Parallel Computing 100 (2020) 102701

Contents lists available at ScienceDirect

Parallel Computing

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

# Exploring GPU acceleration of Deep Neural Networks using Block Circulant Matrices

Shi Dong, Pu Zhao, Xue Lin, David Kaeli *

*Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, 02115, USA*

ARTICLE INFO

*Keywords:*
Block Circulant Matrix
Deep Neural Network
GPU

ABSTRACT

Training a Deep Neural Network (DNN) is a significant computing task since it places high demands on computing resources and memory bandwidth. Many approaches have been proposed to compress the network, while maintaining high model accuracy, reducing the computational demands associated with large-scale DNN training. One attractive approach is to leverage Block Circulant Matrices (BCM), compressing the linear transformation layers, e.g., convolutional and fully-connected layers, that heavily rely on performing General Matrix Multiplications (GEMM). By using BCMs, we can reduce the weight storage for a linear transformation layer from $O(N^2)$ to $O(N)$. BCMs are also more efficient in terms of computational complexity, improving algorithmic complexity from $O(N^2)$ to $O(N\log(N))$.

Previous work has only evaluated DNNs using BCMs targeting FPGAs for inference. There has been little prior work that considers the potential benefits of using BCMs for accelerating DNN training on GPUs. In this paper, we explore acceleration of DNNs using BCM on a state-of-the-art GPU. First, we identify the challenges posed by using BCMs. Next, we perform both general and GPU-specific optimizations that impact: (i) the decomposition and interaction of individual operations, and (ii) the overall GPU kernel design. We modify the algorithmic steps to remove redundant computations, while maintaining mathematical integrity. We also leverage multiple GPU kernel optimizations, considering performance factors, such as occupancy, data sharing/reuse patterns, and memory coalescing. We evaluate the performance of DNN training on an NVIDIA Tesla V100, providing insights into the benefits of our proposed kernel optimizations on a state-of-the-art GPU. Based on our results, we can achieve average speedups of 1.31× and 2.79× for the convolutional layers and fully-connected layers, respectively for AlexNet. We can also achieve average speedups of 1.33× and 3.66× for the convolutional layers and fully-connected layers, respectively for VGGNet-16.

## 1. Introduction

Deep Neural Networks (DNNs) have been found to deliver remarkable inference accuracy in many application domains, including computer vision, speech recognition, and natural language processing [1]. DNNs have been widely deployed to support a broad range of AI applications. Emerging platforms, such as self-driving vehicles [2] and virtual assistants [3], have successfully exploited DNNs to change our daily lives. The fundamental building blocks of DNN are Artificial Neural Networks (ANNs), first introduced around 1960 [4]. Their recent popularity is a product of advances in parallel computing hardware (i.e., GPUs). These platforms can provide significant speedups for matrix-based computations, which dominate execution in DNNs [5]. However, the current trend in DNN models (e.g., VGG [6], Googlenet [7], and Resnet [8]) is to keep adding additional parameters and layers, deepening, and widening the model to achieve better accuracy. As a result, the current computing platforms are facing growing demands on computing resources, memory and interconnect bandwidth, and storage resources.

Researchers have been pursuing new methods to improve the efficiency of DNN execution to overcome these challenges. Compressing the network model is one popular method. One way to compress the model is to use structured weight matrices, such as *circulant matrices* [9,10]. FPGAs and ASICs have been the primary target in these studies [9–12]. The main motivation for using structured matrices has been to improve energy efficiency, while also reducing execution time (especially in embedded applications for IoT products) during inference. However, training is also an integral part of deep learning. Structured matrices based weight compression techniques can benefit training as well, reducing the number of computations and the storage

* Correspondence to: 409 Dana Research Center, Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, 02115, USA.
*E-mail addresses:* dong.sh@northeastern.edu (S. Dong), zhao.pu@northeastern.edu (P. Zhao), xuelin@ece.neu.edu (X. Lin), kaeli@ece.neu.edu (D. Kaeli).

https://doi.org/10.1016/j.parco.2020.102701
Received 11 July 2020; Received in revised form 6 October 2020; Accepted 13 October 2020
Available online 16 October 2020

space for weights. FPGAs/ASICs are limited in their ability to be used effectively during DNN training.

GPUs, on the other hand, are a natural fit for accelerating the training of DNNs, especially when working with tremendous amounts of data (e.g., image classification with data from ImageNet [13]). GPUs have a large number of parallel arithmetic units, allowing thousands of threads to be launched concurrently, while also hiding memory access latency. For this reason, GPUs have been widely deployed to perform training of DNNs. In some settings, DNN training leverages multiple GPUs in a distributed manner [14]. However, naively applying a weight compression technique to DNN training on a GPU can be highly inefficient, even causing a significant slowdown.

In this paper, we explore DNN training acceleration on a GPU, using Block Circulant Matrices (BCMs) [15] for weight compression. BCMs can be used to replace the weight matrices in layers leveraging GEMM (e.g., convolutional and fully-connected layers). As a result, we replace GEMM computation with the BCM algorithm.

A BCM consists of many square circulant matrix blocks, as shown in Fig. 1. The block size determines the compression ratio, which presents a tradeoff between the performance and model accuracy when approximating the desired matrices. The circulant matrices enable the transformation of regular matrix multiplications to *circulant convolutions*, reducing computing complexity by employing Fast Fourier Transforms (FFTs) and Inverse Fast Fourier Transforms (IFFTs).

To study the BCM algorithm for DNN training on a GPU, we first analyze the algorithmic steps in the BCM algorithm. We characterize three different scenarios, considering forward and backward propagation. We analyze the algorithmic complexity of each stage for the different scenarios, highlighting some of the challenges when leveraging BCMs on a GPU.

The contributions of this paper include:

- we extend the BCM algorithm for batched DNN training on a GPU.
- We propose a set of optimizations for BCM algorithm for DNN training, reducing the number of redundant computations during both forward and backward propagation, taking advantage of the mathematical properties of an FFT/IFFT, and modifying selected steps in the algorithm.
- We discuss a series of GPU kernel tuning principles, considering three factors that influence performance: (i) device occupancy, (ii) data reuse, and (iii) memory coalescing for both reads and writes. We define two new metrics that can characterize data reuse and memory coalescing patterns present in BCM-based DNNs. Equipped with these metrics, as well a measure of device occupancy, we can guide our optimizations to accelerate DNN execution effectively.
- We evaluate the execution of both forward and backward propagation on an NVIDIA Tesla V100, employing performance counters to profile the resulting execution. We report on the instruction per cycle (IPC), static/dynamic instruction count, the memory intensity (the total number of memory transactions issued across all GPU kernels), and device occupancy.
- We show that by applying our optimizations with BCM, we can achieve an average speedup of 1.31× and 2.79× for the convolutional layers and fully-connected layers of AlexNet, respectively, and a speedup of 1.33× and 3.66× for the convolutional layers and fully-connected layers of VGGNet-16, respectively. When we leverage BCM during the DNN training process, the model experiences a minimal loss in terms of both convergence rate and inference accuracy. We also show that there can be significant benefits when using BCM to reduce the amount of communication required during a multi-GPU training using two NVIDIA Tesla V100's.

Our proposed methods can benefit all types of DNNs leveraging GEMM. The computation involved in the fully-connected layer is a GEMM operation. The convolutional layer can use GEMM with an additional im2col operation [10]. The computations in an LSTM unit are also GEMM operations. In practice, our proposed methods can significantly reduce the time needed to deploy BCM-based DNNs on edge devices for inference. Previous studies have described the potential of BCM-based DNNs in terms of both performance and energy efficiency on edge devices, with only a minor loss in inference accuracy [9,10,16].

To the best of our knowledge, this is the first work that explores DNN training using the BCM algorithm on a GPU. Note that our goal is not to replace GEMM-based DNNs with BCM-based DNNs completely. Our proposed method serves as an option for DNN training, providing significant performance and weight storage benefits with only a minor loss in the training convergence rate and inference accuracy.

We organize the rest of this paper as follows. In Section 2, we provide an overview of the BCM algorithm and review the targeted GPU architecture and programming model. In Sections 3 and 4, we describe our proposed optimizations and kernel tuning principles used to optimize the BCM algorithm. In Sections 5 and 6, we present our test environment, evaluation results, and reflect on the benefits of our proposed optimizations. In Section 7, we discuss previous studies on weight compression methods, as well as the acceleration of DNN execution on many hardware platforms (including GPUs), and in Section 8, we conclude the paper and outline plans for future work.

## 2. Background

### 2.1. Deep neural networks

A deep neural network consists of different types of layers, used to extract features at different levels of abstraction. GEMM operations play a fundamental role in the linear transformation layers of a DNN (i.e., the fully-connected (FC) and convolutional layers). For example, the FC layer is modeled by connecting each input $X_j$ with numerous artificial neurons [17]. Each connection has an associated learnable weight $W_j$. The $ith$ output is computed as follows:

$$a_i = \sum_{j=1}^{n} W_j * X_j + bias_i \tag{1}$$

where $bias_i$ stands for a learnable bias.

From the above equation, the computation of the FC layer is a matrix–vector multiplication, and it becomes GEMM with multiple sets of inputs. The convolutional layer can also use GEMM after transforming the inputs using an *im2col* operation [18]. The LSTM unit consists of multiple GEMM operations [19]. Other than the linear transformation layers, a mature DNN model may also include layers such as pooling, normalization, activation, and softmax. Many representative DNNs are basically built upon multiple linear transformation layers stacked together with other types of layers. In this paper, we focus on investigating the benefits of using BCMs for the weight matrices in the linear transformation layers, replacing GEMM with the BCM algorithm.

Training deep neural networks is an iterative and time consuming task. The ultimate goal is to learn a set of weights for the neural network model so that it can recognize images, sounds or texts with high accuracy. The detailed mechanism for training can be further divided into a *forward* and a *backward propagation*. The purpose for performing forward propagation during training is to calculate the loss of the overall network based on the current weights. The goal of backward propagation is to compute the derivatives of the loss function with respect to the weights, which are then used to update the weights.

### 2.2. Block circulant weight representation

It has been formally proven that some structured matrices have the universal approximation property [20,21]. Among them, circulant matrices can be used in constructing neural networks given their properties to reduce both the amount of storage and computing complexity through FFTs/IFFTs [15]. Block circulant weight matrices have been

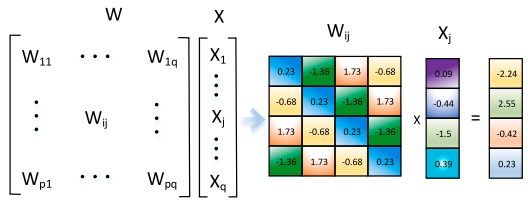


**Fig. 1.** An example of a Block Circulant Matrix (BCM).

proposed [9,10,16] to address the two limitations of a circulant matrix: (1) it must be square, and (2) the compression ratio is invariant, resulting in a high degree of information loss for large matrices.

A BCM is composed of an array of equally-sized square circulant matrices (i.e., blocks), whose size is configurable. This method carefully considers the tradeoff between compression ratio and model performance by selecting the best block size (i.e., a larger block size corresponds to a larger compression ratio and improved computation speed, though a smaller block size corresponds to a higher accuracy).

The convention to describe a BCM is as follows: $\mathbf{W}$ represents the block circulant weight matrix, $\mathbf{X}$ the input, and $\mathbf{a}$ the output. Within $\mathbf{W}$, $p$ denotes the number of blocks row-wise, $q$ the number of blocks column-wise, and $k$ the block size, which is the number of free elements in one circulant matrix block (i.e., $\mathbf{W}_{ij}$, with a total of $k^2$ elements). A block size of $k$ represents the compression ratio for the weight matrix. Weight compression using BCM leads to information loss in the weights. Therefore, a larger block size results a more loss in terms of inference accuracy. Fig. 1 shows an example of $\mathbf{W} * \mathbf{X}$, a circulant matrix block $\mathbf{W}_{ij}$, and one corresponding input vector $\mathbf{X}_j$, in which $i = \{1...p\}$ and $j = \{1...q\}$. Using a BCM, the number of weights needed for each linear transformation layer reduces from $p * q * k^2$ to $p * q * k$.

#### 2.2.1. Training with BCM algorithm

As mentioned in Section 2.1, the training process involves forward and backward propagations. The computation of the forward propagation is $\mathbf{W} * \mathbf{X}$. As the weight matrix is now block circulant, the computation involves two steps that reduce the overall computational complexity. First, within each block, GEMMs are transformed into an "FFT → element-wise multiplication → IFFT", according to the *circulant convolution theorem* [22,23]. Second, the results over index $j$ can be summed to produce the final result.

Backward propagation involves two operations both using $\frac{\partial \mathbf{L}}{\partial \mathbf{a}}$. The first computes the derivative for the inputs $\frac{\partial \mathbf{L}}{\partial \mathbf{X}}$ and the second operation computes the derivative for the weights $\frac{\partial \mathbf{L}}{\partial \mathbf{W}}$.

In the context of the BCM algorithm, $\frac{\partial \mathbf{a}}{\partial \mathbf{W}}$ and $\frac{\partial \mathbf{a}}{\partial \mathbf{X}}$ are both BCMs [15]. Within the scope of each block, $\frac{\partial \mathbf{a}_i}{\partial \mathbf{W}_{ij}}$ is a circulant matrix defined by vector $\mathbf{X'}_j : \{X_{j1}, X_{jk}, X_{j,k-1}, \ldots, X_{j2}\}$ and $\frac{\partial \mathbf{a}_i}{\partial \mathbf{X}_j}$ is a circulant matrix defined by $\mathbf{w'}_{ij} : \{w_{ij1}, w_{ijk}, w_{ij,k-1}, \ldots, w_{ij2}\}$.

Usually, every training iteration should consume one batch of input data, producing one batch of outputs. In this paper, we consider training in batches and develop a set of equations to describe batched DNN training. To characterize the underlying data format, we use index $b$ to indicate the position of data within each batch and $n$ to denote the number of input data elements in one batch. Equations for the forward and backward propagations are shown below:

$$\mathbf{a}_{bi} = \sum_{j=1}^{q} IFFT(FFT(\mathbf{w}_{ij}) \circ FFT(\mathbf{X}_{bj})) \tag{2}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}_{ij}} = \sum_{b=1}^{n} IFFT(FFT(\frac{\partial \mathbf{L}}{\partial \mathbf{a}_{bi}}) \circ FFT(\mathbf{X'}_{bj})) \tag{3}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{X}_{bj}} = \sum_{i=1}^{p} IFFT(FFT(\frac{\partial \mathbf{L}}{\partial \mathbf{a}_{bi}}) \circ FFT(\mathbf{w'}_{ij})) \tag{4}$$

For simplicity, we set the row height and column width of the weight matrices to power of 2. If dimensions of the linear transformation layers are not power of 2, zero padding can be used both on the feature maps and weight matrices. Note that zero padding does not effect computation complexity and is used for regularizing the formats only.

### 2.3. GPU architecture

The architecture of a Graphic Processing Unit (GPU) is designed to improve instruction throughput rather than reduce the latency of a single instruction. The memory subsystem of a GPU is optimized to deliver high memory bandwidth, providing shared memory and memory coalescing units to support GPU memory demands.

In order to support general-purpose GPU programming, GPU vendors have developed a number of software platforms, along with supporting libraries and programming interfaces. Among the many options, CUDA is aimed for programming NVIDIA GPUs. The programming model of CUDA provides an intuitive abstraction of the GPU compute resources for the programmer. Multiple threads are grouped together into thread blocks, and multiple thread blocks are combined together into a grid. Programmers can define the shape of a thread block and a grid using two CUDA-specific identifiers, *BlockDim* and *GridDim*. In addition, *threadIdx* and *blockIdx* (also CUDA-specific identifiers) are used to specify the thread ID and block ID in a multi-dimensional fashion. For example, threadIdx.x represents the thread ID for dimension x, threadIdx.y for y, and threadIdx.z for z. When developing a CUDA kernel, the programmer needs to carefully map the layout of the data in their kernel to a group of threads, specifying the kernel dimension.

## 3. Implementing BCM based DNN training on a GPU

Next, we take a deep dive into the operations performed when training a linear transformation layer using BCMs. First, we consider how to map the BCM algorithm when targeting a GPU. Then, we consider some of the performance challenges present in our baseline GPU implementation.

### 3.1. Decomposing forward and backward propagation

Considering Eqs. (2), (3), and (4), the computations associated with forward and backward propagation can be further broken down into multiple stages. Fig. 2(a) shows the stages required during forward and backward propagations, where each block represents the operations in the equations and the arrows represent data dependencies between two blocks. For instance, the *multiply* block depends on the results computed by the two FFTs blocks. Whenever there is a dependency, synchronization between the two operations is required.

#### 3.1.1. Managing data

All the data, including the intermediate results from each stage, are stored as tensors (multi-dimensional arrays) in GPU memory. The data layout in our baseline implementation uses a conventional tensor data layout. A tensor has $N$ dimensions, denoted as $D_i$, where $i$ can take on values $0, \ldots, N-1$. Each dimension has an associated size value $M_i$, specifying the number of data chunks for the specific dimension. For example, for a tensor-based data that has dimensions of $n \times p \times q \times k$, we can know that this tensor has four dimensions ($D_3$, $D_2$, $D_1$, and $D_0$), each of which has an associated size value of ($M_3$, $M_2$, $M_1$, and $M_0$), corresponding to $n$, $p$, $q$, and $k$, respectively.

The format of each input and output involved in the BCM algorithm is indicated in Table 1. The values $n$, $p$, $q$ and $k$ are described in Section 2.

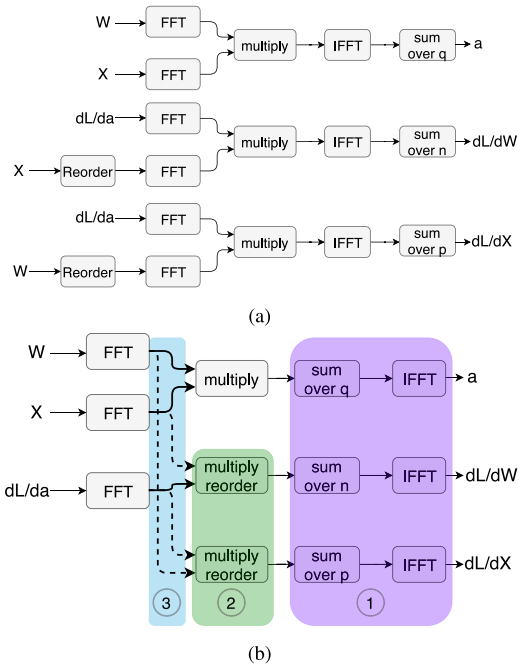


**Fig. 2.** (a) The flow of operations performed during a complete training. (b) An optimized implementation of the same operations.

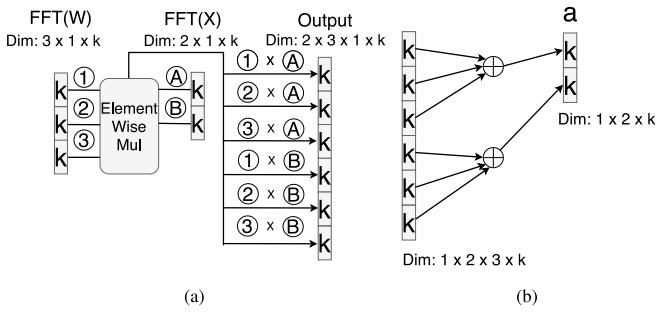


**Fig. 3.** (a) Operational overview of element-wise multiplications performed during forward propagation. (b) Reduction summations performed during forward propagation.

**Table 1**
Dimensions of the input and output data.

| Type | Name | Format |
|---|---|---|
| Inputs, Outputs | $W$, $\frac{\partial L}{\partial W}$ | $p \times q \times k$ |
| | $X$, $\frac{\partial L}{\partial X}$ | $n \times q \times k$ |
| | $a$, $\frac{\partial L}{\partial a}$ | $n \times p \times k$ |

### 3.2. Kernels of the operations

Inspecting the algorithm decomposition in the forward and backward propagation, we see four primary operations: (1) FFTs, (2) IFFTs, (3) element-wise multiplications, and (4) summations. The reorder kernel is already efficient, so we exclude it from this discussion.

To compute the FFT/IFFT efficiently, we select the cuFFT [24] library, a highly optimized FFT/IFFT library specifically designed for NVIDIA GPUs. In our implementation, we use the 1D FFT/IFFT, launched in batches over the inner-most dimension $D_0$.

Figs. 3(a) and 3(b) show two simple examples illustrating the operations involved in the element-wise multiplications and summations during forward propagation, together with the indexing pattern associated with the memory layout. We omit the two operations performed during backward propagation as they share a similar pattern. The major differences are: (1) the format of the input, and (2) the locations read from the inputs.

**Table 2**
Kernel dimensions for element-wise multiplications and summations (Forward Propagation).

| Kernels | Dimension variables | Forward |
|---|---|---|
| Multiplication | blockDim | $(k, 1, 1)$ |
| | gridDim | $(q, p, n)$ |
| Summation | blockDim | $(k, 1, 1)$ |
| | gridDim | $(p, n, 1)$ |

Fig. 3(a) illustrates the operational details for the element-wise multiplications during forward propagation. From the figure, we can see that element-wise multiplications are performed over $k$-sized data chunks. The chunks are read from $FFT(W)$ and $FFT(X)$, with shapes of $3\times1\times k$ and $2\times1\times k$, respectively. In the example presented in the figure, the labels ①–③ correspond to 3 different k-sized data chunks read from $FFT(W)$, and the labels Ⓐ–Ⓑ stand for the 2 different k-sized data chunks read from $FFT(X)$. Data chunks are read for element-wise multiplication on a k-sized chunk basis. The labels on the write path (e.g., ① × Ⓐ) identify the results from the element-wise multiplications (for data chunks ① and Ⓐ in this example). Likewise, the other labels on the write path are the result of element-wise multiplications for their respective data chunks.

Fig. 3(b) shows the reductions (i.e., summations) performed during forward propagation. Given the input, which is the output shown in Fig. 3(a), the summation operations are performed over different dimensions, $q$ for the forward propagation, $n$ for the backward propagation (for the weights), and $p$ for the backward propagation (for the data).

For the element-wise multiplications and summations, we design and implement GPU kernel functions. The basic elements required for a GPU kernel include: (i) mapping the data layout to the kernel dimensions (i.e., the shape of the cooperative thread array, defined by blockDim and gridDim [25]), (ii) specifying the operations to be performed in the kernel, and (iii) managing the kernel input data, stored in GPU memory while the kernel is active.

According to the data layout in the BCM algorithm, the data unit involved in the computations is a $k$-sized data chunk (i.e., the size of the inner-most dimension). Therefore, the most straightforward scheme would be to directly map both element-wise multiplications and summations to a space where the kernel parameters (i.e., blockDim.x, gridDim.x, gridDim.y, and gridDim.z) are determined by the tensor data dimensions. With this mapping mechanism, each thread block has $k$ threads simultaneously running on a GPU. Given this mapping philosophy, the kernel becomes significantly simplified. Table 2 shows the mapping details.

### 3.3. Challenges

First, our flow of operator shows that there are at least five operators for both forward and backward propagation. When executing this multi-stage application on a GPU, we need to launch multiple kernels sequentially due to the constraints of the dependencies between operators, adding the overhead of kernel launch and associated synchronizations to the overall execution time. As such, the benefits of parallelism on a GPU are limited to each kernel.

Second, the kernels executed after FFT and before IFFT, in the BCM operator sequence, suffer from an occupancy issue. Given that we are using the NVIDIA cuFFT library kernels for FFT/IFFT execution, the size of $k$ (the thread block size after kernel dimension mapping) changes to $k/2+1$ after the FFT, due to a memory-saving feature of the cuFFT library [24]. As a result, the thread block size is not always a multiple

of 32 (warp size). This new value for $k$, denoted as $k'$ in this paper, results in lower occupancy and underutilization of the device.

Last but not least, a regular kernel mapping function cannot generally employ the full capabilities of a GPU, primarily due to inefficient use of hardware components, such as memory coalescing units and shared memory. The former coalesces multiple memory transactions issued from the same thread block, and the latter enables fast access to data that is frequently reused by multiple threads in a thread block [25]. There are many opportunities to optimize the two customized kernel. First, the nature of the two kernels results in a complicated memory access pattern (due to tensor data indexing). As such, optimizing memory coalescing efficiency can lead to performance improvements. Second, our implementation produces some predictable (and cacheable) data reuse patterns. For example, in Fig. 3(a), Ⓐ and Ⓑ are reused when writing the results.

## 4. Optimizations

### 4.1. Improving the flow of operations

In this section, we describe the set of optimizations we develop/apply to reduce the number of redundant operations in the BCM algorithm. Our optimizations preserve the mathematical integrity of all steps. As shown in Fig. 2(b), our optimizations consist of 3 enhancements identified in purple, green and blue, and numbered 1, 2, and 3, respectively. These enhancements include: (1) swapping the position of the IFFT and summation operations, (2) removing the *reorder* operation and modifying the associated *multiplication* to guarantee mathematical correctness, and (3) reusing results after computing the FFT for backward propagations for weights and data. In the following paragraphs, we provide details of these steps and show how their use can simplify computations.

Step 1 does not impact the mathematical integrity of the computation, given that an IFFT is a linear transformation. The IFFT can be performed on A and B as individual components or combined in a sum, given that $IFFT(A+B) = IFFT(A)+IFFT(B)$. Therefore, we can swap the position of the *sum* and *IFFT* in the BCM algorithm. The benefit of these simple changes results in a significant reduction in terms of the number of computations needed for computing the IFFT. Similarly, the same optimizations can be applied to the backward propagation as well.

For step 2, the $\mathbf{w}'$ and $\mathbf{X}'$ are in fact the same vectors for $\mathbf{w}$ and $\mathbf{X}$, but shifted by $k-1$ elements. Based on the fundamental characteristics of an FFT [26], the magnitude of $FFT(\mathbf{w}')$ and the magnitude of $FFT(\mathbf{w})$ are the same, with only the sign of the imaginary part flipped. Based on this property, we can remove the *reorder* operation entirely and only modify the associated element-wise multiplications, considering the flipped sign in the imaginary part of the $FFT(\mathbf{w})$ and $FFT(\mathbf{X})$. Applying step 2 leads directly to step 3. Removing the *reorder* operation adds some redundancy that we can exploit for further optimizations, specifically during backward propagation. In the blue shaded area ③ of Fig. 2(b), the data reuse paths are indicated using dashed arrows.

For the computation of $FFT(\mathbf{X})$ and the $FFT(\mathbf{w})$ during backward propagation, we can reuse the corresponding results produced during forward propagation. In addition, the results from the $FFT(\frac{\partial \mathbf{L}}{\partial \mathbf{a}})$ computation can also be reused during backward propagation. In all, we can eliminate two *reorder* operations and three major FFT operations during backward propagations, significantly reducing the number of operations.

### 4.2. Kernel customizations

In this section, we present a number of kernel-specific optimizations for the two customized kernels, based on their execution performance characteristics. To guide these optimizations, we consider three different performance-related factors: (1) occupancy, (2) data reuse pattern, and (2) memory coalescing for both reads and writes. We develop a set of principles that are focused on these three factors. We provide a brief description of each of them.

- Principle I: To achieve high occupancy, the thread block size should be a multiple of 32 (the warp size).
- Principle II: Each thread block should include threads that exhibit temporal locality in their data access pattern.
- Principle III: Given a thread block design based on principles I and II, the memory access distance (defined later) within one thread block exploits memory coalescing.

Given the lack of control over thread scheduling on the actual hardware, tuning the ***mapping function*** (Section 3.2) and ***reorganizing the memory layout*** are the keys to applying our principles. To quantitatively evaluate the benefits of principles II and III, we define two metrics: (i) the degree of data reuse (DR), and (ii) the memory access distance (MD) for both reads and writes. Both metrics are measured within a single thread block. We define the DR and MD metric as below:

- ***DR***: The maximum number of threads that reuse the same data.
- ***MD***: The range of addresses accessed across all memory accesses, divided by the number of threads per thread block.

The DR metric reflects the data reuse pattern in one thread block. The MD metric is used to assess spatial locality. An ideal MD value would equal to 1 storage unit (a float or double), meaning that the memory accesses within one thread block are entirely regular and consecutive, leading to an ideal spatial locality. Any MD value larger than 1 storage unit implies some degree of irregular access. A more significant value for MD translates to a lower spatial locality. In this paper, we focus on reducing the MD value by reorganizing the memory layout. Both metrics are a function of the kernel design. We can easily obtain DR and MD for a given mapping function and memory layout of the input and output data. Note that tuning the DR and MD metrics is a tradeoff, in that increasing DR can increase MD while introducing irregular memory accesses.

#### 4.2.1. Element-wise multiplications

The element-wise multiplication kernel suffers from occupancy issues, such that data reuse and memory coalescing are heavily underexploited. To address these issues, we propose two optimizations (labeled O1 and O2), which both consider principles I and II, but in different priority orders. We also transform the memory layout to decrease MD, as is the goal with principle III.

**The O1** optimization is aimed at addressing the occupancy issue first, then additionally leveraging the data reuse pattern to support better shared memory usage. We achieve this by directly increasing the size of a thread block, including more threads from a second dimension rather than the innermost dimension. More specifically, with the mapping functions indicated in Table 2, given a multiple of 32 for $N$, we can change the mapping of gridDim.x to $(q-N+1)/N$, while keeping the gridDim.y and gridDim.z dimensions unchanged. Fig. 4 illustrates this change in the kernel dimension mapping. However, this can produce an undesirable MD value during backward propagation, resulting in long-strided memory accesses because the two inputs for the multiplication do not share the same innermost dimensions. To address this issue, we reorganize the memory layout, leading to a better chance of memory coalescing. For example, during backward propagation for weights, the dimensions for the two inputs for the multiplication are $n \times p \times k$ and $n \times q \times k$. We reorganize them so that their dimensions become $p \times n \times k$ and $q \times n \times k$.

Likewise, the memory layout of their outputs need to be reorganized as well, in order to avoid large MD values for the writes.

To leverage any temporal reuse pattern in the data, we manipulate the mapping function as well. We keep the blockDim.x unchanged as $N$, and map DR to the blockDim.y dimension, increasing the number of threads that share data. When mapping the DR to BlockDim.y, we specify the number of lower dimension chunks that can reuse one chunk of data. As a result, a number of lower dimension chunks that reuse one chunk of data can be loaded into a single thread block. For

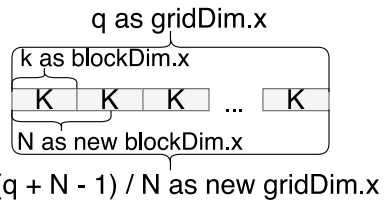


**Fig. 4.** Kernel dimension mapping with a new thread block size.

**Algorithm 1** Kernel Mapping Function of O1

**Inputs:** n, p, q, k, $FFT(\frac{\partial L}{\partial a})$, $FFT(W)$, $FFT(X)$, propagation_type
**Outputs:** blockDim, gridDim
**if** propagation_type == BACKWARD_WEIGHT **then**
    MemoryReorganize($FFT(\frac{\partial L}{\partial a})$) $\rightarrow p \times n \times k$
    MemoryReorganize($FFT(X)$) $\rightarrow q \times n \times k$
    $M\_1, M\_2, M\_3$ = n, q, p
**else if** propagation_type == BACKWARD_DATA **then**
    MemoryReorganize($FFT(W)$) $\rightarrow q \times p \times k$
    $M\_1, M\_2, M\_3$ = p, q, n
**else if** propagation_type == FORWARD **then**
    $M\_1, M\_2, M\_3$ = q, p, n
**end if**
N = k < 32 ? 32 : k
blockDim.x = N
gridDim.x = $(M\_1 - N + 1)/N$
DR = $\frac{1024}{blockDim.x}$
blockDim.y = DR
gridDim.y = $\frac{M\_2 - DR + 1}{DR}$
gridDim.z = $M\_3$

example, chunk A and chunk B reuse the data of chunk C, so we set the BlockDim.y to 2 so one thread block can load both chunk A and chunk B. Threads reading data from chunk A and chunk B can read data from chunk C, stored in shared memory. Given the constraint that a single thread block can only contain 1024 threads [25], DR can be calculated using Eq. (5). This optimization approach considers a tradeoff. A larger DR value can make better use of the shared memory, but results in a smaller blockDim.x dimension, potentially leading to lower spatial locality within a single thread block. From our experimental results, we found that the blockDim.x can be selected using the following rule of thumb: if $k'$ is less than 32, then we set the blockDim.x to 32. Otherwise, we set blockDim.x to $k$.

Algorithm 1 describes the steps required to map kernel O1, and includes reorganization of the memory layout and initialization of kernel parameters.

$$DR = \frac{1024}{blockDim.x} \tag{5}$$

$$DR = \frac{1024}{k} \tag{6}$$

**The O2** optimization, on the other hand, focuses on leveraging the data reuse pattern first. We calculate DR using Eq. (6). Note that the blockDim.x is $k'$ in O2, computed as $k/2+1$. Eq. (6) is, in fact, derived from a different process. Given that $k'$ is not a multiple of 32, we need to find a DR value using two constraints: (1) the DR value has to be a power of 2, and (2) $DR * (k/2 + 1) <= 1024$. The DR value should be the largest power of 2 which satisfies $DR <= \frac{2048}{k+2} < \frac{2048}{k}$, leading to Eq. (6).

After adopting this approach, we find that there is no need to increase the thread block size to improve occupancy, as the thread block size equals to $\frac{1024}{k} * (k/2 + 1)$, which can be further expanded as $512 + \frac{1024}{k}$. Assuming that we only select a power of 2 for $k$, the new thread block size is a multiple of 32 when $k <= 32$. Even when $k$'s value is greater than 32, the impact is negligible, thanks to the constant term (512) for the thread block size.

**Algorithm 2** Kernel Mapping Function of O2

**Inputs:** n, p, q, k, $FFT(\frac{\partial L}{\partial a})$, $FFT(W)$, $FFT(X)$, propagation_type
**Outputs:** blockDim, gridDim
**if** propagation_type == BACKWARD_WEIGHT **then**
    $M\_1, M\_2, M\_3$ = n, q, p
**else if** propagation_type == BACKWARD_DATA **then**
    $M\_1, M\_2, M\_3$ = p, q, n
**else if** propagation_type == FORWARD **then**
    MemoryReorganize($FFT(W)$) $\rightarrow p \times k \times q$
    $M\_1, M\_2, M\_3$ = q, p, n
**end if**
k' = k / 2 + 1
blockDim.x = k'
gridDim.x = $M\_1$
DR = $\frac{1024}{k}$
blockDim.y = DR
gridDim.y = $\frac{M\_2 - DR + 1}{DR}$
gridDim.z = $M\_3$

The backward propagations have smaller MD values when applying O2, versus forward propagation. The MD value for the former is 1, which is ideal, whereas the latter is $q * k'/N$ approximately. When $N < q * k'$ (a common scenario), so the MD is larger than 1. As such, we reorganize the memory layout of $W$ from $p \times q \times k$ to $p \times k \times q$ for forward propagation.

Algorithm 2 describes the steps required to map kernel O2, including reorganization of the memory layout and initialization of kernel parameters.

Comparing with O1, O2 has both advantages and disadvantages. In terms of advantages, first, O2 only needs one memory reorganization operation, while O1 needs three. Second, O2 has a smaller MD value for reads because of the same reason described in the previous paragraph. In terms of disadvantages, O2 has a larger MD value for writes (we need to reorganize the output for the reduction summation). The occupancy issues for O1 and O2 are no longer existing. Achieving a better DR value is a bit tricky for these two optimizations, as the value for DR depends on the size of $k$. When $k < 32$, O2 achieves a better DR value, whereas, O1 and O2 have similar data reuse behaviors when $k >= 32$.

#### 4.2.2. Reduce summation

Based on the execution patterns and memory access patterns illustrated in Fig. 3(b), the kernels only perform a simple summation of the data along a single dimension, resulting in no data reuse. Therefore, we only focus on optimizing the occupancy and MD value for the summation. The baseline mapping function results in undesirable MD values for backward propagations. To improve spatial locality, we reorganize the memory layout of the outputs of the previous kernels (i.e., the element-wise multiplications) so that the specific dimension for performing the summation becomes the second innermost dimension. To address the occupancy issue, we can also use the same approach illustrated in Fig. 4 to increase the thread block size. However, our experimental results show that this can lead to irregular memory access patterns, impacting spatial locality. Therefore, we only apply memory layout reorganization to improve MD on reads for the summations.

Algorithm 3 describes the steps required to map the Reduce Summation kernel. The reduce summation does not require changes to the memory layout since the element-wise multiplication has already remapped the memory layout to a desired format.

**Algorithm 3** Kernel Mapping Function of Reduce Summation

**Inputs:** n, p, q, k, propagation_type
**Outputs:** blockDim, gridDim
**if** propagation_type == BACKWARD_WEIGHT **then**
    $M\_1, M\_2$ = q, p
**else if** propagation_type == BACKWARD_DATA **then**
    $M\_1, M\_2$ = q, n
**else if** propagation_type == FORWARD **then**
    $M\_1, M\_2$ = p, n
**end if**
k' = k / 2 + 1
blockDim.x = k'
gridDim.x = $M\_1$
gridDim.y = $M\_2$

**Algorithm 4** Kernel Mapping Function of Kernel Fusion Method

**Inputs:** n, p, q, k, propagation_type
**Outputs:** blockDim, gridDim
**if** propagation_type == BACKWARD_WEIGHT **then**
    $M\_1, M\_2$ = q, p
**else if** propagation_type == BACKWARD_DATA **then**
    $M\_1, M\_2$ = q, n
**else if** propagation_type == FORWARD **then**
    $M\_1, M\_2$ = p, n
**end if**
k' = k / 2 + 1
blockDim.x = k'
gridDim.x = 1
DR = $\frac{1024}{k}$
blockDim.y = DR
gridDim.y = $\frac{M\_1 - DR + 1}{DR}$
gridDim.z = $M\_2$

#### 4.2.3. Kernel fusion

Returning to our optimized implementation for the BCM algorithm, the first step swaps the position of the IFFT and the summation to reduce the amount of computation required during the IFFT. The optimized BCM algorithm provides us with an opportunity to ***fuse*** these two kernels into one. We apply kernel fusion, considering the same design principles just described, and leverage the same optimizations. We implement a new mapping function for the fused kernels based on O2. We chose O2 over O1 because O2 does a better job of improving the locality of the reads. We omit the memory layout reorganization operations as they are no longer necessary because the performance gains from reorganizing memory layout are negligible after the kernel fusion.

Algorithm 4 describes the steps required to map the Kernel Fusion method.

The kernel fusion approach has the following potential benefits:

- It significantly reduces the number of memory transactions required.
- The memory layout reorganization operations are no longer needed.
- It saves the overhead of launching one additional kernel.
- No intermediate data is produced between these two kernels.

## 5. Experiments

In this paper, we select the Intel Xeon CPU E5-2630 [27] as the CPU platform, and the NVIDIA Tesla V100 [28] as the GPU platform, to run our experiments. In terms of software, we use CUDA 10, and its associated SDKs and math libraries. We use cuBLAS to build GEMM as a baseline. We have extended the DNNMark benchmark framework [5] to implement the BCM algorithm, and leverage DNNMark tools to benchmark our implementations. We use the default compilation optimization level to build the executable.

To demonstrate the effectiveness of our optimization methods, we evaluate a broad range of problem sizes using real-world models trained with the ImageNet [13] and TIMIT [29] datasets. We only consider GEMM-based layers (i.e., the convolutional and fully-connected layers) in DNNs, CNNs, TDNNs, and LSTM models [6,30,31].

First, we evaluate the baseline implementation of the BCM algorithm and our optimized versions versus traditional matrix multiplication (MM). For simplicity, we use three different representative layer configurations, corresponding to the fully-connected and convolutional layers in different models, using the ImageNet dataset (Layer A and B) and fully-connected layer from models using the TIMIT dataset (Layer C). To obtain these representative layer configurations, we calculate the average number of weights across all corresponding layers and select a layer configuration having the same number of weights as close as possible to the average value. For example, we find that the average number of weights across all fully-connected layers in three ImageNet trained models (AlexNet, VGGNet, and ResNet [6,8,30]) is approximately 30 million. As such, we select 4096 and 8192 as the number of output nodes and input nodes, generating 33 million weight parameters. Table 3 describes the layer configuration details of the three representative layers.

Next, we evaluate the best performing optimization method across all GEMM-based layer configurations using models trained with the ImageNet and TIMIT datasets. For the ImageNet dataset, we select AlexNet and VGGNet-16, because they provide good coverage of a wide range of layer configurations in popular CNN models. For example, ResNet and VGGNet share the same collection of layer configurations. For the TIMIT dataset, we select the TDNN and LSTM models. For the convolutional and fully-connected layers in AlexNet and VGGNet-16, we use conv$x$ and fc$y$, where $x$ and $y$ are the index values for the convolutional and fully-connected layers, respectively. Note that we only present one layer configuration if multiple layers share the same configuration. For example, conv6 and conv7 of VGGNet-16 share a common layer configuration. For the fully-connected layer in TDNN and LSTM models, we use fc$x$-$y$ to represent the layers, where the $x$ value corresponds to the layer configuration corresponding to the number of nodes (i.e., 256, 512, 1024, and 2048), and the $y$ index specifying the layer type (i.e., input layer, hidden layer, and output layer) [31]. To capture the execution behavior, while varying the block size, we select multiple commonly-used block sizes (16, 32, and 64).

Last, we collect a range of performance counters to analyze different optimization, including: instructions per cycle, number of instructions executed, number of memory transactions, and device occupancy. To obtain the value of the performance counters, we leverage nvprof [25], a tool that can collect the hardware performance counters on NVIDIA GPUs.

## 6. Performance evaluation

### 6.1. Results

First, we present the performance evaluation results of our baseline implementation of the BCM algorithm, denoted as *BCM*, and then our optimized versions. We compare them against using a traditional matrix multiplication (MM) based representation (cuBLAS [32]). We label our optimized results using *O1+Sum*, *O2+Sum*, and *Kernel Fusion*, indicating which of our three optimized versions was used. *O1+Sum* implements multiplication applying the O1 optimization, along with an optimized summation. Version *O2+Sum* implements multiplication using O2 and the optimized summation. *Kernel Fusion* fuses the multiplication with O2 optimization and the summation, as described in Section 4.2.3.

**Table 3**
Experimental setup of three representative layer configurations in our experiments.

| Layer name | Dataset | Type of layer | Number of outputs | Number of inputs | Filter size | Number of weights | Batch size |
|---|---|---|---|---|---|---|---|
| Layer A | ImageNet | Fully-Connected Layer | 4096 | 8192 | N/A | 33 Million | 128 |
| Layer B | ImageNet | Convolutional Layer | N/A | 14 × 14 | 3 × 3 × 256 × 512 | 1.2 Million | 64 |
| Layer C | TIMIT | Fully-Connected Layer | 1024 | 1024 | N/A | 1 Million | 128 |

**Table 4**
Speedup of the Kernel Fusion approach over baseline BCM for Layer A, B, and C.

| Block size | 16 | 32 | 64 |
|---|---|---|---|
| Layer A | 43.3× | 22.4× | 11.3× |
| Layer B | 27.3× | 10.7× | 4.6× |
| Layer C | 14.7× | 5.2× | 2.6× |

Fig. 5(a)–5(c) show the speedup of BCM and our optimized versions over MM, across Layers A, B, and C, as described in Section 5. From the figures, MM outperforms BCM, our baseline implementation, as described in Section 4. The O1+Sum and O2+Sum optimizations produce substantial performance improvements over BCM. However, they cannot outperform MM. The Kernel Fusion approach outperforms the O1+Sum and O2+Sum optimizations. Each has a different impact on performance, depending on the problem size (a function of the number of weights, number of inputs, and batch size) when compared against MM. For Layer A, the Kernel Fusion approach can achieve a speedup of 1.12×, 2.1×, and 3.5× over MM across block sizes of 16, 32, and 64, respectively. For Layer B, the Kernel Fusion approach can also achieve a speedup of 1.06×, 1.39×, and 1.64× over MM for block sizes of 16, 32, and 64, respectively. For Layer C, the benefit of the Kernel Fusion approach is limited. From our results, we find that our proposed optimization approach can achieve better performance than MM when the problem size is sufficiently large. Although our optimization approach has limited benefits when problem sizes are small, it still outperforms the baseline implementation (BCM), significantly reducing the execution time needed for training a block-circulant DNN. Table 4 lists the speedup of the Kernel Fusion approach over the baseline version. From the table, the Kernel Fusion approach can achieve an average speedup of 25.7× for Layer A, 14.2× for Layer B, and 7.5× for Layer C.

We find that problem size has an impact on the performance of our optimization approach. Thus, we vary the batch size for Layers A, B, and C and present the speedup for different problem sizes in Fig. 6(a)–6(c). In this experiment, we evaluate the best performing optimization, i.e., the Kernel Fusion approach, across different block sizes, labeled as *BCM16*, *BCM32*, and *BCM64*. The baseline is MM. From the figures, we can notice that the speedup increases with increased batch size. When the batch size is sufficiently large, the trend becomes less obvious.

In Fig. 7(a)–7(c), we present the speedup of the Kernel Fusion approach over MM, across all GEMM-based layer configurations from AlexNet and VGGNet-16, trained with the ImageNet dataset, and the TDNN and LSTM models, trained with the TIMIT dataset. From our results of using a block size of 64, we see that for AlexNet we can achieve an average speedup of 1.31× for the convolutional layers and 2.79× for the fully-connected layers. For VGGNet-16 with a block size of 64, we can achieve an average speedup of 1.33× for the convolutional layers and 3.66× for the fully-connected layers. Note that two fully-connected layers in VGGNet-16 share the same layer configurations of fc2 and fc3 in AlexNet. For TDNN and LSTM, the performance gains are limited. Only when the number of weights is larger than 1.9 million can we achieve a speedup of 2.12×, given block size of 64.

### 6.2. Performance analysis

In this section, we select a number of hardware performance counters to compare performance. We capture the number of instructions executed, the number of memory transactions, and the GPU occupancy.

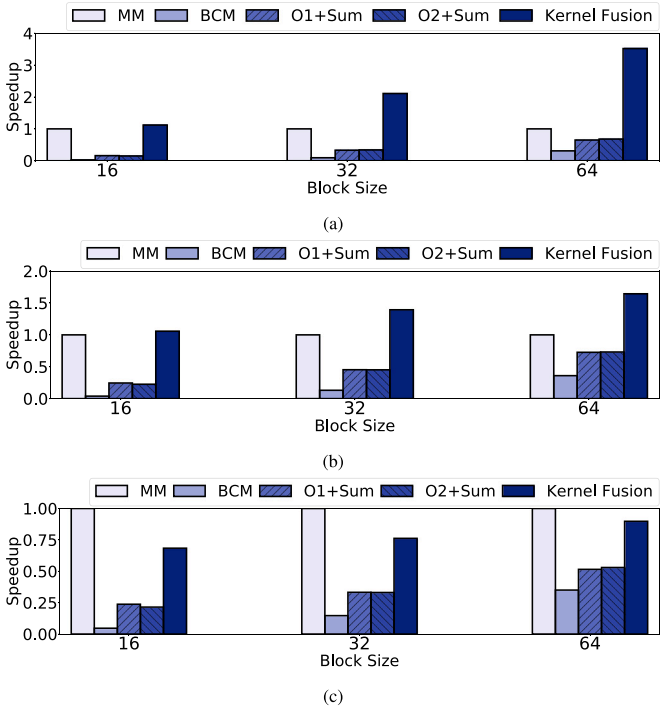


**Fig. 5.** Speedup of the BCM baseline and our optimized versions, as compared to a matrix multiplication implementation for (a) Layer A, (b) Layer B, and (c) Layer C.

We use these metrics to evaluate the effectiveness of each optimization, and discuss the reasons for the resulting performance.

We present the implications of using the customized kernels in Figs. 8–12. For simplicity, we only discuss the results of Layer A, as other layers exhibit similar trends.

In Fig. 8, we can see that the IPC is always higher for the optimized versions. For different optimizations, while each provides a different degree of performance improvement, we find that the trends in each optimization are consistent. We conclude that the number of eligible warps that can run per cycle has a strong impact on performance. The O2+Sum implementation results in the lowest number of eligible warps per cycle, as compared to the other optimized versions. We found that the resulting performance of O2+Sum is heavily impacted by the behavior of memory operations. As we explored the performance achieved in each optimized version, we found that the blockDim.x in the kernel mapping function in the multiplication kernels plays an important role. Across our implementations, a larger blockDim.x can launch more spatially local memory accesses, leading to a higher degree of memory coalescing. This translates to fewer memory transactions. O2+Sum uses $k'$ for its blockDim.x, whereas O1+Sum and Kernel Fusion use $N$ and $k' \times M_i$, respectively ($k'$ is the new block size after the FFT operation, $N$ is the new blockDim.x, as shown in Fig. 4, and $M_i$ is the size of the dimension over which summation is performed). Obviously, a blockDim.x of O2+Sum is smaller than the other two values. Although we use O2, Kernel Fusion significantly improves the memory efficiency by reducing the number of memory transactions needed, leading to higher IPC.

From the same figure, we also see that the IPC of the O2+Sum optimization using a block size of 16 results in comparable performance compared to the other two schemes, This is because using a

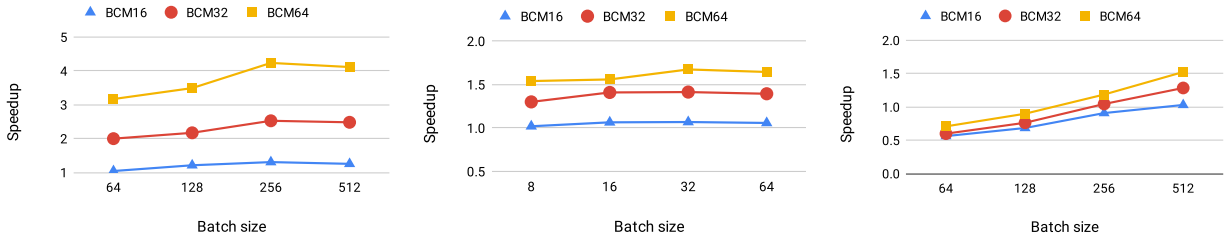


**Fig. 6.** Speedup of baseline BCM version and our optimized versions, as compared to a matrix multiplication implementation for (a) Layer A, (b) Layer B, and (c) Layer C.

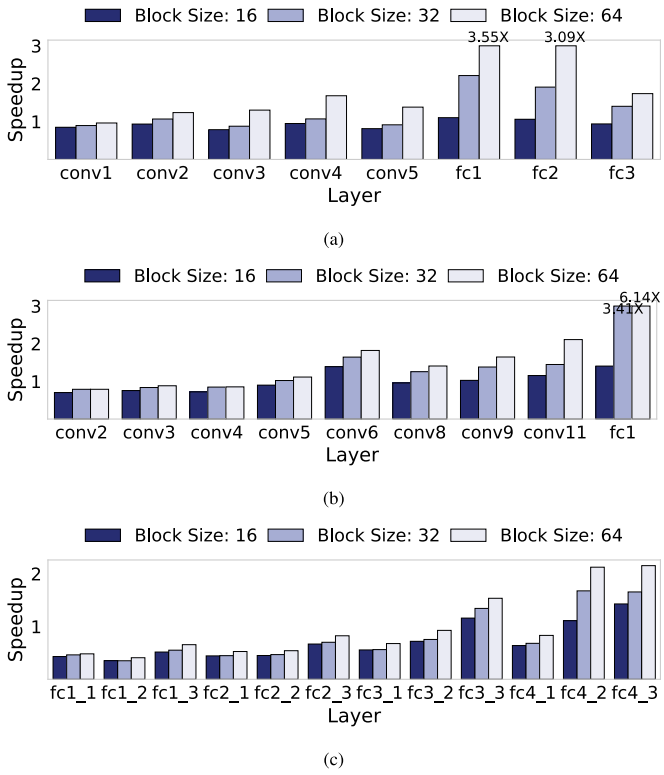


**Fig. 7.** Speedup of the Kernel Fusion approach, as compared to a matrix multiplication implementation, for all layer configurations in (a) AlexNet, (b) VGGNet-16, and (c) TDNN and LSTM.

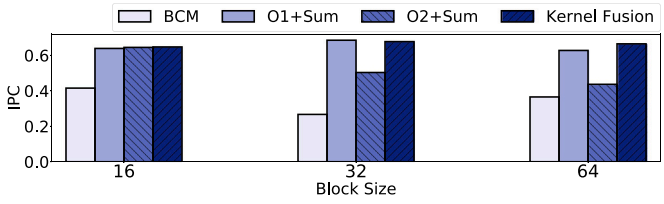


**Fig. 8.** IPC of the baseline and optimized versions.

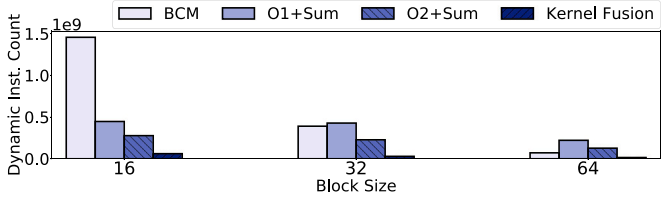


**Fig. 9.** Dynamic instruction count of the baseline and optimized versions.

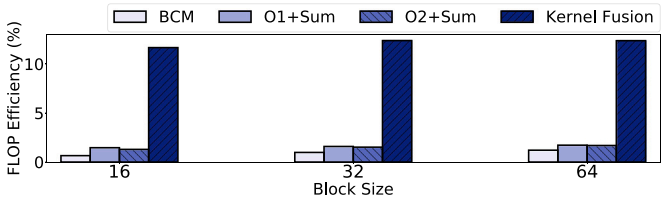


**Fig. 10.** FLOP efficiency of the baseline and optimized versions.

**Table 5**
The static instruction count of the baseline and optimized versions.

| BCM | O1+Sum | O2+Sum | Kernel fusion |
|---|---|---|---|
| 2003 | 2363 | 2693 | 4797 |

smaller block size leads to higher DR (as discussed in Section 4). The O2+Sum optimization is making better utilization of shared memory, thus compensating for lower memory coalescing, as described above.

Table 5 lists the static instruction count for different optimization methods. The static instruction count reflects the programming style used in each optimization, immutable in the executable. From our results, we can see that the baseline version has a low static instruction count, given that the implementation is straightforward. The implementations of other optimizations (i.e., O1+Sum, O2+Sum, and Kernel Fusion) result in higher static instruction counts, because the implementations require more integer operations to calculate the data indices and additional branch logic for avoiding out-of-bounds data indexing. The level of program complexity determines the number of static instructions.

Fig. 9 presents the dynamic instruction count (i.e., the number of instructions executed) for different block sizes. Analyzing this metric sheds light on the execution efficiency of each optimization. From the figure, Kernel Fusion results in the highest execution efficiency (smallest dynamic instruction count) as compared to the other optimizations.

To better quantify the execution efficiency, we use FLOP efficiency, which is the achieved percentage of peak floating-point operations [25]. This metric captures the execution efficiency on a GPU. As shown in Fig. 10, the O1+Sum and O2+Sum optimizations result in higher FLOP efficiency versus the baseline version. Kernel Fusion achieves the highest FLOP efficiency, outperforming the other optimizations and the baseline version. Kernel Fusion removes a large number of memory transactions, reducing memory pressure more effectively than the other codes.

Next, we present the number of memory transactions in Figs. 11(a) and 11(b), reporting the number of memory reads and writes, respectively. We can see that the Kernel Fusion optimization significantly reduces the number of memory transactions for both reads and writes. We can also see that the number of memory transactions drops as the block size increases. This is because, as the block size grows, there are fewer circulant matrix blocks. As such, the number of inter-block memory accesses is reduced, leading to a more regular memory access pattern, and hence, fewer memory transactions.

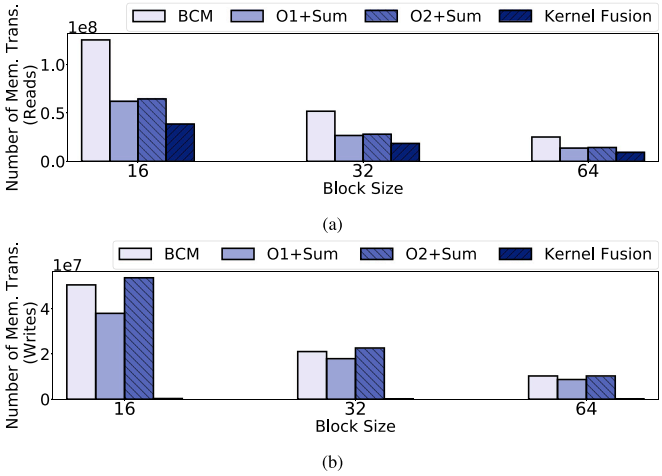


**Fig. 11.** (a) Number of memory transactions (Reads) for the baseline and optimized versions. (b) Number of memory transactions (Writes) for the baseline and optimized versions.

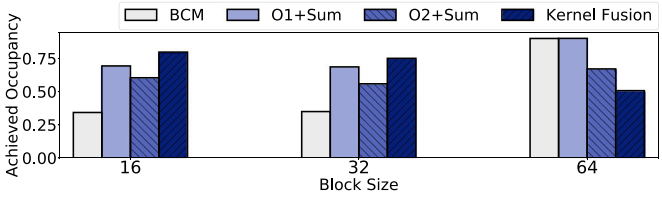


**Fig. 12.** The occupancy achieved for the baseline and optimized versions.

Fig. 12 shows the effectiveness of each optimization in terms of GPU occupancy, across different block sizes (16 and 32). However, as the block size $k$ grows to 64, Kernel Fusion results in a lower occupancy than that of other implementations. This is because its thread block size decreases monotonically as a function of $k$, i.e., $512+\frac{1024}{k}$. Unlike Kernel Fusion, the O2+Sum optimization achieves an unexpected increase in occupancy when using a block size of 64, even though it includes similar optimizations as used in Kernel Fusion. This is due to the impact of increased occupancy of the summation kernel. The multiplication kernel in O2+Sum actually has reduced occupancy, for the same reason as was experienced using Kernel Fusion.

### 6.3. Compression rate v.s. training performance

To begin, we use the BCM algorithm to train a DNN model that has 4 convolutional layers, 3 fully connected layers and a softmax layer, with the CIFAR-10 dataset [33]. We use block sizes of 16 (BCM16), 32 (BCM32) and 64 (BCM64). Figs. 13(a) and 13(b) compare the training convergence rate and classification accuracy for GEMM versus our BCM algorithm. From the figures, we can see that the BCM algorithm obtains a similar convergence rate as compared with the GEMM approach. We also find that the BCM algorithm achieves a similar classification accuracy as using GEMM. In this experiment, the accuracy degrades by 1.4% for BCM16. The accuracy degrades by 2.3% for BCM32, and by 5.2% for BCM64. We find that the accuracy degradation is closely correlated with the training convergence rate. The less the accuracy degrades, the less training convergence suffers.

Next, to demonstrate the effectiveness of training using the BCM algorithm, in Fig. 14(a) we show the compression rate of models trained with different datasets, including MNIST [34], SHVN [35], CIFAR-10 [33], ImageNet [13] and TIMIT [29]. We can achieve a significant reduction in model size, as shown in the figure. We also report the accuracy degradation of the same models in Fig. 14(b). For MNIST and SVHN, the block size is 32. For CIFAR-10, ImageNet and TIMIT, the block size is 16. For each model, the degradation is smaller than 4%, as compared with the original models.

### 6.4. Multi-GPU DNN training

Multi-GPU DNN training has been widely deployed to shorten the training time [36]. In multi-GPU training, the same DNN model is duplicated on distributed GPU nodes and the forward and backward propagation are executed independently with different batches of input data. Only the gradients need to be collected from the different nodes and aggregated to update the weights.

Using BCM also benefits multi-GPU training as it can significantly reduce the amount of data transferred during the weight update stage. The BCM algorithm reduces the weights in one layer by a factor of $k$ (i.e., the block size). Given a DNN model, we quantify the original weight size and reduced weight size in Eqs. (7) and (8), respectively. $W$ is the original weight size. $W_{bcm}$ is the reduced weight size. $N$ is the number of layers with learnable weights. $W_i$ is the number of weights of the $ith$ layer. $k_i$ is the block size used for the $ith$ layer. The compression ratio can be calculated using $W/W_{bcm}$. A higher compression ratio results in more efficient data communications when using distributed training, leading to better scalability. We use Caffe [37] to conduct an experiment involving two PCIe-connected NVIDIA V100 GPUs and perform training for a simple model. We replace one regular linear transformation layer with a block circulant layer using the Kernel Fusion optimization. When the block size is 64, we achieve a 6.25X speedup over using a matrix multiplication.

$$W = \sum_{i=1}^{N} W_i \tag{7}$$

$$W_{bcm} = \sum_{i=1}^{N} \frac{W_i}{k_i} \tag{8}$$

## 7. Related work

Prior work has explored improving scalability, performance, and energy efficiency for deep learning applications. Two orthogonal trends have dominated this prior research.

The first trend focuses on using *custom hardware to accelerate DNNs*. Representative work include Google's TPU [38,39] and IBM's TrueNorth [40–42]. There have been a number of optimizations proposed, including dataflow designs [43], increasing the degree of parallelism [44], improving the data mapping [45], reducing off-chip memory transfers [46], improved bit-representations and quantization [47], zero skipping [48], and many others.

The second trend focuses on *model compression*. Several algorithm-level techniques have been proposed to compress models and accelerate DNNs, including weight quantization [49,50], connection pruning [51,52], and low-rank approximation [53,54]. These approaches offer reasonable parameter reductions (e.g., Han et al. reported a 9× to 13× improvement [51,52]), with only a minor impact on accuracy.

In contrast to the above approaches, Cheng et al. studied parameter redundancy in DNNs by using circulant matrices for weight representation of the fully-connected layers [15]. They proposed the corresponding accelerated inference and training algorithms for the fully connected layers. As a follow on work, Ding et al. evaluated the performance versus accuracy tradeoff of using block-circulant weight matrices [10]. They generalized the approach for both fully-connected layers and convolutional layers, and provided a cross-platform hardware design and optimization solution for deep learning systems.

There have also been a number of previous studies focused on accelerating DNNs on GPUs. Many deep learning frameworks, such as Tensorflow [55], Caffe [37], and CNTK [56], leverage GPUs for acceleration. Eliuk et al. developed a distributed DNN library to accelerate

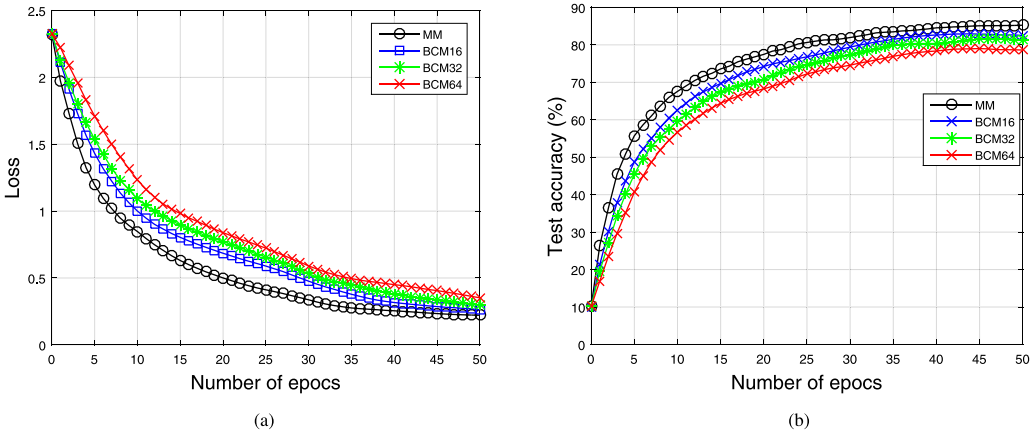


**Fig. 13.** (a) Training convergence rate on CIFAR-10 dataset. (b) Test accuracy on CIFAR-10 dataset.

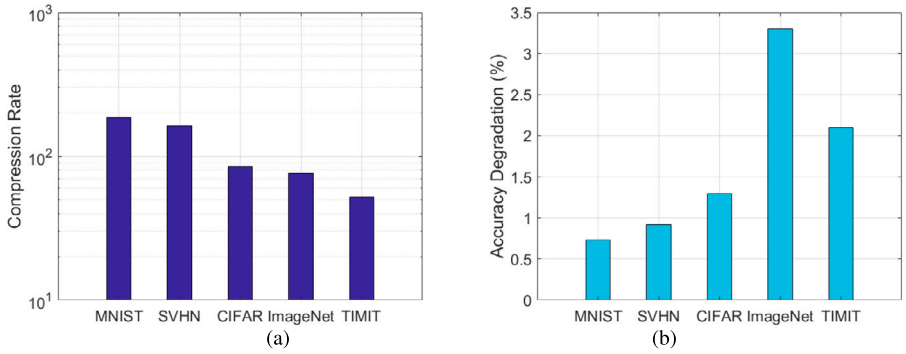


**Fig. 14.** (a) Compression rate and (b) Accuracy degradation of models trained with datasets, including MNIST, SHVN, CIFAR-10, ImageNet, and TIMIT.

DNN execution on a GPU cluster [57]. Awan et al. extended Caffe to enable distributed training on a GPU cluster [36]. Dong et al. characterized the microarchitectural implications of a convolutional neural network execution on a single GPU and suggested optimizations for acceleration [58]. However, all previous approaches did not consider compression techniques, focusing primarily on lossless optimizations.

Each of the previous studies has their own merits and limitations. First, most of the previous work was *not* focused on using general purpose processors, such as CPUs and GPUs. Therefore, they tend to be limited in terms of the size of the network model (i.e., based on the limits of a FPGA) and the flexibility of the model. Second, most of the previous work using GPUs did not consider weight compression techniques, primarily focusing on lossless optimizations. Our work addresses the challenges of optimizing one DNN weight compression technique on a GPU, accelerating DNN execution with a tolerable accuracy loss.

## 8. Conclusion and future work

In this paper, we present our work accelerating DNN training using Block Circulant Matrices. Using BCMs can accelerate model training for both forward and backward propagation. We redesigned the operations used to perform backward propagation in the context of commonly-used batch-based training. We identified the basic operations required in the BCM algorithm and presented our GPU implementation. After highlighting some of the challenges that the BCM algorithm encounters on a GPU platform, we proposed a number of optimizations to improve the performance. From our results, we can achieve an average speedup of 1.31× and 2.79× for the convolutional layers and fully-connected layers of AlexNet, respectively, and a speedup of 1.33× and 3.66× for the convolutional layers and fully-connected layers of VGGNet-16, respectively. We can achieve an improved compression ratio for the weights, while experiencing only a minor loss in the training convergence rate and model accuracy.

In our future work, we plan to explore the benefits of mapping multiple stages of the BCM algorithm to different GPUs, and executing them in a pipelined fashion [59]. We plan to further optimize memory usage when deploying the BCM algorithm, especially since memory usage becomes a growing concern as DNNs grow deeper and wider. We also plan to explore other weight compression techniques on GPU, such as leveraging Permuted Diagonal Matrices [60].

## Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.parco.2020.102701.

## Acknowledgment

This work was supported by the National Science Foundation, USA [NSF CCF AiF #1733701].

## References

[1] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.
[2] E. Guizzo, How google's self-driving car works, 2016.
[3] Siri Team, Deep learning for siri's voice: On-device deep mixture density networks for hybrid unit selection synthesis, 2017.
[4] B. Reagen, R. Adolf, P. Whatmough, Deep Learning for Computer Architects, Morgan & Claypool Publishers, 2017.
[5] S. Dong, D. Kaeli, DNNMark: A deep neural network benchmark suite for GPUs, in: Proceedings of the General Purpose GPUs, GPGPU-10, ACM, New York, NY, USA, 2017, pp. 63–72, http://dx.doi.org/10.1145/3038228.3038239.

[6] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, in: International Conference on Learning Representations, 2015.

[7] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: 2015 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2015, pp. 1–9.

[8] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2016, pp. 770–778.

[9] S. Liao, Z. Li, X. Lin, Q. Qiu, Y. Wang, B. Yuan, Energy-efficient, high-performance, highly-compressed deep neural network design using block-circulant matrices, in: Proceedings of the 36th International Conference on Computer-Aided Design, IEEE Press, 2017, pp. 458–465.

[10] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Lin, C ir CNN: accelerating and compressing deep neural networks using block-circulant weight matrices, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2017, pp. 395–408.

[11] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, Z. Zhang, Accelerating binarized convolutional neural networks with software-programmable FPGAs, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17, ACM, New York, NY, USA, 2017, pp. 15–24, http://dx.doi.org/10.1145/3020078.3021741.

[12] H. Alemdar, N. Caldwell, V. Leroy, A. Prost-Boucle, F. Pétrot, Ternary neural networks for resource-efficient AI applications, 2016.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, ImageNet: A large-scale hierarchical image database, in: CVPR09, 2009.

[14] S.A. Mojumder, M.S. Louis, Y. Sun, A.K. Ziabari, J.L. Abellán, J. Kim, D. Kaeli, A. Joshi, Profiling DNN workloads on a volta-based DGX-1 system, in: 2018 IEEE International Symposium on Workload Characterization, IISWC, 2018, pp. 122–133.

[15] Y. Cheng, F.X. Yu, R.S. Feris, S. Kumar, A. Choudhary, S.-F. Chang, An exploration of parameter redundancy in deep networks with circulant projections, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 2857–2865.

[16] Y. Wang, C. Ding, Z. Li, G. Yuan, S. Liao, X. Ma, B. Yuan, X. Qian, J. Tang, Q. Qiu, X. Lin, Towards ultra-high performance and energy efficiency of deep learning systems: an algorithm-hardware co-optimization framework, in: Proceedings of the 32nd AAAI Conference on Artificial Intelligence, AAAI, 2018.

[17] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, 1958, pp. 65–386,

[18] NVIDIA, cuDNN: GPU accelerated deep learning, 2016.

[19] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780, http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[20] L. Zhao, S. Liao, Y. Wang, Z. Li, J. Tang, B. Yuan, Theoretical properties for neural networks with weight matrices of low displacement rank, in: International Conference on Machine Learning, 2017, pp. 4082–4090.

[21] V. Sindhwani, T. Sainath, S. Kumar, Structured transforms for small-footprint deep learning, in: Advances in Neural Information Processing Systems, 2015, pp. 3088–3096.

[22] V.Y. Pan, Structured Matrices and Polynomials: Unified Superfast Algorithms, Springer Science & Business Media, 2012.

[23] D. Bini, V. Pan, W. Eberly, Polynomial and matrix computations volume 1: Fundamental algorithms, SIAM Rev. 38 (1) (1996) 161–164.

[24] NVIDIA, CUDA CUFFT library, 2016.

[25] NVIDIA, CUDA toolkit documentation, 2016.

[26] K.R. Rao, D.N. Kim, J.-J. Hwang, Fast Fourier Transform - Algorithms and Applications, first ed., Springer Publishing Company, Incorporated, 2010.

[27] Intel, Intel(R) Xeon(R) Processor E5-2630 v3, 2014.

[28] NVIDIA, NVIDIA TESLA V100 GPU architecture, 2017.

[29] J. Garofolo, L. Lamel, W. Fisher, J. Fiscus, D. Pallett, N. Dahlgren, V. Zue, TIMIT Acoustic-phonetic continuous speech corpus, Linguist. Data Consortium (1992).

[30] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: F. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger (Eds.), Curran Associates, Inc., 2012, pp. 1097–1105,

[31] J. Michalek, J. Vanek, A survey of recent DNN architectures on the TIMIT phone recognition task.

[32] NVIDIA, cuBLAS: Dense linear algebra on GPUs, 2016.

[33] A. Krizhevsky, V. Nair, G. Hinton, CIFAR-10 (Canadian Institute for Advanced Research).

[34] Y. LeCun, C. Cortes, MNIST handwritten digit database, 2010.

[35] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, A.Y. Ng, Reading digits in natural images with unsupervised feature learning, in: NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011, 2011.

[36] A.A. Awan, K. Hamidouche, J.M. Hashmi, D.K. Panda, S-caffe: Co-designing MPI runtimes and caffe for scalable deep learning on modern GPU clusters, in: Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '17, ACM, New York, NY, USA, 2017, pp. 193–205, http://dx.doi.org/10.1145/3018743.3018769.

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

[38] N. Jouppi, Google supercharges machine learning tasks with TPU custom chip, 2016.

[39] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T.V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C.R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, D.H. Yoon, In-datacenter performance analysis of a tensor processing unit, 2017.

[40] P.A. Merolla, J.V. Arthur, R. Alvarez-Icaza, A.S. Cassidy, J. Sawada, F. Akopyan, B.L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S.K. Esser, R. Appuswamy, B. Taba, A. Amir, M.D. Flickner, W.P. Risk, R. Manohar, D.S. Modha, A million spiking-neuron integrated circuit with a scalable communication network and interface, Science 345 (6197) (2014) 668–673.

[41] S.K. Esser, R. Appuswamy, P. Merolla, J.V. Arthur, D.S. Modha, Backpropagation for energy-efficient neuromorphic computing, in: NIPS, 2015, pp. 1117–1125.

[42] S.K. Esser, P.A. Merolla, J.V. Arthur, A.S. Cassidy, R. Appuswamy, A. Andreopoulos, D.J. Berg, J.L. McKinstry, T. Melano, D.R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M.D. Flickner, D.S. Modha, Convolutional networks for fast, energy-efficient neuromorphic computing, National Acad Sciences, 2016, pp. 11441—11446,

[43] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, O. Temam, Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning, ACM Sigplan Notices 49 (4) (2014) 269–284.

[44] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, H. Yang, Going deeper with embedded FPGA platform for convolutional neural network, in: FPGA, ACM, 2016, pp. 26–35.

[45] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, O. Temam, Dadiannao: A machine-learning supercomputer, in: MICRO, IEEE Computer Society, 2014, pp. 609–622.

[46] Y.H. Chen, T. Krishna, J.S. Emer, V. Sze, Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks, IEEE J. Solid-State Circuits 52 (1) (2017) 127–138.

[47] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M.A. Horowitz, W.J. Dally, EIE: efficient inference engine on compressed deep neural network, in: ISCA, IEEE Press, 2016, pp. 243–254.

[48] P. Judd, J. Albericio, A. Moshovos, Stripes: Bit-serial deep neural network computing, in: MICRO, IEEE, 2016, pp. 1–12.

[49] D. Lin, S. Talathi, S. Annapureddy, Fixed point quantization of deep convolutional networks, in: ICML, 2016, pp. 2849–2858.

[50] J. Wu, C. Leng, Y. Wang, Q. Hu, J. Cheng, Quantized convolutional neural networks for mobile devices, in: CVPR, 2016.

[51] S. Han, H. Mao, W.J. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015.

[52] S. Han, J. Pool, J. Tran, W.J. Dally, Learning both weights and connections for efficient neural network, in: NIPS, 2015, pp. 1135–1143.

[53] M. Jaderberg, A. Vedaldi, A. Zisserman, Speeding up convolutional neural networks with low rank expansions, 2014.

[54] C. Tai, T. Xiao, X. Wang, W. E., Convolutional neural networks with low-rank regularization, 2015.

[55] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: A system for large-scale machine learning, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation, {OSDI} 16, 2016, pp. 265–283.

[56] F. Seide, A. Agarwal, CNTK: Microsoft's Open-Source Deep-Learning Toolkit, ACM, pp. 2135–2135, http://dx.doi.org/10.1145/2939672.2945397.

[57] S. Eliuk, C. Upright, A. Skjellum, Dmath: A scalable linear algebra and math library for heterogeneous GP-GPU architectures, 2016.

[58] S. Dong, X. Gong, Y. Sun, T. Baruah, D. Kaeli, Characterizing the microarchitectural implications of a convolutional neural network (CNN) execution on GPUs, in: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, ACM, New York, NY, USA, 2018, pp. 96–106, http://dx.doi.org/10.1145/3184407.3184423.

[59] T. Gautier, J.V. Ferreira Lima, N. Maillard, B. Raffin, Locality-aware work stealing on multi-CPU and multi-GPU architectures, in: 6th Workshop on Programmability Issues for Heterogeneous Multicores, MULTIPROG, Berlin, Germany, 2013.

[60] C. Deng, S. Liao, Y. Xie, K.K. Parhi, X. Qian, B. Yuan, Permdnn: Efficient compressed DNN architecture with permuted diagonal matrices, in: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2018, pp. 189–202, http://dx.doi.org/10.1109/MICRO.2018.00024.