Batched Data Layout-Optimization for Batched Im2col-based Convolution Method on CPUs

Hongzhi Zhao , *Member, IEEE,*, Xun Liu , Ruiyang Chen, Deyang Wang, and Jinxiang Xie

***Abstract*—Most researches focus on designing optimization strategies of batched convolutions for DNN inference tasks based on the architectural characteristics of GPUs and NPUs, ignoring CPUs involving in batch processing as the essential computational resources in many edge devices. Im2col-based convolution is a common convolution method widely applied in deep learning frameworks. However, the existing im2col-based convolution method repeats access to the same weight data after a certain interval in the batched convolution calculations, and results in the reuse distances for accessing the same weight data, increasing memory accesses and impacting the computation efficiency of batched im2col-based convolutions. The proposed algorithm batched data layout-optimization for batched im2col-based convolution method (BDLO) improves the computation efficiency of batched im2col-based convolutions by optimizing the data layout of the batched input feature maps to eliminate the reuse distances of the weight data to zero in the processes of batched im2col-based convolutions. As a result, compared to the im2col-based convolution method used in LibTorch on Eigen and OpenBLAS, the experimental results show that the computation efficiencies of BDLO are higher at the most benchmarks, especially with the small sizes of the input feature maps with *NCHW* layout. BDLO achieves the best performances with approximately $1.348\times$ speedup for *NCHW*, an L1 data cache miss rate of 0.294%, and an L2 data cache miss rate of 5.76%, which are far lower than those of the im2col-based convolution method used in LibTorch.**

***Index Terms*—Deep neural networks (DNNs), batched im2col-based convolutions, CPUs, reuse distances, data layouts.**

I. Introduction

APPLICATIONS of convolutions in the deep neural networks (DNNs) greatly promote the evolution of artificial intelligence, achieving great successes in image recognition [1]–[3], object detection [4]–[6], and semantic segmentation [7], [8]. The wide application of DNN algorithms has generated a large number of DNN inference tasks. In recent years, both academia and industry have focused on deploying DNN algorithms on resource-constrained devices near the edge [11]. The DNN inference tasks are usually processed in a batch for higher processing efficiency [15].

CPUs are essential components in many edge devices. Since the limited computational and storage resources of edge devices, CPUs often need to participate in batch processing for DNN inference tasks [11]. Most researches focus on designing optimization strategies of batch processing for DNN inference tasks based on the architectural characteristics of devices like GPUs [14] and NPUs, while overlooking the involvement of CPUs in batch processing.

A DNN consists of various types of layers, including convolutional layers, activation layers, pooling layers and more. Among these layers, the convolutional layer is one of the most important but the slowest and most memory-intensive ones in the convolutional DNNs. Inferences of the DNNs are the computationally-intensive tasks with convolution layers accounting for roughly 80% of the total convolutional DNN running time [12]. Therefore, it's significant to optimize the computing processes of the convolution for reducing the latency of the convolutions, and improving the efficiency of the calculation.

Im2col-based convolution, also known as im2col + general matrix multiply (GEMM), is commonly employed in implementing the convolutions. This approach is widely applied in deep learning (DL) frameworks [16], [17] to enhance the computational efficiency of convolutional layers. The core of im2col-based convolution is lowering the convolution to GEMM. Im2col-based convolution transforms an input feature map of a convolution into an input matrix at first, allowing most of the data in the input feature map to be stored continuously and redundantly in memory. The weight data are also converted into a weight matrix without redundant storage. Then, highly optimized GEMM routines in BLAS libraries [18], [19] are invoked to calculate the multiplications of the input matrix and weight matrix, generating the output feature maps of a convolution. Many optimization works related to im2col-based convolution typically concentrate on reducing memory overhead by improving the data layout of the input matrix [9], [13] to increase the computational efficiency of convolutions.

The emphasis of these methods is to optimize the computation process of a single im2col-based convolution. When dealing with the batched convolution calculations, N input feature maps are often transformed into N input matrices. Then N GEMM routines are called to perform the batched im2col-based convolutions. We refer to this batch computation method as the original batch computation. The original batch computation logic is simple and easy to implement, and mainstream DL frameworks also adopt this approach for batch computation. Since the need to call N GEMM routines, the original batch computation repeats access to the same weight data after a certain interval during the computation process. During this interval, CPUs access other weight data. This access pattern results in a reuse distance for accessing the same weight data. The reuse distance represents the number of unique memory locations accessed between two consecutive

This work was supported by the Fundamental Research Funds for the Central Universities under Grant 2023YJS025.

Hongzhi Zhao, Xun Liu, Ruiyang Chen, Deyang Wang, and Jinxiang Xie are with School of Computer and Information Technology, Beijing Jiaotong University, Beijing, 100044, China (e-mal: hzzhao@bjtu.edu.cn).

accesses to the same memory location.

Benefiting from the multi-level cache hierarchy of CPUs, the access efficiency of CPUs will be improved if the computational data can reside in the cache for an extended period. However, the reuse distances of weight data directly impact the residence time of weight data in the cache, resulting in frequent memory accesses, decreasing cache hit rate, and consequently affecting the computational efficiency of batch processing.

In this paper, we propose a novel batch im2col convolution method on CPUs, termed Batched Data Layout-Optimization for Batched Im2col-based Convolution Method (BDLO). The core objective of BDLO is reducing the memory accesses by eliminating the reuse distances of weight data during a single batch convolution computation process, thus enhancing the computational efficiency in performing batched convolution computations on CPUs. BDLO achieves this by optimizing the data layouts of batched input matrices in the batch computations, converting N input feature maps into one large input matrix. Then BDLO requires the GEMM routine only once for the batched convolutions, without modifications, but invokes existing GEMM routines [18], [19]. The GEMM routine generates a hybrid output matrix consisting of N output feature maps. And this hybrid output matrix needs to be separated into N individual output feature maps for the subsequent layer computations. When the computations of the next layer involve the non-cross-channel operations, such as in batch normalization layers and pooling layers, BDLO integrates the separation operations of the hybrid output matrix into the computation processes of the next layer, lowing the latency of these separation operations.

Experimental results demonstrate that for the input feature maps with the *NCHW* data layout format, the best performances of BDLO are approximately $1.321\times$ speedup compared to the im2col-based convolution used in LibTorch. Additionally, BDLO can result in lower convolution latency for the input feature maps with the *NCHW* data layout compared to the input feature maps with the *NHWC* data layout, while the sizes of the input feature maps are small. This implies that when CPUs and GPUs collaborate in processing the DNN inference tasks, CPUs can handle data passed from GPUs using the same data layout as GPUs without the need for data layout conversion. This is highly advantageous for edge devices with limited computational and storage resources.

Our contributions can be summarized as follows.

1) To the best of our knowledge, this is the first work to improve the computation efficiency of batched convolutions on CPUs by eliminating the reuse distances of weight data for reducing the memory accesses. Our main contribution is that we propose BDLO, which optimizes the batched data layouts, completely eliminating the reuse distances of the same weight data during the batched convolution computations. BDLO enhances cache access efficiency and reduces the latency of batch convolution computation.
2) When the computations in the next layer are the non-cross-channel computations, such as in batch normalization layers and pooling layers, BDLO integrates the hybrid output matrix separation operations into the computation processes of the next layer.
3) With the experimental results, our work demonstrates that the computational efficiency of data using the *NHWC* data layout is not always higher than that of the *NCHW* data layout on CPUs. From the perspective of a unified computational data layout, BDLO provides a new technical support to improve the computational efficiency of heterogeneous multi-core collaborative batch processing for DNN inference tasks.

II. Related Works

The optimization on im2col-based convolution. The optimization researches on im2col-based convolution are usually concerned about memory overhead, minimizing memory footprint in convolution is critical for DNN applications [9].

Several methods have been proposed to eliminate or reduce the copy of the image tensor in memory for easing memory overhead. YaConv [12] prevents unnecessary copies of image elements, and make direct use of unmodified GEMM building blocks from high performance numerical libraries, improving cache utilization. YaConv reshapes the input tensor and packs it in the contiguous layout for outer-product GEMM matrices. And the weight tensor is packed in the same way that is done in the GEMM routine of the im2col-based convolution. YaConv avoids unnecessary additional copies of image elements and uses unchanged GEMM microkernels.

The primitive block algorithm of GEMM decomposes the whole convolution into multiple blocks with two-round packing operations, generating substantial memory overhead. ECBC [13] simplifies two-round packing operations to one-round packing operations in the im2col transformations, directly producing the objective tensors of the two-round packing operations. And the objective tensors are packed into a shared tensor when required. This method conducts the convolutions and the im2col transformations in a blockwise way more concisely and greatly reduces the memory overhead in the computations.

III. Background and Motivation

Direct convolution is the straightforward implementation of convolution. In the direct convolutions, the convolutional kernels (sliding windows) slide over the input feature map, conducting element-wise multiplications and accumulating the results within the sliding windows. The sum within each sliding window contributes a single data point to the output feature map on the difference channels. Since the poor cache locality, direct convolution is often less efficient compared to other convolutional methods in many cases. Im2col-based convolution transforms the input feature maps into matrices with redundancy. And the convolutions can be performed as the matrix-matrix multiplications, which shows better cache locality than the direct convolutions. And the im2col-based convolutions can leverage highly optimized BLAS libraries [18], [19], which are extensively utilized in popular deep learning frameworks such as PyTorch [16], MXNet [20], and TensorFlow [17].

TABLE I
SUMMARY OF NOTATIONS

Symbol	Description
N	Batch size
C_{in}/C_{out}	Number of input/output channels
$W_{in}/W_{out}/W_{kernel}$	Width of inputs/outputs/wights
$H_{in}/H_{out}/H_{kernel}$	Height of inputs/outputs /kernels
S	Stride
P	Padding
M_{I}	Input matrix
M_{W}	Weight matrix
M_{O}	Output matrix
M_{HI}	Hybrid Input matrix
$M_{\mathrm{HI}}^{\mathrm{T}}$	Transpose of Hybrid Input matrix
M_{HO}	Hybrid Output matrix
$\otimes$	Convolution
$\times$	Matrix multiplication
$*$	Scalar multiplication

Our work is closely related to the im2col-based convolution algorithm for the batched input feature maps. The calculation processes of im2col convolution are comprised of three main steps: im2col transformation, matrix multiplication, and results reshaping. In this section, we first describe the im2col transformation, matrix multiplication & result transformation, and batch GEMM. Then we elaborate our motivation based on the mentioned backgrounds. The symbols used in the paper are described in Table I.

A. Im2col Transformation

During the im2col transformation, a row-major input feature map is unfolded into a column-major matrix, which is referred to as the input matrix M_{I}. In this matrix, some data from the input feature map may be repeated.

Assume that one input feature map consists of C_{in} channels, each with $H_{in} * W_{in}$ elements, where $C_{in} = 3, H_{in} = 3$, and $W_{in} = 3$ as an instance shown in Fig. 1(a). The data layout, *CHW* or *HWC*, defines the storage direction for the data in the input feature map. *CHW* denotes that the data in the same channel are contiguously stored in the memory. And *HWC* denotes that the data at the same position are contiguously stored across multiple channels. With this storage characteristic, Fig. 1(a) illustrates the input matrices converted by the input feature maps with the different data layouts. The sizes of input matrices for *CHW* and *HWC* are determined before they are populated with data. For *CHW* data layout, the input matrix size is $(H_{out}*W_{out})*(C_{in}*H_{kernel}*W_{kernel})$, and for *HWC* data layout, it is $(C_{in}*H_{kernel}*W_{kernel})*(H_{out}*W_{out})$, with H_{out} and W_{out} being calculated using (1a) and (1b). Then filling data into each column of the input matrices. The input matrices for *CHW* and *HWC* shown in Fig. 1(a) are resulted from making good use of the data layouts of the input feature maps. And these implementations of the im2col transformation are utilized in the mainstream DL framework PyTorch [16].

$$H_{out} = \lfloor (H_{in} + 2 * P - H_{kernel})/S \rfloor + 1 \tag{1a}$$
$$W_{out} = \lfloor (W_{in} + 2 * P - W_{kernel})/S \rfloor + 1 \tag{1b}$$

The weights also need to be unfolded into a column-major matrix without any repeated data, referred to as the weight matrix M_{W}. And the weights always keep the same data layout as the input feature maps. Weights consist of C_{out} kernels, each of size $C_{in} * H_{kernel} * W_{kernel}$, assuming that $C_{out} = 2, H_{kernel} = 2$, and $W_{kernel} = 2$ as shown in Fig. 1(b). Each kernel is unfolded into a column of the weight matrix. The C_{out} columns are contiguously stored in the memory. Therefore, the size of the weight matrix is $(C_{in} * H_{kernel} * W_{kernel}) * C_{out}$. Despite having the same sizes, the data inside of M_{W} are stored in different order for the *NCHW* layout and *NHWC* layout.

B. Matrix Multiplication & Result Transformation

The process of im2col transformation involves converting a 3-dimensional input feature map and 4-dimensional weights into an input matrix and weight matrix, respectively. These matrices enable the efficient utilization of GEMM routines which are supported by BLAS libraries on CPUs and integrated in many DL frameworks. The GEMM routines can be called with the different parameter sets, such as sgemm in PyTorch. The key parameters of the GEMM routines usually consist of (TRANSPOSE $transA$, TRANSPOSE $transB$, int m, int n, int k, float $alpha$, float $*A$, int lda, float $*B$, int ldb, float $beta$, float $*C$, int ldc). The parameters $transA$ and $transB$ specify whether the matrices (A and B) are transposed before the matrix multiplication. m, n and k are the dimensions of the matrices involved in the multiplication. $alpha$ and $beta$ are usually set to 1 and 0 in convolutions, respectively. lda, ldb and ldc are the leading dimensions of matrices. The leading dimension is the number of elements between successive rows for the row-major matrix in memory.

Fig. 1(c) shows the abstract process of the above GEMM routines. We define that the direction of reading the calculating data is the calculate direction. The calculate directions are the same with the storage directions of A, B and C, fully leveraging column storage for these matrices in the memory and resulting in a higher cache hit rate. For *CHW* data layout, A is the input matrix M_{I} and B is the weight matrix M_{W}. For *HWC* data layout, A is the transposed matrix of the weight matrix M_{W} and B is the input matrix M_{I}. In a GEMM routine, each data of B is consecutively accessed for all relevant calculations and is not accessed later. This consecutive access pattern contributes to the cache-friendliness of the CPU's memory hierarchy. For an example as shown in Fig.1 (d), we assume that calling a GEMM routine for the matrices with *CHW* data layout, where A is the input matrix M_{I} and B is the weight matrix M_{W}. The weight data 10 is consecutively accessed and will not be accessed in the remaining computations of the GEMM routine.

After invoking a GEMM routine, the output matrix M_{O} is generated with the size of $m * n$. In many optimizations of the im2col-based convolutions[xx][xx], the data layout of the output matrix may not adhere to *NCHW* or *NHWC*. However, the next layers normally expect *NCHW* layout or *NHWC* layout. Hence, the data layout of the output matrix will be transformed into *NCHW* or *NHWC* for the next layers.

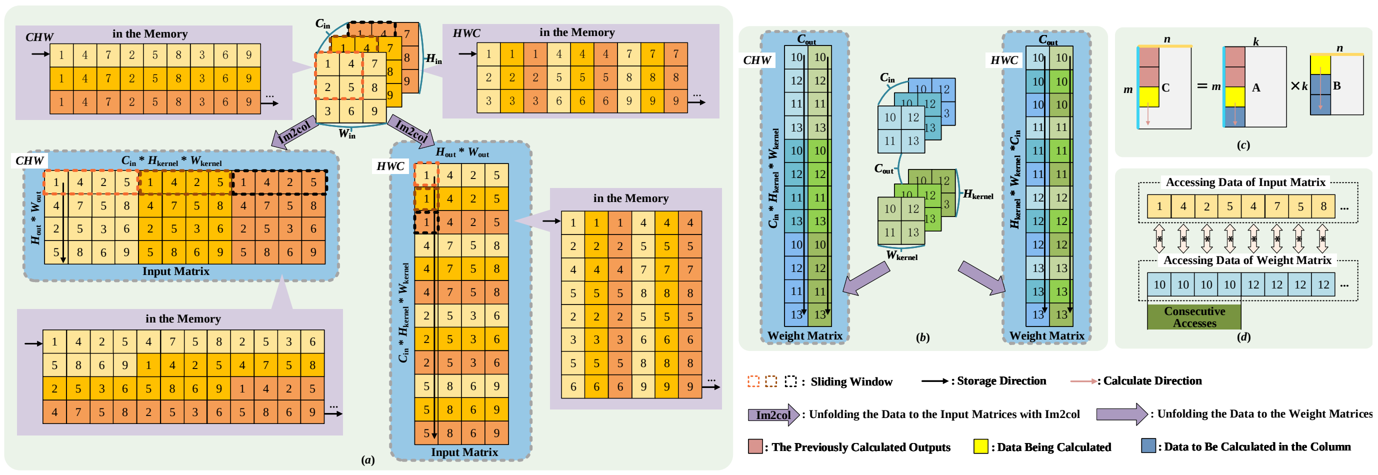

in the Memory
CHW
HWC
Input Matrix
Weight Matrix
Im2col
Sliding Window
Storage Direction
Calculate Direction
Unfolding the Data to the Input Matrices with Im2col
Unfolding the Data to the Weight Matrices
The Previously Calculated Outputs
Data Being Calculated
Data to Be Calculated in the Column
Accessing Data of Input Matrix
Accessing Data of Weight Matrix
Consecutive Accesses

Fig. 1. (a) The input feature maps for *NCHW* and *NHWC* data layouts are each unfolded into two types of input matrices, respectively. (b) The weights for *NCHW* and *NHWC* data layouts are each unfolded into two types of weight matrices, respectively. (c) The abstract process involves the GEMM routines supported by BLAS libraries on CPUs and integrated in mainstream DL frameworks. (d) The data accesses of input matrix and weight matrix in a GEMM routine.

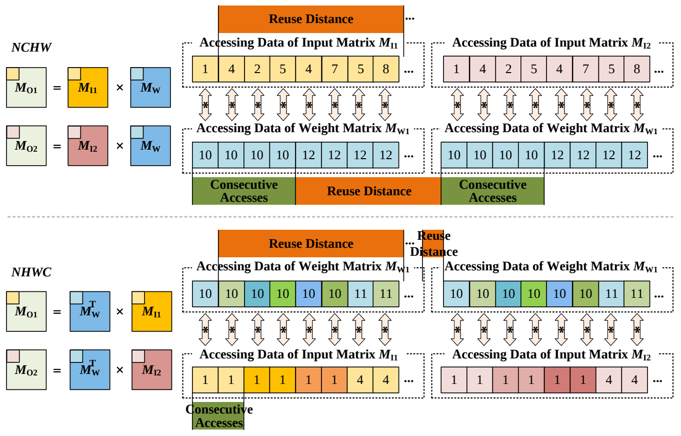

NCHW
NHWC
Reuse Distance
Accessing Data of Input Matrix M_{I1}
Accessing Data of Input Matrix M_{I2}
Accessing Data of Weight Matrix M_{W1}
Consecutive Accesses

Fig. 2. The reuse distances of the data for the N GEMM routines.

C. Batch GEMM

If N input feature maps (where $N > 1$) have the same sizes, each input feature maps will be converted into the input matrix using the im2col transformation method described above. The start address of the next input matrix is contiguous with the end address of the last input matrix. The batched input matrices, with batch size of N, are represented as a 3D input matrix with the size of $N*(H_{out}*W_{out})*(C_{in}*H_{kernel}*W_{kernel})$. And the weights are also unfolded into the weight matrix M_{W} as shown in Fig. 1(b). The frameworks[xx] and some existing works[xx][xx] execute the matrix multiplications for batched input matrices by calling the GEMM routine N times. Each GEMM routine multiplies one input matrix in the batch by M_{W}.

Fig. 2 illustrates the matrix multiplications for batched input matrices with *NCHW* layout and *NHWC* layout respectively. For *NCHW* data layout, when viewing the N GEMM routines as a larger routine, the consecutive accesses for the same data in M_{W} are discretely distributed across the larger routine, as depicted in Fig. 2. In the N GEMM routines, the weight data 10 is accessed again after finishing the previous GEMM routine, generating the reuse distance of weight 10. Reuse distance measures the number of unique memory locations accessed between two consecutive accesses to the same memory location. The long reuse distances may result in that the data are replaced from the caches. In each routine, the data in M_{I} are not consecutively accessed, but the data are repeatedly accessed every once in a while. For *NHWC* data layout, since the input matrices in the routines are different, the consecutive accesses for the same data in M_{I} only exist in every single routine. In the N GEMM routines, the data in M_{W} are also not consecutively accessed, but the weight data are repeatedly accessed later. Hence, the reuse distances of weight data are scattered in the N GEMM routines.

D. Motivation

In the implementations of the above batched im2col-based convolutions, we observe that 1) the weight matrix M_{W} must be repeatedly accessed in each GEMM routine, generating large reuse distances, and 2) the consecutive accesses of individual data occur only within the right matrix in the matrix multiplication. The mode of the above batched im2col-based convolutions ignores the fact that the large reuse distances of the data directly impact the data residence time in the cache, resulting in increased memory accesses and incurring additional time costs.

Fortunately, we are inspired by the observations that the reuse distances of each weight data can be eliminated when each data is continuously multiplied by all the corresponding data in the batched input matrices. And the continuous multiplications of each data in the weight matrix can be achieved by adjusting the data layouts of the batched input matrices. Hence, these observations motivate us to design a better computation method for batched im2col-based convolutions on CPUs, eliminating the reuse distances for data in the weight matrix, reducing the accesses from memory and improving the efficiency of batched im2col-based convolutions.

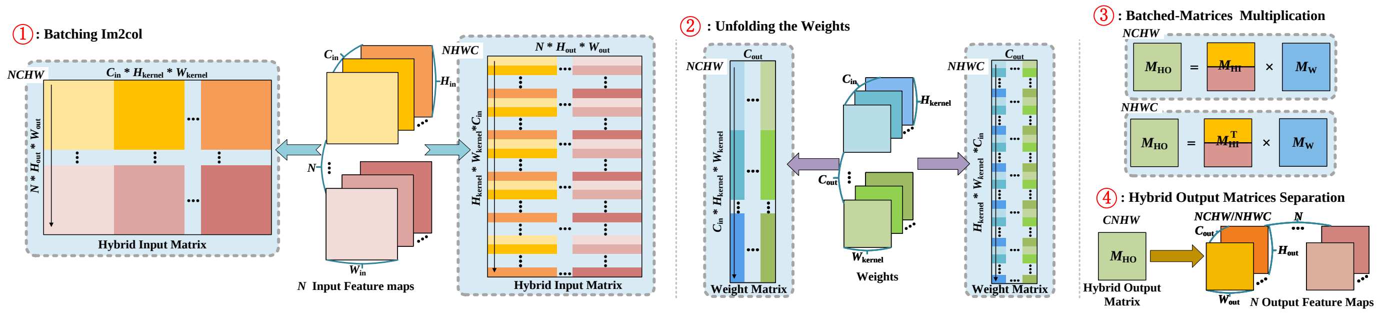


Fig. 3. The four computation steps of Batched Data Layout-Optimization based on Batched Im2col-based Convolution Method (BDLO) for *NCHW* and *NHWC* data layouts.

IV. Methodology

Focus on the problems referred above, we propose a novel batched im2col-based convolution method, termed Batched Data Layout-Optimization for Batched Im2col-based Convolution Method on CPUs(BDLO). Our method completely exterminates the reuse distances for the weight matrix data to zero in the batched im2col-based convolutions with optimizing the data layouts of the batched input matrices. BDLO ensures the production of the same output feature maps as the im2col-based convolution and increases computation efficiency for the batched im2col-based convolutions on CPUs. BDLO can be applied to the input feature maps with both the *NCHW* and *NHWC* data layouts.

A. Overview

BDLO integrates the convolution computations of N input feature maps into one convolutional implementation, which consists of four steps: ① batching im2col, ② unfolding the weights, ③ batched-matrices multiplication, and ④ hybrid output matrices separation. While using BDLO, the input feature maps with the *NCHW* and *NHWC* data layouts undergo the identical computation processes, with some differences in the implementation details in step ① batching im2col and step ④ hybrid output matrices separation. Fig. 3 illustrates the four steps of BDLO for the batched the input feature maps with *NCHW* and *NHWC* data layouts, respectively.

In step ① batching im2col, the N batched input feature maps are converted into a column-major input matrix. The size of the column-major input matrix is $(N * H_{out} * W_{out}) * (C_{in} * H_{kernel} * W_{kernel})$ for *NCHW* layout and $(H_{kernel} * W_{kernel} * C_{in}) * (N * H_{out} * W_{out})$ for *NHWC* layout. The data from the N input feature maps are stored within the input matrix using multiple matrices hybrid layout, referred to as the hybrid input matrix M_{HI}.

In step ② unfolding the weights, the weights are also unfolded into a column-major matrix, referred to as the weight matrix M_{W}, without any repeated data. This process is consistent with the weight matrix conversion described in Section II-A. The size of the weight matrix is $(C_{in} * H_{kernel} * W_{kernel}) * C_{out}$ for *NCHW* layout and $(H_{kernel} * W_{kernel} * C_{in}) * C_{out}$ for *NHWC* layout. And the data storage of the weight matrix is shown in Fig. 3 ②.

After generating the hybrid input matrix and the weight matrix, the GEMM routine as described in Section II-B is called in step ③ batched-matrices multiplication. As elaborated in Fig. 3 ③, for *NCHW* layout, the hybrid input matrix $M_{\mathrm{HI}}[m][n]$ serves as the left matrix, and the weight matrix serves $M_{\mathrm{W}}[n][k]$ as the right matrix in the the GEMM routine, where $m = N * H_{out} * W_{out}$, $k = C_{in} * H_{kernel} * W_{kernel}$ and $n = C_{out}$. For *NHWC* layout, the transpose of the hybrid input matrix $M_{\mathrm{HI}}^{\mathrm{T}}[m][n]$ serves as the left matrix, and the weight matrix $M_{\mathrm{W}}[n][k]$ serves as the right matrix in the the GEMM routine, where $m = N * H_{out} * W_{out}$, $k = H_{kernel} * W_{kernel} * C_{in}$ and $n = C_{out}$. And the GEMM routine is only invoked once in the computation processes of BDLO, producing a column-major output matrix referred to as hybrid output matrix $M_{\mathrm{HO}}[m][n]$ with the *CNHW* data layout.

The computations of the next layer normally expect *NCHW* layout or *NHWC* layout. If the next layer involves the computation operations across the channels, such as convolutional layers, full-connected layers, and layer normalization layers, BDLO will explicitly perform step ④ hybrid output matrices separation to get the N output feature maps with the objective layout as the input data for the next layer. If the computations of the next layer are the non-cross-channel computations, such as pooling layers, activation layers, and batch normalization layers, the hybrid output matrix with *CNHW* layout can be directly utilized as the input data for the next layer. And the each output data of the next layer is directly positioned at the corresponding location in its output feature maps with *NCHW* layout or *NHWC* layout, as opposed to implementing the data movements of step ④ after obtaining all output data of the next layer. The implementation of step ④ hybrid output matrices separation is seamlessly integrated into the computing processes of the next layer, implicitly converting the data layout of the hybrid output matrix into *NCHW* layout or *NHWC* layout.

The key technologies of BDLO are the multiple matrices hybrid layout of the hybrid input matrix and the hybrid output matrix separation strategy. By employing the multiple matrices hybrid layout, all data relevant to the calculations of each weight data are stored consecutively in either a column of the hybrid input matrix or a column of its transpose. This layout facilitates consecutive accesses during matrix multiplication without encountering reuse distances, ultimately reducing memory accesses. Hybrid output matrix separation strategy decreases the time of data layout conversions for the hybrid output matrix when the computations of the next layer involve the non-cross-channel operations. We will describe these two key technologies in detail in the following.

B. Multiple Matrices Hybrid Layout

Multiple matrices hybrid layout is designed for the consecutive storage of all data in memory, with each data relevant to the computations involving the same weight data. In step ① batching im2col, the batched input feature maps are transformed to form a column-major hybrid input matrix M_{HI} with the multiple matrices hybrid layout. The implementations of step ① batching im2col vary between *NCHW* layout and *NHWC* layout. Algorithm 1 gives the processes of the batching im2col.

Inside line 3 to line 22 of Algorithm 1, for *NCHW* data layout, step ① batching im2col converts the N input feature maps into a column-major hybrid input matrix with the size of $(N{*}H_{out}{*}W_{out}){*}(C_{in}{*}H_{kernel}{*}W_{kernel})$, as shown in Fig 3 ①. The hybrid input matrix can be viewed as the concatenation of N ordinary input matrices along $(H_{out} * W_{out})$ dimension. Each ordinary input matrix has a size of $(H_{out}{*}W_{out}){*}(C_{in}{*}H_{kernel} * W_{kernel})$. The x-th column of each ordinary input matrix $(0 \leq x < C_{in} * H_{kernel} * W_{kernel})$ is referred to as a subcolumn for the x-th column of the hybrid input matrix. These N subcolumns are then spliced together to generate the x-th column of the hybrid input matrix. Since the column-major storage format, the start address of the x-th column is contiguous with the end address of the $(x-1)$-th column in the hybrid input matrix, resulting in the multiple matrices hybrid layout.

Assume that the first data for the y-th subcolumn of the x-th column $(0 \leq x < C_{in} * H_{kernel} * W_{kernel}, 0 \leq y < N)$ can be represented as a four-dimensional coordinate (y, ch, h, w) in M_{HI}. ch, h, and w are respectively defined in line 5 to line 8 of Algorithm 1, denoting the channel, height and width. The offset dst from the first data to (y, ch, h, w) in M_{HI} can be calculated using the equation in line 10 of Algorithm 1, based on the position (y, ch, h, w). In line 11, src is calculated as the offset from the first data in the first input feature map to the first data of the channel where (y, ch, h, w) is located in the y-th input feature map. The offset src of (y, ch, h, w) consists of $y * (C_{in} * H_{in} * W_{in})$ in the batch size dimension and $ch{*}(H_{in}{*}W_{in})$ in the channel dimension. In Fig. 4(a), the components of dst and src are illustrated. In line 12 to line 20, the corresponding $H_{out} * W_{out}$ data in the batched input feature maps, located with the starting address src, are copied to positions with the starting address dst, thereby filling the y-th subcolumn of the x-th column in M_{HI}.

Inside line 24 to line 40 of Algorithm 1, for *NHWC* layout, the N input feature maps are transformed into N column-major input matrices, each with a size of $(H_{kernel} * W_{kernel} * C_{in}) * (H_{out} * W_{out})$. And these N input matrices are concatenated one by one and stored sequentially in the memory, resulting in a M_{HI} with the size of $(H_{kernel} * W_{kernel} * C_{in}) * (N * H_{out} * W_{out})$ as shown in Fig. 3 ①.

The first data of the k-th column $(0 \leq k < N * H_{out} * W_{out})$ can be represented as a four-dimensional coordinate $(k\_b, h, w, 0)$ in M_{HI}. k_b, h, and w are respectively defined

Algorithm 1 Batching Im2col of BDLO

```
Input: Input feature maps $I$ with $N{*}C_{in}{*}H_{in}{*}W_{in}$, $H_{kernel}$, $W_{kernel}$, $H_{out}$, $W_{out}$, $S$, $P$
Output: Hybrid input matrix $\mathbf{M}_{\text{HI}}$
 1: Interpret $l$ as data layout of $I$
 2: if $l$ is NCHW then
 3:   Initialize $\mathbf{M}_{\text{HI}}[N{*}H_{out}{*}W_{out}][C_{in}{*}H_{kernel}{*}W_{kernel}]$ as all zeros
 4:   for $x \in [0, C_{in} * H_{kernel} * W_{kernel} - 1]$ in parallel do
 5:     $ch = x/(H_{kernel} * W_{kernel})$
 6:     $rest = x\%(H_{kernel} * W_{kernel})$
 7:     $h = rest/W_{kernel}$
 8:     $w = rest\%W_{kernel}$
 9:     for $y \in [0, N - 1]$ do
10:       $dst = ch{*}(N{*}H_{out}{*}W_{out}{*}H_{kernel}{*}W_{kernel}) + h * (N * H_{out} * W_{out} * W_{kernel}) + w * (N * H_{out} * W_{out}) + y * (H_{out} * W_{out})$
11:       $src = y * (C_{in} * H_{in} * W_{in}) + ch * (H_{in} * W_{in})$
12:       for $u \in [0, H_{out} - 1]$ do
13:         $iu = u * S - P + h$
14:         for $v \in [0, W_{out} - 1]$ do
15:           $iv = v * S - P + w$
16:           if $(iu >= 0$ and $iu < H_{in})$ and $(iv >= 0$ and $iv < W_{in})$ then
17:             memcpy($\mathbf{M}_{\text{HI}} + dst + u * W_{out} + v, I + src + iu * H_{in} + iv$, sizeof($scalar\_t$) $* 1$)
18:           end if
19:         end for
20:       end for
21:     end for
22:   end for
23: else if $l$ is NHWC then
24:   Initialize $\mathbf{M}_{\text{HI}}[H_{kernel}{*}W_{kernel}{*}C_{in}][N{*}H_{out}{*}W_{out}]$ as all zeros
25:   for $k \in [0, N * H_{out} * W_{out} - 1]$ in parallel do
26:     data_index_init$(0, h, H_{out}, w, W_{out})$
27:     $k\_b = k/(H_{out} * W_{out})$
28:     $dst = k\_b{*}(H_{kernel}{*}W_{kernel}{*}C_{in}){*}(H_{out}{*}W_{out}) + h * (W_{out} * H_{kernel} * W_{kernel} * C_{in}) + w * (H_{kernel} * W_{kernel} * C_{in})$
29:     $src = k\_b * (H_{in} * W_{in} * C_{in})$
30:     for $u \in [0, H_{kernel} - 1]$ do
31:       $iu = h * S - P + u$
32:       for $v \in [0, W_{kernel} - 1]$ do
33:         $iv = w * S - P + v$
34:         if $(iu >= 0$ and $iu < H_{in})$ and $(iv >= 0$ and $iv < W_{in})$ then
35:           memcpy($\mathbf{M}_{\text{HI}} + dst + u * W_{out} * C_{in} + v * C_{in}, I + src + iu * W_{in} * C_{in} + iv * C_{in}$, sizeof($scalar\_t$) $* C_{in}$)
36:         end if
37:       end for
38:     end for
39:     data_index_step$(h, H_{out}, w, W_{out})$
40:   end for
41: end if
```

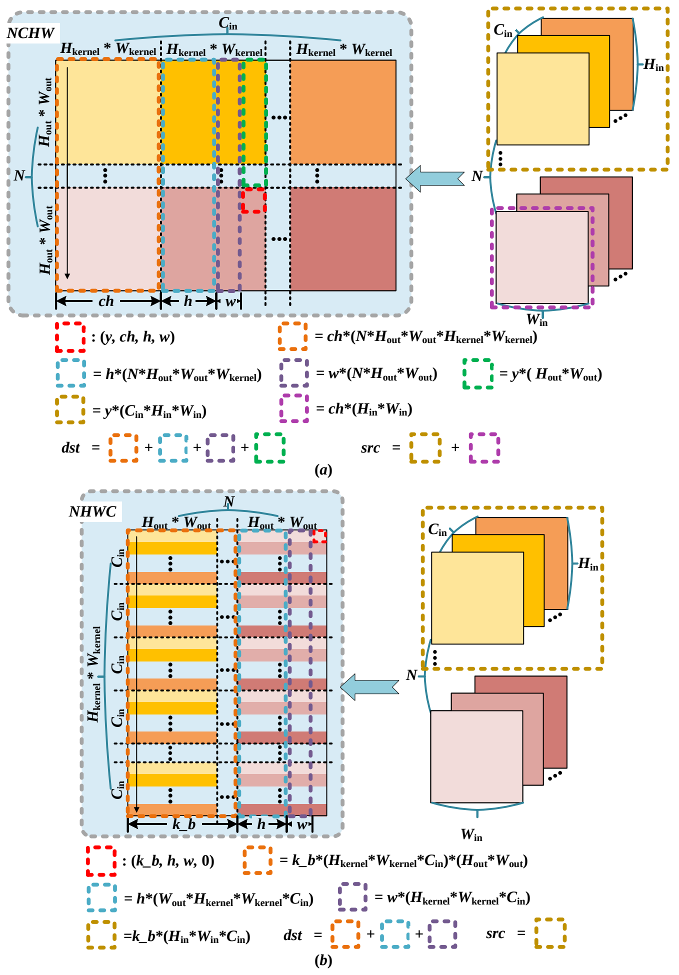


Fig. 4. The components of dst and src for (a) *NCHW* data layout and (b) *NHWC* data layout.

in line 25 to line 27 of Algorithm 1, denoting the batch size, height and width. The offset dst from the first data to $(k\_b, h, w, 0)$ in M_{HI} can be calculated using the equation in line 28. And in line 29, src is the offset from the first data in the first input feature map to the first data in the k_b-th input feature map. Fig. 4(b) shows the components of dst and src for *NHWC* layout, respectively. In line 30 to line 38, the corresponding $H_{kernel} * W_{kernel} * C_{in}$ data in the batched input feature maps with the starting address src are copied to positions with the starting address dst, thereby filling the k-th column in M_{HI}. Prior to batched-matrices multiplication, M_{HI} is transposed to ensure that the data relevant to computations involving the same weight data are stored consecutively. And the transpose of M_{HI} uses multiple matrices hybrid layout, referred to as M_{HI}^{T}. The M_{HI} for the *NCHW* layout can always match a column with identical data in the transpose of M_{HI} for *NHWC* layout.

C. Hybrid Output Matrix Separation Strategy

Step ③ batched-matrices multiplication generates the hybrid output matrix M_{HO} with the *CNHW* data layout which means that the data at the same channel in every output feature map are consecutively stored in memory. Therefore, the *CNHW* data layout needs to be transformed to the *NCHW* layout or *NHWC* layout in step ④ hybrid output matrix separation. The implementations of the hybrid output matrix separation strategy are different for the diverse computation types of the next layer.

If the next layer involves the computation operations across the channels, such as convolutional layers, full-connected layers, and layer normalization layers, we will explicitly invoke permute function[xx] or transpose function[xx] in the DL frameworks for M_{HO} to get the N output feature maps with the objective layout.

If the computations of the next layer are the non-cross-channel computations, such as pooling layers, activation layers, and batch normalization layers, M_{HO} with *CNHW* layout will be directly utilized as the input data for the inference computations of the next layer. When obtaining an output data of the next layer, we utilize (2) to calculate the corresponding position $(batch, channel, height, width)$ in the batched output feature maps of the next layer with the *NCHW* layout.

$$channel = ind/(H_{out}^{'} * W_{out}^{'})/N \tag{2a}$$

$$batch = ind/(H_{out}^{'} * W_{out}^{'})\%N \tag{2b}$$

$$height = ind\%(H_{out}^{'} * W_{out}^{'})/W_{out}^{'} \tag{2c}$$

$$width = ind\%(H_{out}^{'} * W_{out}^{'})\%W_{out}^{'} \tag{2d}$$

ind $(0 \leq ind < C_{out} * N * H_{out}^{'} * W_{out}^{'})$ denotes that the next layer produces the ind-th output data. $H_{out}^{'}$ and $W_{out}^{'}$ are the height and width of the output feature maps for the next layer. Then the ind-th output data is directly stored in the batched output feature maps at the position $(batch, channel, height, width)$. When the next layer produces all output data, the batched output feature maps of the next layer with the *NCHW* layout are formed at the same time. For *NHWC* layout, the corresponding position is $(batch, height, width, channel)$ and also can be calculated by (2). dst_{ind} $(0 \leq dst_{ind} < C_{out} * N * H_{out}^{'} * W_{out}^{'})$ is the offset from the first data to the position in the batched output feature maps of the next layer, as defined in (3), which is used for the specific storage operations.

$$dst_{ind} = \begin{cases} batch * (C_{out} * H_{out}^{'} * W_{out}^{'}) \\ +channel * (H_{out}^{'} * W_{out}^{'}) \\ +height * W_{out}^{'} + width, & \text{if } l \text{ is } \textit{NCHW} \\ \\ batch * (H_{out}^{'} * W_{out}^{'} * C_{out}) \\ +height * (W_{out}^{'} * C_{out}) \\ +width * C_{out} + channel, & \text{if } l \text{ is } \textit{NHWC}. \end{cases} \tag{3}$$

Thus the operations of the hybrid output matrix separation are implicitly performed in the processes of the next layer, reducing the repeated data accesses from the memory.

D. Case Study

We take an example to describe the working process of our proposed BDLO, batched data layout-optimization based

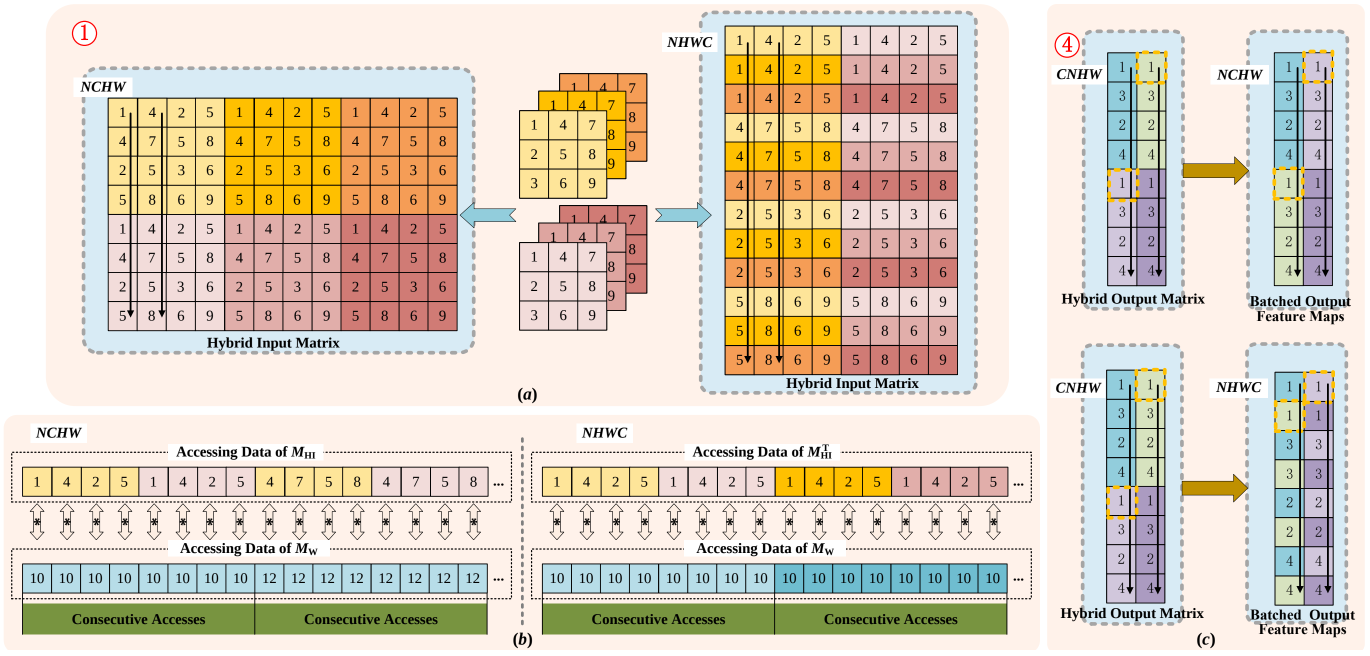


Fig. 5. The case study for BDLO.

on batched im2col-based convolution method in detail. We assume the convolution parameters are set as the following: $N = 2$, $H_{in} = W_{in} = 3$, $C_{in} = 3$, $H_{kernel} = W_{kernel} = 2$, $C_{out} = 2$, $S = 1$, $P = 0$, and $H_{out} = W_{out} = 2$.

At first, 2 input feature maps are converted to a column-major M_{HI} with multiple matrices hybrid layout in step ① batching im2col, as shown in Fig. 5(a). For *NCHW* layout, the first four data in each column of $M_{\text{HI}}$are copied from the first input feature map, and the remaining data come from the second input feature map. For *NHWC* layout, the data in the first four columns of M_{HI} are sourced from the sliding windows on the first input feature map, as shown in Fig. 1(a). Similarly, the last four columns are produced by the same operations on the second feature map. These eight columns are stored contiguously. Then we transpose the M_{HI} with the *NHWC* layout to obtain a M^{T}_{HI}. The second column {1, 4, 2, 5, 1, 4, 2, 5} of M^{T}_{HI} with the *NHWC* layout can match the fifth column of M_{HI} with the *NCHW* layout. And the weights are unfolded into M_{W} in step ② unfolding the weight, as shown in Fig. 1(b). The data inside of M_{W} are stored in different order for the *NCHW* layout and *NHWC* layout. The elements in the first column of M_{W} with *NCHW* layout are {10, 12, 11, 13, 10, 12, 11, 13, 10, 12, 11, 13} as an instance, while for *NHWC* layout, they are {10, 10, 10, 11, 11, 11, 12, 12, 12, 13, 13, 13}.

Then in step ③ batched-matrices multiplication, M_{W} is always positioned to the right of the multiplication sign, as depicted in Fig. 3 ③. Furthermore, Fig. 5(b) illustrates some details of the data accesses. Benefited from the multiple matrices hybrid layout, weight data 10 is consecutively accessed for the all related multiplications and will not be repeatedly accessed in the batched-matrices multiplication. And so as the other weight data. Therefore, the reuse distances of the weight data are reduced to zero in the processes of the batched-matrices multiplication.

And in both the *NCHW* layout and *NHWC* layout, step ③ yields a M_{HO} with the *CNHW* layout. Assume that the next layer is batch normalization layer, and the M_{HO} is its inputs. The layout of the M_{HO} is transformed to *NCHW* or *NHWC* in the computation processes of the next layer. And the batched output feature maps of the next layer are unfolded for the observation data storage in the memory, where $H^{'}_{out} = W^{'}_{out} = 2$, as depicted in Fig. 5(c). The output data 1 of the first column, circled in yellow ink, is the fifth $(ind = 4)$ output data of the next layer. Hence the position of output data 1 in the batched output feature maps can be calculated by using (2). For *NCHW* layout, the position is (1, 0, 0, 0) and $dst_{ind} = 8$. For *NHWC* layout, the position is (1, 0, 0, 0) and $dst_{ind} = 8$. Output data 1 should be stored at the offset dst_{ind} in batched output feature maps. Similarly, another output data 1 at the second column, circled in yellow ink, is the ninth $(ind = 8)$ output data of the next layer. For *NCHW* layout, the position is (0, 1, 0, 0) and $dst_{ind} = 4$. For *NHWC* layout, the position is (0, 0, 0, 1) and $dst_{ind} = 1$.

V. Evaluation

A. Experimental Setup

The experiments are conducted on Linux server TaiShan 2280 v2 with Kunpeng 920 CPU [21] in C++ using single 32-bit precision. Some key parameters of Kunpeng 920 CPU are as the following: L1i cache 64KB, L1d cache 64KB, L2 cache 512 KB, L3 cache 32 MB and frequency 2.60GHz. The basic benchmarks are shown in Table 2. These benchmarks come from various popular DNN algorithms [23], [24]. By default, $N = 8$, $S = 1$, and $P = 1$. All tests are conducted using a single thread and a single core on Kunpeng 920 CPU to prevent interference of computation partition from the multi-threaded processing, which is the subject of our further work.

TABLE II
BENCHMARKS OF CONVOLUTIONAL LAYERS

Name	Input feature maps $(C_{in}, H_{in} \times W_{in})$	weights $(C_{out}, H_{kernel} \times W_{kernel})$
$c1$	$3, 224 \times 224$	$32, 3 \times 3$
$c2$	$32, 112 \times 112$	$96, 3 \times 3$
$c3$	$96, 112 \times 112$	$96, 3 \times 3$
$c4$	$144, 56 \times 56$	$144, 3 \times 3$
$c5$	$192, 28 \times 28$	$192, 3 \times 3$
$c6$	$384, 14 \times 14$	$384, 3 \times 3$
$c7$	$576, 14 \times 112$	$576, 3 \times 3$
$c8$	$960, 7 \times 112$	$960, 3 \times 3$

To evaluate the proposed algorithm, in this section, we present the comparison between BDLO and a used industrial-level im2col-based convolution on LibTorch platform [22], the C++ distributions of PyTorch. The im2col-based convolution on LibTorch is the baseline convolution, referred to as LibTorch. And we employ Eigen [18] and OpenBLAS [19] as high-performance matrix multiplication routine. The im2col transformation discussed in Section II-A is utilized in LibTorch, and implementation of step ① batching im2col is adapted from it inside LibTorch, minimizing the implementations difference. Note that we do not claim that it is the optimal implementation for batching im2col, and our primary focus is obtaining the column-major hybrid input matrix with the multiple matrices hybrid layout, rather than on the processes of producing it. Since the MEC [9] is not open-sourced and exhibits worse performance [13] than the convolution methods in Caffe using Eigen and OpenBLAS, it is not selected as the baseline.

B. Results on BDLO

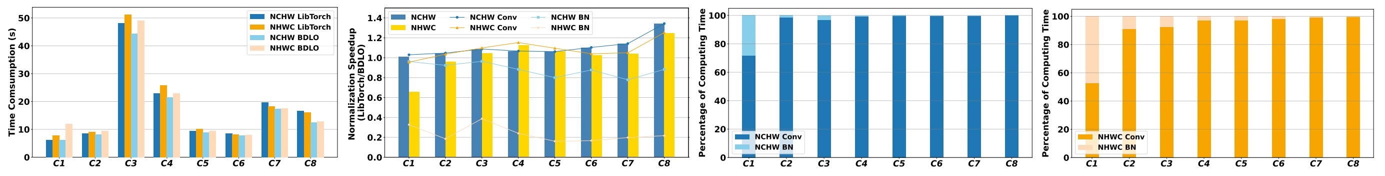
We typically conduct a convolutional layer of the benchmarks followed by either a batch normalization layer or a $(2 \times 2, S = 2)$ maxpooling layer in each test by default, as these are common operations in DNN algorithms. These are referred to as Conv+BN and Conv+Pool, respectively. For BDLO, the batch normalization layer and the maxpooling layer contain the implicit hybrid output matrix separation. To evaluate the computation efficiency, we run the Conv+BN and Conv+Pool 50 times each after warming up the CPU, and then record the runtime. We report the speedup of BDLO normalized to im2col-based convolution in LibTorch with either Eigen or OpenBLAS. The tests are performed for both the input feature maps with *NCHW* layout and *NHWC* layout.

The results of time consumption, normalization speedup and percentage of computing time are summarized in Fig. 6. The subfigures of time consumption display the overall runtime of Conv+BN and Conv+Pool for BDLO and im2col-based convolution in LibTorch, represented by the bars. Inside the subfigures of normalization speedup, we not only depict the overall normalization speedups (LibTorch/BDLO) of the Conv+BN and Conv+Pool with the bars, but also the individual normalization speedups (LibTorch/BDLO) of the convolutional layer and the next layer with the lines. And subfigures for percentage of computing time display the proportions of the convolutional layer and the next layer in the overall runtime while using BDLO.

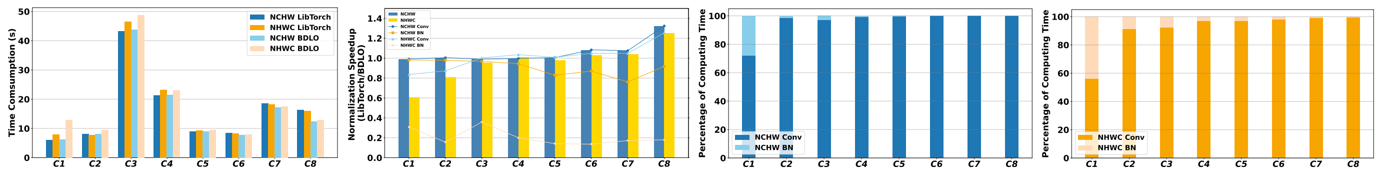
For Conv+BN on Eigen, as shown in Fig. 6(a), the overall runtime for *NCHW* layout is always lower than it for *NHWC* layout when using BDLO. In each benchmark, the overall runtime for *NCHW* layout is consistently the minimum time consumption. And we can see that BDLO achieves the best performances with approximately $1.342\times$ overall speedup (LibTorch/BDLO) for *NCHW* layout and $1.247\times$ for *NHWC* layout, both observed in $c8$. The individual normalization speedups (LibTorch/BDLO) of the convolutional layer are consistently higher than $1.0\times$ when using BDLO, setting the upper limits of the overall normalization speedups (LibTorch/BDLO). On the other hand, the individual normalization speedups (LibTorch/BDLO) of the batch normalization layer are below $1.0\times$ when using BDLO, in especially for *NHWC* layout. This is a result of the addition of the hybrid output matrix separation in the batch normalization layer. Since the batch normalization layer accounts for a relatively small portion of the overall runtime, except for $c1$, the negative impacts from this layer reduce the overall normalization speedups (LibTorch/BDLO) by approximately 1%. For Conv+BN on OpenBLAS, as shown in Fig. 6(b), BDLO achieves the best performances with approximately $1.323\times$ overall speedup (LibTorch/BDLO) for *NCHW* layout and $1.250\times$ for *NHWC* layout, both observed in $c8$. But BDLO demonstrates better computational efficiencies in $c2$, $c5$, $c6$, $c7$ and $c8$ for *NCHW* layout, and in $c4$, $c6$, $c7$ and $c8$ for *NHWC* layout, slightly worse than Conv+BN on Eigen. The discrepancies of speedups result from variations in the matrix multiplication routines used in the Eigen and OpenBLAS.

For Conv+Pool on Eigen, as shown in Fig. 6(c), the overall runtime for *NCHW* layout exhibits the minimum time consumption in most benchmarks, and BDLO achieves the best performances in $c8$ with approximately $1.348\times$ overall speedup (LibTorch/BDLO) for *NCHW* layout and $1.211\times$ for *NHWC* layout. BDLO demonstrates better computational efficiency than LibTorch except in $c1$ for *NCHW* layout, and in $c1$ and $c2$ for *NHWC* layout. The proportions of the maxpooling layer in the overall runtime are larger than those of the batch normalization layer. The higher proportions of the maxpooling layer increase the negative impacts on the overall speedups (LibTorch/BDLO), potentially leading to speedups dropping below $1.0\times$ in some cases, such as in $c1$ for *NCHW* layout and $c2$ for *NHWC* layout. For Conv+Pool on OpenBLAS, as shown in Fig. 6(d), the best performances of BDLO are approximately $1.321\times$ overall speedup (LibTorch/BDLO) for *NCHW* layout and $1.245\times$ for *NHWC* layout, both observed in $c8$. The overall speedups of Conv+Pool on OpenBLAS are inferior to those of Conv+Pool on Eigen, but BDLO still obtains the speedups higher than $1.0\times$ in $c4$, $c6$, $c7$ and $c8$.

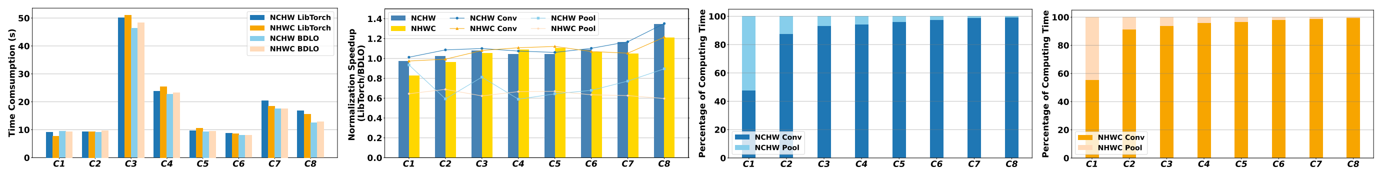
On the whole, while using BDLO, the computation efficiencies of batched convolutions for *NCHW* are higher at the most compared with that for *NHWC* and the im2col-based convolution in LibTorch, especially with the small sizes of the input feature maps. This is opposite to a common cognition that *NHWC* layout is more suitable than *NCHW* layout for the convolution computations on CPUs. On the



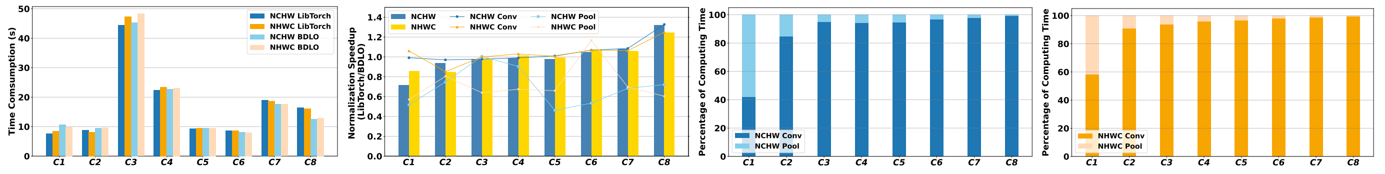
(a) Conv+BN (Eigen)



(b) Conv+BN (OpenBLAS)



(c) Conv+Pool (Eigen)



(d) Conv+Pool (OpenBLAS)

Fig. 6. The results of time consumption, normalization speedup (LibTorch/BDLO) and percentage of BDLO computing time for both the input feature maps with *NCHW* layout and *NHWC* layout with either Eigen or OpenBLAS.

other hand, *NCHW* layout is regarded as the more appropriate layout for GPUs[xx] and NPUs[xx]. Therefore, BDLO provides a technology support to avoid transforming the *NCHW* layout to the *NHWC* layout on CPUs while performing the batched im2col-based convolutions with heterogeneous multi-core collaborative processing, thus maintaining consistent data layouts and decreasing the time consumption on the data layout transformations.

C. Results on Hybrid Output Matrix Separation

The layers integrated with hybrid output matrix separation operations are more time-consuming compared to those in LibTorch. And in this section, we state that implicit hybrid output matrix separation in the next layer can effectively reduce the time consumption compared with explicitly invoking the data layout transformation function before the next layer, when the computations of the next layer are the non-cross-channel. In general, the data layout transformation function is permute function in LibTorch. We abbreviate BDLO using explicitly invoking the data layout transformation function as 'explicit separation' and that using implicit hybrid output matrix separation as 'implicit separation'.

The normalization speedups (LibTorch/BDLO) of BDLO using explicit and implicit separation are displayed in Table III(a) and (b), for input feature maps with *NCHW* and the *NHWC* layouts, employing Eigen and OpenBLAS. Additionally, Table III(a) and (b) present the differences between the normalization speedups using implicit separation and those using explicit separation. The differences exceeding 0 indicate represent that BDLO using implicit separation achieves higher computational efficiency.

For Conv+BN and Conv+Pool, as shown in Table III(a) and (b), the normalization speedups (LibTorch/BDLO) exceed $1.0\times$ in many benchmarks for both explicit separation and implicit separation. In the Conv+BN benchmarks, BDLO using implicit separation is more efficient than that using explicit separation for *NCHW* layout. For *NHWC* layout, BDLO using implicit separation achieves better efficiencies when employing OpenBLAS. The differences below 0 are only about 1%, with 21.8% for the best and 4.3% for an average, indicating that BDLO using implicit separation is more efficient than that using explicit separation at most cases for Conv+BN. Similarly, for Conv+Pool shown in Table III(b), BDLO using implicit separation is more efficient than that using explicit separation for the most Conv+Pool benchmarks, with 23.8% higher for the best and 5.4% for an average. Furthermore, focusing on BN and Pool, as depicted in Table III(c) and (d), the speedups (explicit/implicit) of BN and Pool below $1.0\times$ only exist in a few benchmarks. The best speedup (explicit/implicit) of BN is $2.406\times$ in $c7$ for *NCHW* layout on OpenBLAS, and that of Pool is $2.273\times$ in $c3$ for *NHWC* layout on OpenBLAS.

And it is noted that if the speedup (explicit/implicit) of BN is more than $1.0\times$, the differences for the speedups

TABLE III
COMPARISON BETWEEN EXPLICIT AND IMPLICIT HYBRID OUTPUT MATRIX SEPARATION

Name	*NCHW* (Eigen)			*NHWC* (Eigen)			*NCHW* (OpenBLAS)			*NHWC* (OpenBLAS)		
	Explicit	Implicit	Difference	Explicit	Implicit	Difference	Explicit	Implicit	Difference	Explicit	Implicit	Difference
$c1$	0.792	1.010	**0.218**	0.621	0.657	**0.037**	0.828	0.989	**0.160**	0.624	0.605	-0.019
$c2$	1.034	1.045	**0.011**	0.962	0.959	-0.002	0.993	1.004	**0.011**	0.789	0.811	**0.022**
$c3$	1.067	1.084	**0.017**	1.011	1.044	**0.033**	0.951	0.989	**0.038**	0.924	0.953	**0.029**
$c4$	1.065	1.066	**0.001**	1.112	1.124	**0.012**	1.002	0.997	-0.005	1.009	1.009	**0.001**
$c5$	1.037	1.060	**0.023**	1.076	1.066	-0.011	0.988	1.006	**0.018**	0.996	0.981	-0.014
$c6$	1.099	1.103	**0.004**	1.032	1.024	-0.008	1.060	1.083	**0.023**	1.032	1.032	0.000
$c7$	1.139	1.141	**0.003**	1.041	1.041	0.000	1.064	1.075	**0.011**	1.029	1.040	**0.011**
$c8$	1.346	1.342	-0.003	1.252	1.247	-0.005	1.265	1.323	**0.058**	1.229	1.250	**0.021**

(a) Conv+BN Normalization Speedup (LibTorch/BDLO)

Name	*NCHW* (Eigen)			*NHWC* (Eigen)			*NCHW* (OpenBLAS)			*NHWC* (OpenBLAS)		
	Explicit	Implicit	Difference	Explicit	Implicit	Difference	Explicit	Implicit	Difference	Explicit	Implicit	Difference
$c1$	0.743	0.863	**0.119**	0.612	0.769	**0.157**	0.862	0.844	-0.019	0.655	0.893	**0.238**
$c2$	1.013	1.013	0.000	0.897	0.932	**0.036**	0.971	0.939	-0.032	0.806	0.854	**0.047**
$c3$	1.058	1.083	**0.025**	0.986	1.026	**0.040**	0.959	0.998	**0.039**	0.908	0.985	**0.077**
$c4$	1.056	1.057	**0.001**	1.062	1.088	**0.026**	0.979	0.982	**0.003**	0.985	1.029	**0.044**
$c5$	1.054	1.053	-0.001	1.063	1.072	**0.010**	0.981	0.978	-0.003	0.975	1.000	**0.025**
$c6$	1.103	1.104	**0.001**	1.061	1.064	**0.003**	1.083	1.060	-0.024	1.054	1.067	**0.013**
$c7$	1.071	1.076	**0.005**	1.055	1.060	**0.005**	1.078	1.077	-0.001	1.054	1.067	**0.013**
$c8$	1.354	1.357	**0.003**	1.196	1.202	**0.005**	1.325	1.324	-0.001	1.242	1.247	**0.005**

(b) Conv+Pool Normalization Speedup (LibTorch/BDLO)

Name	*NCHW* (Eigen)	*NHWC* (Eigen)	*NCHW* (OpenBLAS)	*NHWC* (OpenBLAS)
$c1$	**1.917**	**1.115**	**1.751**	**1.133**
$c2$	**1.769**	0.884	**2.103**	**1.143**
$c3$	**1.861**	**1.397**	**1.883**	**1.276**
$c4$	**1.774**	**1.334**	**2.052**	**1.553**
$c5$	**1.638**	0.820	**1.776**	0.980
$c6$	**1.976**	0.717	**2.398**	0.755
$c7$	**1.931**	0.905	**2.406**	**1.124**
$c8$	**1.870**	0.991	**2.345**	0.936

(c) BN Normalization Speedup (Explicit/Implicit)

Name	*NCHW* (Eigen)	*NHWC* (Eigen)	*NCHW* (OpenBLAS)	*NHWC* (OpenBLAS)
$c1$	**1.298**	**1.528**	0.976	**1.965**
$c2$	**1.056**	**1.608**	0.800	**1.679**
$c3$	**1.446**	**1.667**	**2.017**	**2.273**
$c4$	**1.075**	**1.689**	**1.006**	**1.892**
$c5$	**1.042**	**1.412**	0.817	**1.654**
$c6$	**1.121**	**1.375**	0.859	**1.540**
$c7$	**1.462**	**1.440**	0.853	**1.582**
$c8$	**1.188**	**1.202**	0.945	**1.650**

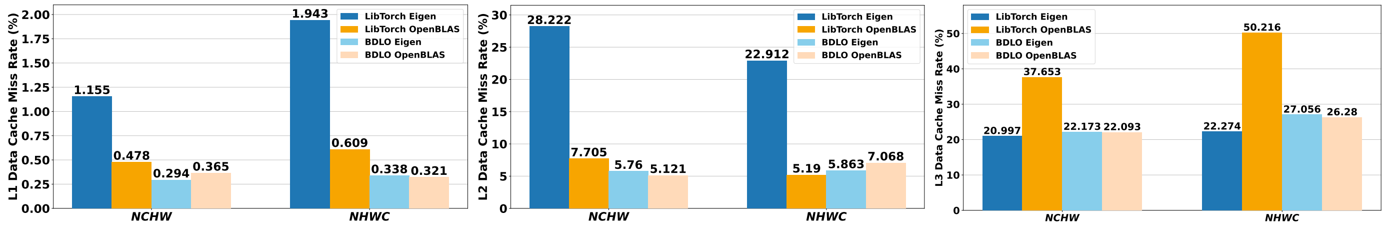
(d) Pool Normalization Speedup (Explicit/Implicit)

(LibTorch/BDLO) of Conv+BN using explicit separation and implicit separation will exceed zero with high probability, even if considering the fluctuation in time consumption. So as Conv+Pool. This can illustrate that the efficiency of implicit hybrid output matrix separation used in BDLO is better than explicitly invoking the data layout transformation function before the next layer. Implicit hybrid output matrix separation used in BDLO can be expanded to other non-cross-channel layers as well.

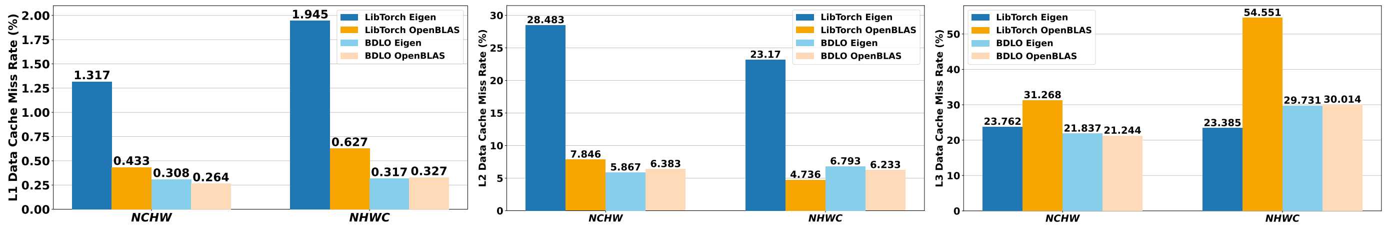
D. Cache Miss Rate

Cache miss rates can directly reflects the actual reasons for the higher efficiency of BDLO. As we discussed in Section III, the core of higher efficiency for BDLO is reducing the memory accesses by decreasing the reuse distances of weight data to zero in batched im2col-based convolutions, which can be reflected on the cache miss rates. Kunpeng 920 CPU has 3-level caches, and the L1 caches are closest to the computing cores of CPUs, followed by the L2 caches and L3 caches. CPUs will attempt to access data in the next level caches if they fail to find the required data in the last level caches. A lower L1 data cache miss rate denotes a greater probability for the computing cores accessing data in the L1 caches, and the same principle applies to L2 and L3 caches. Fig. 7 displays the L1 data, L2 data and L3 cache miss rates while performing Conv+BN and Conv+Pool with $c8$.

For L1 data cache miss rates, as shown in Fig. 7, BDLO always exhibits the lowest L1 data cache miss rates of about 0.3%. For Conv+BN, L1 data cache miss rates of LibTorch are much higher, especially on Eigen, reaching up to 1.155% for the *NCHW* and 1.943% for the *NHWC*, which are $2.928\times$ and $4.749\times$ higher than those of BDLO, respectively. Similarly, L1 data cache miss rates of LibTorch are $3.276\times$ and $5.136\times$ higher than those of BDLO for the *NCHW* and *NHWC*, respectively. On OpenBLAS, L1 data cache miss rates of BDLO are approximately $0.478\times$ lower than those of LibTorch. And the L2 data cache miss rates of BDLO are approximately 6%, which is similar to those of LibTorch on OpenBLAS, while those of LibTorch on Eigen can be as high as 28.222%. The values of L1 and L2 data cache miss rates prove that multiple matrices hybrid layout indeed enhances the access efficiency for the weight data by eliminating the reuse distances in batched im2col-based convolutions, as elaborated in Section III.



(a) Conv+BN



(b) Conv+Pool

Fig. 7. L1 data cache miss rates, L2 data cache miss rates and L3 cache miss rates (%) of $c8$ for LibTorch and BDLO

BDLO and LibTorch both exhibit the L3 cache miss rates exceeding 20%. Particularly, the L3 cache miss rates of LibTorch on OpenBLAS are more than 30%, with a maximum of 54.551% for *NHWC* layout. Combining with L1 and L2 data cache miss rates of BDLO, they demonstrate that BDLO reduces the memory accesses, thus improving the computation efficiency of batched im2col-based convolutions.

Furthermore, we perceive the access characteristics of Eigen and OpenBLAS for the batched im2col-based convolutions from the Fig. 7. Eigen shows much higher L1 and L2 data cache miss rates compared to OpenBLAS, while the trend is opposite for L3 cache miss rates.

VI. Conclusion

In this article, we propose a novel and efficient batched im2col-based convolution method on CPUs, termed Batched Data Layout-Optimization for Batched Im2col-based Convolution Method (BDLO). In the existing im2col-based convolutions for the batching processes, We observe that the weight matrix must be repeatedly accessed in each GEMM routine, generating large reuse distances and resulting in increased memory accesses. Therefore, we optimize the data layout for batched input feature maps to eliminate the reuse distances of weight data in the processes of GEMM. To the best of our knowledge, this is the first work to improve the computation efficiency of batched convolutions on CPUs by eliminating the reuse distances of weight data for reducing the memory accesses. BDLO converts N batched input feature maps into one hybrid input matrix and invokes the GEMM routine only once for the batched convolutions, producing a hybrid output matrix with *CNHW* layout. For efficient data layout transformations, BDLO integrates the hybrid output matrices separation operations into the computation processes of the next layer, lowing the latency of these separation operations, if the computations of the next layer are the non-cross-channel computations.

BDLO is tested on many benchmarks from the frequently-used DNN algorithms with different data layouts *NCHW* and *NHWC*, using Kunpeng 920 CPU. Compared to the im2col-based convolution method used in LibTorch on Eigen and OpenBLAS, the experimental results show that the computation efficiencies of BDLO are higher at the most benchmarks, especially with the small sizes of the input feature maps with *NCHW* layout. BDLO achieves the best performance with approximately $1.348\times$ speedup for *NCHW*. Hence, BDLO provides a technology support to avoid transforming the *NCHW* to the *NHWC* layout on CPUs while performing the batched im2col-based convolutions with heterogeneous multi-core collaborative processing.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Red Hook, NY, USA: Curran Associates, 2012, pp. 1097–1105.
[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–14.
[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[4] W. Liu et al., "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2016, pp. 21–37.
[5] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*. [Online]. Available: http://arxiv.org/abs/1804.02767
[6] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards realtime object detection with region proposal networks," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 1. Cambridge, MA, USA: MIT Press, 2015, pp. 91–99.
[7] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 3431–3440.
[8] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid scene parsing network," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 2881–2890.
[9] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, 2017, pp. 815–824.
[10] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on CPUs," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 1025–1040.

[11] S. Heidari, M. Ghasemi, Y. G. Kim, C. J. Wu, and S. Vrudhula (2022), ”CAMDNN: Content-Aware Mapping of a Network of Deep Neural Networks on Edge MPSoCs,” IEEE Trans. on Computers, 71(12), 3191-3202.

[12] Korostelev, I., L. De Carvalho, J. P., Moreira, J., and Amaral, J. N. (2023). YaConv: Convolution with low cache footprint. ACM Transactions on Architecture and Code Optimization, 20(1), 1-18.

[13] Zhao, T., Hu, Q., He, X., Xu, W., Wang, J., Leng, C., and Cheng, J. (2021). ECBC: Efficient convolution via blocked columnizing. IEEE Transactions on Neural Networks and Learning Systems.

[14] Zhang, Y., Wang, Y., Mo, Z., Zhou, Y., Sun, T., Xu, G., ... and Yang, L. (2022, December). Accelerating small matrix multiplications by adaptive batching strategy on GPU. In 2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys) (pp. 882-887). IEEE.

[15] Y. Choi, Y. Kim and M. Rhu, Lazy batching: An sla-aware batching system for cloud machine learning inference[C]. 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Korea (South), 2021, 493-506. https://doi.org/10.1109/HPCA51647.2021.00049.

[16] Pytorch, Pytorch chinese tutorial & documentation. 2019. Available online. https://pytorch.apachecn.org/#/.

[17] M. Abadi et al., “Tensorflow: A system for large-scale machine learning,” in Proc. 12th USENIX Symp. Operating Syst. Design Implement. (OSDI), 2016, pp. 265–283.

[18] G. Guennebaud et al. (2010). Eigen V3. [Online]. Available: http://eigen.tuxfamily.org

[19] X. Y. Zhang, Q. Wang, W. Saar, et al. Openblas: An optimized blas library. In Texas Advanced Computing Center. 2016. Available online. http://www.openblas.net/.

[20] T. Chen et al., “MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems,” in Proc. Neural Inf. Process. Syst. Workshop Mach. Learn. Syst., 2015, pp. 1–6.

[21] Xia, J., Cheng, C., Zhou, X., Hu, Y., and Chun, P. (2021). Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services. IEEE Micro, 41(5), 67-75.

[22] LibTorch. https://pytorch.org/cppdocs/installing.html

[23] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L. C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4510-4520).

[24] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826).