Optimizing Convolutional Neural Networks on the Sunway TaihuLight Supercomputer

WENLAI ZHAO, HAOHUAN FU, JIARUI FANG, WEIJIE ZHENG, LIN GAN, and GUANGWEN YANG, Tsinghua University

The Sunway TaihuLight supercomputer is powered by SW26010, a new 260-core processor designed with on-chip fusion of heterogeneous cores. In this article, we present our work on optimizing the training process of convolutional neural networks (CNNs) on the Sunway TaihuLight supercomputer. Specifically, a highly efficient library (swDNN) and a customized Caffe framework (swCaffe) are proposed. Architecture-oriented optimization methods targeting the many-core architecture of SW26010 are introduced and are able to achieve 48 times speedup for the convolution routine in swDNN and 4 times speedup for the complete training process of the VGG-16 network using swCaffe, compared to the unoptimized algorithm and framework. Compared to the cuDNN library and the Caffe framework based on the NVIDIA K40m GPU, the proposed swDNN library and swCaffe framework on SW26010 have nearly half the performance of K40m in single -precision and have 3.6 times and 1.8 times speedup over K40m in double precision, respectively.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Multicore architectures**;

Additional Key Words and Phrases: Convolutional neural network, deep learning, heterogeneous many-core architecture, Sunway TaihuLight supercomputer

This article is an extension of a conference paper: "swDNN: A Library for Accelerating Deep Learning Applications on Sunway TaihuLight" published in IPDPS 2017 [10]. We consider this work an improved edition of the conference paper with new contributions listed as follows:

- We present a more systemic algorithm design and optimization process with methods related to the local directive memory usage, register communication, and instruction pipeline, including a modified performance model and a new register blocking strategy based on the previous work.
- We propose algorithm design and optimization methods to support single precision on SW26010.
- We present the swDNN library with a four core-group (CG) parallelization to support different CNN layers.
- We propose swCaffe, an optimized Caffe framework that can support a highly efficient CNN training process on the Sunway TaihuLight supercomputer.
- We present algorithm and framework evaluation with both float and double precision for training practical CNN models to provide more comprehensive performance results.

This work was supported in part by the National Key R&D Program of China (grant 2016YFA0602200), by the National Natural Science Foundation of China (grants 4137411, 91530323, 61702297, and 61672312), and by the China Postdoctoral Science Foundation (2016M601031).

Q1

Authors' addresses: W. Zhao, J. Fang, W. Zheng, L. Gan, and G. Yang, Department of Computer Science and Technology, Tsinghua University, Beijing 100084; H. Fu (corresponding author), Department of Earth System Science, Tsinghua University, Beijing 100084, China; email: haohuan@tsinghua.edu.cn. All authors are concurrently with the National Supercomputing Center in Wuxi, Wuxi, 214000, Jiangsu Province, China.

https://doi.org/10.1145/3177885

ACM Reference format:
Wenlai Zhao, Haohuan Fu, Jiarui Fang, Weijie Zheng, Lin Gan, and Guangwen Yang. 2018. Optimizing Convolutional Neural Networks on the Sunway TaihuLight Supercomputer. *ACM Trans. Archit. Code Optim.* 15, 1, Article 13 (March 2018), 26 pages.
https://doi.org/10.1145/3177885

1 INTRODUCTION

The convolutional neural network (CNN [14]) is one of the most successful deep learning models in modern artificial intelligence applications [8, 12, 20, 22, 24]. The training process of CNN involves a large amount of computation and has become a popular research topic in the field of high performance computing (HPC). GPUs have currently been considered as the most efficient hardware choice for deep learning tasks and can support high-level deep learning frameworks [1, 4, 7, 13].

Sunway TaihuLight [11], a supercomputer that ranks number one in the latest release (November 2017) of the TOP 500 list with over 100 PFlops computing capacity, is powered by the SW26010 many-core processor, which is designed with on-chip fusion of heterogeneous cores and is able to provide a peak double-precision performance of 3.06 TFlops. SW26010 introduces several unique features that could potentially accelerate the training process of CNNs, such as user-controlled local directive memory (LDM), hardware-supported register-level data sharing, and a unified memory space shared by all processing elements.

Our previous publication [10] introduced the optimization of convolutional algorithm targeting the many-core architecture of SW26010. As an extension of the previous work, in this article we present a more systemic optimization process to accelerate CNN training tasks on the Sunway TaihuLight supercomputer. Specifically, a highly efficient library and a customized Caffe framework for the SW26010 many-core processor are proposed.

The major contributions of this article include the following:

- We propose algorithm design and optimization methods related to the LDM usage, register communication, and instruction pipeline, guided by a performance model. The optimized convolution routine can achieve 48 times speedup over the basic implementation on SW26010.
- A customized deep learning library for the SW26010 many-core processor is developed, called *swDNN*, to provide the support for various computation and data processing layers in CNN models.
- An optimized Caffe framework for the SW26010 many-core processor is proposed, called *swCaffe*, which is integrated with the swDNN library and supports a four core-group (CG) parallelization on a SW26010 processor. The swCaffe framework can achieve about 4 times speedup over a BLAS-based Caffe framework on SW26010.

Evaluation results also show that the proposed convolution implementation and the swCaffe framework have nearly half the performance of the NVIDIA K40m GPU in single precision while achieving 3.6 times and 1.8 times speedup over K40m in double precision, respectively.

The article is organized as follows. Section 2 introduces the background of the work, including the CNN algorithms, the detailed architecture of SW26010, and the related work on the optimization of CNN algorithms. Section 3 presents the performance model and architecture-oriented optimization methods targeting the convolution algorithm, including the evaluation of the implementation. Section 4 presents the swDNN library and the swCaffe framework, as well as the

Table 1. Configurations of a Convolutional Layer

N_i	Number of input feature maps
R_i	Height of an input feature map
C_i	Width of an input feature map
N_o	Number of output feature maps
R_o	Height of an output feature map
C_o	Width of an output feature map
K	Size of convolution kernel

evaluation of the complete training process with swCaffe on a SW26010 many-core processor. Section 5 presents our conclusion.

2 BACKGROUND

2.1 Convolutional Neural Networks

CNNs usually contain multiple computing layers, among which *convolutional layers* usually account for the majority of the computing time (greater than 90%). We first give the description of the convolutional layer configurations, listed in Table 1. The input data of a convolutional layer consists of N_i channels, each of which can be considered as a feature map with size of $R_i \times C_i$. Similarly, the output of a convolutional layer consists of N_o feature maps with size of $R_o \times C_o$. To calculate the values in an output feature map, N_i convolutional kernels with size of $K \times K$ and 1 bias value are required. Each kernel convolutes with an input feature map. The output value equals the sum of N_i convolution results and the bias value. Therefore, there are $N_i \times N_o$ convolutional kernels and N_o bias in a convolutional layer.

The training process of a CNN model is based on the stochastic gradient descent (SGD) algorithm. In each training step, the network is trained with a batch of samples. We define the batch size as B_s, and the original algorithm of a convolutional layer in a training iteration can be described as Algorithm 1. The input data, output data, and convolution weights are organized in four-dimension tensors, and there are seven nested loops in the algorithm, which provides possibilities for the parallel optimization on many-core processors like SW26010.

In addition to convolutional layers, a CNN usually contains other kinds of layers, such as pooling layers, fully connected layers, softmax layers, and other data processing layers, such as activation function layers and normalization layers. Different CNN models have different network structures, which describe how different kinds of layers are stacked in the neural network.

The major algorithm of fully connected layers is matrix multiplication, which can be supported by the high-performance basic linear algebra subprograms (BLAS). Other layers, such as pooling, activation functions, and softmax, can be considered as data processing layers, and they are not the critical points of performance optimization.

2.2 SW26010 Many-Core Architecture

Figure 1 shows the architecture of a SW26010 many-core processor. SW26010 consists of four CGs, and each CG includes 65 cores: one management processing element (MPE), and 64 computing processing elements (CPEs) organized as an 8×8 mesh. The MPE and CPE are both complete 64-bit RISC cores but serve different roles in a computing task.

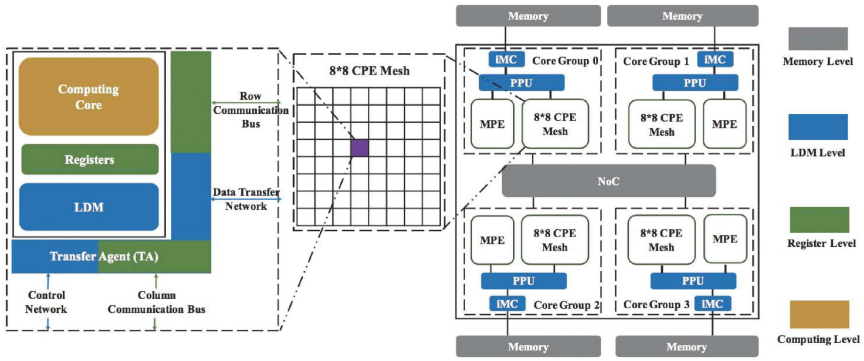


Fig. 1. SW26010 architecture.

ALGORITHM 1: Original Algorithm of a Convolutional Layer

1: //$IN[B_s][N_i][R_i][C_i]$, $OUT[B_s][N_o][R_o][C_o]$, $CONVW[N_o][N_i][K_r][K_c]$, and $b[N_o]$ are input/output feature maps, convolutional kernels, and bias
2: //$K_r = K_c = K$ represent the number of rows and columns of a 2-dimensional convolutional kernel
3: //The output images OUT are initialized with the bias b
4: **for** $cB := 0 : 1 : B_s$ **do**
5: **for** $cN_o := 0 : 1 : N_o$ **do**
6: **for** $cR_o := 0 : 1 : R_o$ **do**
7: **for** $cC_o := 0 : 1 : C_o$ **do**
8: **for** $cN_i := 0 : 1 : N_i$ **do**
9: **for** $cK_r := 0 : 1 : K_r$ **do**
10: **for** $cK_c := 0 : 1 : K_c$ **do**
11: $OUT[cB][cN_o][cR_o][cC_o] += CONVW[cN_o][cN_i][Kr - 1 - cK_r][K_c - 1 - cKc] * IN[cB][cN_i][cR_o + cK_r][cC_o + cKc]$;
12: **end for**
13: **end for**
14: **end for**
15: **end for**
16: **end for**
17: **end for**
18: **end for**

An MPE has a 32KB L1 instruction cache, a 32KB L1 data cache, and a 256KB L2 cache, supporting the complete interrupt functions, memory management, superscalar, and out-of-order instruction issue/execution.

The CPE is designed for maximizing the aggregated computing throughput while minimizing the complexity of the microarchitecture. Each CPE has a 16KB L1 instruction cache and a 64KB LDM. The LDM can be considered as a user-controlled fast buffer, which allows orchestrated memory usage strategies for different implementations, so the LDM-level optimization is one of the important ways to improve the computation throughput.

A CPE has 32 vector registers (256 bits) and two execution pipelines (P0 and P1). P0 supports scalar and vectorized computing operations of both floating-point and integer, whereas P1 supports scalar and vectorized data load/store, compare, jump operations, and scalar integer operations. The double pipelines provide an opportunity for the overlapping of data accessing and

computation operations. Therefore, register-level and instruction-level optimizations are also important to performance.

Inside the 8×8 CPE mesh, there is a control network, a data transfer network (connecting the CPEs to the memory interface), eight column communication buses, and eight row communication buses. Each CPE has two 1,024-bit send buffers and two 1,024-bit receive buffers for column and row communication separately. The communication buses and buffers enable fast register-level data communication between CPEs of same column and same row, providing an important data sharing and cooperation capability within the CPE mesh.

In the instruction set, there are customized load/store instructions to support both vectorized data access and data sharing in a nonblocking mode. For example, a *vldr* instruction first loads 256-bit data into a vector register and then performs the *row broadcast*; a *vlddec* instruction first loads 64-bit data into a scalar register, then extends (copies) the data to fill a vector register, and finally performs the *column broadcast*. Based on these instructions, highly efficient data access and register communication can be realized.

Each CG connects to a memory controller (MC), through which 8GB memory space can be accessed and shared by the MPE and the CPE mesh. The maximum memory bandwidth of an MC is 36GB/s. An on-chip network (NoC) connects four CGs, so the memory of a CG can also be shared to other CGs. Users can explicitly set the size of each CG's private memory space and the size of the shared memory space. Through NoC, data sharing between four CGs can be implemented without memory data copy, which enables highly efficient CG-level parallelism for communication-intensive problems. Under the sharing mode, the maximum memory bandwidth of four CGs is up to 144GB/s.

2.3 Related Works

A straightforward implementation of the original convolution algorithm involves strong data dependency in the innermost accumulation computation. To improve the parallelism, several optimization methods are proposed, which can be summarized into the following three categories.

- *Time-domain transformation methods* are first introduced in the early phase of CNN optimization research [2, 6, 15]. By expanding convolution operations into matrix multiplications, the performance can be improved with the help of the BLAS on different hardware platforms. However, additional data transformation is required, which either consumes more memory space and extra data copy operations or involves complicated memory address remapping. Therefore, the memory consumption and bandwidth are major problems for time-domain transformation methods, and the overall performance is limited by the performance of BLAS.
- *Frequency-domain transformation methods* can reduce the arithmetic complexity of convolution operations. FFT-based [18, 23] and Winograd's filtering-based [16] convolution algorithms are proposed and perform well in cases with both large and small convolution kernel sizes. Similar to time domain–based methods, additional data transformation, as well as extra memory consumption, is required, and the overall performance is limited by the performance of transformation.
- *Direct convolution optimization methods* can reduce the data dependency by redesigning the convolution algorithm with loop reordering and data blocking, so as to improve the parallelism of the core computation. Instead of relying on existing BLAS or FFT libraries, direct convolution implementations require hardware-oriented optimization methods to take full advantage of the hardware architecture, and therefore the overall performance can approach the peak performance of the processor. Moreover, by carefully designing the

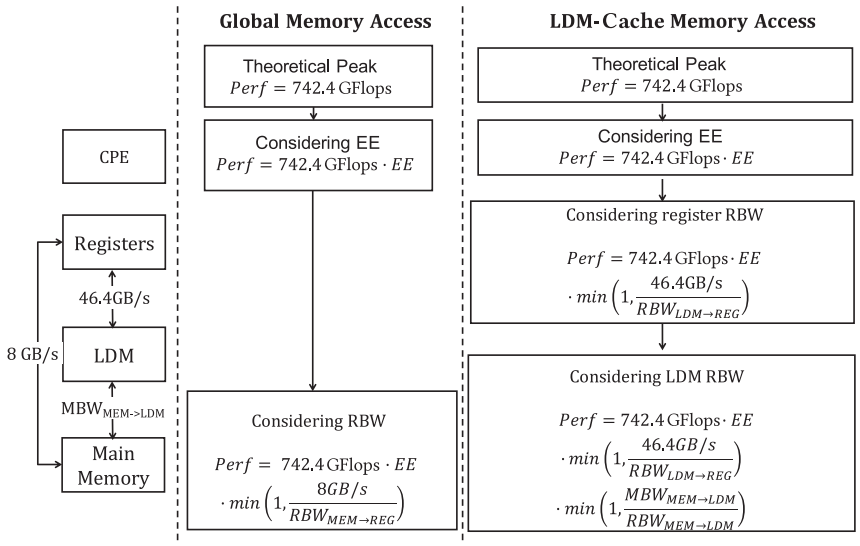


Fig. 2. Performance model for one CG (EE, execution efficiency; RBW, required bandwidth; MBW, measured bandwidth).

data blocking strategies, additional data transformation and extra memory consumption can be avoided, which is more suitable for memory and bandwidth bounded architectures.

In addition to the algorithm optimization, various hardware accelerators are employed to accelerate the convolution computation, such as GPU, FPGA, and ASIC, focusing on both classification and training process of CNNs. FPGAs [19, 25–27] and ASICs [3, 5, 9, 17] are usually used for classification tasks due to the customizability of data precision, low latency, and high energy efficiency. GPUs have currently dominated the competition of the HPC platforms for training tasks. Especially, NVIDIA launched GPU like V100, which includes deep learning specific units such as tensor cores. Correspondingly, the cuDNN [6] library was released to provide highly efficient routines for deep learning algorithms on NVIDIA GPUs and can be neatly integrated to widely used deep learning frameworks such as Caffe [13] and TensorFlow [1].

To explore the potential of training CNNs on other off-the-shelf many-core processors, in this article we present the detailed architecture-oriented optimization methods for the convolution algorithm on the SW26010 processor. Then we show the design of the deep learning library and the customized Caffe framework dedicated for the SW26010 processor, so as to support a highly efficient training process of the CNN on the Sunway TaihuLight supercomputer.

3 CONVOLUTION ALGORITHM OPTIMIZATION

We first introduce a performance model that shows the features of the SW26010 architecture and indicates the key factors that could affect the performance of an implementation. Guided by the performance model, we redesign the convolution algorithm and propose LDM-related, register-related, and instruction-related methods for further optimizations.

3.1 Performance Model

We consider different factors that affect the performance of one CG and propose a performance model shown in Figure 2. The frequency of a CPE is 1.45GHz and the vectorization size is 4.

Assuming that each CPE executes one vector floating-point multiplication and addition (*vfmad*) instruction, the peak performance of a CG can be derived as:

$$2 \times 4 \times 1.45 \times 64 = 742.4\text{GFlops} \quad (1)$$

For an implementation, we define the *execution efficiency* (EE) as the ratio of *vfmad* instructions to the total execution cycles. Therefore, considering the loss from EE, the theoretical performance of an implementation is $742.4\text{GFlops} \cdot EE$.

Before a computing instruction can be executed, we need to make sure that the data has been loaded into registers. For a *vfmad* instruction, 12 double-precision numbers ($12 \times 64 = 768$ bits) are needed. In Figure 2, the *required bandwidth* (RBW) of an implementation is defined as the minimum data access bandwidth that could overlap the data access and computation.

A CPE supports two data access patterns to load the data into registers. One is the global memory access (*gload* instruction), which can load 64 bits of data into a scalar register directly from main memory. In this case, to guarantee the overlapping of computation and data access, the data accessed by a *gload* instruction should be involved in at least 12 *vfmad* instructions (768 bits : 64 bits). Here we define the *computation to data access ratio* (CDR), which represents the ratio of computation instructions (*vfmad*) to data access instructions. In the global memory access pattern, to overlap the computation and data access, the CDR should be greater than 12, which can hardly be met by most algorithms. Therefore, the global memory access pattern is relatively low efficient.

The performance model of the global memory access pattern is shown in Figure 2. The maximum memory bandwidth of one CG is about 8GB/s. We denote the RBW by $RBW_{MEM->REG}$. Here we assume that the computation and the data access are parallel processes and are independent, which can be realized through some optimization methods, such as double buffering (see Section 3.3.2). Therefore, if the $RBW_{MEM->REG}$ is greater than 8GB/s, it will lower the performance by a rate of $\frac{8\,\text{GB/s}}{RBW_{MEM->REG}}$.

The other memory access pattern is to use the LDM as a data cache, which means that the data will be loaded first from the main memory into the LDM and then from the LDM into registers. There are two stages of data accessing in this case. We denote the RBW of the two stages by $RBW_{MEM->LDM}$ and $RBW_{LDM->REG}$. When loading data from the LDM to registers, vectorized load instruction (*vload*) is supported. Each *vload* instruction can load 256-bit (32 Bytes) data into a vector register. The execution of load/store instructions usually takes three or four CPU cycles and is a nonblocking process so that we can issue an instruction every cycle and the bandwidth between the LDM and registers is $32\,\text{Bytes} \times 1.45\,\text{GHz} = 46.4\,\text{GB/s}$.

Data is transferred from main memory to the LDM through the direct memory access interface (DMA), and the theoretical maximum bandwidth of the DMA is 36GB/s. A DMA put/get operation will access one or more *memory blocks*, which has a size of 128Bytes for SW26010. The latency of a DMA request from CPEs is usually more than 100 CPU cycles. Therefore, theoretically, successive DMA operations with large granularity can make full use of the DMA bandwidth. Practically, the actual bandwidth is not a constant value and is variant with the size of continuous memory access blocks of one CPE. We write a microbenchmark on one CG to measure the actual DMA bandwidth and present the results in Table 2, where *Size* indicates the granularity of a DMA operation. We denote the measured DMA bandwidth (MBW) by $MBW_{MEM->LDM}$. We can see that the bandwidth of the DMA ranges from 4GB/s to 36GB/s. In general, a higher bandwidth is achieved when using a block size larger than 256Bytes and aligned in 128Bytes.

Figure 2 also shows the performance model of the LDM-cache memory access pattern. Here the required CDR is 3 (768 bits : 256 bits), which is more easily accomplished compared to the global memory access pattern. Our design is based on the LDM-cache memory access pattern.

Table 2. Measured DMA Bandwidth on One CG(GB/s)

Size(Byte)	*Get*	*Put*	*Size(Byte)*	*Get*	*Put*
32	4.31	2.56	512	27.42	30.34
64	9.00	9.20	576	25.96	28.91
128	17.25	18.83	640	29.05	32.00
192	17.94	19.82	1024	29.79	33.44
256	22.44	25.80	2048	31.32	35.19
384	22.88	24.67	4096	32.05	36.01

According to the performance model, we propose optimization methods to overlap the computation and data access, to increase the $MBW_{MEM->LDM}$, EE, and to reduce the $RBW_{MEM->LDM}$ and $RBW_{LDM->REG}$.

3.2 Algorithm Design

Considering the original algorithm of a convolutional layer (Algorithm 1), the inner loops perform a $K \times K$ convolution. Usually, the value of K is relatively small and is odd, such as 3, 5, 7. Therefore, it is hard to map the inner loops onto the CPE mesh and is also inefficient for the vectorization of core computation.

To improve the parallelism, we reschedule the seven nested loops, making the inner computation to be a matrix multiplication with dimensions N_i, N_o, and B_s, which are relatively large in most convolution layers and are suitable for mapping the inner computation onto the CPE mesh. Algorithm 2 shows the optimized algorithm based on matrix multiplication. We call the inner matrix multiplication operation the *core computation*. To complete the computation of an output matrix (D_o) of size $N_o \times B_s$, each CPE is responsible for a block of size $\frac{N_o}{8} \times \frac{B_s}{8}$. Correspondingly, the input data of a CPE includes a tile of the input matrix W (of size $\frac{N_o}{8} \times N_i$) and a tile of the input matrix D_i (of size $N_i \times \frac{B_s}{8}$), both of which can be shared between the CPEs either in the same row or in the same column. Therefore, for the core computation, the amount of data to be accessed by a CPE is $(N_i \times \frac{N_o}{8} + N_i \times \frac{B_s}{8} + \frac{N_o}{8} \times \frac{B_s}{8})$. The amount of *vmadd* instructions is $(N_i \times \frac{N_o}{8} \times \frac{B_s}{8})/4$. We use *vload* instruction for data access, so the theoretical CDR of the core computation is

$$\frac{(N_i \times \frac{N_o}{8} \times \frac{B_s}{8})/4}{(N_i \times \frac{N_o}{8} + N_i \times \frac{B_s}{8} + \frac{N_o}{8} \times \frac{B_s}{8})/4}. \tag{2}$$

Assuming that N_i, N_o, and B_s have the same value, the CDR can meet the requirement ($CDR \geq 3$) of the LDM-cache pattern when the value is larger than 51, which can be realized in most of the convolution layers. For values that are not a multiple of 8, zero padding can be adopted and will not cause too much decrease in performance. Therefore, for brevity, we focus on the configurations that are a multiple of 8 in the following discussion. The following sections will show the detailed implementation and optimization methods based on Algorithm 2.

3.3 LDM-Related Optimization

LDM-related optimization methods are focused on an effective implementation for outer loops of the algorithm. The targets are to realize the overlap of data access from main memory to the LDM and the core computation of the CPE mesh, so as to increase $MBW_{MEM->LDM}$ and reduce $RBW_{MEM->LDM}$.

ALGORITHM 2: Matrix Multiplication–Based Convolution Algorithm

1: //$IN[B_s][N_i][R_i][C_i]$, $OUT[B_s][N_o][R_o][C_o]$, $CONVW[N_o][N_i][K_r][K_c]$, and $b[N_o]$ are input/output feature maps, convolutional kernels, and bias
2: //$K_r = K_c = K$ represent the number of rows and columns of a two-dimensional convolutional kernel
3: //The output images OUT are initialized with the bias b
4: **for** $cR_o := 0 : 1 : R_o$ **do**
5: **for** $cC_o := 0 : 1 : C_o$ **do**
6: $D_o[0 : N_o][0 : B_s] = (OUT[0 : B_s][0 : N_o][cR_o][cC_o])^T$
7: **for** $cK_r := 0 : 1 : K_r$ **do**
8: **for** $cK_c := 0 : 1 : K_c$ **do**
9: $W[0 : N_o][0 : N_i] = CONVW[0 : N_o][0 : N_i][K - 1 - cK_r][K - 1 - cK_c]$
10: $D_i[0 : N_i][0 : B_s] = (IN[0 : B_s][0 : N_i][cR_o + cK_r][cC_o + cK_c])^T$
11: **Core computation**: $D_o += W \times D_i$
12: **end for**
13: **end for**
14: $OUT[0 : B_s][0 : N_o][cR_o][cC_o] = (D_o[0 : N_o][0 : B_s])^T$
15: **end for**
16: **end for**

3.3.1 Optimized Data Layout. The input data of the core computation is a part of the input/output feature maps and the convolutional kernels. Based on the original data layout, data in W, D_i, D_o is not stored continuously in IN, OUT, and $CONVW$, so the $MBW_{MEM->LDM}$ will be limited due to small data access block. To increase $MBW_{MEM->LDM}$, we redesign the data layout of the input/output feature maps and the convolutional kernels as $IN[R_i][C_i][N_i][B_s]$, $OUT[R_o][C_o][N_o][B_s]$, and $CONVW[K_r][K_c][N_o][N_i]$. In addition, we rotate the convolutional kernels on K_r and K_c dimensions to eliminate the coordinate transform in line 6 of Algorithm 2. For IN and OUT, we put B_s as the lowest dimension, which can eliminate the data transposition in lines 3, 7, and 11 of Algorithm 2, and can support vectorized operations on the B_s dimension in the core computation.

3.3.2 Double Buffering. Double buffering is adopted to overlap the data access from main memory to the LDM and the core computation. Because the DMA is asynchronous, we design two LDM buffers of the same size. While the data in one buffer is used for core computation, the data to be used in next core computation can be loaded into another buffer. Note that the double buffering design halves the maximum available space of the LDM for one computation iteration, which means that for one CPE, only a 32KB LDM is available for the core computation.

3.3.3 LDM Blocking. We consider the total LDM usage of 64 CPEs in the core computation with different convolutional-layer configurations. It can be described as follows:

$$(N_i \times N_o + N_i \times B_s + N_o \times B_s) \times DataLen, \tag{3}$$

where $DataLen$ is the number of bytes for the data type. Assuming that N_i, N_o, and B_s are equal to 256, which are relatively large configurations for most convolutional layers, and the data type is double precision, the total LDM usage of 64 CPEs is $3 \times 256 \times 256 \times 8$ Bytes = 1,536 KBytes. By using register communication techniques, the data stored in one CPE's LDM can be shared to other CPEs (more details will be shown in Section 3.4.1), so the exact LDM usage of each CPE is 1,536 KB/64 = 24 KB. Therefore, for most convolutional layers, a 32KB LDM is enough for the core computation, and in other words, it is possible to take advantage of the remaining LDM spaces to improve the overall performance of the implementation.

ALGORITHM 3: Optimized Algorithm With LDM Blocking

1: //$IN[R_i][C_i][N_i][B_s]$, $OUT[R_o][C_o][N_o][B_s]$, $CONVW[K_r][K_c][N_o][N_i]$, and $b[N_o]$ are input/output feature maps, convolutional kernels, and bias
2: //$K_r = K_c = K$ represent the number of rows and columns of a two-dimensional convolutional kernel
3: //$W, \tilde{W}$ and $D_i, \tilde{D_i}$ represent the double buffering for weight and input feature maps
4: //The output images OUT are initialized with the bias b
5: **for** $cR_o := 0 : 1 : R_o$ **do**
6: **for** $cC_o := 0 : b_C : C_o$ **do**
7: DMA get $D_o[0 : b_C][0 : N_o][0 : B_s] \leftarrow OUT[cR_o][cC_o : cC_o + b_C][0 : N_o][0 : B_s]$
8: **for** $cK_r := 0 : 1 : K_r$ **do**
9: **for** $cK_c := 0 : 1 : K_c$ **do**
10: DMA get:
11: $\tilde{W}[0 : N_o][0 : N_i] \leftarrow CONVW[cK_r][cK_c][0 : N_o][0 : N_i]$
12: $\tilde{D_i}[0 : b_C][0 : N_i][0 : B_s] \leftarrow IN[cR_o + cK_r][cC_o + cK_c : cC_o + cK_r + b_C][0 : N_i][0 : B_s]$
13: **//Core computation**:
14: **for** $cb_C := 0 : 1 : b_C$ **do**
15: $D_o[cb_C] += W \times D_i[cb_C]$
16: **end for**
17: Check DMA get $\tilde{W}, \tilde{D_i}$ finished.
18: Exchange W, D_i with $\tilde{W}, \tilde{D_i}$
19: **end for**
20: **end for**
21: DMA put $D_o[0 : b_C][0 : N_o][0 : B_s] \rightarrow OUT[cR_o][cC_o : cC_o + b_C][0 : N_o][0 : B_s]$
22: **end for**
23: **end for**

In the convolution algorithm, the convolutional kernel is shared by the computation of values in the same output image. In the core computation of Algorithm 2, the data of convolutional kernel (W) is only used for one core computation corresponding to the values in the output feature maps at coordinate (cRo, cCo). To improve the data reuse of W, and in the meantime to improve the CDR of the core computation, we propose an LDM blocking strategy shown in Algorithm 3.

In the core computation of Algorithm 3, we load b_C times more data of input/output feature maps and reuse the data of convolutional kernels to complete b_C matrix multiplication computation. The $RBW_{MEM->LDM}$ is reduced, and the CDR of a CPE is

$$\frac{b_C \times N_i \times \frac{N_o}{8} \times \frac{B_s}{8} / 4}{(N_i \times \frac{N_o}{8} + b_C \times N_i \times \frac{B_s}{8} + b_C \times \frac{N_o}{8} \times \frac{B_s}{8})/4}, \tag{4}$$

which is greater than Equation (2). The larger b_C we choose, the greater CDR we can get. However, b_C is limited by the available size of the LDM, and we can maximize the value to take full advantage of the LDM.

3.4 Register-Related Optimization

Register-related optimization methods mainly focus on effectively mapping the core computation onto an 8×8 CPE mesh. Two key problems are targeted in our work: (i) to realize the register-level data sharing between CPEsto reduce the $RBW_{LDM->REG}$ for each CPE, and (ii) to take full use of the vector register to implement the computation efficiently on a CPE.

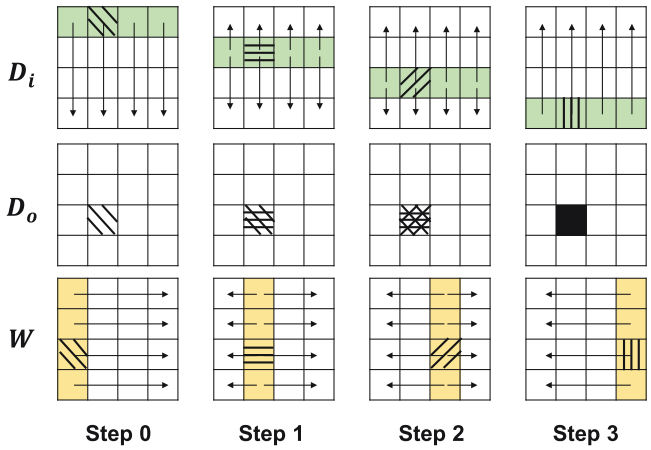


Fig. 3. Register communication example on 4×4 CPE mesh.

3.4.1 Register Communication. In the core computation, a CPE is responsible for a $\frac{N_o}{8} \times \frac{B_s}{8}$ block of D_o, which requires an $\frac{N_o}{8} \times N_i$ tile of W and an $N_i \times \frac{B_s}{8}$ tile of N_i. CPEs in the same row of the mesh share the tile of W, and CPEs in the same row of the mesh share the tile of N_i, which perfectly matches the register communication feature of the CPE mesh. However, there are some limitations of the register communication feature: (i) the send and receive buffers designed for the register communication are simply FIFOs with limited size (4×256 bits); (ii) the data received though the register communication buses has no information of the source CPE; and (iii) if the send and receive buffer are both full, the source CPE will halt.

Considering the limitations, we carefully design a register communication strategy for matrix multiplication computation. For simplicity, we take a 4×4 CPE mesh as an example to introduce the design, shown in Figure 3. We label the CPEs with coordinates $(0,0)$–$(3,3)$ from top left to bottom right. D_i, W, and D_o are divided into 4×4 parts and are labeled as $D_i(0,0)$–$D_i(3,3)$, $W(0,0)$–$W(3,3)$, and $D_o(0,0)$–$D_o(3,3)$. For a given pair of (i,j), the computation of $D_o(i,j)$ can be described as follows:

$$D_o(i,j) += \sum_{k=0}^{3} W(i,k) \times D_i(k,j), \tag{5}$$

which can be done in four steps by CPE(i,j). $D_i(i,j)$, $W(i,j)$, and $D_o(i,j)$ are preloaded into the LDM of CPE(i,j) before executing the core computation. Without loss of generality, we take CPE$(2,1)$ as an example to show the process.

- *Step 0*: First, for all $j \in \{0,1,2,3\}$, CPE$(0,j)$ loads data of $D_i(0,j)$ from the LDM and sends the data to other CPEs in the same column by register communication. Thus, CPE$(2,1)$ can receive the data of $D_i(0,1)$. Then, for all $i \in \{0,1,2,3\}$, CPE$(i,0)$ loads data of $W(i,0)$ from the LDM and sends the data to CPEs in the same row. CPE$(2,1)$ can receive the data of $W(2,0)$. $D_o(2,1)$ can be loaded from the LDM of CPE so that the computation of $D_o(2,1) += W(2,0) \times D_i(0,1)$ can be done.
- *Step 1*: First, CPEs with coordinates $(1,j)$ load data of $D_i(1,j)$ from the LDM and send the data to CPEs in the same column. Then, CPEs with coordinates $(i,1)$ load data of $W(i,1)$ and send CPEs in the same row. Thus, CPE$(2,1)$ can receive the data of $D_i(1,1)$ through

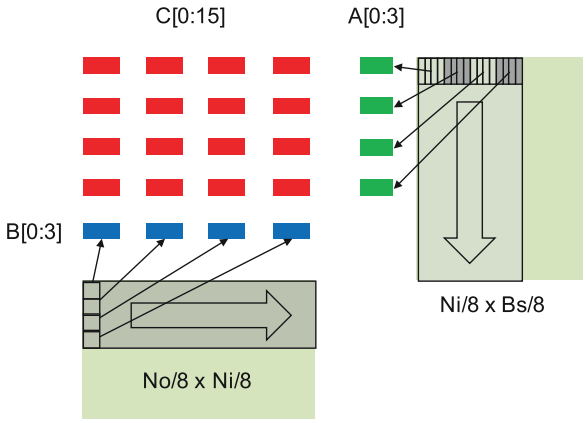


Fig. 4. Register blocking strategy on one CPE.

column register communication, and can load $W(2,1)$ and $D_o(2,1)$ from the LDM, to compute $D_o(2,1) += W(2,1) \times D_i(1,1)$.

- *Step 2*: CPEs with coordinates $(2,j)$ and $(i,2)$ load the data of $D_i(2,j)$ and $W(i,2)$, and send to the same column and same row, respectively. Then CPE$(2,1)$ can receive the data of $W(2,2)$ through row register communication and load $W(2,2)$ and $D_o(2,1)$ from the LDM. The computation of $D_o(2,1) += W(2,2) \times D_i(2,1)$ can be done.
- *Step 3*: Similarly, CPEs with coordinates $(3,j)$ and $(i,3)$ load and send the data of $D_i(3,j)$ and $W(i,3)$, respectively. Correspondingly, CPE$(2,1)$ can receive $W(2,3)$ and $D_i(3,1)$ through row and column register communication, and finally finish the computation of $D_o(2,1) += W(2,3) \times D_i(3,1)$.

Based on the proposed register communication strategy, the core computation can be done on an 8×8 CPE mesh following eight steps with highly efficient data sharing between CPEs.

3.4.2 Register Blocking. In each step of the register communication process, the computation task of a CPE is to calculate the matrix multiplication of $W(i,j)$ and $D_i(i,j)$. The size of the blocks are $(\frac{N_o}{8} \times \frac{N_i}{8})$ and $(\frac{N_i}{8} \times \frac{B_s}{8})$, respectively.

For each CPE, there are only 32 vector registers, including the zero register and the stack pointer (*sp*) register, so the number of available registers is less than 30 for the implementation. We should consider to use vectorized computation to improve the data reuse in registers, and to reduce the data dependency to achieve an efficient instruction flow. Therefore, we propose a register blocking strategy to implement the computation in each step. Figure 4 shows the details.

We use four vector registers to load D_i, denoted by $A[0:3]$, and four vector registers to load W, denoted by $B[0:3]$. In addition, 16 vector registers are used for storing the data of D_o, denoted by $C[0:15]$. We define the following process as a *kernel task* of the register blocking design:

- First, we load 16 values in a row of $D(i,j)$ into A[0:3], which can be done by 4 *vload* instructions. We load 4 values in a column of $W(i,j)$ and duplicate the values to fill B[0:3], which can be done by 4 *vlde* instructions.
- Second, we load 4×16 values from $D_o(i,j)$ into C[0:15] using 16 *vload* instructions.
- Third, for $i,j \in \{0,1,2,3\}$, we calculate Equation (6) using 16 *vfmad* instructions.

$$C[i+4*j] += A[i] \times B[j] \tag{6}$$

(a)

```
InLoop:
1  vload A[0],ptrA,0
2  vload A[1],ptrA,4
3  vload A[2],ptrA,8
4  add ptrA, offsetA, ptrA
     vload A[3],ptrA,12
5  vlde B[0],ptrB,0
6  vlde B[1],ptrB,4
7  vlde B[2],ptrB,8
8  add ptrB, offsetB, ptrB
     vlde B[3],ptrB,12
9  vfmad A[0], B[0], C[0]
10 vfmad A[1], B[0], C[1]
11 vfmad A[2], B[0], C[2]
12 vfmad A[3], B[0], C[3]
13 vfmad A[0], B[1], C[4]
14 vfmad A[1], B[1], C[5]
15 vfmad A[2], B[1], C[6]
16 vfmad A[3], B[1], C[7]
17 vfmad A[0], B[2], C[8]
18 vfmad A[1], B[2], C[9]
19 vfmad A[2], B[2], C[10]
20 vfmad A[3], B[2], C[11]
21 vfmad A[0], B[3], C[12]
22 vfmad A[1], B[3], C[13]
23 vfmad A[2], B[3], C[14]
     cmp cNi, (Ni/8-1)
24 vfmad A[3], B[3], C[15]
     add cNi, 1, cNi
25 cmp cNi, Ni
26 bne InLoop
```

(b)

```
InLoop:
1  vfmad A[0], B[0], C[0]
     vlde B[1],ptrB,4
2  vfmad A[1], B[0], C[1]
     vlde B[2],ptrB,8
3  vfmad A[2], B[0], C[2]
     vlde B[3],ptrB,12
4  vfmad A[3], B[0], C[3]
     add ptrB, offsetB, ptrB
5  vfmad A[0], B[1], C[4]
     add cNi, 1, cNi
6  vfmad A[1], B[1], C[5]
     cmp cNi, (Ni/8-1)
7  vfmad A[2], B[1], C[6]
8  vfmad A[3], B[1], C[7]
9  vfmad A[0], B[2], C[8]
10 vfmad A[1], B[2], C[9]
11 vfmad A[2], B[2], C[10]
12 vfmad A[3], B[2], C[11]
     vlde B[0],ptrB,0
13 vfmad A[0], B[3], C[12]
     vload A[0],ptrA,0
14 vfmad A[1], B[3], C[13]
     vload A[1],ptrA,4
15 vfmad A[2], B[3], C[14]
     vload A[2],ptrA,8
16 vfmad A[3], B[3], C[15]
     vload A[3],ptrA,12
17 add ptrA, offsetA, ptrA
     bne InLoop
```

Fig. 5. Instruction-related optimization for the kernel task.

In addition, 24 registers are used in the kernel task. As we can see from Figure 4, to finish the calculation of 4×16 values of $D_o(i, j)$, $\frac{N_i}{8}$ kernel tasks are required. During this process, A[0:3] and B[0:3] are reloaded for $\frac{N_i}{8}$ times, whereas C[0:15] only need to be loaded once in the first kernel task, which improves the data reuse at register level and thus reduces the $RBW_{LDM->REG}$. Because there is no data dependency between the $vfmad$ instructions in a kernel task, one instruction can be issued in each CPU cycle, which can increase the EE of the implementation.

3.5 Instruction-Related Optimization

We adopt instruction-related optimization methods to overlap the data loading and computation instructions and to further improve the EE in the kernel task. Figure 5(a) shows the instruction flow based on a direct implementation of the kernel task. It takes 26 CPU cycles to issue the instructions, among which there are 16 $vfmad$ instructions. The EE is $16/26 = 61.5\%$. As we can

ALGORITHM 4: 4-CG Implementation of the Convolution Algorithm

```
1: //Assume that IN[R_i][C_i][N_i][B_s], OUT[R_o][C_o][N_o][B_s], CONVW[K_r][K_c][N_o][N_i], and b[N_o] are
   input/output feature maps, convolutional kernels, and bias
2: //K_r = K_c = K represent the number of rows and columns of a two-dimensional convolutional kernel
3: //W, W̃ and D_i, D̃_i represent the double buffering for weight and input feature maps
4: //The output images OUT are initialized with the bias b
5: //Parallel execution on four CGs
6: for cg := 0 : 1 : 4 do
7:   for cR_o := 0 : 1 : R_o/4 do
8:     for cC_o := 0 : b_C : C_o do
9:       DMA get D_o[0 : b_C][0 : N_o][0 : B_s] ← OUT[cg × R_o/4 + R_o][cC_o : cC_o + b_C][0 : N_o][0 : B_s]
10:      for cK_r := 0 : 1 : K_r do
11:        for cK_c := 0 : 1 : K_c do
12:          DMA get:
13:          W̃[0 : N_o][0 : N_i] ← CONVW[cK_r][cK_c][0 : N_o][0 : N_i]
14:          D̃_i[0 : b_C][0 : N_i][0 : B_s] ← IN[cg × R_o/4 + cR_o + cK_r][cC_o + cK_c : cC_o + cK_r + b_C][0 :
             N_i][0 : B_s]
15:          for cb_C := 0 : 1 : b_C do
16:            Core computation: D_o[cb_C]+ = W × D_i[cb_C]
17:          end for
18:          Check DMA get W̃, D̃_i finished.
19:          Exchange W, D_i with W̃, D̃_i
20:        end for
21:      end for
22:    end for
23:    OUT[cg × R_o/4][cC_o : cC_o + b_C][0 : N_o][0 : B_s] = D_o[0 : b_C][0 : N_o][0 : B_s]
24:  end for
25: end for
```

see, in cycles 4, 8, 23, and 24, two instructions can be issued to pipeline P0 and P1 simultaneously, because there is no data dependency and the instructions can be executed on P0 and P1 separately. Only data loading instructions ($vldr$ can load the data into a vector register and send out through row register communication) are issued in the first few cycles, which will lower the EE of the implementation.

Considering that $\frac{N_i}{8}$ kernel tasks are required to calculate a 4×16 block of $D_o(i, j)$, we unroll the $\frac{N_i}{8}$ kernel tasks and reorder the instructions to overlap the $vldr$ instructions of a kernel task with the $vfmad$ instructions at the end of the previous kernel task. The implementation after loop unrolling and instructions reordering is shown in Figure 5(b), where only 17 CPU cycles are required to finish a kernel task and the EE is improved to $16/17 = 94.1\%$.

3.6 CG-Level Parallel Scheme

Based on the preceding optimization methods, the convolution algorithms can be mapped onto a CG efficiently. Considering that there are four CGs in a SW26010 processor, we can further design the parallel scheme for four CGs. The simplest but most efficient way is to introduce parallelism on the outermost loop (R_o). As discussed in Section 2.2, data can be shared by four CGs without extra data copy. Therefore, we can set the data of input/output feature map, convolutional kernel, and bias to the shared mode, and implement a four-CG convolution algorithm as shown in Algorithm 4.

ALGORITHM 5: Configuration Generation Algorithm

```
Test Set 1 : Bs = 128, Ro = Co = 32, K = 3
for No = 64; No <= 384; No+ = 64 do
    for Ni = 64; Ni <= 384; Ni+ = 64 do
        CONV(Bs, Ni, No, Ro, Co, K)
    end for
end for
Test Set 2 : Bs = 128, Ni = No = 128, K = 3
for Ro = Co = 8; Ro <= 128; Ro+ = Ro, Co+ = Co do
    CONV(Bs, Ni, No, Ro, Co, K)
end for
Test Set 3 : Bs = 128, Ni = No = 128, Ro = Co = 64
for K = 3; K <= 11; K+ = 2 do
    CONV(Bs, Ni, No, Ro, Co, K)
end for
Test Set 4 : Ni = No = 128, Ro = Co = 64, K = 128
for Bs = 32; Bs <= 512; Bs∗ = 2 do
    CONV(Bs, Ni, No, Ro, Co, K)
end for
```

3.7 Single-Precision Support

During the preceding design and optimization process, we consider double precision (64 bits) as the data representation for both feature maps and weights. However, unlike in most scientific applications, single precision (32 bits) is sufficient for training CNN models in deep learning applications. Therefore, to support practical CNN training tasks, we further improve our optimized algorithm implementation to support single-precision operations.

Originally designed for supporting major scientific applications that mostly rely on double-precision data types, the features of the SW26010 architecture, such as vectorized instructions and register communication operations, are generally more suitable to handling double-precision operations. There is no special optimization on hardware for single-precision operations on SW26010. Therefore, theoretically, the peak performance for single-precision computation is equal to that for double precision. In practice, there will be performance loss due to the lack of support for single-precision operation in the instruction set, which will be discussed in the following.

A straightforward way to support single precision is to redesign the kernel task instruction flow based on single-precision instructions. The major problem is that there is no instruction like *vldr* or *vlddec* for single-precision data in the instruction set of SW26010. Instead, we should first load four single-precision data into a vector register using the *vlds* or *vldse* instruction, then call register communication using the *putr* or *putc* instruction. Therefore, for single precision, the instruction flow of the kernel task has eight more instructions than the double-precision implementation shown in Figure 5, and more importantly, these instructions cannot be overlapped by computation instructions due to the register dependency. Eight more cycles in the kernel task will lower the EE to $16/(17 + 8) = 64\%$, which indicates that the overall performance loss will be more than 30% (compared to 94.1%).

In the straightforward way, all data accessed in the kernel task requires extra cycles. Considering that there is data reused in the core computation (e.g., W will be reused cb_C times), we propose another way to reduce the overall extra cycles for single-precision data access, called

Table 3. Specifications of SW26010 and the K40m/K80m GPU

Specifications		SW26010	NVIDIA K40m	NVIDIA K80m
Release Year		2014	2013	2014
TDP		250W	235W	375W
Number of Cores		260	2,880 (15 SM[1])	4,992 (26 SM)
Memory	Capacity	32GB	12GB	24GB
	Bandwidth	144GB/s	288GB/s	480GB/s
Peak Perf.	Float	3.02TFlops	4.29TFlops	8.74TFlops
	Double	3.02TFlops	1.43TFlops	2.91TFlops

[1]Each streaming multiprocessor (SM) has 192 CUDA cores.

a *float2double* implementation. After we load the data from the main memory to the LDM, we cast the single-precision data in the LDM to double precision and then do the core computation in double precision. Correspondingly, we cast the double-precision data to single precision before storing the computation results back to the main memory. The data casting can be implemented using a flow of *vlds*/*vsts* (for single-precision) and *vldd*/*vstd* (for double-precision) instructions.

3.8 Evaluation

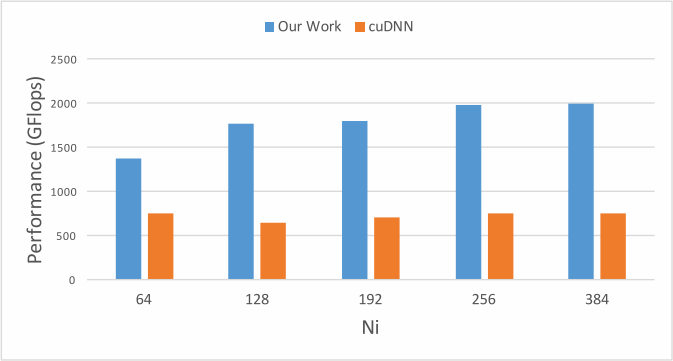
To show the performance improvement obtained from the proposed algorithm design and optimization methods, we first evaluate the performance of the implementation based on double precision.

Different convolutional layer configurations listed in Table 1 will lead to different practical performance. Since the configurations change with CNN models and applications irregularly, it is unnecessary to traverse all possibilities. Therefore, we derive the test cases according to Algorithm 5, where four sets of test cases are generated targeting different values of N_i/N_o, $Ro(Co)$ K, and B_s separately.

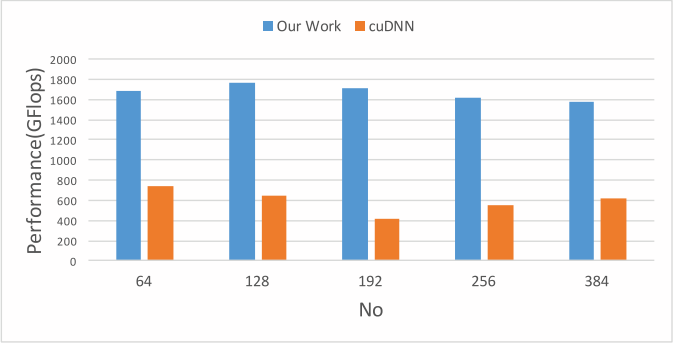
Table 3 lists the specifications of SW26010 and NVIDIA K40/K80 GPUs. Taking the peak performance in both single and double precision into consideration, we choose the K40m GPU as a comparison to SW26010 in our evaluation. We run the test cases using our implementation and the convolution subroutine of cuDNN (v5.1) on the NVIDIA K40m GPU. The evaluation results are summarized into four categories to show how the performance changes with different configurations as shown in Figures 6 and 7.

As we can see from Figure 6(a), the performance of our implementation is more sensitive to the value of N_i. As discussed in Section 3.4.2, in each step of the register communication process, $\frac{N_i}{8}$ kernel tasks are executed. Therefore, larger N_i will lead to a longer process with consecutive kernel tasks, which can provide better performance. Figure 7(a) shows that the performance with a small value of R_o (and C_o) is relatively low, which is because we use a double buffering design to achieve the overlap of the data access from main memory to the LDM and the core computation. The design can be considered as a pipeline, and there is a starting phase at the beginning of the process. Small R_o and C_o will shorten the pipeline and therefore lower the overall performance. The performance with different N_o and different K is relatively stable according to Figure 6(b) and Figure 7(b).

In Figure 7(c), small B_s (e.g., 32 and 64) cannot take full use of the 8×8 CPE mesh, so the performance penalty of the proposed implementation is quite apparent. For large B_s (e.g., 256 and



(a) Average performance of different Ni



(b) Average performance of different No

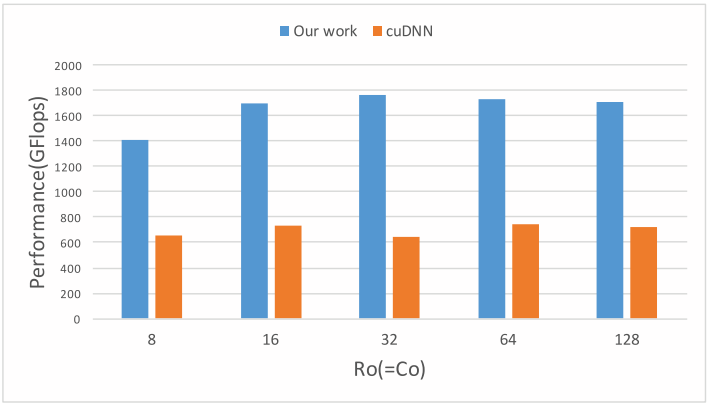
Fig. 6. Performance evaluation on Ni and No (vs. cuDNNv5.1 on the K40m GPU in double precision).

512), the performance improvement is also apparent since large B_s will benefit the performance of the innermost matrix multiplication computation in our design.

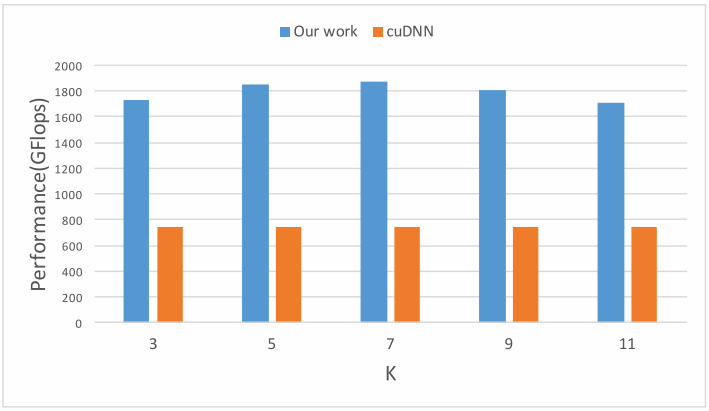
Considering all test cases, the performance of our implementation ranges from 1.3TFlops to 2.0TFlops, and the average performance is about 1.68TFlops, which is about 56% of the peak performance of SW26010. For the evaluation of cuDNN on the K40m GPU, the average performance is about 0.47TFlops. The peak double-precision performance of K40m is 1.43TFlops, so the efficiency of cuDNN is about 32%. Compared to cuDNN, our work can achieve about 3.6 times speedup on performance and about 24% improvement on hardware efficiency.

To illustrate the effectiveness of the optimization methods proposed in this article, we show the performance of the implementations after adopting different optimization methods in Figure 8. In our work, we take the implementation of Algorithm 2 as the basic version and follow the steps of adopting vectorization design, LDM-related optimization, register-related optimization, and instruction-related optimization successively, which forms an optimization process guided by the performance model. Finally, we propose four-CG parallelization design and introduce the implementation based on Algorithm 4. As we can see, in the optimization process, distinct performance improvement can be achieved in each step, and 48 times speedup is achieved in total.

Generally speaking, some of the proposed optimization techniques, such as vectorization, register blocking, and instruction-related optimization, are also applicable to other heterogeneous architectures, such as the GPU and Intel Xeon Phi. In our work, we consider the features of



(a) Average performance of different Ro(Co)



(b) Average performance of different K

(c) Average performance of different Bs

Fig. 7. Performance evaluation on Ro(Co), K, and Bs (vs. cuDNNv5.1 on the K40m GPU in double precision).

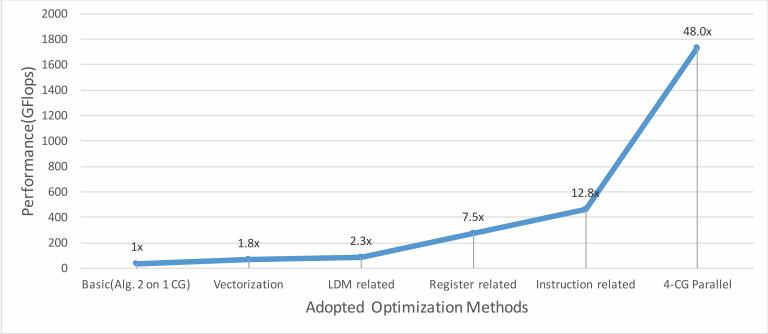


Fig. 8. Performance improvement after adopting different optimization methods.

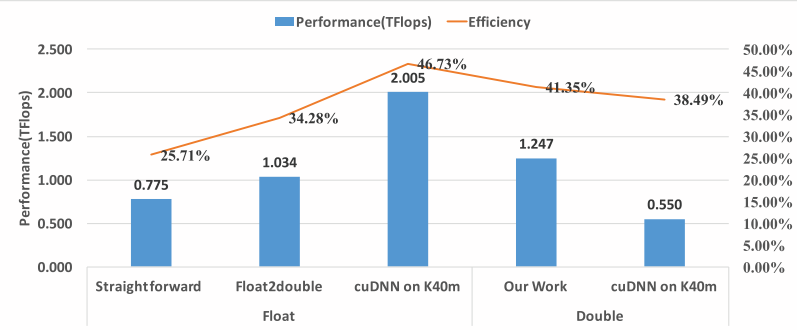


Fig. 9. Average performance and efficiency in float/double precision (training VGG-16 model).

the SW26010 many-core architecture and customize the practical optimization strategies for these general optimization techniques. In addition, other optimization techniques, such as LDM utilization and register communication, are specific to the SW26010 architecture. In summary, both the architecture-specific optimization techniques and the architecture-customized strategies for general optimization techniques are considered as *architecture-oriented* optimization methods, which can also be general for other application and algorithm optimization problems on the SW26010 many-core architecture.

We further evaluate the performance of our convolution implementation based on single-precision data representation, which is generally used in the practical training process of CNN models. As discussed in Section 3.7, a straightforward implementation and a float2double implementation are proposed. In the experiment, we train VGG-16 [21] with both float and double data precision based on our work on SW26010 and cuDNN on K40m. There are 13 convolutional layers with different configurations in VGG-16. We show the average performance and hardware efficiency of the convolutional layers in Figure 9.

Considering the performance on SW26010, the float2double implementation has about 10% improvement on the hardware efficiency over the straightforward implementation. Therefore, we adopt the float2double implementation when training CNN models with single precision. To support more efficient computation in lower data precisions, such as single or half precision, further improvement to the SW26010 architecture is necessary and should target two main aspects. The first is to provide optimized SIMD operations for lower data precisions, such as 8× SIMD in single precision or 16× SIMD in half precision, which can be realized by using the current 256-bit vector

registers and adopting hardware optimization for the ALU part in CPEs. The second aspect is to support more completed low precision instructions (e.g., *vldr* for single/half precision), so as to improve the overlapping of computation and data access (or data transferring).

4 TRAINING PROCESS OPTIMIZATION

Based on the proposed algorithm optimization methods in Section 3, we further design the swDNN library and a customized Caffe framework, called *swCaffe*, which can provide a complete solution to train CNN models on the Sunway TaihuLight supercomputer.

4.1 swDNN Library

Section 3 focused on detailed algorithm and code optimization methods for convolution algorithm, which is the most computational intensively part in a CNN. To provide a high-performance solution for the complete training process of a CNN on the Sunway TaihuLight supercomputer, we further put efforts on optimizing the computation of all kinds of layers with all possible conditions, as well as the backward propagation process to support practical CNN models.

First, we consider different conditions for a convolutional layer. The proposed implementation performs well when the numbers of input and output channels are large enough to assign the tasks to all CPEs. Usually for the first few layers in a practical CNN, the number of channels is small. In this case, the performance of the proposed implementation is poor, so we provide an alternate implementation based on a time-domain transformation method proposed by Jia et al. [13], which contains an Img2Col function to transform the input maps to a matrix and a general matrix-matrix multiplication (GEMM) function to do the computation. The GEMM implementation on SW26010 shares the same core computation with the proposed convolution algorithm, so we can skip over the optimization details for brevity. Different implementations are chosen under different conditions, together to support all kinds of convolutional layers.

The fully connected layer, which is realized through matrix multiplication, involves the second largest amount of computation in a CNN. The implementation is also based on GEMM.

In addition to the computation-intensive layers, such as convolutional and fully connected layers, other layers can be considered as memory-intensive layers, such as pooling layers, normalization layers, and activation function layers. Memory access bandwidth is the key factor that affects the performance of these layers. As shown in Algorithms 1 and 3, the original data layout is (B_s, N_o, R_o, C_o) and the optimized data layout is (R_o, C_o,N_o, B_s). Therefore, as discussed in Section 3.1, we propose parallel implementations using CPEs with task partition along the output channel dimension(N_o), which in both data layout cases can guarantee a successive memory access with large granularity, so as to take fully advantage of the memory bandwidth. Similarly, data transformation operations, such as the data layout and Img2Col transformation can also be accelerated using CPEs.

Usually the output layer of a CNN is a softmax layer. The algorithm of softmax is hard to be parallelized, and the computation amount of the softmax layer is rather small. Therefore, there is no need to design a CPE-based implementation for the softmax layer.

Each iteration of the training process contains a forward process and a backward process. The preceding implementations are focused on the forward process. The output of a forward process is the classification results given by the current model. In the backward process, we first evaluate the *error* of the output results referring to the true labels of the input samples. Then we propagate the error back from the output layer to the input layer and adjust the weights in the layers to minimize the error. In each layer, the backward process shares similar computation patterns with the forward process but involves approximately twofold computation operations for both error propagation and weight update. Therefore, the algorithm design and optimization for the

Table 4. Summary of the swDNN Library

Layers	Conditions	Using CPE	Parallel Strategy	
			1-CG	4-CG
Convolution	Ni and No >= 64	YES	Data Transform + Proposed methods	
	Ni or No < 64	YES	Img2Col + GEMM	On batch size
Fully connected		YES	GEMM	On batch size
Pooling	Max/Min/Avg	YES	On output channel	On batch size
Activation Function	ReLU, Tanh, etc.	YES	On output channel	On batch size
Normalization		YES	On output channel	On batch size
Softmaxr		NO	None (only MPE)	On batch size

backward process of a layer is similar to the forward process but has different input/output data. We implement CPE-based backward process for each layer to provide a highly efficient backward propagation in the training process.

Integrating the preceding implementations for different layers and corresponding data transformation functions, we present a library for accelerating deep neural networks on the SW26010 many-core architecture, called *swDNN*. A summary of the swDNN library is shown in Table 4. For each subroutine in swDNN, we provide two implementations. The basic implementation utilizes one CG of SW26010. For the training process of large CNN models, we provide four-CG parallel implementation to take advantage of the all-shared memory. The four-CG parallel strategy is to adopt the task partition along the outermost dimension of the data. For the convolutional layers with optimized data layout, the outermost dimension is R_o, as introduced in Section 3.6. For other layers with the original data layout, the outermost dimension is batch size (B_s).

4.2 swCaffe Framework

To support more efficient CNN model development and training task deployment, we port Caffe, an open-source deep learning framework, onto the Sunway TaihuLight supercomputer. The original Caffe calls the BLAS library to do the arithmetic computation. On Sunway TaihuLight, swBLAS is one of the fundamental libraries that provide CPE-based implementations on one CG. We consider the Caffe framework depending on swBLAS as the basic version, which has no specialized optimization for the CNN models.

Based on the basic version, we propose three optimization methods to customize the Caffe framework for the SW26010 many-core architecture, and finally we present swCaffe.

First, we implement swDNN-based layers, as listed in Table 4, to substitute for the original implementations of different layers in Caffe.

Second, we add new data transformation layer to swCaffe. In most CNN models, there are consecutive convolutional layers and pooling layers that can be accelerated with optimized data layout, such as the 2nd to 5th convolutional layers in AlexNet [14] and the 2nd to 13th convolutional layers in VGG-16. Here we take the convolutional layers in VGG-16 as examples. If we do data transformation for input/output feature maps and weights in each layer, the data transformation time is about 27% of the total execution time of all convolutional layers, as listed in Table 5. We add a dedicated data transformation layer into swCaffe so that for the consecutive convolutional layers, the data transformation of input/output feature maps is performed only once. The data transformation time is reduced to 16% of the total execution time of all convolutional layers. Specifically, the data transformation time for feature maps is reduced to about one fourth (from 3.16s to 0.73s).

Table 5. Computation and Data Transformation Time of swCaffe With/Without a Data Transformation Layer (One Iteration of Training VGG-16 with $B_s = 128$)

			Data Transformation			
		Computation	Weights	Feature Maps	Total	Total
Without Data Trans. Layer	Time(s)	13.55	1.86	3.16	5.02	18.57
	Percentage	73%	10%	17%	27%	100%
With Data Trans. Layer	Time(s)	13.49	1.83	0.73	2.56	16.05
	Percentage	84%	11%	5%	16%	100%

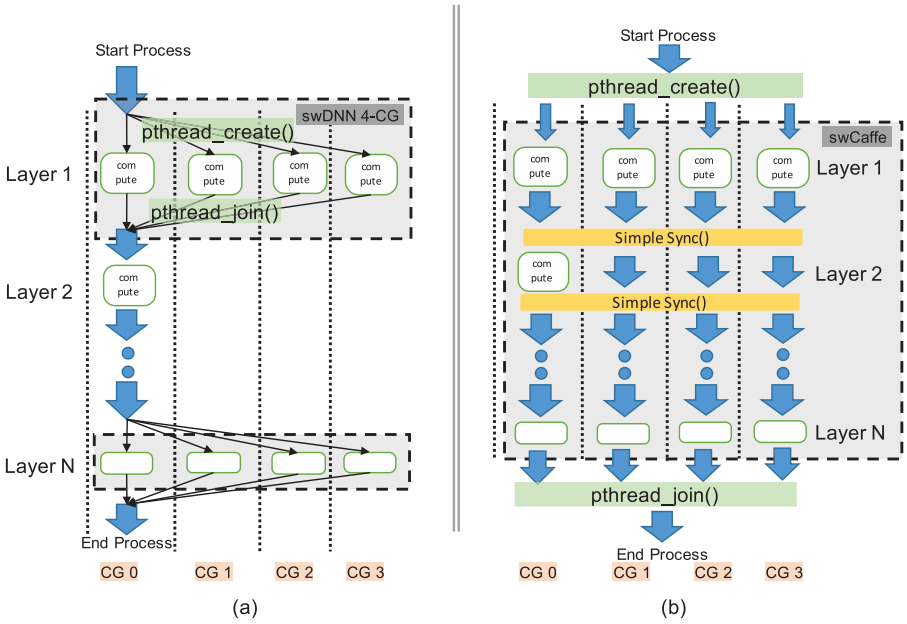


Fig. 10. (a) Caffe using a four-CG implementation in swDNN. (b) Four-CG design for swCaffe.

559 Third, we extend swCaffe to support a four-CG parallel in the complete training process. A
560 straightforward way is to utilize four-CG implementations in swDNN for each layer, and the par-
561 allel process is shown in Figure 10(a). As we can see, the training process is started on CG 0. For
562 layers that have four-CG parallel implementations in swDNN, we call *pthread_create* to start three
563 computing threads on CG1-CG3. After the computation finished, we call *pthread_join* to release
564 the computing threads and continue the process on the main thread. In a CNN model, when most
565 of the layers are based on swDNN, calling pthread_create and pthread_join repeatedly will lead to
566 a relatively large overhead for creating or releasing the thread context.
567 Addressing the preceding problems, we propose a framework-level parallelization design as
568 shown in Figure 10(b). At the beginning of the process, we call pthread_create to start four threads
569 on four CGs, all of which will be activated during the whole training process. For layers that can be
570 implemented in parallel, such as Layer 1 in Figure 10, computation can be done in four CGs without
571 extra overhead. For layers that cannot be implemented in parallel, such as Layer 2, we first call a
572 simple synchronization function to guarantee that all four threads are at the same stage, then do the

ALGORITHM 6: Description of Synchronization Function

```
1: //Initialization
2: set NThread = 4
3: //Signal[NThread] is used to initiate synchronization
4: //Respond[NThread] is used to confirm synchronization
5: for i := 0 : 1 : NThread do
6:     set Signal[i] = 0
7:     set Respond[i] = 0
8: end for
9: //Function definition
10: define Simple_Sync():
11: set thread_id = get_thread_id()
12: if thread_id == 0 then
13:     for i := 1 : 1 : NThread do
14:         set Signal[i] = 1 //initiating synchronization on CG 0
15:     end for
16:     set nRespond = NThread − 1
17:     while nRespond > 0 do
18:         //waiting for the confirmation from CG 1, 2, 3
19:         for i := 1 : 1 : NThread do
20:             if Respond[i] == 1 then
21:                 set nRespond = nRespond − 1
22:                 set Respond[i] = 0
23:             end if
24:         end for
25:     end while
26: else
27:     //waiting for synchronization on CG 1,2,3
28:     while Signal[thread_id]! = 1 do
29:         //waiting for synchronization signal
30:     end while
31:     set Signal[thread_id] = 0
32:     set Respond[thread_id] = 1 //set the confirmation
33: end if
```

computation on CG 0, and finally call the synchronization function again to continue the process on four CGs. Algorithm 6 describes the synchronization function(*Simple_Sync()*), which is based on an handshake (initiation-confirmation) strategy through the semaphore (*Signal*[*NThread*] and *Respond*[*NThread*]) stored in the shared memory.

4.3 Evaluation

The performance of a complete training process is evaluated based on the training VGG-16 model, which is one of the typical and widely used CNN models. To show the performance improvement obtained from different framework-level optimization methods, we train VGG-16 using three versions of Caffe on Sunway TaihuLight, including the following:

- *Caffe-swBLAS*: The basic swBLAS-based Caffe, utilizing only one CG on SW26010.
- *Caffe-swDNN*: Caffe with swDNN-based layer implementations, utilizing four CGs on SW26010.

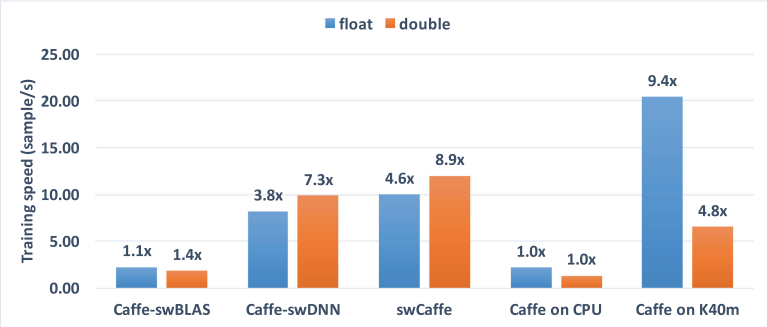


Fig. 11. Performance evaluation of the training VGG-16 model.

- *swCaffe*: swDNN-based Caffe with customized data transformation layers and framework parallelization design for four CGs.

For comparison, we provide the performance of training VGG-16 with Caffe on Intel multicore CPUs (2 × E5-2670v3, 24 cores, with 128GB memory) and the NVIDIA K40m GPU. The training dataset is the ImageNet (ILSVRC) 2012 image classification dataset. We use *sample per second (sample/s)* as the metric to show the average training speed. Results are shown in Figure 11.

As we can see, for the single-precision–based training process, the proposed swCaffe framework can achieve about 4.6 times speedup over the basic swBLAS-based framework, mainly because of the utilization of four CGs. In addition, the optimization targeting data transformation layers and four-CG parallelization can provide about 20% (speedup from 3.8× to 4.6×) performance improvement. Overall, the proposed optimization methods are proven to be effective.

Compared to CPU and GPU results, swCaffe is 4.6 times more efficient than two 12-core CPUs (based on OpenBLAS) and is nearly half the performance of K40m (based on cuDNNv5.1). As a supplement, the performance of the double-precision–based training process is also provided. The double-precision performance of swCaffe is even higher than the single-precision performance on SW26010, and is 8.9 and 1.8 times that of CPUs and the K40m GPU, respectively.

5 CONCLUSIONS

In this article, we present our work on optimizing the CNN on the SW26010 many-core processor. We propose architecture-oriented optimization methods for the algorithm implementation and framework parallelization. Based on the proposed optimization methods, we develop a customized deep learning library (swDNN) and a customized Caffe framework (swCaffe).

Evaluation results show that the proposed optimization methods can bring 48 times performance improvement to the convolution routine in swDNN compared to the basic implementation. The optimized swCaffe framework achieves 4 times performance improvement for the complete training process of the VGG-16 network compared to the original Caffe with swBLAS. Moreover, the proposed convolution routine in swDNN and the swCaffe framework show nearly half the performance of the cuDNN library (on a K40m GPU) in single precision while achieving 3.6 times and 1.8 times speedup over cuDNN (on a K40m GPU) in double precision, respectively.

The presented work can provide highly efficient solutions for training CNN models with the SW26010 many-core processor. Moreover, it proves the capability of deploying large-scale deep learning applications on the Sunway TaihuLight supercomputer.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. arXiv:1603.04467.

[2] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition*.

[3] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGPLAN Notices* 49, 269–284.

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

[5] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, et al. 2014. DaDianNao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 609–622.

[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. arXiv:1410.0759.

[7] Ronan Collobert, Samy Bengio, and Johnny Marithoz. 2002. *Torch: A Modular Machine Learning Software Library*. Idiap.

[8] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. 2011. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio Speech and Language Processing* 20, 1, 30–42.

[9] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. *ACM SIGARCH Computer Architecture News* 43, 92–104.

[10] Jiarui Fang, Haohuan Fu, Wenlai Zhao, Bingwei Chen, Weijie Zheng, and Guangwen Yang. 2017. swDNN: A library for accelerating deep learning applications on Sunway TaihuLight. In *Proceedings of the Parallel and Distributed Processing Symposium*. 615–624.

[11] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, et al. 2016. The Sunway TaihuLight supercomputer: System and applications. *Science China Information Sciences* 59, 7, 072001.

[12] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel Rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, and Tara N. Sainath. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29, 6, 82–97.

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

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[15] Andrew Lavin. 2015. maxDNN: An efficient convolution kernel for deep learning with maxwell GPUs. arXiv:1501.06633.

[16] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.

[17] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A polyvalent machine learning accelerator. *ACM SIGARCH Computer Architecture News* 43, 369–381.

[18] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast training of convolutional networks through FFTs. arXiv:1312.5851.

[19] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, et al. 2016. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, 26–35.

[20] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529, 7587, 484.

[21] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556.

[22] Yi Sun, Xiaogang Wang, and Xiaoou Tang. 2014. Deeply learned face representations are sparse, selective, and robust. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2892–2900.

[23] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. arXiv:1412.7580.

672 [24] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Computer Vision—*
673 *ECCV 2014*. Lecture Notes in Computer Science, Vol. 8689. Springer, 818–833.
674 [25] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accel-
675 erator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium*
676 *on Field-Programmable Gate Arrays*. ACM, New York, NY, 161–170.
677 [26] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. 2016. Energy-efficient CNN implementa-
678 tion on a deeply pipelined FPGA cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics*
679 *and Design*. ACM, New York, NY, 326–331.
680 [27] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. 2016. F-
681 CNN: An FPGA-based framework for training convolutional neural networks. In *Proceedings of the IEEE International*
682 *Conference on Application-Specific Systems, Architectures, and Processors*. 107–114.

683 Received June 2017; revised December 2017; accepted January 2018