# Optimizing Convolutional Neural Networks on Sunway TaihuLight Supercomputer

**Abstract:** Sunway TaihuLight supercomputer is announced in June 2016 and currently ranks the first place on the Top 500 List. The supercomputer is powered by the SW26010, which is a new many-core processor designed with on-chip heterogeneous techniques. In this paper, we present our work on exploring the potential of SW26010 architecture for convolutional neural networks, which is one of the most effective deep learning models and involves a large amount of computations in the training tasks. Based on the characteristics of SW26010 processor, we derive a performance model to identify the most suitable approach of mapping convolutional computations onto the many-core architecture. Optimization methods targeting local directive memory usage, vector register usage and instruction pipelines are proposed for further performance improvement, guided by the performance model. With the proposed design and optimization methods, we manage to achieve a double-precision performance of 1.6 TFlops for the convolution computation, up to 54% of the theoretical peak performance. Compared with cuDNN on NVIDIA Tesla K40m GPU, our work results in 1.9 to 9.7 times speedup on performance and 14% improvement on hardware efficiency according to the evaluation with 132 test cases.

**Key words:** Convolutional Neural Network, Deep Learning, Heterogeneous Many-core Architecture, Sunway TaihuLight Supercomputer

## 1 Introduction

Convolutional neural network (CNN[1]) is one of the most successful deep learning models. The training process of CNNs involves a large amount of computations, and has become one a popular research topics in HPC field. GPUs have currently been considered as the most efficient hardware choice for deep learning tasks, and have widely adopted in both academia and industry. However, with the increasing complexity of CNNs, higher demands are put forward on not only processors and accelerators, but also on systems, such as an customized HPC server or even a cluster, where remains challenges for higher efficient solutions than the state-of-art GPUs and GPU-based HPC platforms.

Sunway TaihuLight[2], a supercomputer that ranks the first in the world with over 100 PFlops computing capacity, is powered by the SW26010 many-core processor, which is designed with on-chip heterogeneous techniques and can provides a peak double-precision performance of 3.06 TFlops. SW26010 introduces a number of unique features that could potentially help the training process of CNNs, such as the user-controlled local directive memory (LDM), the hardware-supported register-level data sharing, and a unified memory space shared by all processing elements.

To explore the potential, in this paper, we present our work on designing and optimizing the convolution neural network based on SW26010 many-core architecture. The major contributions of this work include:

- Based on the characteristics of SW26010, we derive a performance model to identify the most suitable approach of mapping convolutional computations onto the many-core architecture.
- We design LDM usage strategies, including double buffering and LDM blocking, to improve the efficiency and to reduce the required memory bandwidth between main memory and LDM.
- We design register communication and register blocking strategies to take fully use of the vector registers in the computing processing elements and to implement the core computation efficiently.
- Based on the double-pipeline architecture of the computing processing elements, we adopt loop enrolling and instruction re-ordering for the core computation, which improves the execution efficiency of the instruction flow.

An evaluation with 132 test cases is presented and the results show that our implementation can provide an average double-precision performance of about 1.6 TFlops, achieving 54% of

**Algorithm 1 Original algorithm of a convolutional layer**

1: //Assume that $IN[B_s][N_i][R_i][C_i]$, $OUT[B_s][N_o][R_o][C_o]$, $CONVW[N_o][N_i][K_r][K_c]$ 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 initialed 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**

the theoretical peak performance of SW26010. Compared with cuDNN on NVIDIA Tesla K40m GPU, our work results in 1.9 to 9.7 times speedup on performance and 14% improvement on hardware efficiency, which proves the capability of SW26010 processor and Sunway TaihuLight supercomputer to training large scale CNNs.

## 2 Background

### 2.1 Convolutional Neural Networks

CNNs usually contain multiple computing layers, among which convolutional layers usually take the majority of computing time (over 90%) in most of CNNs. Therefore, we focus on the optimization of convolutional computation in this paper.

**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 |

We first give the description of convolutional layer configurations listed in Tab. 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 to 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 $B_s$, then 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 4-dimension tensors and there are 7 nested loops in the algorithm, which provides possibilities for parallel optimization on many-core processors like SW26010.

### 2.2 SW26010 Many-core Architecture

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

The MPE has 32KB L1 instruction cache, 32KB L1 data cache and 256KB L2 cache, and supports the complete interrupt functions, memory management, superscalar, and out-of-order instruction issue/execution. The MPE is good at handling the management, task schedule, and data communications.

The CPE is designed for maximizing the aggregated computing throughput while minimizing the complexity of the micro-architecture. Each CPE has 16-KB L1 instruction cache and 64KB local directive memory (LDM). The LDM can be considered as a user-controlled fast buffer, which allows orchestrated memory usage strategies for different implementations, so that 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

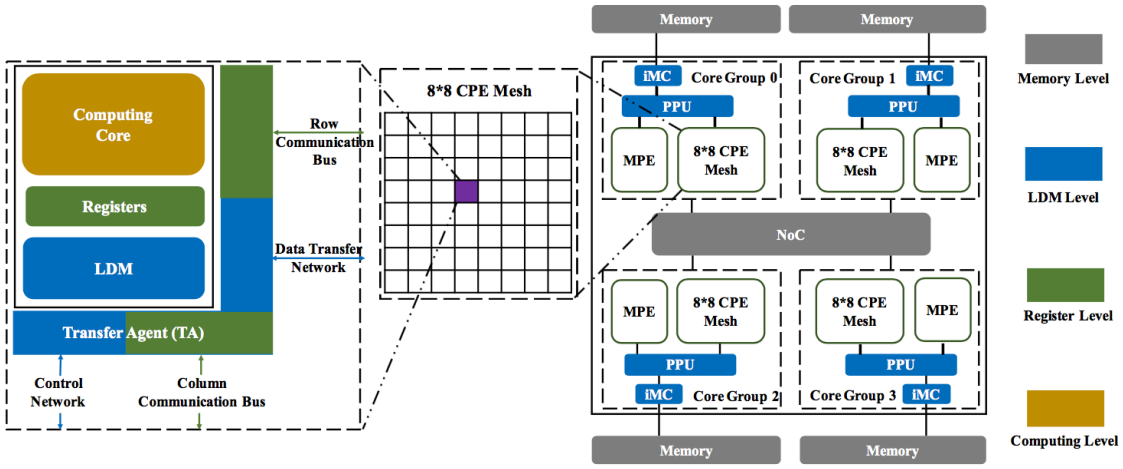


**Fig. 1 SW26010 architecture**

pipelines (P0 and P1). P0 supports scalar and vectorized computing operations of both floating-point and integer, while P1 supports data load/store, compare, jump operations and scalar integer operations. The double-pipeline design provides probability for the overlapping of data accessing and computation operations. Therefore, register level and instruction level optimization can improve the performance of core computation.

Inside the $8 \times 8$ CPE mesh, there is a control network, a data transfer network (connecting the CPEs to the memory interface), 8 column communication buses, and 8 row communication buses. The column and row communication buses enable fast register-level data communication between CPEs in the same column and same row, providing important data sharing and cooperation capability within the CPE mesh.

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 that the memory of a CG can also be shared to other CGs. Users can explicitly set the size of each CGs 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, different optimized implementations are proposed, which can be summarized into the following three categories.

- **Time-domain transformation methods** are first introduced in the early phase of CNN optimizations researches[3–5]. By lowering the convolution operation into matrix multiplication, the performance can be improved with the help of high efficient 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, 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[6, 7] and Winograd's filtering[8] based convolution algorithm are proposed and performs well in cases with both large and small convolution kernel sizes. Similar to time-domain based methods, additional FFT transformation is required as well as extra memory consumption, and the overall performance is limited by the performance of FFT.

- **Direct convolution optimization methods** can reduce the data dependency by re-designing the convolution algorithm with loop reordering and data blocking, so that 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 to the peak performance of the processor. Moreover, by carefully designing the data blocking strategies, additional data transformation and extra memory consumption can be avoided, which is more suitable for memory and bandwidth bounded architectures.

Besides 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[9–11] and ASICs[12–15] 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 has launched several deep learning specific GPUs as well as cuDNN[16] library,

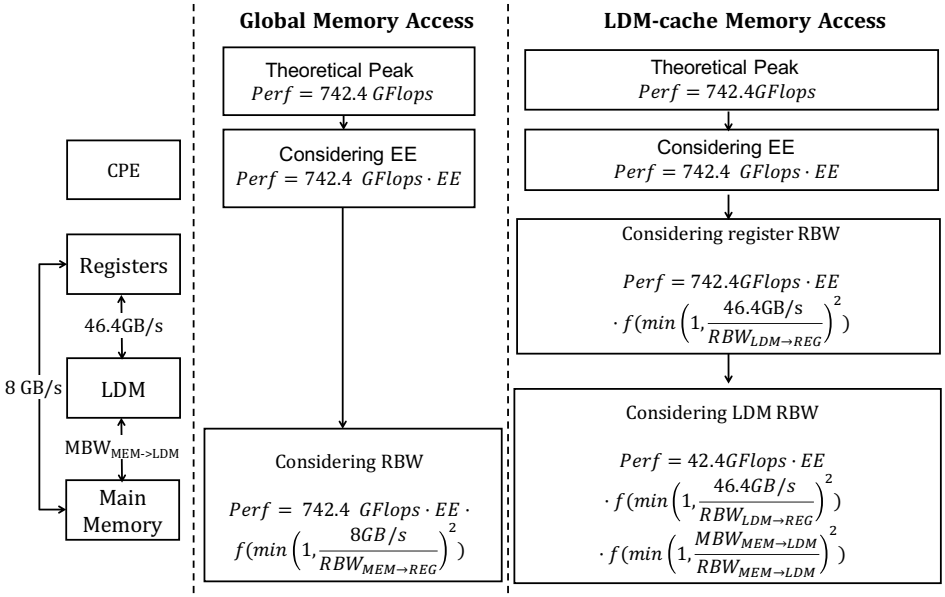


**Fig. 2 Performance model for one CG**

which provides a flexible API for deep learning workloads and is neatly integrated to widely used deep learning frameworks, such as Caffe[17], Tensorflow[18], etc.

To explore the potential of training CNNs on other off-the-shelf many-core processors, in this paper, we present our work on optimizing CNN algorithm on the SW26010 many-core processor. Based on the unique architectural features of SW26010, we propose customized direct convolution algorithm with a series of optimization methods to improve the performance.

# 3 Optimization Methods

## 3.1 Performance model

We consider different factors that affect the actual performance of one CG and propose a performance model shown in Fig. 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:

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

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 $742.4GFlops \cdot EE$.

Before a computing instruction can be executed, we need to make sure the data has been loaded into registers. For a *vfmad* instruction, 12 double-precision floating point numbers ($12 \times 64 = 768bits$) are needed. In Fig. 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 data into registers. One is the global memory access (*gload* instruction), which means the data is directly loaded from the main memory to registers. Each *gload* instruction can load 64 bits data into a scalar register. In this case, to guarantee the computation and data access can be fully overlapped, the data accessed by a *gload* instruction should be involved in at least 12 *vfmad* instructions (768bits : 64bits). Here we define the *computation to data access ratio* (**CDR**), which represents the ratio of computation instructions (*vfmad*) to data access instructions. As we can see, in global memory access pattern, to overlap the computation and data access, CDR should be greater than 12, which can hardly be met by most algorithms. Therefore, global memory access pattern is relatively low efficient. The performance model of global memory access pattern is shown in Fig. 2. The maximum memory bandwidth of one CG is about 8GB/s. We donate the RBW as $RBW_{MEM->REG}$. $f(\cdot)$ is a monotone increasing function with $f(1) = 1$, which describes how the bandwidth limitation affects the performance.

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

Data is transferred from main memory to LDM through direct memory access interface (DMA), and the theoretical maximum bandwidth of DMA is 36GB/s. However, 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 micro-benchmark on one CG to measure the actual DMA bandwidth and present the results in Tab. 2, where $Size$ indicates the size of continuous memory access data block of one CPE. We donate the measured DMA bandwidth as $MBW_{MEM->LDM}$. We can see that the bandwidth of DMA ranges from 4 GB/s to 36 GB/s. In general, a

**Algorithm 2 Matrix-multiplication-based convolution algorithm**

1: //Assume that $IN[B_s][N_i][R_i][C_i]$, $OUT[B_s][N_o][R_o][C_o]$, $CONVW[N_o][N_i][K_r][K_c]$ 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 initialed 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:  **//Core computation**: $D_o += W \times D_i$
10:  $W[0 : N_o][0 : N_i] = CONVW[0 : N_o][0 : N_i][K - 1 - cK_r][K - 1 - cK_c]$
11:  $D_i[0 : N_i][0 : B_s] = (IN[0 : B_s][0 : N_i][cR_o + cK_r][cC_o + cK_c])^T$
12:  **for** $cN_o := 0 : 1 : N_o$ **do**
13:  **for** $cB := 0 : 1 : B_s$ **do**
14:  **for** $cN_i := 0 : 1 : N_i$ **do**
15:  $D_o[cN_o][cB] += W[cN_o][cN_i] \times D_i[cN_i][cB]$
16:  **end for**
17:  **end for**
18:  **end for**
19:  **end for**
20:  **end for**
21:  $OUT[0 : B_s][0 : N_o][cR_o][cC_o] = (D_o[0 : N_o][0 : B_s])^T$
22:  **end for**
23: **end for**

higher bandwidth is achieved when using a block size larger than 256B and aligned in 128B.

**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 |

Figure 2 also shows the performance model of LDM-cache memory access pattern. The required CDR is 3 (768bits : 256bits), which is easier to be accomplished compared with the global memory access pattern. Therefore, our design is based on LDM-cache memory access pattern. According to the performance model, we propose optimization methods to achieve the overlap of computation and data access, to increase the $MBW_{MEM->LDM}$, $EE$ and 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, like 3, 5, 7, etc. 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 parallelism, we re-scheduled the 7 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. To complete the computation of an output matrix of size $N_o \times B_s$ ($D_o$), 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:

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

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

**Algorithm 3 Optimized algorithm with LDM blocking**

1: //Assume that $IN[B_s][N_i][R_i][C_i]$, $OUT[B_s][N_o][R_o][C_o]$, $CONVW[N_o][N_i][K_r][K_c]$ 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 initialed with the bias $b$
4: **for** $cR_o := 0 : 1 : R_o$ **do**
5:  **for** $cC_o := 0 : b_C : C_o$ **do**
6:  $D_o[0 : b_C][0 : N_o][0 : B_s] = OUT[cR_o][cC_o : cC_o + b_C][0 : N_o][0 : B_s]$
7:  **for** $cK_r := 0 : 1 : K_r$ **do**
8:  **for** $cK_c := 0 : 1 : K_c$ **do**
9:  $W[0 : N_o][0 : N_i] = CONVW[cK_r][cK_c][0 : N_o][0 : N_i]$
10:  $D_i[0 : b_C][0 : N_i][0 : B_s] = IN[cR_o + cK_r][cC_o + cK_c : cC_o + cK_r + b_C][0 : N_i][0 : B_s]$
11:  //**Core computation**: $D_o[0 : b_C] += W \times D_i[0 : b_C]$
12:  **for** $cb_C := 0 : 1 : b_C$ **do**
13:  **for** $cN_o := 0 : 1 : N_o$ **do**
14:  **for** $cB := 0 : 1 : B_s$ **do**
15:  **for** $cN_i := 0 : 1 : N_i$ **do**
16:  $D_o[cb_C][cN_o][cB] += W[cN_o][cN_i] \times D_i[cb_C][cN_i][cB]$
17:  **end for**
18:  **end for**
19:  **end for**
20:  **end for**
21:  **end for**
22:  **end for**
23:  $OUT[cR_o][cC_o : cC_o + b_C][0 : N_o][0 : B_s] = D_o[0 : b_C][0 : N_o][0 : B_s]$
24:  **end for**
25: **end for**

and the core computation of the CPE mesh, so that to increase $MBW_{MEM->LDM}$ and reduce $RBW_{MEM->LDM}$.

### 3.3.1 Optimized Data Layout

The input data of the core computation are 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 that the $MBW_{MEM->LDM}$ will be limited due to small data access block. To increase $MBW_{MEM->LDM}$, we re-designed 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 $CONVW[K_r][K_c][N_o][N_i]$. Besides, we rotated the convolutional kernels on $K_r$ and $K_c$ dimensions to eliminate the coordinate transform in line 8 of Algorithm 2. For $IN$ and $OUT$, we put $B_s$ as the lowest dimension, which can eliminate the data transposition in line 9 and 10 of Algorithm 2, and can support vectorized operations on $B_s$ dimension in the core computation.

### 3.3.2 Double Buffering

Double buffering is adopted to overlap the data access from main memory to LDM and the core computation. Because DMA is asynchronized, 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 at next computation iteration can be loaded into another buffer. The double buffering design halves the maximum available space of LDM for one computation iteration, which means for one CPE, only 32KB LDM is available for the core computation.

### 3.3.3 LDM Blocking

We consider the LDM usage of the core computation with different convolutional layer configurations. It can be described as:

$$(N_i \times N_o + N_i \times B_s + N_o \times B_s) \times DataLen \quad (3)$$

where $DataLen$ is the number of bytes for the data type. Assuming $N_i$, $N_o$, and $B_s$ are equal to 256, which are relatively large configurations for most convolutional layers, the LDM usage of each CPE is 24KB. Therefore, for most convolutional layers, 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.

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 matrix multiplication computation corresponding to the values in the output feature maps with 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:

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

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

## 3.4 Register-Related Optimization

Register-related optimization methods are mainly focused on effectively mapping the core computation onto $8 \times 8$ CPE mesh. Two key problems are targeted in our work: (i) to realize the register-level data sharing between CPEs, so that to reduce the $RBW_{LDM->REG}$ for each CPE; (ii) to take fully use of the vector register to implement the computation efficiently on a CPE.

### 3.4.1 Register Communication

As discussed in Section 3.2, a CPE is responsible for a $\frac{N_o}{8} \times \frac{B_s}{8}$ block of $D_o$, and requires an $\frac{N_o}{8} \times N_i$ tile of $W$ and an $N_i \times \frac{B_s}{8}$ tile of $N_i$. For the matrix multiplication computation, 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 limitations of the register communication feature: (i) the send and receive buffers designed for register communication are just FIFOs with limited size ($4 \times 256bits$); (ii) the data received though the register communication buses has no information of which CPE it is from; (iii) if the send buffer and receive buffer are both full, the CPE at the sending end will halt.

Considering the limitations, we carefully design a register communication strategy for the matrix multiplication computation. For simplicity, we take a $4 \times 4$ CPE mesh as an example to introduce the design, shown in Fig. 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 $4 \times 4$ 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:

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

which can be done in 4 steps by CPE$(i,j)$. $D_i(i,j)$, $W(i,j)$ and $D_o(i,j)$ are pre-loaded 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.

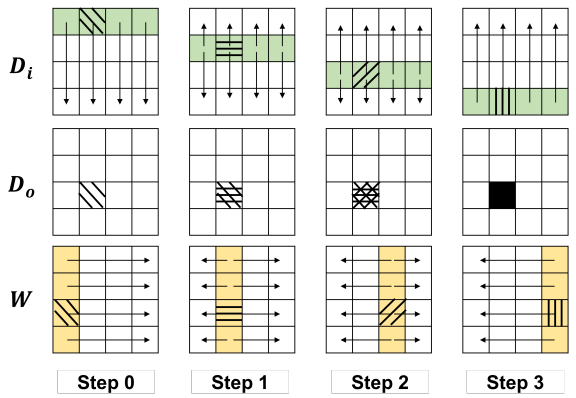


**Fig. 3 Register communication example on $4 \times 4$ CPE mesh**

- **Step 0** First, for all $j \in \{0,1,2,3\}$, CPE$(0,j)$ loads data of $D_i(0,j)$ from LDM and send 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 LDM and send 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$(2,1)$, 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 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 column register communication, and can load $W(2,1)$ and $D_o(2,1)$ from LDM, so that 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 $D(2,j)$ 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 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 $D(3,j)$ 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 $8 \times 8$ CPE mesh following 8 steps, and meanwhile, highly efficient data sharing between CPEs is achieved.

### 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 register, including zero register and stack pointer ($sp$) register, which means the number of available registers is less than 30 for the implementation. Besides, we should consider to use vectorized computation, to improve the data reuse in registers, and to reduce the data dependency in order to achieve efficient instruction flow. Therefore, we propose a register blocking strategy to implement the computation in each step. Figure 4 shows the details.

We use 4 vector registers to load $D_i$, denoted as $A[0:3]$, and 4 vector registers to load $W$, denoted as $B[0:3]$. 16 vector registers are used for storing the data of $D_o$, denoted as $C[0:15]$. 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 val-

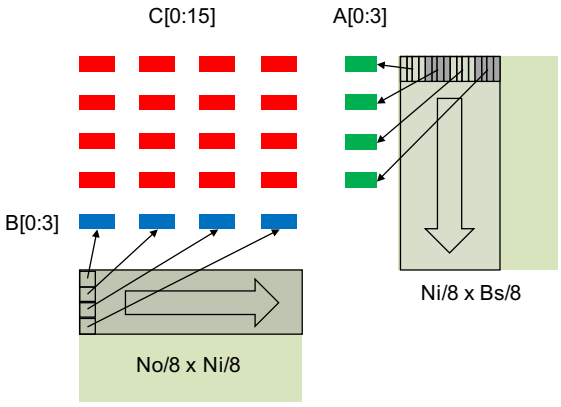


**Fig. 4 Register blocking strategy on one CPE**

ues 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 $4 \times 16$ values in $D_o(i,j)$ into C[0:15], which can be done by 16 $vload$ instructions.
- Third, for $i,j \in \{0,1,2,3\}$, we calculate:

$$C[i+4*j] += A[i] \times B[j] \quad (6)$$

which can be done by 16 $vfmad$ instructions.

24 registers are used in the kernel task. As we can see from Fig. 4, to finish the calculation of $4 \times 16$ 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 while 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 $RBW_{LDM->REG}$. 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 $EE$ 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 see, in cycle 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 issue 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 $4 \times 16$ 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 shows in Fig. 5(b), where only 17 CPU cycles are required to finish a kernel task and the $EE$ is improved to $16/17 = 94.1\%$.

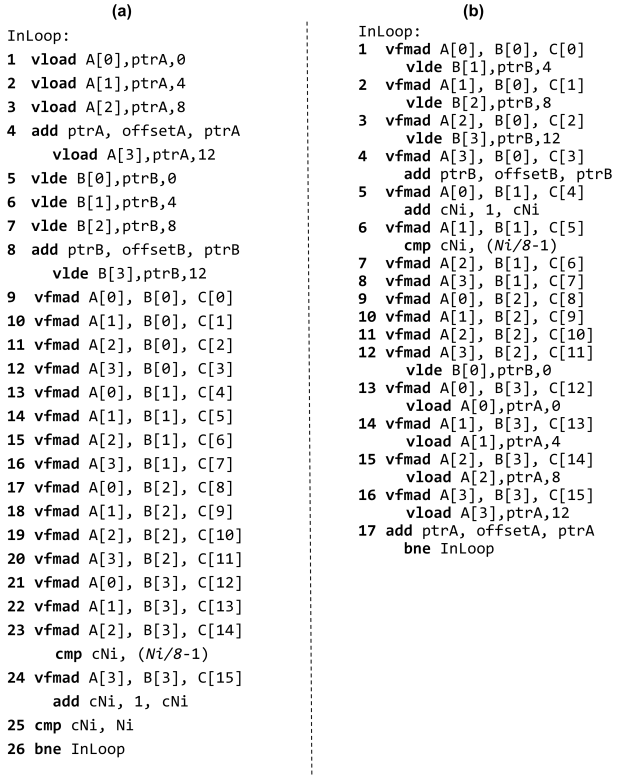


**Fig. 5 Instruction-related optimization for the kernel task**

## 3.6 Core-Group Level Parallel Scheme

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

# 4 Results and Discussion

Because the arithmetic architecture of SW26010 is designed for scientific computation and does not provide an easy doubling or quadrupling of the performance by using single or even half precision, we use double-precision floating-point for evaluation.

Different convolutional layer configurations, including $B_s$ and those listed in Table 1, will lead to different performance of the implementation. Since the configurations change with CNN models and applications, it is unnecessary to traverse all possibilities. Therefore, we derive the test cases according to the following analysis.

First, $R_o$ and $C_o$ range from tens to hundreds in different models and different layers. However, in our implementation, $R_o$ and $C_o$ determine the length of the outer loops, which will not affect the performance of the core computation. Therefore, to show the optimization results on the core computation, we set $R_o = C_o = 64$ as constant configurations in our test cases.

**Algorithm 4 4-CG implementation of convolution algorithm**

1: //Assume that $IN[B_s][N_i][R_i][C_i]$, $OUT[B_s][N_o][R_o][C_o]$, $CONVW[N_o][N_i][K_r][K_c]$ 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 initialed with the bias $b$
4: //Parallel execution on 4 CGs
5: **for** $cg := 0 : 1 : 4$ **do**
6: **for** $cR_o := 0 : 1 : \frac{R_o}{4}$ **do**
7: **for** $cC_o := 0 : b_C : C_o$ **do**
8: $D_o[0 : b_C][0 : N_o][0 : B_s] = OUT[cg \times \frac{R_o}{4} + R_o][cC_o : cC_o + b_C][0 : N_o][0 : B_s]$
9: **for** $cK_r := 0 : 1 : K_r$ **do**
10: **for** $cK_c := 0 : 1 : K_c$ **do**
11: $W[0 : N_o][0 : N_i] = CONVW[cK_r][cK_c][0 : N_o][0 : N_i]$
12: $D_i[0 : b_C][0 : N_i][0 : B_s] = IN[cg \times \frac{R_o}{4} + cR_o + cK_r][cC_o + cK_c : cC_o + cK_r + b_C][0 : N_i][0 : B_s]$
13: **for** $cb_C := 0 : 1 : b_C$ **do**
14: **//Core computation:**
15: $D_o[cb_C][:][:] += W[:][:] \times D_i[cb_C][:][:]$
16: **end for**
17: **end for**
18: **end for**
19: **end for**
20: $OUT[cg \times \frac{R_o}{4}][cC_o : cC_o + b_C][0 : N_o][0 : B_s] = D_o[0 : b_C][0 : N_o][0 : B_s]$
21: **end for**
22: **end for**

$B_s$ is not a configuration of a CNN model, but is related to the training process. Either too large or too small will affect the convergence of the training process. We choose $B_s = 128$ as a constant value, which is feasible for both stand-alone and distributed training.

**Algorithm 5 Configuration generation algorithm of Set 1**

1: $N_i = 128, B_s = 128, R_o = 64, C_o = 64$
2: **for** $N_o = 128; N_o <= 256; N_o += 64$ **do**
3: **for** $K = 3; K <= 21; K += 2$ **do**
4: $CONV(B_s, N_i, N_o, R_o, C_o, K)$;
5: **end for**
6: **end for**

We observe different scenarios for $K$, $N_i$, and $N_o$. In the initial layers of a CNN model, the size of input/output feature maps is large. In order not to involve too much computation and memory consumption, $K$ is usually set to a relatively large value (such as 11), and $N_i$, $N_o$ are relatively small (such as 64). For the following layers , the sizeof input/output feature maps is small, thus $K$ goes smaller (e.g. 3), and $N_i$, $N_o$ are getting larger (e.g. 128, 256 or 384), so that to provide more features for the classification layer. From the above, we generate two sets of test cases using Algorithm 5 and Algorithm 6, considering the practical configurations for $K$, $N_i$, and $N_o$. For comparison, we run the test cases with both our implementation on SW26010 processor and the convolution subroutine of cuDNN(v5.1) on NVIDIA K40m GPU.

Algorithm 5 generates **Set 1**, which is designed to show the performance with different values of $K$. The results are shown in Fig. 6, where the test cases are numbered as **1** to **30** following the order that generated.

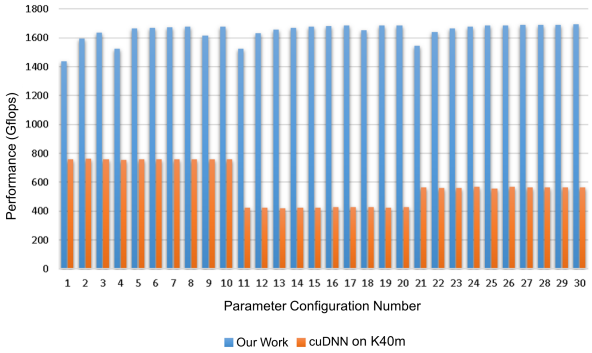


**Fig. 6 Double-precision performance results of our implementation for different filter sizes ranging from $3 \times 3$ to $21 \times 21$, compared with the K40m GPU with cuDNNv5.**

Algorithm 6 generates **Set 2**, which is designed to show the performance with different $N_i$ and $N_o$. The results are shown in Fig. 7 and the test cases are numbered as **1** to **102**, among which, No. 1 to No. 21 have smaller $N_i$ and $N_o$ values, and No. 22 to No. 102 have larger $N_i$ and $N_o$ values.

As we can see from Fig. 6 and Fig. 6, the performance of our implementation decreases a little when $N_i$, $N_o$ and $K$ are small, but in general, it is relatively stable under different parameter configurations. The average double-precision performance is around 1.6 TFlops, which is about 54% of the peak performance of SW26010 processor. As a contrast, the performance of cuDNN changes appreciably with different values of $N_o$, but is not very sensitive to $N_i$ and $K$, which means the optimization

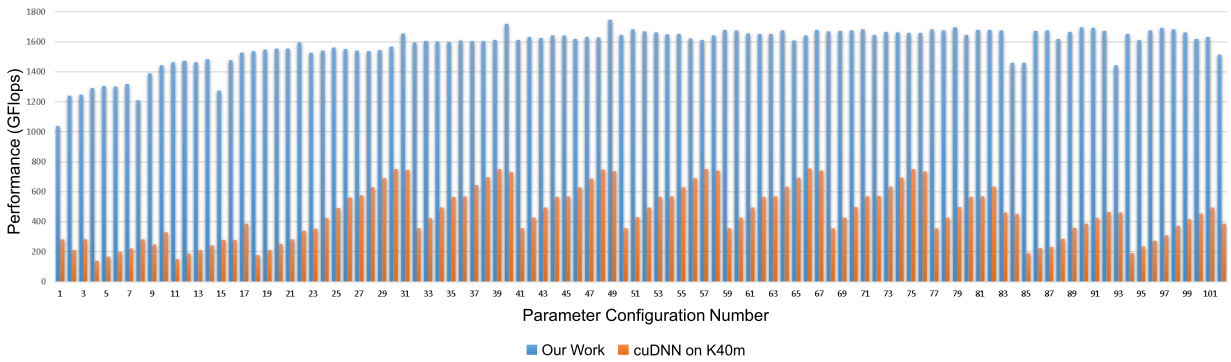


**Fig. 7 Double-precision performance results of our convolution kernels with different (Ni,No) ranging from (64,64) to (384, 384), compared with the K40m GPU results with cuDNNv5.**

**Algorithm 6 Configuration generation algorithm of Set 2**

$K = 3, B_s = 128, R_o = 64, C_o = 64$
Test case No. 1 to No. 21
**for** $N_i = 64; N_i <= 128; N_i += 32$ **do**
  **for** $N_o = 64; N_o <= 256; N_o += 32$ **do**
    $CONV(B_s, N_i, N_o, R_o, C_o, K)$;
  **end for**
**end for**

Test case No. 22 to No. 102
**for** $N_i = 128; N_i <= 384; N_i += 32$ **do**
  **for** $N_o = 128; N_o <= 384; N_o += 32$ **do**
    $CONV(B_s, N_i, N_o, R_o, C_o, K)$;
  **end for**
**end for**

methods related to the loop of $N_o$ are adopted by the cuDNN implementation. The average double-precision performance of cuDNN on K40m is less than 600GFlops, which is around 40% of the peak performance (1.43TFlops). Therefore, compared with cuDNN on K40m, our implementation can achieve 1.9 to 9.7 times speedup on performance and about 14% improvement on hardware efficiency.

## 5 Conclusions

In this paper, we present our work on optimizing the convolutional neural network on SW26010 many-core processor, which is designed with on-chip heterogeneous techniques and is adopted in the new announce Sunway TaihuLight supercomputer. We first derive a performance model based on the characteristics of SW26010 processor. Then we re-designed the algorithm of the convolutional operation to efficiently map the computations onto the many-core architecture. To further explore an optimized implementation, we propose optimization methods targeting LDM usage, vector register usage and instruction pipeline, guided by the performance model. With the proposed design and optimization, we manage to achieve an average double-precision performance of 1.6 TFlops (54% of the peak performance) under 132 different test cases. Compared with cuDNN on NVIDIA Tesla K40m GPU, our work results in 1.9 to 9.7 times speedup and about 14% improvement of efficiency. Our work provides possibilities for training large convolutional neural networks on Sunway TaihuLight supercomputer, taking advantage of the strong computation capabilities and high efficient distributed data communication.

## References

[1] Krizhevsky, A., Sutskever, I., Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

[2] Haohuan Fu, Junfeng Liao, et al. The sunway taihulight supercomputer: system and applications. Science China Information Sciences, pages 116, 2016.

[3] Kumar Chellapilla, Sidd Puri, Patrice Simard, et al. High performance convolutional neural networks for document processing. In Workshop on Frontiers in Handwriting Recognition, 2006.

[4] Sharan Chetlur, Cliff Woolley, et al. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759, 2014.

[5] Andrew Lavin. maxdnn: an efficient convolution kernel for deep learning with maxwell gpus. arXiv:1501.06633, 2015.

[6] Nicolas Vasilache, Jeff Johnson, at al. Fast convolutional nets with fbfft: A gpu performance evaluation. arXiv preprint arXiv:1412.7580, 2014.

[7] Michae l Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. CoRR, abs/1312.5851, 2013.

[8] Andrew Lavin. Fast algorithms for convolutional neural networks. arXiv preprint arXiv:1509.09308, 2015.

[9] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, et al. Opti- mizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA In- ternational Symposium on Field-Programmable Gate Arrays, pages 161170. ACM, 2015.

[10] Jiantao Qiu, Jie Wang, et al. Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 2635. ACM, 2016.

[11] Chen Zhang, Di Wu, Jiayu Sun, at al. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In Proceedings of the 2016 International Symposium on Low Power Electronics and Design, pages 326331. ACM, 2016.

[12] Tianshi Chen, Zidong Du, et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In ACM Sigplan Notices, volume 49, pages 269284. ACM, 2014.

[13] Yunji Chen, Tao Luo, Shaoli Liu, et al. Dadiannao: A machine-learning supercomputer. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchi- tecture, pages 609622. IEEE Computer Society, 2014.

[14] Daofu Liu, Tianshi Chen, et al. Pudiannao: A polyvalent machine learning accelerator. In ACM SIGARCH Computer Architecture News, volume 43, pages 369381. ACM, 2015.

[15] Zidong Du, Robert Fasthuber, et al. Shidiannao: shifting vision processing closer to the sensor. In ACM SIGARCH Computer Architecture News, volume 43, pages 92104. ACM, 2015.

[16] Sharan Chetlur, Cliff Woolley, et al. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759, 2014.

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, et al. Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia, pages 675678. ACM, 2014.

[18] Martn Abadi, Ashish Agarwal, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467, 2016.