BitFlow: Exploiting Vector Parallelism for Binary Neural Networks on CPU

Yuwei Hu[1], Jidong Zhai[2], Dinghua Li[1], Yifan Gong[1], Yuhao Zhu[3], Wei Liu[1], Lei Su[1], and Jiangming Jin[1]

[1]TuSimple Inc
[2]Department of Computer Science, Tsinghua University
[3]Department of Computer Science, University of Rochester

***Abstract*— Deep learning has revolutionized computer vision and other fields since its big bang in 2012. However, it is challenging to deploy Deep Neural Networks (DNNs) into real-world applications due to their high computational complexity. Binary Neural Networks (BNNs) dramatically reduce computational complexity by replacing most arithmetic operations with bitwise operations. Existing implementations of BNNs have been focusing on GPU or FPGA, and using the conventional image-to-column method that doesn't perform well for binary convolution due to low arithmetic intensity and unfriendly pattern for bitwise operations. We propose BitFlow, a gemm-operator-network three-level optimization framework for fully exploiting the computing power of BNNs on CPU. BitFlow features a new class of algorithm named PressedConv for efficient binary convolution using locality-aware layout and vector parallelism. We evaluate BitFlow with the VGG network. On a single core of Intel Xeon Phi, BitFlow obtains 1.8x speedup over unoptimized BNN implementations, and 11.5x speedup over counterpart full-precision DNNs. Over 64 cores, BitFlow enables BNNs to run 1.1x faster than counterpart full-precision DNNs on GPU (GTX 1080).**

I. INTRODUCTION

Deep learning has revolutionized computer vision and other fields since 2012. However, it is challenging to deploy Deep Neural Networks (DNNs) into real-world applications due to their high compute and storage requirements. For example, the VGG-16 model is over 500 MB in size and requires about 40 billion operations per inference. Improving storage efficiency allows for more complex models that offer higher accuracy; improving inference speed allows for quick response to changing events such as early detection of pedestrians in auto driving systems. Therefore, network compression and acceleration is a timely research topic.

Binary Neural Networks (BNNs), first proposed in 2016 by Courbariaux et al. [3], has been demonstrated as an effective method that unifies compression and acceleration. In particular, BNNs refer to neural networks with weights and activations constrained to +1 (represented as 1) or -1 (represented as 0). By replacing floating point operations with bitwise operations (i.e. xor and bit-count), BNNs achieve an compression ratio of $32\times$ compared with full precision networks while dramatically accelerating the inference speed at the cost of little accuracy loss [25]. BNNs offer the prospects of enabling efficient deep learning on performance, memory, and power constrained environment such as autonomous driving and always-on IoT systems [15].

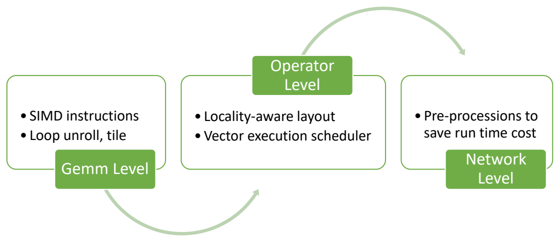


Fig. 1: System Overview

Our work specifically focuses on developing a framework for exploiting the compute and storage efficiency of BNNs on CPUs. The reason we choose to focus on general-purpose CPUs, instead of GPUs [12] or FPGAs [20][24], is that running BNNs on CPU helps free GPU and FPGA resources for other more complex tasks. For instance, auto driving systems typically require multiple perception models such as detection, tracking, and segmentation to be evaluated concurrently. Offloading BNNs to CPUs improves the overall system throughput and latency. In addition, modern CPUs are equipped with SIMD instruction extensions such as SSE and AVX in x86 [13] and NEON and SVE in ARM architectures [11]. SIMD hardware is particularly well-suited for the highly parallel nature of bitwise operations in BNN inference.

We propose BitFlow, a novel approach for optimizing BNN on CPUs. Our **key insight** is that existing implementations of BNNs use the conventional image-to-column method [9] that, although is well-suited for floating-point convolutions, performs poorly on binary convolutions due to the low arithmetic intensity and the unfriendly dataflow pattern of bitwise operations. Instead, BitFlow proposes a new class of algorithm called *PressedConv* for efficient binary convolution. PressedConv uses a unique locality-aware layout for BNN operators and exploits the SIMD-parallelism in modern processor architectures.

We implement BitFlow as a stand-alone inference engine for BNNs without external dependencies. This not only results in a highly optimized implementation of BNNs, but also substantially simplifies its deployment in practical applications. We evaluate BitFlow with the VGG network [17]. On a single core of Intel Xeon Phi, BitFlow obtains 1.8x speedup over unoptimized BNN implementations, and 11.5x speedup over counterpart full-precision DNNs. Over 64 cores, BitFlow enables BNNs to run 1.1x faster than counterpart full-precision DNNs on GPU (GTX 1080). It is worth noting that BitFlow is a generic BNN framework and is generally applicable to other SIMD hardware and architectures.

This paper makes the following contributions:

1) BitFlow is the first framework for exploiting computing power of BNNs on CPU in a systematic way.
2) We propose a new class of algorithm named PressedConv for efficient binary convolution using locality-aware layout and vector parallelism.
3) We evaluate BitFlow with the VGG network. The results show that BitFlow achieves 83% speedup over unoptimized BNN implementations, and even outperforms a GPU implementation of full-precision DNNs.

In section II, we review the background of BNNs together with other neural network compression and acceleration methods, and also the conventional image-to-column convolution method. In section III, we describe our algorithms for efficient binary convolution using locality-aware layout and vector parallelism. In section IV, we describe the implementation of BitFlow framework, and the optimizations conducted on gemm, operator, network levels respectively. In section V, we describe the evaluation methodology and report the experiments results, followed by discussions and conclusions.

II. BACKGROUND

A. Network Compression and Acceleration

Network Pruning. Network pruning has proven to be an effective method to reduce the network size by removing non-informative neural connections. Ciresan [2] proposes to drop the weights randomly to achieve good performance. Han [7] reduces parameter number and computational cost without loss of accuracy on several benchmarks: the parameters below a certain threshold are removed to form a sparse network, then the network is re-trained for the remaining connections.

Network Quantization. In quantized networks, the objective is to train DNNs whose quantized weights don't significantly impact the network's precision. For example, Courbariaux et. al [4] shows that 10-bits are enough for Maxout networks, and how more efficient multiplications can be performed with fixed-point arithmetic. Continuing this trend, Hwang [8] proposed fixed-point DNN with ternary weights {-1,0,+1}. Their training leveraged an optimized backtracking procedure for fixed-point data, obtaining precision very close to that of the floating-point baseline.

Binary Neural Networks (BNNs). To maximize the compression ratio of network, neural network binarization methods are developed. BinaryConnect [5] proposes to constrain the weights to binary values of +1 or -1 and achieves state-of-the-art results on MNIST, CIFAR-10 datasets. BinaryNet [3] further extends the idea of parameter binarization by converting activations to +1 or -1.

The weights of a BNN can be stored in the bits of a 32-bit unsigned int, and this procedure is called bit-packing. One immediate advantage of bit-packing is to drastically reduce the memory usage by a 32× factor. An even more significant advantage is the ability to process multiple values at the same time. Assume $\vec{\boldsymbol{A}}$ and $\vec{\boldsymbol{B}}$ are binarized vectors of length N (N is multiple of 32), A_i is the i_{th} element of *packed* $\vec{\boldsymbol{A}}$, B_i is the i_{th} element of *packed* $\vec{\boldsymbol{B}}$, then the inner product of $\vec{\boldsymbol{A}}$ and $\vec{\boldsymbol{B}}$ is calculated by:

$$\vec{\boldsymbol{A}} \cdot \vec{\boldsymbol{B}} = N - 2 \times \left(\sum_{i=0}^{N/32} \text{bitcount}\left(\text{XOR}\left(A_i, B_i\right)\right) \right) \tag{1}$$

Equation 1 is the secret why BNN brings acceleration. Computationally intensive Floating-point Multiply and Add operations (FMAs) are replaced by xor (for multiplications) and bit-count (for accumulations), enabling significant computational speedup. This represents the cornerstone over which we build the BitFlow framework.

B. Convolution Operation

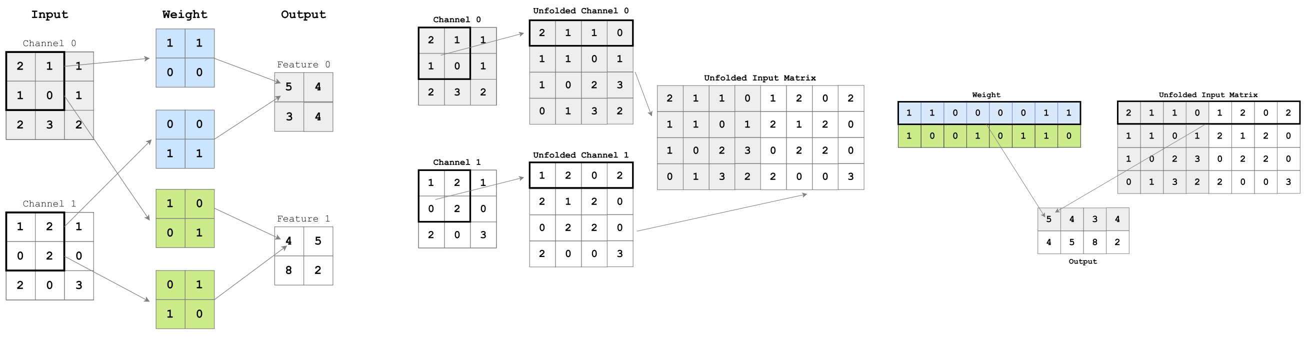
A convolution operator correlates a bank of K filters with C channels and size $h \times w$ against a minibatch of N images with C channels and size $H \times W$. We denote filter elements as $W_{k,c,i,j}$ and image elements as $I_{n,c,x,y}$. The computation of a single convolution output $O_{n,k,x,y}$ is given by the formula:

$$O_{n,k,x,y} = \sum_{c=1}^{C} \sum_{i=1}^{h} \sum_{j=1}^{w} I_{n,c,x+i,y+j} W_{k,c,i,j} \tag{2}$$

Fig. 2a shows an example of a convolution on a two dimensional image of size 3 × 3 with two input channels (Channel 0 and Channel 1). The convolution has two output features (Feature 0 and Feature 1), and each feature has individual sets of weights that correspond to each input channel. The weights for the first channel are the top two matrices and the weights for the second channel are the bottom two matrices. To produce the first element of Feature 0, the convolution computes the inner product of the sub-region of Channel 0 within the black boundary and the feature's weights that correspond to Channel 0. The convolution then sums this result with the inner product of the sub-region of Channel 1 within the black boundary and the feature's weights that correspond to Channel 1.

The conventional execution method for convolution is image-to-column. It consists of two processes:

1) **Unfold:** The first step is to unfold the input activation vector into a matrix, as illustrated in Fig. 2b. For each input channel, the unfolding procedure flattens the inputs to each kernel into a row vector. The sequential concatenation of each row vector produces the unfolded representation of a channel. Then the unfolded input channels are stacked from left to right to produce the final unfolded input matrix.



(a) 2D convolution (b) Unfolding (c) Gemm

Fig. 2: (a) An example of a 2D convolution using a 2×2 kernel with 2 input channels of size 3×3 and 2 output features. (b) Unfolding a 2D image input for convolving with a 2×2 kernel. (c) Computing 2D convolution using Gemm.

2) **Gemm:** The second step performs a matrix-matrix multiplication, with one matrix consisting of the layer's weights and the other consisting of the unfolded activations, as shown in Fig. 2c. It constructs the weight matrix by stacking row vectors that correspond to the flattened representation of the weights for each feature.

This image-to-column convolution approach reduces convolution to a Gemm (matrix multiplication) problem, so that BLAS libraries such as MKL [19] and OpenBLAS [23], can be utilized to achieve good performance.

III. Efficient Binary Operators

A BNN is composed of a sequence of operators whose weights W_k^b and activations a_k^b are binarized to the values $\{-1, +1\}$. The superscript b in the notation indicates binary quantities. Weights and activations are $\{-1, +1\}$, but at the hardware level they must be encoded as $\{0, 1\}$. We follow the convention to encode $-1 \rightarrow 0$ and $+1 \rightarrow 1$. Among many possible choices, e.g. stochastic binarization [14], we employ the following activation function due to its efficient implementation:

$$x^b = sign(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases} \qquad (3)$$

This section describes the algorithms for efficient binary neural network operators in detail, with a focus on binary convolution. The conventional image-to-column method is abandoned due to low arithmetic intensity and unfriendly pattern for bitwise operations when applied to binary convolution, and a new class of algorithm named PressedConv using locality-aware layout and vector parallelism is introduced.

A. Limits of Image-to-column for Binary Convolution

As discussed in section II-B, the conventional image-to-column method reduces convolution to Gemm so that it can benefit from optimized BLAS libraries. This approach is widely used in float-value convolution, but it doesn't perform well when applied to binary convolution due to two limits.

First limit is the overhead brought by the unfolding procedure, which reduces the maximum achievable fraction of the intrinsic Arithmetic Intensity (AIT) of the convolution operation. Arithmetic Intensity (AIT) refers to the ratio of the number of arithmetic operations to the number of memory operations in a computation [6]. It is an important characterization of the performance of a convolution algorithm, and a high AIT is necessary to get high performance because memory operations are slower than arithmetic operations [18]. The AIT of a convolution is $\frac{|A|}{|I|+|W|+|O|}$, where $|A|$ is the number of arithmetic operations and $|I| + |W| + |O|$ is the number of memory accesses. The sets I, W, and O correspond to the input, weight, and output, respectively. Their sizes are calculated as follows:

$$|A| = 2N_f N_x N_y N_c F_x F_y \qquad (4)$$

$$|I| = N_x N_y N_c \qquad (5)$$

$$|W| = N_f N_c F_x F_y \qquad (6)$$

$$|O| = N_f (N_x - F_x + 1)(N_y - F_y + 1) \qquad (7)$$

The unfolding procedure increases the size of the input by approximately a factor of $F_x F_y$. In addition, the unfolded input need to be stored before the Gemm, doubling the number of memory access to the unfolded input. Therefore, the minimum number of memory accesses in image-to-column method is $2|U|+|W|+|O|$, where $|U|$ is the unfolded input with size:

$$|U| = (N_x - F_x + 1)(N_y - F_y + 1)N_c F_x F_y \qquad (8)$$

The resulting fraction of the intrinsic AIT of convolution that image-to-column method can achieve is at most r, where $r = \frac{|I|+|W|+|O|}{2|U|+|W|+|O|}$. For binary convolution, the sizes of input and weight are reduced by a factor of 32 after bit-packing, and the computational complexity is also dramatically reduced, which amplifies the overhead of unfolding procedure and makes AIT even lower.

Another limit of image-to-column method when applied to binary convolution is that after input tensor is unfolded into matrix of $M \times N$, N won't be multiple of 32 in most cases, making bit-packing and SIMD execution inefficient.

B. *PressedConv Algorithm*

We propose PressedConv, a new class of algorithms for efficient binary convolution using locality-aware layout and vector parallelism. PressedConv conducts bit-packing of input tensor and filters along the channel dimension, thus pressing them by a factor of 32 (or 64, 128, 256, 512), and then computes convolution of the pressed input tensor and filters.

Locality-aware Layout. One important property of convolution operator to notice is that the channel dimension is multiple of 32 in most cases. Taking VGG as an example, the channel dimension is 64 in conv 2.1, 128 in conv 3.1, 256 in conv 4.1, and 512 in conv 5.1. This determines that the channel dimension is the best choice along which bit-packing should be conducted. For efficient bit-packing, we adopt NHWC (batch, height, weight, channel) data layout in BitFlow, rather than NCHW, the default setting in mainstream deep learning frameworks e.g. Caffe [10], MXNet [1].

In BitFlow, each element of a tensor $A \in \mathbb{R}^{H\times W\times C}$ is identified by the triplet h, w, c, where $h \in [0, H)$ indicates the height dimension, $w \in [0, W)$ indicates the width dimension, and $c \in [0, C)$ indicates the channel dimension. A is stored in memory using row-major order with interleaved channels. According to the layout, the element $A_{h,w,c}$ is found at position $(hW + w)C + c$ in linear memory. Bit-packing is performed along the channel dimension, and this enables efficient memory access, which would have not been possible if either height or width dimension has been chosen instead. This layout is optimal for retrieving a pixel neighborhood as needed by convolution without requiring the layout to be changed. Moreover, the result will also be stored in the NHWC layout automatically at zero cost.

Vector Execution Scheduler. Modern (co)processors have a feature that one can apply an operation on several data elements in a single instruction - referred to as SIMD (Single Instruction Multiple Data) parallelism. SSE (Streaming SIMD Extension) is designed by Intel and introduced in 1999 in their Pentium III series of processors and it uses 128-bit vector unit. Advanced Vector Extension (AVX) is proposed in 2008 and AVX256 expands most integer commands to 256 bits. AVX512 expands AVX to 512-bit and is available on Intel Xeon Phi coprocessor. Using a 512-bit vector unit, 16 single precision (or 8 double precision) floating point operations can be performed at a single vector operation. In our work, we utilize vector bitwise operations, e.g. `_mm_xor_si128 (__m128i a, __m128i b)` that computes the bitwise XOR of the 128-bit value in a and the 128-bit value in b, `_mm256_xor_si256 (__m256i a, __m256i b)` that computes the bitwise XOR of the 256-bit value in a and the 256-bit value in b, and `_mm512_xor_si512 (__m512i a, __m512i b)` that computes the bitwise XOR of the 512-bit value in a and the 512-bit value in b.

Assume that the input tensor is $H \times W \times C$, and the filter is $h \times w \times C$. As described above, we first pack binarized input and filters into chunks of 32 bits, and then further pack

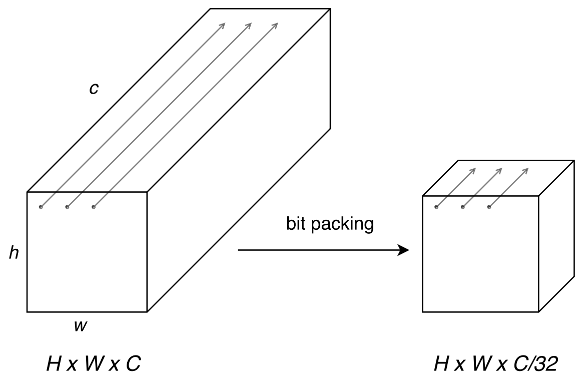


Fig. 3: Bit-packing of input tensor along channel dimension, pressing the tensor by a factor of 32.

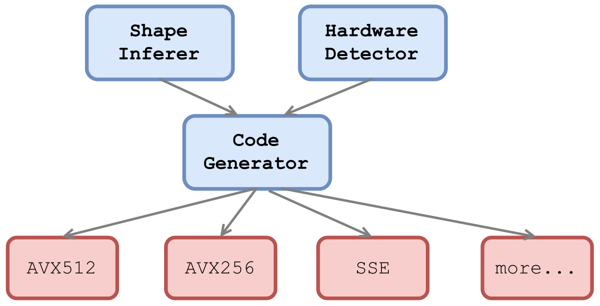


Fig. 4: Vector Execution Scheduler

them into data types of _m128i or _m256i, _m512i to utilize SIMD instructions for efficient bitwise operations. The sizes of input tensor and filters vary for different neural networks. When C is multiple of 512 and AVX512 is available on the hardware, 512-bit instruction can be utilized, however, when this condition is not satisfied, some strategies are needed to ensure that we can still use SIMD efficiently. A general vector execution scheduler is introduced to select the optimal computing kernel for different settings.

The vector execution scheduler consists of three components: shape inferer, hardware detector, and code generator. The shape inferer calculates the output dimensions of each convolution operator in a neural network given the input size and filter sizes. The hardware detector detects whether a certain vector instruction set is available on the hardware platform. The code generator selects the optimal computing kernel for different settings based on the following rules:

1) When channel dimension is multiple of 512, pack unsigned ints into _m512i and utilize AVX512.
2) When channel dimension is multiple of 256, pack unsigned ints into _m256i and utilize AVX256.
3) When channel dimension is multiple of 128, pack unsigned ints into _m128i and utilize SSE.
4) When channel dimension is multiple of 32, use intrinsic bitwise instruction; else, pad extra zeros.

Currently the vector execution scheduler supports SSE, AVX256, and AVX512 because we have implemented computing kernels using these SIMD instruction sets. The scheduler can be easily extended to support more SIMD instruction

TABLE I: SIMD instructions used in this work

SIMD Instruction	Description
`_mm128_xor_si128 (a, b)`	Compute the bitwise XOR of 128 bits in a and b
`_mm256_xor_si256 (a, b)`	Compute the bitwise XOR of 256 bits in a and b
`_mm512_xor_si512 (a, b)`	Compute the bitwise XOR of 512 bits in a and b
`_mm512_maskz_xor_epi64 (k, a, b)`	Compute the bitwise XOR of packed 64-bit integers in a and b using zeromask k
`_mm512_popcnt_epi64 (a)`	Count the number of logical 1 bits in a
`_mm512_maskz_popcnt_epi64 (k, a)`	Count the number of logical 1 bits in a using zeromask k

Algorithm 1 PressedConv

```
step 1: Bit-packing of input tensor along channel dimension.
step 2: Bit-packing of filter along channel dimension.
step 3: Convolution of pressed input tensor and filter. Do multiplications using xor, accumulations using bit-count. Utilize vector parallelism on the C dimension; utilize multi-core parallelism on the fused H and W dimension.
```

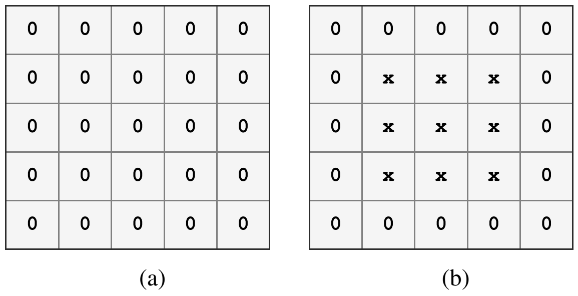


Fig. 5: Automatic padding at zero cost through memory pre-allocation. (a) Pre-allocate memory of size 5×5 for the output. (b) Store the convolution results in the middle 3×3 part, leaving the marginal part unchanged as zero.

sets, such as NEON on ARM architecture.

Multi-core parallelism is also utilized. Since BitFlow targets at low-latency of inference tasks (not training), only the case when batch=1 is considered. Therefore, multi-core parallelism is performed over the height and width dimension.

Address Zero Padding. One issue to address in convolution is zero padding [16]. In convolutional neural networks, in order to avoid that the pixels on the border are not fully utilized, extra pixels of zero value are padded. This procedure introduces overhead that is non-trivial since the computing complexity of binary convolution is drastically reduced compared with float-value convolution. We address zero padding by memory pre-allocation. Assume that the output size of a convolution operator is $H \times W$ (one channel) without padding, and we want to pad it to $(H+2) \times (W+2)$. Not following the first-convolution-then-padding convention, we pre-allocate memory of size $(H + 2) \times (W + 2)$ for the output, and store the convolution results in the middle $H \times W$ part, leaving the marginal part unchanged (zero as initialized). In this way, zero padding is realized at zero cost. Fig. 5 illustrates how this strategy works when $H = 3$ and $W = 3$.

C. Extension to Other Binary Operators

The PressedConv algorithm can be extended and applied to other binary operators, e.g. binary fully connected operator and binary max pooling operator, which are also building blocks of a binary neural network. Binary fully connected operator is in essence doing binary matrix matrix multiplication (Gemm). Assume that the input is $M \times N$, the weight is $N \times K$, we utilize vector parallelism over the N dimension, and multi-core parallelism over the K dimension. As for binary max pooling operator, it also adopts $NHWC$ data layout and conducts bit-packing along the C dimension as in PressedConv algorithm. The difference is that xor and bit-count are replaced by bitwise or, which is used to get the max of a sequence of ones and zeros.

IV. Implementation

We implement BitFlow as a stand-alone inference engine for BNNs without external dependencies. This not only results in a highly optimized implementation of BNNs, but also substantially simplifies its deployment in practical applications, such as on mobile or embedded devices.

Some of the most important data structures in BitFlow are listed in Table II. `bit64_u` uses bit field and union to do binarization and bit-packing efficiently; `m128_u`, `m256_u`, and `m512_u` are used to leverage vectorized xor and bit-count operations.

BitFlow decouples the task of efficiently implementing BNNs into three sub-tasks: gemm-level optimization, operator-level optimization, and network-level optimization. This approach enables us to fully exploit the optimization opportunities on each level.

Gemm-Level Optimization. There has been intensive research on how to implement sGemm (single float general matrix multiplication) on CPU efficiently [21][22]. Some techniques used in sGemm can be directly applied to bGemm (binary Gemm), e.g. tiling and loop unrolling. One difference is that bGemm has additional binarization and bit-packing stages. The data structures of `bit64_t` and `bit64_u` in BitFlow enable us to do bit-packing together with binarization in one step. Moreover, it is common practice to transpose matrix B before computing $A \times B$ for more friendly memory accesses. We conduct transposition of matrix B implicitly with binarization and bit-packing, which means that we store the results of pit-packing in a transposed pattern. To sum up, we fuse binarization, bit-packing, and transposition into a single operation, as illustrated in Table III.

TABLE II: Data structures in BitFlow

```
// for efficient binarization
typedef struct{
   unsigned int b0 :1;
   unsigned int b1 :1;
   unsigned int b2 :1;
   unsigned int b3 :1;
   ...
   ...
   unsigned int b62:1;
   unsigned int b63:1;
}bit64_t;
// for efficient bit-packing
typedef union{
   bit64_t b;
   uint64_t u;
}bit64_u;
// used in SSE
typedef union{
   __m128i m;
   int64_t i[2];
}m128_u;
// used in AVX256
typedef union{
   __m256i m;
   int64_t i[4];
}m256_u;
// used in AVX512
typedef union{
   __m256i m;
   int64_t i[8];
}m512_u;
```

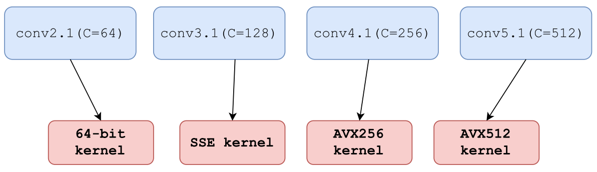


Fig. 6: Mapping from operators to computing kernels

Operator-Level Optimization. As discussed in section III, BitFlow abandons the conventional image-to-column method due to low arithmetic intensity and unfriendly pattern for bitwise operations when applied to binary convolution, and introduces a new class of algorithm named Pressed-Conv using locality-aware layout and vector parallelism for efficient binary convolution. To ensure that we use SIMD efficiently, a general vector execution scheduler is introduced to select the optimal computing kernel for different sizes of input and filter. Taking VGG as an example, the channel dimension is 3 in conv1.1 and the selected kernel pads extra zeros to the channel dimension; it is 64 in conv2.1 and the selected kernel utilizes intrinsic bitwise operations; it is 128 in conv3.1 and the selected kernel utilizes SSE; it is 256 in conv4.1 and the selected kernel utilizes AVX256; it is 512 in conv5.1 and the selected kernel utilizes AVX512 if available e.g. on Intel Xeon Phi, otherwise AVX256 e.g. Intel Core i7. The mapping from operators to computing kernels is illustrated in Fig. 6.

TABLE III: Fused binarization, bit-packing, and transposition in BitFlow

```
/* pack weight: B(float) -> Bb(bit) */
float *p;
int i, j;
bit64_u bit64;
for (j = 0; j < k; j+=1)
{
   for (i = 0; i < n; i+=64)
   {
      p = &B[i*k+j];
      // fuse bit-packing into binarization
      bit64.b.b0  = p[ 0]>=0.0f;
      bit64.b.b1  = p[ 1*k]>=0.0f;
      bit64.b.b2  = p[ 2*k]>=0.0f;
      bit64.b.b3  = p[ 3*k]>=0.0f;
      ...
      ...
      bit64.b.b62 = p[62*k]>=0.0f;
      bit64.b.b63 = p[63*k]>=0.0f;
      // do transposition implicitly
      Bb[(j*n+i)>>6] = bit64.u;
   }
}
```

TABLE IV: VGG architecture. K is the number of filters. For fully connected operators (fc6, fc7), K is the number of weight matrix's columns, and the input has only two dimensions $H \times W$

Operator	$H \times W \times C$	K	stride
conv2.1	$112 \times 112 \times 64$	128	1×1
conv3.1	$56 \times 56 \times 128$	256	1×1
conv4.1	$28 \times 28 \times 256$	512	1×1
conv5.1	$14 \times 14 \times 512$	512	1×1
fc6	25088×4096	4096	None
fc7	4096×4096	1000	None
pool4	$28 \times 28 \times 512$	None	2×2
pool5	$14 \times 14 \times 512$	None	2×2

Network-Level Optimization. Compared with full-precision DNN, BNN introduces binarization and bit-packing stages. Noticing that weights are constant in inference, we conduct binarization and bit-packing of weights during network initialization, once and for all. Besides, we pre-allocate all the memory needed for storing the output and intermediate results by analysis of the neural network as a static computational graph. These pre-processions save run time cost.

V. Evaluation

Experiments Setup. We evaluate BitFlow with VGG, a classic neural network model used in a wide range of computer vision tasks. The VGG network uses 3×3 filters exclusively in the convolution operators. We select 4 convolution operators, 2 fully connected operators, and 2 pooling operators as the benchmarks, as summarized in Table IV. We compared the speed of BitFlow with unoptimized BNN implementations and counterpart full-precision

TABLE V: Accuracy and model size comparison of binarized VGG with full-precision VGG

	full-precision VGG	binarized VGG
MNIST (%)	99.4	98.2
CIFAR10 (%)	92.5	87.8
Model Size (MB)	534	17

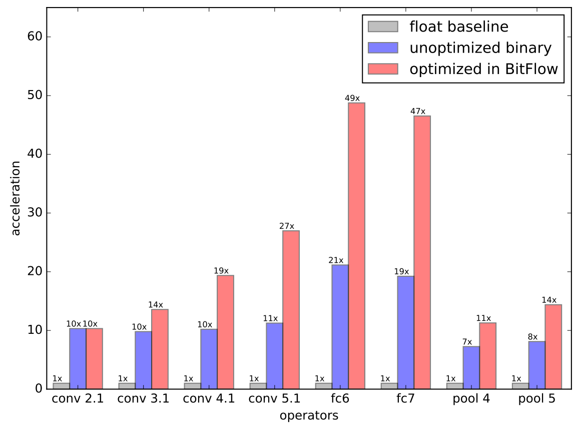


Fig. 7: Performance improvement brought by vectorization over unoptimized BNN implementations, with counterpart float-value operators as the baseline, tested on a single core of Intel Xeon Phi 7210.

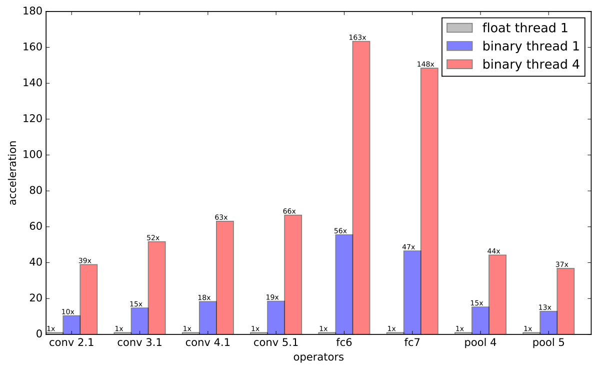


Fig. 8: Multi-core performance of BitFlow on Intel(R) Core(TM) i7-7700HQ

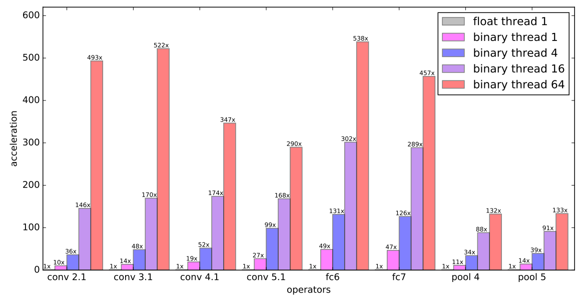


Fig. 9: Multi-core performance of BitFlow on Intel(R) Xeon Phi(TM) CPU 7210

operators. Experiments are conducted on Intel(R) Core(TM) i7-7700HQ and Intel(R) Xeon Phi(TM) CPU 7210, both on a single core and over multi-cores. Furthermore, we compared the best performance of BitFlow with counterpart full-precision operators implemented on GPU (GTX 1080).

Accuracy. First, we run binarized VGG network using BitFlow on datasets of MNIST and CIFAR10, and compared its accuracy with counterpart full-precision VGG network. On MNIST, 98.2% accuracy is achieved, which is 1.2% lower than full-precision VGG; on CIFAR10, the accuracy of binarized VGG is 87.8%, which is 4.7% lower than full-precision VGG, as shown in Table V.

Vector Parallelism Performance. Vectorization brings 83% speedup over unoptimized BNN implementations on average, as shown in Fig. 7. For conv2.1, both BitFlow and unoptimized (i.e. unvectorized) binary kernel achieve 10× acceleration over float-value convolution as the baseline, which is reasonable since the channel dimension is 64 and no SIMD instruction is utilized. For conv3.1, BitFlow is 1.4× faster than unoptimized binary kernel and 14× faster over the baseline, and this is because that SSE 128-bit instruction is utilized in BitFlow to accelerate the computing. For conv4.1, AVX256 is utilized and brings 1.9× acceleration over unoptimized binary kernel. For conv5.1, AVX512 is utilized and brings 2.5× acceleration. For fc6 and fc7, 2.3× acceleration over unoptimized binary kernel and approximately 50× acceleration over float-value operators are observed, which benefits not only from AVX512 vector instruction, but also other gemm optimization techniques (e.g. tiling and loop unrolling). For pool4 and pool5, the acceleration is not that obvious as convolution operator and fully connected operator, and this is because that the computing complexity of pooling is relatively low and the overhead of bit-packing diminishes the acceleration of bitwise operations. On average vector parallelism utilizing SIMD instructions brings 83% speedup over unoptimized BNN implementations.

Multi-core Parallelism Performance. BitFlow scales well to multi-cores on both i7-7700HQ and Xeon Phi for most operators. Fig. 8 shows the scalability of BitFlow over multi-cores on i7-7700HQ, with counterpart float-value operators as the baseline. As we can see, a near-linear scalability is obtained on the selected 8 benchmarks. For conv2.1, 4-core runs 3.9x faster than a single core, and for conv3.1, conv4.1, conv5.1, a 3× acceleration is observed. This is because that the H and W dimension of conv2.1 is large, which contributes to better scalability over multi-cores. As the neural network becomes deeper, the H and W dimension diminishes, which leads to a slightly worse scalability of conv3.1, conv4.1 and conv5.1. Fig. 9 shows the scalability of BitFlow over multi-cores tested on Xeon Phi 7210. As we can see, conv2.1 scales well to even 64 cores and achieves 49.3× acceleration over single-core and 493× acceleration over the float-value baseline. For conv4.1, BitFlow stops scaling well over 16 cores, and compared with 16-cores, no more than 2× acceleration is obtained on 64-cores. Similarly for conv5.1, BitFlow stops scaling well over 4 cores, and compared with 4-cores, no more than 2×

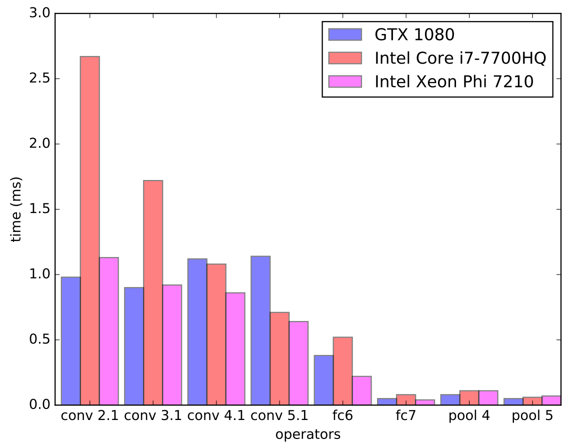


Fig. 10: Wall-clock time comparison of BitFlow with counterpart float-value operators on GPU (GTX 1080)

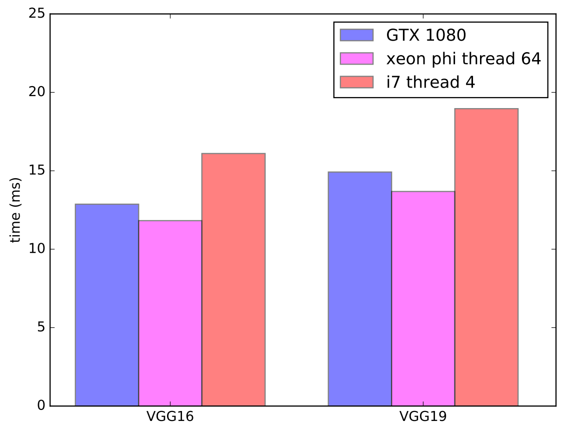


Fig. 11: VGG end to end time comparison

acceleration is obtained on 16-cores.

Comparison with GPU. We compared the wall-clock time of BitFlow with counterpart float-value operators on GPU (GTX 1080), and observed speed of BitFlow comparable with GTX 1080 on Intel i7-7700HQ, and faster on Intel Xeon Phi 7210, as illustrated in Fig. 10. The best performance of BitFlow on Intel i7-7700HQ (4 threads) and Intel Xeon Phi 7210 (64 threads) is used. As we can see, BitFlow on Intel i7-7700HQ (4 threads) is slower than GTX 1080 for conv2.1 and conv3.1, and faster for conv3.1 and conv4.1 because of the efficient utilization of SIMD instructions (AVX256), which again verifies the effectiveness of vector parallelism for fast bitwise operations. On Intel Xeon Phi 7210, the speed of BitFlow for conv2.1 is comparable with that of GTX1080, which is because of good multi-core scalability. For fully connected operators, Intel i7-7700HQ is comparable with GTX 1080, while Intel Xeon Phi 7210 is faster.

Furthermore, we compared VGG end to end time of BitFlow with counterpart full-precision network on GTX 1080, as shown in Fig. 11. The speed of VGG on GPU is tested using keras with tensorflow 1.2 as the backend. The best performance of BitFlow on Intel i7-7700HQ (4 threads) and Intel Xeon Phi 7210 (64 threads) is used. VGG19 and VGG16 have similar architectures, except that VGG19 has 3 more convolution operators. As we can see, the end to end inference time of VGG16 is 12.87ms on GTX 1080, 16.10ms on Intel i7-7700HQ, and 11.82ms on Intel Xeon Phi 7210. For VGG19, the the end to end inference time is 14.92ms on GTX 1080, 18.96ms on Intel i7-7700HQ, and 13.68ms on Intel Xeon Phi 7210. BitFlow on Intel Xeon Phi 7210 brings 8.9% speedup over GTX 1080 for VGG16, and 9.1% speedup for VGG19.

To sum up, we evaluate BitFlow with the VGG network. The results show that: BitFlow's vectorization method brings 83% speedup over unoptimized BNN implementations; BitFlow scales well to multi-cores on both i7-7700HQ and Xeon Phi for most operators; over 64 cores, BitFlow enables binarized VGG to run 1.1$\times$ faster than full-precision VGG.

VI. Conclusion

BNNs unify storage compression and speed acceleration, and is critical for deep learning inference in constrained environment. Existing BNN implementations do not achieve the full potential of BNN because it largely inherits the image-to-column convolution strategy used by full-precision DNNs and thus leads to poor performance scaling. This paper proposes PressedConv, a new convolution algorithm specifically designed for BNN. Based on PressedConv, we present BitFlow, a hierarchical framework that optimizes BNN performance at gemm-, operator-, and network-level. Based on evaluation on the popular VGG network, we show that BitFlow obtains 83% speedup over unoptimized BNN implementations on a single core machine, and 10% speedup over GPU implementations of full-precision DNNs on a 64 core machine. BitFlow shows a promising first step toward utilizing binarization for fast and storage-efficient deep learning.

References

[1] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[2] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1237–1242. AAAI Press, 2011.

[3] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[4] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.

[5] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363, 2015.

[6] Pieter Ghysels, P Kłosiewicz, and Wim Vanroose. Improving the arithmetic intensity of multigrid with the help of polynomial smoothers. *Numerical Linear Algebra with Applications*, 19(2):253–267, 2012.

[7] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

[8] K. Hwang and W. Sung. Fixed-point feedforward deep neural network design using weights +1, 0, and -1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, Oct 2014.

[9] Yangqing Jia. Convolution in caffe: a memo, 2015.

[10] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[11] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou. Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1107–1116, May 2013.

[12] Fabrizio Pedersoli, George Tzanetakis, and Andrea Tagliasacchi. Espresso: Efficient forward propagation for bcnns. *CoRR*, abs/1705.07175, 2017.

[13] Simon J. Pennycook, Chris J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring simd for molecular dynamics, using intel®xeon®processors and intel®xeon phi&#153; coprocessors. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 1085–1097, Washington, DC, USA, 2013. IEEE Computer Society.

[14] Tapani Raiko, Mathias Berglund, Guillaume Alain, and Laurent Dinh. Techniques for learning binary stochastic feedforward neural networks. *arXiv preprint arXiv:1406.2989*, 2014.

[15] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

[16] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pages 958–962, 2003.

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[18] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[19] Intel Software. Intel math kernel library (intel mkl), 2017.

[20] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119, 2016.

[21] Field G Van Zee and Robert A Van De Geijn. Blis: A framework for rapidly instantiating blas functionality. *ACM Transactions on Mathematical Software (TOMS)*, 41(3):14, 2015.

[22] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, page 20. ACM, 2016.

[23] Z Xianyi, W Qian, and W Saar. Openblas: an optimized blas library.(2016), 2016.

[24] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 15–24, New York, NY, USA, 2017. ACM.

[25] L. Zhuang, Y. Xu, B. Ni, and H. Xu. Flexible Network Binarization with Layer-wise Priority. *ArXiv e-prints*, September 2017.