same column and adjacent rows of the matrix I^* are bit-packed into a single $N-bit$ number of the matrix I_b^*. These N elements of I^* are stored uncontinuous in memory. Specifically, for these N elements, address offset between each two adjacent elements is $\gamma o_h o_w$. This means that in traditional methods, the memory access during the procedure of bit-packing is terribly uncontinuous, which is unfriendly to cache.

For BitStream, during the bit-packing procedure, each N elements at the same location and adjacent channels of the input tensor I are bit-packed into a single $N-bit$ number of I_b. Because the input tensor I is stored in memory with order $i_h \times i_w \times i_c$, these N elements are stored in a continuous area of memory. So in BitStream, the memory access during this procedure is totally continuous.

3.5.3. Computation complexity

The computation complexity of BitStream is reduced in two aspects. First, during the procedure of bit-packing, for traditional implementations of BNN, bit-packing procedure is performed on the matrix I^* of size $i_c k_h k_w \times o_h o_w$. During this stage, $i_c k_h k_w \times o_h o_w$ times of *shift* and *OR* operations are nedded. For BitStream, bit-packing procedure is done just before channel-first im2col. During this stage, only $i_c i_h i_w$ times of *shift* and *OR* operations are needed. Under the assumption that $i_h = o_h = i_w = o_w$, $k_h k_w \times$ of operations are saved. Second, the pooling layer. In traditional implementations of BNN, all the other layers, excluding convolutional layers, are computed in floating-point precision. Suppose the kernel size of pooling layer is k_p^2. Then for traditional implementations of BNN, $\frac{o_c o_h o_w}{4} \times k_p^2$ floating-point comparison operations are needed. In BitStream, pooling layer is calculated through *OR* operations. During this stage, only $\frac{o_h o_w}{4} \times \frac{o_c k_p^2}{N}$ *OR* operations are needed. In this way, computation complexity of BitStream can be largely reduced.

4. Experimental results

In this section, we report the experimental results. To evaluate our algorithm, we implement BitStream with C+ and multi-threaded OpenMP. We benchmark our algorithm on two platforms listed as follows:

- **A72 x 2**: Dure-core Cortex-A72 up to 1.8GHZ with Linux.
- **A53 x 4**: Quad-core Cortex-A53 up to 1.5GHZ with Linux.

To compare with other convolution algorithms, we evaluate several popular methods listed as follows:

- **Im2col – Open**: Floating-point im2col-based convolution implemented in Caffe [10], the General Matrix to Matrix Multiplication is calculated with the highly optimized library OpenBLAS [26].
- **BMXNet**: The state-of-the-art of the traditional implementations of BNN. To compare with it fairly, we download the code of BMXNet [25] from [24] and port it to Caffe.

4.1. Experiments on different convolutional layers

We take 8 convolutional layers appearing frequently in most modern CNN models [8,13,21,22] listed in Table 2. For each layer, we run each algorithm 50 times and take the average execution time. Results are shown in Fig. 4. We can see that BitStream outperforms all the other convolution algorithms.

The left side of Fig. 4 shows the normalized runtime of different algorithms on benchmark convolutional layers. Compared to floating-point im2col-based convolution, BitStream can achieve more than $10\times$ acceleration on all the layers except *conv*6. On 5 out of all the 8 layers, BitStream can get even $30\times$ acceleration. Compared to BMXNet, the state-of-the-art of traditional implementations of BNN, BitStream can get more than $5\times$ acceleration on all the layers except *conv*6. On *conv*2, *conv*3, *conv*7 and *conv*8, BitStream can be even more than $10\times$ faster.

The superiority of BitStream on speed is more obvious on the platform of $A53 \times 4$. See the right side of Fig. 4. BitStream is more than $15\times$ faster compared to im2col-based convolution on all the layers. Compared to BMXNet, BitStream is overall $6\times$ faster. This is mainly because that the cache size of $A53 \times 4$ CPUs is smaller than that of $A72 \times 2$ CPUs, so improving the continuousness of memory access and reducing the memory consumption during computation can bring more improvement of computation efficiency.

4.2. Experiments on different networks

For experiments on different networks, we benchmark our algorithm on four popular networks. VGG-16 [21], Alexnet [13], Resnet-18 [8] and a VGG-like network on CIFAR-10, as the same architecture described in [3]. In all the binary models, the first layer and the last layer are not binarized. Results are shown in Fig. 5.

Table 2
Benchmarks of convolution.

Conv.	$[i_c, i_h, i_w]$	$[o_c, k_h, k_w]$	$[s_h(s_w), p_h(p_w)]$
conv1	[96, 27, 27]	[256, 5, 5]	[1, 2]
conv2	[256, 13, 13]	[384, 3, 3]	[1, 1]
conv3	[384, 13, 13]	[384, 3, 3]	[1, 1]
conv4	[64, 56, 56]	[192, 3, 3]	[1, 1]
conv5	[64, 56, 56]	[64, 3, 3]	[1, 1]
conv6	[64, 56, 56]	[128, 3, 3]	[2, 1]
conv7	[128, 28, 28]	[128, 3, 3]	[1, 1]
conv8	[256, 14, 14]	[256, 3, 3]	[1, 1]

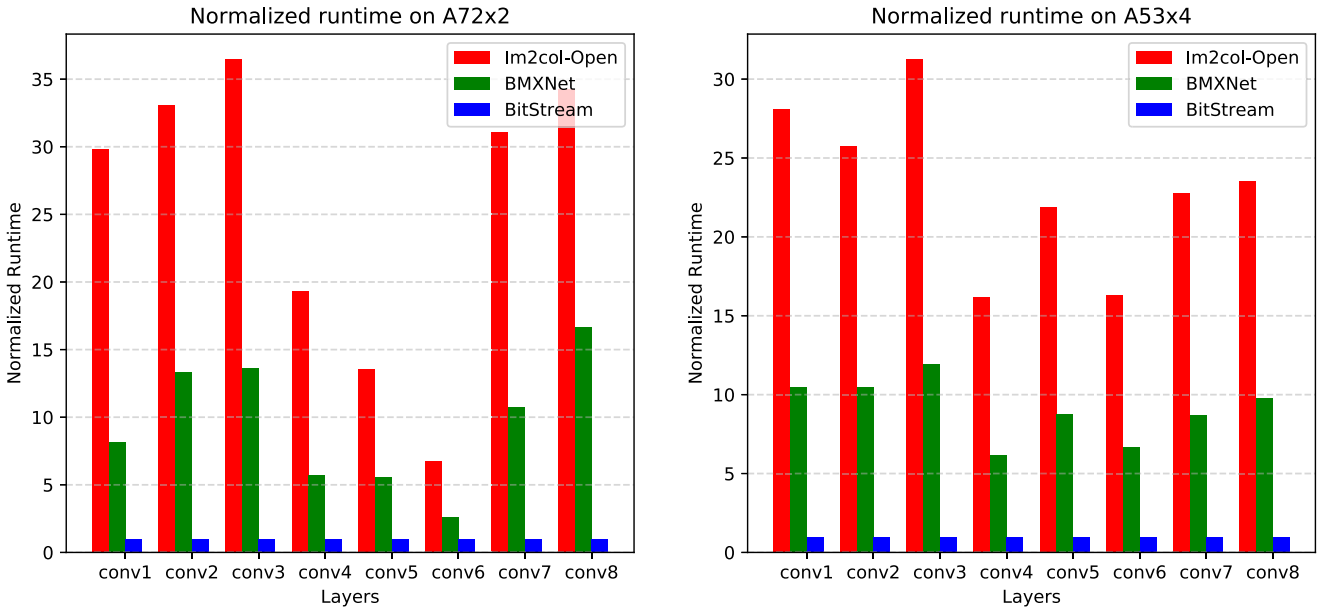

Fig. 4. Normalized runtime of different algorithms on different layers on $A72 \times 2$ (left) and $A53 \times 4$ (right).

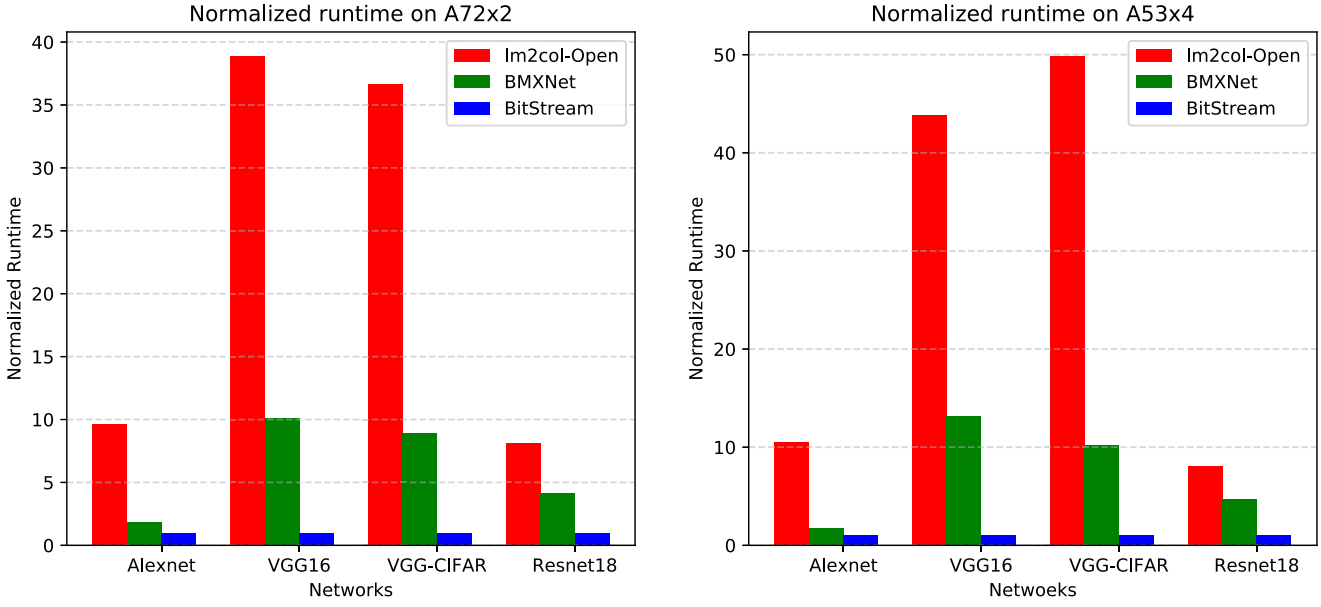

Fig. 5. Normalized runtime of different algorithms on different networks on $A72 \times 2$ (left) and $A53 \times 4$ (right).

4.3. Memory consumption

We have analysed theoretically in Section 3.5 that BitStream can largely reduce the memory consumption compared to traditional methods, in this section, we report the experimental results of memory consumption of different algorithms. Note that in these results, all the memory consumption, including memory for parameters, inputs and outputs of layers, and intermediate memory needed during computation are considered. We evaluate the memory consumption of different algorithms on different layers and networks decribed in the previous Section, results are shown in Fig. 6.

The left side of Fig. 6 shows the comparison of memory consumption of different algorithms on different layers. Interestingly, in the layer *conv*5, the memory consumption of BMXNet is even larger than that of floating-point im2col-based convolution. And in layers *conv*4, *conv*6 and *conv*7, the memory consumption of BMXNet and floating-point im2col-based convolution are similar. This is mainly because that BMXNet needs an extra area of memory to store the bit-packed input I_b^* (see Fig. 3). In some situations, the size of this area of memory may be larger than the size saved by binarization of convolutional parameters. We can see from this figure that compared to BMXNet, BitStream can save overall $5\times$ memory on benchmark convolutional layers.

The right side of Fig. 6 shows the normalized memory consumption of different algorithms on different networks. We can see that on most networks, more than $2.5\times$ memory is saved by BitStream compared to BMXNet. This ratio is smaller than the results shown in the left side, this is because that in all the binary models, the first layer and last layer are not binarized, the memory consumption of all the algorithms on these layers are the same. The ratio of memory saved by BitStream is then reduced.

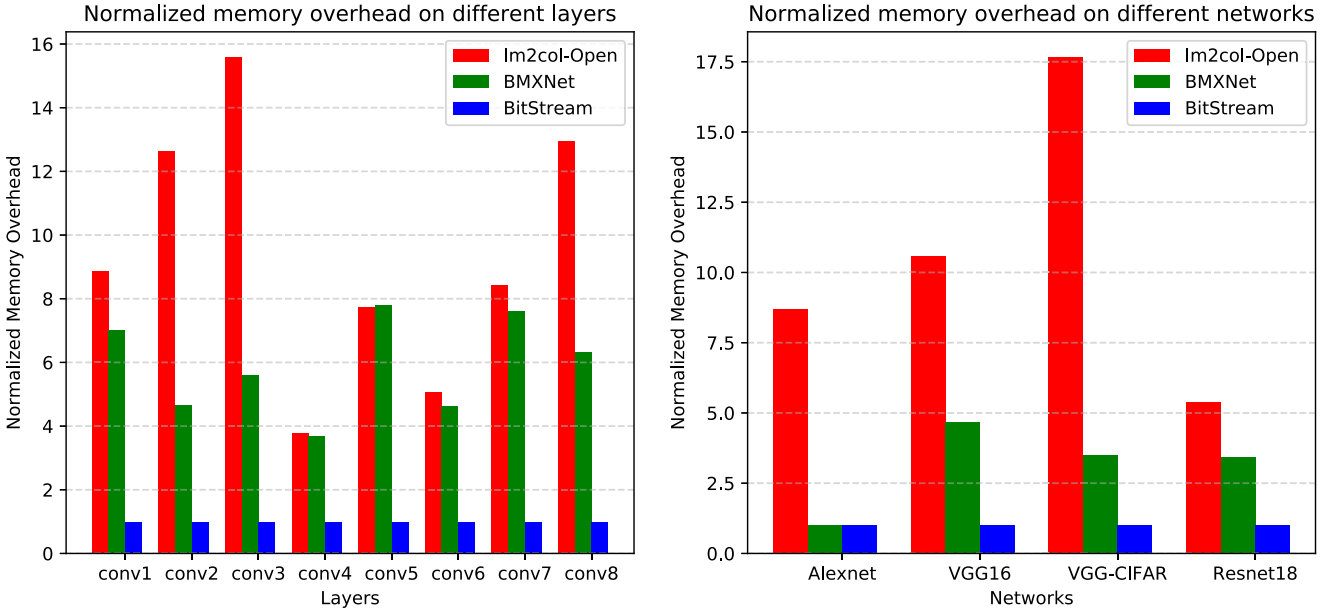

Fig. 6. Normalized memory overhead of different algorithms on different layers (left) and different networks (right).

5. Conclusion

In this paper, we propose BitStream, a general framework for efficient inference of Binary Neural Networks (BNN) on CPUs. In BitStream, we propose a simple but novel memory management strategy, as well as a new computation flow for BNN. Compared to existing implementations of BNN, our algorithm can not only reduce memory consumption, but also improve the continuous of memory access during the inference of BNN. Computation complexity is also largely reduced. Comprehensive analyses and experiments on different networks and hardware platforms show that our algorithm outperforms traditional implementations of BNN in the aspects of both memory consumption and computation efficiency.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P.A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zhang, Tensorflow: a system for large-scale machine learning, CoRR (2016) arXiv:1605.08695.

[2] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems, CoRR (2015) arXiv:1512.01274.

[3] M. Courbariaux, Y. Bengio, Binarynet: training deep neural networks with weights and activations constrained to +1 or −1, CoRR (2016) arXiv:1602.02830.

[4] M. Courbariaux, Y. Bengio, J. David, Binaryconnect: training deep neural networks with binary weights during propagations, CoRR (2015) arXiv:1511.00363.

[5] N.D. Lane, P. Georgiev, Can Deep Learning Revolutionize Mobile Sensing? in: Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications, 2015, pp. 117–122.

[6] C. Dos Santos, M. Gatti de Bayser, Deep convolutional neural networks for sentiment analysis of short texts, COLING, 2014.

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

[8] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, CoRR (2015) arXiv:1512.03385.

[9] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, L.V. Gool, AI Benchmark: running deep neural networks on android smartphones, CoRR (2018) arXiv:1810.01109.

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: convolutional architecture for fast feature embedding, 2014 arXiv:1408.5093.

[11] Y. Kim, Convolutional neural networks for sentence classification, CoRR (2014) arXiv:1408.5882.

[12] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, D. Shin, Compression of deep convolutional neural networks for fast and low power mobile applications, CoRR (2015) arXiv:1511.06530.

[13] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet Classification with Deep Convolutional Neural Networks, in: F. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger (Eds.), Advances in Neural Information Processing Systems 25, Curran Associates, Inc., 2012, pp. 1097–1105.

[14] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S.E. Reed, C. Fu, A.C. Berg, SSD: Single shot multibox detector, CoRR (2015) arXiv:1512.02325.

[15] S.S.L. Oskouei, H. Golestani, M. Kachuee, M. Hashemi, H. Mohammadzade, S. Ghiasi, Gpu-based acceleration of deep convolutional neural networks on mobile platforms, CoRR (2015) arXiv:1511.07376.

[16] E. Park, D. Kim, S. Kim, Y. Kim, G. Kim, S. Yoon, S. Yoo, Big/little deep neural network for ultra low power inference, in: 2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2015, pp. 124–132.

[17] F. Pedersoli, G. Tzanetakis, A. Tagliasacchi, Espresso: Efficient forward propagation for binary deep neural networks, in: International Conference on Learning Representations, 2018.

[18] M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi, Xnor-net: imagenet classification using binary convolutional neural networks, CoRR (2016) arXiv:1603.05279v4.

[19] J. Redmon, A. Farhadi, Yolov3: An incremental improvement, CoRR (2018) arXiv:1804.02767.

[20] S. Ren, K. He, R.B. Girshick, J. Sun, Faster R-CNN: towards real-time object detection with region proposal networks, CoRR (2015) arXiv:1506.01497.

[21] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, CoRR (2014) arXiv:1409.1556.

[22] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S.E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, CoRR (2014) arXiv:1409.4842.

[23] Y. Umuroglu, N.J. Fraser, G. Gambardella, M. Blott, P.H.W. Leong, M. Jahre, K.A. Vissers, FINN: A framework for fast, scalable binarized neural network inference, CoRR (2016) arXiv:1612.07119.

[24] H. Yang, M. Fritzsche, C. Bartz, C. Meinel, Bmxnet: An open-source binary neural network implementation, 2017a, (https://github.com/hpi-xnor/BMXNet).

[25] H. Yang, M. Fritzsche, C. Bartz, C. Meinel, Bmxnet: an open-source binary neural network implementation based on mxnet, CoRR (2017) arXiv:1705.09864.

[26] X. Zhang, Openblas: An optimized blas library, 2018, (http://www.openblas.net/).

[27] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, Z. Zhang, Accelerating binarized convolutional neural networks with software-programmable FPGAS, FPGA, 2018, pp. 15–24.

[28] X. Bai, H. Yang, J. Zhou, P. Ren, J. Cheng, Data-Dependent Hashing Based on p-Stable Distribution, IEEE Trans. Image Processing 23 (12) (2014) 5033–5046.

[29] X. Bai, C. Liu, P. Ren, J. Zhou, H. Zhao, Y. Su, Object Classification via Feature Fusion Based Marginalized Kernels, IEEE Geosci. Remote Sensing Lett 12 (1) (2015) 8–12.

[30] X. Bai, C. Yan, H. Yang, L. Bai, J. Zhou, Edwin Robert Hancock: Adaptive hash retrieval with kernel based similarity, Pattern Recognition 75 (2018) 136–148.

[31] C. Wang, X. Bai, S. Wang, J. Zhou, P. Ren, Multiscale Visual Attention Networks for Object Detection in VHR Remote Sensing Images, IEEE Geosci. Remote Sensing Lett. 16 (2) (2019) 310–314.