swDNN: A Library for Accelerating Deep Learning Applications on Sunway TaihuLight

Jiarui Fang*†‡, Haohuan Fu*‡, Wenlai Zhao*†‡, Bingwei Chen*†‡, Weijie Zheng*†‡, Guangwen Yang*†‡
†*Department of Computer Science & Technology, Tsinghua University*
**Ministry of Education Key Lab. for Earth System Modeling, Department of Earth System Science, Tsinghua University*
‡*National Supercomputing Center in Wuxi*

***Abstract*—To explore the potential of training complex deep neural networks (DNNs) on other commercial chips rather than GPUs, we report our work on swDNN, which is a highly-efficient library for accelerating deep learning applications on the newly announced world-leading supercomputer, Sunway TaihuLight. Targeting SW26010 processor, we derive a performance model that guides us in the process of identifying the most suitable approach for mapping the convolutional neural networks (CNNs) onto the 260 cores within the chip. By performing a systematic optimization that explores major factors, such as organization of convolution loops, blocking techniques, register data communication schemes, as well as reordering strategies for the two pipelines of instructions, we manage to achieve a double-precision performance over 1.6 Tflops for the convolution kernel, achieving 54% of the theoretical peak. Compared with Tesla K40m with cuDNNv5, swDNN results in 1.91-9.75x performance speedup in an evaluation with over 100 parameter configurations.**

***Keywords*-Deep Neural Network, Convolutional Neural Network, Deep learning, Many-core Architecture**

I. Introduction

Originated from the original concept proposed in the 1980s, various deep neural networks (DNN) have proven their effectiveness in a number of application domains. Starting from an automated recognition of images [1][2][3] and audios [4][5], the recent technology innovations have further expanded the territory to the some more challenging domains, such as TV games [6], go [7], and driverless cars [8]. With the problems involving more complicated scenarios and the demand for a better accuracy, both the complexity and the depth of the DNNs have been continuously increasing, from tens of layers in the early competitors in ImageNet to the current hundreds of layers. The increase in both the number of parameters and the depth leads to a combined explosion of the parameter space that we need to explore in the training process, thus demanding even more computing power for training a better machine-based intelligence.

One direct result of the increasing complexity of DNNs and the accompanies increasing demand for computing power, is the increasing adoption of large-scale GPU clusters in almost all the leading companies in the corresponding domain ([9][10]). While there are still algorithmic difficulties for scaling the training process of one huge network to the entire cluster with thousands of GPUs, the high density arithmetic units on the GPUs do help a lot in the training process of various DNNs. Therefore, architecture-wise, NVIDIA's GPU cards still seem the only commercial option on the current market. Although there have already been a lot of alternative architectures that demonstrate their potential either as high-recognized research papers (DianNao [11], DaDianNao [12], PuDianNao [13], ShiDianNao [14]), or secret weapons of current dominating players (google TPU), we still have not seen any strong off-the-shelf competitors in the arena of DNN hardware.

Sunway TaihuLight, a supercomputer that ranks the first in the world [15] with over 100 Pflops computing capacity, is powered by a new SW26010 many-core processor. Providing a peak double-precision performance of 3.06 Tflops with a power consumption of only 300 watts, SW26010 has made TaihuLight not only the fastest but also the greenest supercomputer in the world. In addition to its exceptional performance and power efficiency, SW26010 also introduces a number of unique features that could potentially help the training process of DNNs, such as the on-chip fusion of both management cores and computing core clusters, the support of a user-controlled fast buffer for the 256 computing cores, hardware-supported schemes for register communication across different cores, as well as a unified memory space shared by the four core groups (each including 65 cores).

To provide an alternative platform for parallel DNN training, as well as to explore various architectural features that could lead to DNN designs with different efficiencies, we build this library called swDNN to accelerate deep learning applications (especially focused on the training part) on Sunway TaihuLight. Our current work focuses on convolutional neural network (CNN), which is one of the most widely used DNN in various application scenarios, and will expand to other forms of DNNs at a later stage. Our major contributions in this work includes:

1) based on the analysis of the DNN algorithm and the SW26010 architecture, we derive a performance model that not only demonstrates the major factors that could boost or limit the resulting performance, but also guides us to a number of most suitable mappings of the algorithm to the architecture for different problem

scenarios;
2) a customized register communication scheme that targets at maximizing the data reuse in the convolution kernels, which reduces the memory bandwidth requirement for almost an order of magnitude, and pushes the performance to a next level;
3) a careful design of the most suitable pipelining of instructions that reduces the idling time of computation units by maximizing the overlap of memory operation instructions and computation instructions, thus maximizing the overall training performance on SW26010.

After a systematic exploration of all these unique hardware features of SW26010, our optimized swDNN framework, at the current stage, can provide a double-precision performance of over 1.6 Tflops for the convolution kernels, achieving over 50% of the theoretical peak. The significant performance improvements achieved from a careful utilization of SW26010s architectural features and a systematic optimization process demonstrate that these unique features and corresponding optimization schemes are potential candidates to be included in future DNN architectures as well as DNN-specific compilation tools. Our source code is available at [16].

II. Related Works

Training deep neural networks usually demands a huge amount of computing resources and is extremely time and energy consuming. Many efforts have been made by researchers from both academic and industrial communities to accelerate the training task targeting from the core computing kernels to the entire training process, based on high-performance computing platforms, such as those with heterogeneous accelerators like GPUs, FPGAs, and even customized ASICs.

GPUs have currently dominated the competition of the HPC platforms for DNN training. NVIDIA cuDNN [9] library provides a flexible API for deep learning workloads, and it is neatly integrated to widely used deep learning frameworks, such as Caffe [17], Tensorflow [18], etc. Other works like maxDNN [19], Caffe con Troll (CcT) [20], fbfft [21] and Winograd's minimal filtering algorithms [22] for CNN are focused on specific GPU architecture or specific algorithm design and can achieve better performance in certain cases.

FPGA-based accelerators can also provide solutions with high performance as well as high power efficiency. Works in [23], [24] proposed optimized design for the convolutional kernel which achieves considerable performance with single FPGA. To explore high performance, works in [25] and [26] scale the design to multi-FPGA platforms. However, even though higher power efficiency can be achieved, the overall performance of FPGAs is limited by the total amount of hardware computation resources.

Besides general programmable accelerators, customized ASIC for machine learning and deep learning algorithms is another research hot-pot and demonstrates attractive performance and energy efficiency on both classification and training tasks. DianNao [11] emphasized the impact of memory, performance and energy, designed an accelerator for the large scale CNN and DNN, which achieved 452 GOPS throughput in a small area with low power consumption. DaDianNao [12] introduced a multi-chip architecture for machine learning which is 460.65$\times$ faster than a single GPU. PuDianNao [13] accommodated other six representative machine learning techniques along with deep neural networks. Focusing on visual recognition, the accelerator ShiDianNao [14] performed 30$\times$ faster than high-end GPUs.

While we see great potential in both performance and power efficiency for FPGA and customized ASIC based DNN solutions, GPU still remains the only commercial option that provides training performance at the scale of tera-flops per chip. To investigate the performance potential of running and training DNNs on other off-the-shelf many-core chips, in this work, we explore the possibility to support DNN applications (with a specific focus on CNNs) on the newly-annoucned SW26010 processor. Guided by a neat performance model, we manage to identify the most suitable orgnization of loops, blockings of data items, and sequence of two pipelines of instructions, and achieve a performance of 1.6 Tflops in double precision for the convolution kernels. The results demonstrate the strong capability of SW26010 for performing DNN-related computations, and also the benefits brought by SW26010's unique architectural features.

III. Mapping CNN to SW26010: A Performance Model

A. CNN (Convolutional Neural Networks)

CNNs usually contain multiple computing layers, and these layers can be divided as the extractor and the classifier according to their different functions. The extractor layers, such as convolutional layer and subsampling layer, filter the high dimensional input images into various features. The classifier layers, such as fully connected artificial neural network and SVM, use these low dimensional features to decide the categories input images belong to, or calculate the likelihood of each possible category.

In CNN, large data is utilized for the training of the connected weights and the filters, and the result of the recognition on new data is obtained by the forward process of the trained networks. Therefore, due to its large computing requirement, training is a more suitable scenario for supercomputers.

For the convenience of statement, the corresponding parameters of convolutional layer are collected in Table I. The input is N_i images of size $C_i \times R_i$, and the output contains N_o images of size $C_o \times R_o$. For each input image and each output, they are connected by a convolutional filter W with

$K_c \times K_r$ size. The pseudo code of a convolutional layer can be written as that in Listing 1. In most of CNNs, the convolution operator takes the majority of computing time (over 90%). This paper will focus on the implementation on the convolution operator.

Table I: Parameters of convolutional layers

Parameter	Meaning
N_i	Number of input feature maps
N_o	Number of output feature maps
R_i	Height of input image
C_i	Width of input image
R_o	Height of output image
C_o	Width of output image
K_r	Height of filter kernel
K_c	Width of filter kernel

Listing 1: Pseudo code of a convolutional layer

```
for(cB = 0; cB < B; ++cB)
 for(cCo = 0; cCo < Co; ++cCo)
  for(cRo = 0; cRo < Ro; ++cRo)
   for(cNi=0; cNi < Ni; ++cNi)
    for(cNo = 0; cNo < No; ++cNo)
     for(cKr = 0; cKr < Kr; ++cKr)
      for(cKc = 0; cKc < Kc; ++cKc)
       out[cRo][cCo][cNo][cB] +=
        in[cRo+cKr][cCo+cKc][cNo][cB]
       *filter[cKc][cKr][cCo][cRo];
```

B. The SW26010 Many-Core Processor

As mentioned above, the world-leading performance and efficiency of Sunway TaihuLight is mainly enabled by China's homemade SW26010 many-core processor [15]. As shown in Fig. 1, each processor consists of four *core groups (CGs)*. Each CG includes 65 cores: one *management processing element (MPE)*, and 64 *Computing Processor Element (CPEs)*, organized as an 8 by 8 mesh. The MPE and CPE are both complete 64-bit RISC cores but serve different roles during the computation. The MPE, supporting the complete interrupt functions, memory management, superscalar, and out-of-order issue/execution, is good at handling the management, task schedule, and data communications. The CPE is designed for the purpose of maximizing the aggregated computing throughput while minimizing the complexity of the micro-architecture.

Each CG connects to its own 8GB DDR3 memory through the *Memory Controller (MC)*, shared by the MPE and the CPE mesh. The on-chip network (NoC) connects four CGs with *System Interface (SI)*. Memory of four CGs are also connected through the NoC. Users can explicitly set the size of each CG's private memory space, and the size of the memory space shared among the four CGs.

Compared with the other multi-core or many-core processors, the SW26010 design demonstrates a number of different features: **(i)** As for the memory hierarchy, while the MPE adopts a more traditional cache hierarchy (32-KB L1 instruction cache, 32-KB L1 data cache, and a 256-KB L2 cache for both instruction and data), each CPE only provides a 16-KB L1 instruction cache, and relies on a 64KB *Local directive Memory (LDM)* (also known as *Scratch Pad Memory (SPM)*) as a user-controlled fast buffer. This user-controlled 'cache', while increases the programming challenges for an efficient utilization of the fast buffer, provides the option to implement a customized buffering scheme that can improve the overall performance significantly in certain cases. **(ii)** Inside each CPE mesh, we have a control network, a data transfer network (connecting the CPEs to the memory interface), 8 column communication buses, and 8 row communication buses. The 8 column and row communication buses enable fast register communication channels across the 8 by 8 CPE mesh, providing an important data sharing capability at the CPE level. **(iii)** Each CPE includes two pipelines ($P0$, and $P1$) for the instruction decoding, issuing, and execution. $P0$ is for floating-point operations, and both floating-point and integer vector operations. $P1$ is for memory-related operations. Both $P0$ and $P1$ support integer scalar operations. Therefore, identifying the right form of instruction-level parallelism can potentially resolve the dependences in the instruction sequences, and further improve the computation throughput.

C. The challenges for mapping CNN to SW26010

According to definition of basic convolution, there are two major approaches to implement multi-channel convolution operations. One is the spatial-domain based methods that directly sum up the products of input image pixel values with corresponding filter elements to obtain output pixel values [22]. In addition, the summation operations can be organized into General Matrix-Multiplication (GEMM) by lowering the convolutions into a matrix multiplication [9], [19]. The other one is the frequency-domain based methods that can be finished with dot product operations after transforming the input images and filter kernels from spatial domain to frequency-domain with FFT operators[21].

As the FFT used in frequency-domain based methods has higher requirements for the memory bandwidth and involves global communication from different processing threads, the spatial-domain based methods seem a better fit to the SW26010 many-core architecture. Therefore, in this work, we design our memory access and computing patterns for convolution operation of CNNs based on spatial-domain method.

According to the above analysis of both the algorithm and the architecture, we can identify the following major factors that may limit the performance of CNN on SW26010: **(i)** The relatively low memory bandwidth, especially when compared with the high computing performance. The DDR3 memory interface provides a peak bandwidth of 36 GB/s for each CG (64 CPEs), a total bandwidth of 144 GB/s for the entire processor. The NVIDIA K80 GPU, with a similar double-precision performance of 2.91 Tflops, provides an

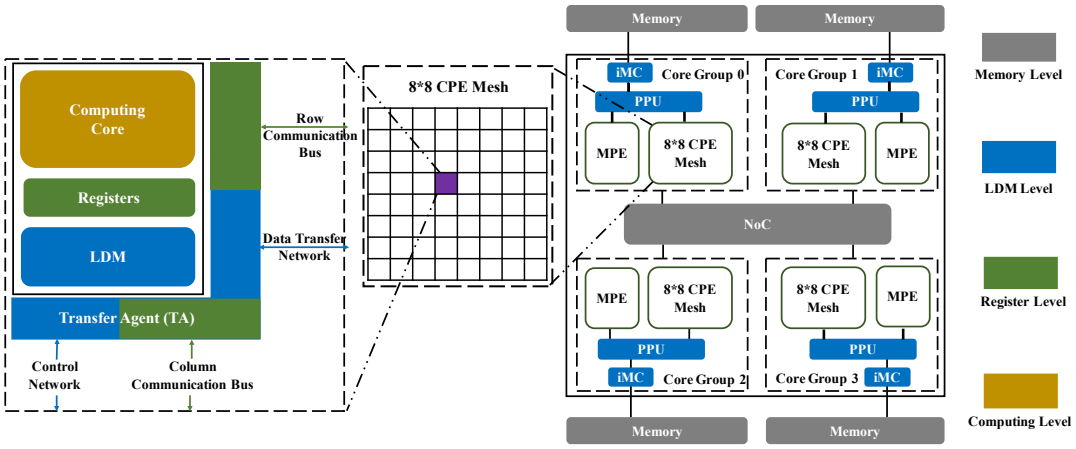

Figure 1: The general architecture of the SW26010 many-core processor.

aggregated memory bandwidth of 480 GB/s. In contrast, SW26010's memory bandwidth can hardly match the 3.06 Tflops double-precision performance and can easily fall into memory-bound cases. While the LDM of each CPE provides an option for manual caching optimizations, the DDR3 interface generally requires aligned memory access patterns (in blocks of 128 bytes) to achieve close to optimal bandwidth. Therefore, while CNN is generally considered as a computing-intensive kernel, we still need a careful memory access scheme to alleviate the memory bandwidth constraints. **(ii)** The algorithm of CNN involves all-to-all connections between inputs, filter kernels, and outputs. As a result, a parallel CNN design generally requires frequent data communication among different processing elements. However, in SW26010, the CPEs do not have a shared buffer for such frequent data communications. Therefore, we have to rely on a fine-grained data sharing scheme based on row and column communication buses in the CPE mesh.

D. Performance model

Based on the above analysis of both the CNN algorithm and the Sunway processor architecture, we can easily see that the memory bandwidth becomes the major bound in our process of identifying the best mapping of CNN to SW26010. To further quantize the limiting factors at different levels of the memory hierarchy, we derive a three-level (register (REG), local data memory (LDM), memory (MEM)) performance model to guide us to the most suitable way of design convolution implementation in a many-core architecture, as shown in Fig. 2. In different scenarios, the CPE mesh can access the data items either directly from the global memory, or from the three-level (REG-LDM-MEM) memory hierarchy. In either cases, we estimate the minimum requirement memory bandwidth (denoted as RBW) by the roofline model [27] to support the peak floating-point throughput for each CG. Because the amount of computation increases with the square of the input data in convolution operations, the actual computing performance can then be estimated based on the square of the ratio between RBW and the actual measured memory bandwidth (denoted as MBW) at different levels. If the RBW is smaller than the MBW at the corresponding level, then the memory bandwidth is no longer the performance bound.

In the first case, the CPE mesh can directly access the data items from MEM by using $gload$ instructions (performance estimated in the middle column of Fig.2). Such a direct memory access pattern does not take advantage of any possible data sharing, thus requiring the largest bandwidth of $RBW_{directMEM}$ = 139.20 GB/s in such case. Moreover, the actual interface of $gload$ only provides a physical bandwidth of 8 GB/s, leading to an extremely low utilization of the floating-point computing capability($(8/139.2)^2$=0.32%). In the second case, the CPE mesh accesses the data items through the MEM-LDM-REG hierarchy (performance estimated in the right column of Fig. 2), i.e. we apply DMA operations to load data into LDM first, and then perform $load$ and $store$ instructions to move data into the register file for the computation afterwards. By going through two extra levels of controls on LDM and register, we can then achieve effective data sharing, thus reducing the actual data accesses that we need to make from the global memory. Similarly, at each level, we estimate the required bandwidth to support the maximum throughput of computing, and compare the required bandwidth to the actual physical bandwidth provided, so as to derive the estimated performance of the CNN kernel. In both cases, we also introduce a concept of execution efficiency (EE) to account for the cases that the CPE is not providing the maximum level of floating-point throughput (due to the floating-point operation pipeline stalls or the pipeline is occupied with non-computing instructions and non-vectorized operations).

The $MBW_{MEM \to LDM}$ between the global memory and the LDM is not a constant value and is variant with the size of continuous memory access blocks of one CPE. We wrote

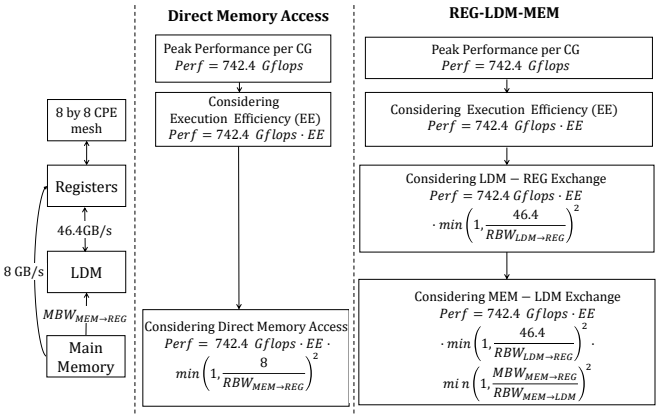

Figure 2: The performance model of our CNN kernel design on one CG of SW26010.

a micro-benchmark on one CG to measure the effective DMA bandwidth and present the results in Table II, in which $Size(Bye)$ indicates the sizes of continuous memory access data block of one CPE. We can see that the effective bandwidth for DMA load and store ranges from 4 GB/s to 36 GB/s. In general, a higher bandwidth is achieved when using a block size larger than 256B and aligned in 128B. Therefore, we should arrange the leading blocking size of data layouts to satisfy these constraints when design our convolution operations.

Table II: Measured DMA Bandwidths (GBps) on 1CG

$Size(Byte)$	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

Another design issue we need to consider is the way to scale the training process across four CGs. With the support on partitioning between private memory space and shared memory space across the four CGs in one SW26010, we can partition output images into four parts along the row, and assign each CG to process one fourth of the output images. Our experiments demonstrate that such a partition scheme can generally achieve near linear scaling among the four CGs in one processor.

IV. LDM-Related Optimizations

A. LDM Blocking

Due to extremely low efficiency of the direct memory access mode, we adopt the REG-LDM-MEM memory access in our CNN design. As described in Section III-D, When using such memory access mode, data should be explicitly loaded into LDM using DMA requests first. An LDM blocking strategy is adopted to partition the input/output images and filter kernels into smaller blocks. Such a blocking strategy helps to keep convolution data in the fast buffer for future data reuse. As shown in Listing 1, a naive implementation of convolution is with 7 for loops $(B, N_i, N_o, K_r, K_c, C_o, R_o)$. *Loop scheduling* and *loop blocking* can be applied to these 7 loops. The following insights will guild us towards more efficient designs.

- We can not apply blocking to every dimension due to the limited size of LDM, although blocking on each loop dimension will decrease the $RBW_{MEM-LDM}$. We should choose the dimension that leads to the most significant reduction of RBW for blocking first.
- The DMA bandwidth between memory and LDM will be affected by the size of leading dimension of loops. We should choose the loop scheduling plan which makes leading dimension large enough to ensure a better DMA bandwidth.
- To increase data reuse in LDM, we should arrange the DMA operations at outer loops as far as possible.

Based on such a design philosophy, we can derive a serial of algorithmic transformations, to achieve an improved effective memory bandwidth, and a resulting improved computing performance.

A image-size-aware version is illustrated in Algorithm 1. Its RBW of MEM-LDM is illustrated in Equation 1, where b_B and b_{C_o} is blocking size on B and C_o dimensions; T is the peak performance; DS is the size of data type. A batch-size-aware version is illustrated in Algorithm 2. Its RBW of MEM-LDM is illustrated in Equation 2.

$$RBW_{Mem\to LDM} = \frac{(N_o + b_{C_o} b_B)DS}{2 b_{C_o} b_B N_o / T} = \frac{(\frac{1}{b_{C_o} b_B} + \frac{1}{N_o})DS}{2/T} \quad (1)$$

Algorithm 1 Image Size Aware Version

```
for b_BStart = 0 : b_B : B do
  for Ro_Start = 0 : Ro do
    for Co_Start = 0 : bCo : Co do
      for cKr = 0 : K_r do
        for cKc = 0 : K_c do
          DMA get D_i ← N_i × b_B channels input images (Co_Start +
          cKc : Co_Start + cKc + bCo, Ro_Start + cKr) start at
          b_BStart.
          DMA get W ← N_i × N_o channels filter kernels (cKc, cKr)
          start at b_BStart
          D_o+ = D_i × W
        end for
      end for
      DMA put b_B × N_o channels output images(Co_Start : Co_Start +
      bCo, Ro_Start) ← D_o
    end for
  end for
end for
```

For both versions, a large output channel N_o will reduce the RBW. If the batch size is large enough to reduce the RBW to a lower level, we can adopt the batch-size-aware

version. Otherwise, we can perform blocking on the column dimension with the image-size-aware version.

$$RBW_{Mem\to LDM} = \frac{(B+K_cN_o)DS}{2K_cBN_o/T} = \frac{(\frac{1}{K_cN_o}+\frac{1}{B})DS}{2/T} \tag{2}$$

Algorithm 2 Batch Size Aware Version

```
1: for Co_start = 0 : b_Co : C_o − 1 do
2:   for cR_o = 0 : R_o − 1 do
3:     for cK_r = 0 : K_r − 1 do
4:       cR_i = cR_o + cK_r
5:       for cC_i = Co_start : Co_start + b_Co + K_c − 1 do
6:         DMA get D_i ← N_i × B channels of input images(cC_i, cR_i)
7:         for cKc = 0 : K_c − 1 do
8:           DMA get W ← N_i × N_o channels of filter kernels
             (cK_c, cK_r)
9:           cC_o = cC_i − cK_c
10:          if cC_o >= Co_start and cC_o < Co_start + K_c then
11:            D_o(cC_o)+ = W × D_i
12:          end if
13:        end for
14:      end for
15:    end for
16:    DMA put N_i × B channels of output images (Co_start : Co_start +
       b_Co, cR_o) ← D_o
17:  end for
18: end for
```

Double Buffering is adopted to overlap DMA with computing. While the data is computed in one LDM buffer, the data to be used at next iteration is loaded into another LDM buffer by DMA. In our above descriptions, blocking is not performed on N_i and N_o dimensions. However, if LDM space is not enough for large N_i or N_o, we still need to apply loop blocking on these dimensions. Conversely, if free LDM space is sufficient, we can promote the DMA operation to outer loop to further reduce RBW. For Algorithm 1, we can promote the DMA operation at line 6 to line 4 and read input image tile of size $(Co_{start} : Co_{start} + K_r + b_{Co})$. For Algorithm 2, we can promote the DMA operation at line 8 to line 4 and read filter tile of size $(:, cK_r)$.

V. Register-Related Optimizations

A. Register Communication

Both Algorithm 1 (lines 8) and Algorithm 2 (line 10) perform a general matrix-matrix multiplication (GEMM) operation on data in LDM. One unique feature of the SW26010 architecture is the register communication mechanism inside the 8×8 CPEs mesh, which is designated to support data transfer between 8 CPEs within the same row/column. We can optimization LDM-GEMM with the register communication provided by SW26010.

8 *row communication buses* and 8 *column communication buses* form the data exchange channels for the 8 by 8 CPE mesh. Register-level communication is achieved by a pair of *Put* and *Get* operations through *row/column communication buses*. The sender CPE uses the *Put* operation to send a 256-bit register file to the *Transfer Buffer* of a receiver CPE,

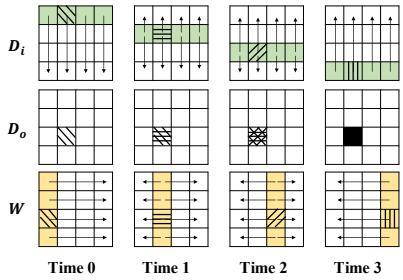

Figure 3: Schematic of register communication on CPEs for matrix multiplication.

while the receiver CPE uses the *Get* operation to fetch the 256-bit data from the *Transfer Buffer* to the local general-purpose register file. A producer-consumer strategies is implemented to ensure multi-Put and multi-Get operations. In addition to *Put* and *Get* operations, SW26010 also provides mechanisms to broadcast and multicast 256-bit data items.

We design a data distribution plan for images and filters on the 8× 8 CPE mesh to ensure no duplicate data stay on different CPEs and minimize MEM-LMD bandwidth requirement. As for input images, the N_i/8 channels of images are resident on a column of mesh. Each CPE of 8 CPEs on one column have $1/8$ batchs of image pixels. As for filters, the N_o/8 channels of output channels are resident on a column of mesh. Each CPE of 8 CPEs on one column have $Ni/8$ input channels of filter elements. No duplicated data are resident on two different CPEs. Therefore, each core can finish a convolution operation with 1/64 data and can achieve a partial result. To get a finial result, each CPE requires data on other cores. With this plan, the required data of one CPE is resident on the CPEs of the same column and the same row and we can use register communication to fetch remote data from local transfer buffers to local GPRs.

We demonstrate the basic idea in a simplified 4 by 4 CPE mesh, shown in Figure 3. In Figure 3, input image D_i, filter W and output image D_o are divided into 4×4 parts, each CPE has the corresponding input, filter and output data. That is, for each (i, j), CPE(i, j) owns $D_i(i, j)$, $W(i, j)$ and $D_o(i, j)$. Initially, the values in D_o are set to 0. For the statement convenience, the following shows the calculation of $D_o(2, 1)$ using register communication. The value of $D_o(2, 1)$ relies on the row 2 values of filter W ($W(2, 0)$, $W(2, 1)$, $W(2, 2)$ and $W(2, 3)$) and the column 1 values of input image D_i ($D_i(0, 1)$, $D_i(1, 1)$, $D_i(2, 1)$ and $D_i(3, 1)$). At *step 0*, each column 0 CPE (in yellow) sends its own filter W value to corresponding CPE in column $1 \sim 3$ though row communication bus, and each row 0 CPE (in green) sends its input D_i data to corresponding CPE in row $1 \sim 3$ via column communication bus. For example, CPE$(0, 0)$ sends $W(0, 0)$ to CPE$(0, 1)$, CPE$(0, 2)$ and CPE$(0, 3)$, and sends $D_i(0, 0)$ to CPE$(1, 0)$, CPE$(2, 0)$ and CPE$(3, 0)$. After receiving, each CPE calculate the matrix multiplication of

the received W and D_i, and the result is added to D_o values it owns. For $D_o(2,1)$, CPE$(2,1)$ currently received $W(2,0)$ and $D_i(0,1)$, and now $D_o(2,1) = W(2,0) \times D_i(0,1)$. Next at *step 1*, column 1 CPEs send W to other columns, and row 1 CPEs send D_i to other rows. Now $D_o(2,1) = W(2,0) \times D_i(0,1) + W(2,1) \times D_i(1,1)$. Next time 2, column 2 sends W, row 2 sends D_i, and $D_o(2,1) = W(2,0) \times D_i(0,1) + W(2,1) \times D_i(1,1) + W(2,2) \times D_i(2,1)$. Finally, at *step 3*, column 3 sends W, row 3 sends D_i, and CPE$(2,1)$ owns the complete value of $D_o(2,1)$. Similar to CPE$(2,1)$, other CPE(i,j) has already computed the corresponding $D_o(i,j)$ after four steps.

B. Register Blocking

Register Blocking can further improve the data reuse in registers, thus reducing required bandwidth between LDM and registers. There exist two blocking different approaches at the register level. As shown in Figure 4, one way usually adopted by direct convolution plan is that we perform a 2D spatial-convolution on C_i and R_i dimensions in registers; the other way adopted by blocked-GEMM convolution plan is that we perform a 2D spatial-convolution on B and N_o dimensions in registers.

In the first way, we fix a $rb_{K_r} \times rb_{K_c}$ filter kernel in registers and load a block of $rb_{C_i} \times rb_{R_i}$ input image pixels from LDM to register for convolution. After convoluted, the results are stored to $rb_{C_o} \times rb_{R_o}$ ($rb_{C_o} = rb_{C_i} - K_c + 1$ and $rb_{R_o} = rb_{R_i} - K_r + 1$) output pixels in LDM from registers. In the second way, we load rb_B input pixels and rb_{N_i} filter data from LDM into registers and fix $rb_B rb_{N_o}$ output pixels in registers for updating iteratively. Equation 3 illustrates RBW for one CPE of the first way. It is hard to lower the RBW, because the RBW is mainly dependent on rb_{K_r} and rb_{K_c}, the maximum values of which are limited by the network parameter K_r and K_c. That can explain why we do not adopt a direct convolution plan at beginning. Equation 4 illustrates RBW of the second way. In contrast, the RBW changes with register block input pixels rb_B and rb_{N_o}. By blocking on B and N_o rather than dimensions of filter kernel size, we enable register more flexible for different parameter configurations. We can use such blocking plan for our register-communication-based GEMM implementation.

$$RBW_{LDM \to Reg} = \frac{(rb_{R_i} rb_{C_i} + rb_{bC_o} rb_{R_o}) DS}{2 rb_{K_r} rb_{K_c} rb_{C_o} rb_{R_o} / T} \quad (3)$$

$$RBW_{LDM \to Reg} = \frac{(rb_B + rb_{N_o}) DS}{2 rb_B rb_{N_o} / T} \quad (4)$$

C. Vectorization-Oriented Data Layout

Both the MPE and CEPs in each CG support 256-bit vectorized instructions, which enable 4 simultaneous double-precision or single-precision floating-points operations. We

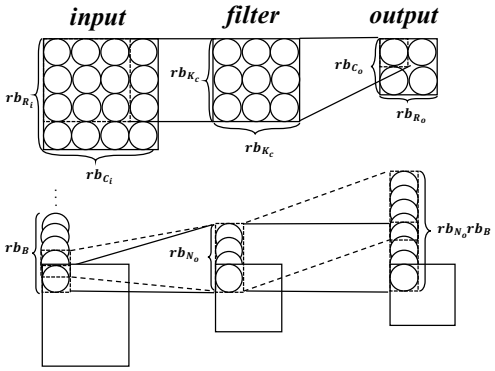

Figure 4: Two register blocking plans with the data blocked in registers are shown in dashed boxes. The upper convolutes input images on C_i and R_i dimensions. The lower convolutes input images on B and N_o dimensions.

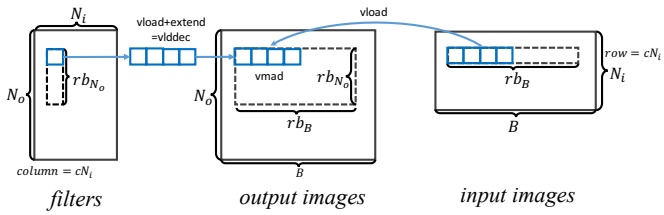

Figure 5: Vectorization and register blocking for Matrix-Matrix Multiplication. Dotted boxes illustrate the data blocked in registers.

use *vldr/vldc(vldder/vlddec)* primitives, which are equivalent to *vload+putr/putc*(*vldde+putr/putc*), to load a vector data from LDM to registers and then broadcast it to transfer buffers of other CPEs on the same row/column. The primitive *getr/getc* to load vector data from transfer buffer into registers. For the convolution operation in Algorithm 1 (line 8) and Algorithm 2 (line 11), our vectorization plan is shown in Figure 5. Four floating-point image pixels are packed into a vector structure and are loaded from LDM into registers. One filter element is loaded from LDM and is replicated in quadruples to form a vector structure in register with *vldde* primitive. Vectorized multiply-add *vfmad* operation is performed with them afterwards. In addition, we design the following data layouts for vectorization:

- For the image-size-aware version, the 4D input and output images are organized as $(4, C, R, N, B/4)$;
- For the batch-size-aware version, the 4D input and output images are organized as $(4, B/4, C, R, N)$.

We choose an appropriate blocking size (rb_B, rb_{N_o}) for Equation 4 to ensure the $RBW_{LDM \to Reg}$ is less than 46.4 GB/s. Because we require to load a single float-point filter elements and extend it into a SIMD vector, it leads to 4× bandwidth cost to load rb_{N_o} filter elements. As shown in Equation 5, a reasonable parameter setting is rb_B=16 and

rb_{N_o}=4, which leads the required bandwidth to 23.2 GB/s (far less than architecture bandwidth 46.4 GB/s).

$$RBW\_SIMD_{LDM\rightarrow Reg} = \frac{(rb_B + 4 \times rb'_{N_o})DS}{(2rb_B rb_{N_o})/T}$$
$$= \frac{(16 + 4 \times 4) \times 8Byte}{(2 \times 16 \times 4)/(1.45GHz * 8)} = 23.2GB/s < 46.4GB/s \quad (5)$$

VI. Instruction Reordering

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

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

Figure 6: Instruction reordering for innermost loop. The left is original assembly code and the right is rescheduled assembly code.

A. Instruction Pipelines

Each CPE consists of two execution pipelines, called *P0* and *P1*. Both of them can handle some basic scalar integer operations. Besides, floating-point operations and vector operations can only be handled on *P0*. Control transfer operations, load/store and register communication operations for both scalar and vector can only be handle on *P1*. The two execution pipelines share an *Instruction Decoder* (ID), and an instruction queue is maintained in the ID stage. In each cycle, two instructions in the front of the queue are checked by the ID and can be issued into two pipelines simultaneously if all the following conditions are satisfied:

1) Both instructions have no conflicts with the unfinished instructions issued before.
2) The two instructions have no *Read After Write* (RAW) or *Write After Write* (WAW) conflicts.
3) The two instructions can be handled by two execution pipelines separately.

For algorithms with floating-point operations as core computation, maximizing the efficiency of *P0* can improve the overall performance. Theoretically, with *P1* handling data load/store, control transfer and other scaler integer operations, we can make $P0$ fully-pipelined for floating-point operations during the core computing process, which requires an orchestrated instruction flow. However, current optimization tools in the Sunway C compiler can not provide an optimized solution. Therefore, we propose an optimization process for double-pipeline instruction reordering to explore higher efficiency for the core computing process.

B. Instruction Reordering Optimization

A GEMM kernel calculating $C+ = A \times B$ in the left side of Figure 6, the execution time of the original instruction flow is 26 (8*vload* + 1*cmp* + 1*bnw* + 16*vmad* = 26) cycles per iteration. The innermost loop contains $N_i/8$ iterations. Under optimal circumstances, $P0$ only executes 16 $vfmad$ instructions every iteration. Therefore, the execution efficiency now is $16/26 = 61.5\%$. Based on the original instruction flow, the following three steps can guide our further optimizations to improve the execution efficiency.

1) Dependence analysis: The load operation has a 4-cycle latency, so that the load operation should be issued 4 cycles before the data is used. In order to issue the *vfmadd* instruction as early as possible, we should load $A[0]$ and $B[0]$ first, then load $A[1]$-$A[3]$ and $B[1]$-$B[3]$ sequentially. The latency of *vfmadd* is 7 cycles, but there is no data dependency on C in each iteration, so the *vfmadd* instructions could be issued into $P0$ in a fully-pipelined sequence without reordering.

2) Intra-loop pipelining and reordering: Based on the above analyses, we can hide data load operation with *vfmad*s in one loop. We first move the load operation of $B[0]$ forward to cycle 1. The load and the address update operations of $A[0]$-$A[3]$ can be issued at cycle 2-5. Then the first *vfmadd* operation can be issued at cycle 6 (4 cycles after loading $A[0]$). The load and address update operations of $B[1]$-$B[2]$ can be issued to $P1$ while $P0$ is handling the first 4 *vfmadd* operations at cycle 6-11. The loop control operations can also be handled by $P1$, so each of them can be issued together with a *vfmadd* operation.

3) Inter-loop pipelining and reordering: After intra-loop pipelining and reordering, there is no floating-point operations in the first 5 cycles due to the data dependency constraint. Considering a multi-iteration process, when the total iteration number is more than 2, we can issue the load instructions of $A[0]$-$A[3]$ and $B[0]$ to *P1* together with the *vfmadd* instructions in the previous iteration. As shown in the right side of Figure 6, we need to design an initial section before the loop starts and an exit section for the last iteration. In this case, the initial section takes 5 cycles, each iteration takes 17 cycles and the exit section takes 16 cycles. The execution efficiency with innermost iterations is

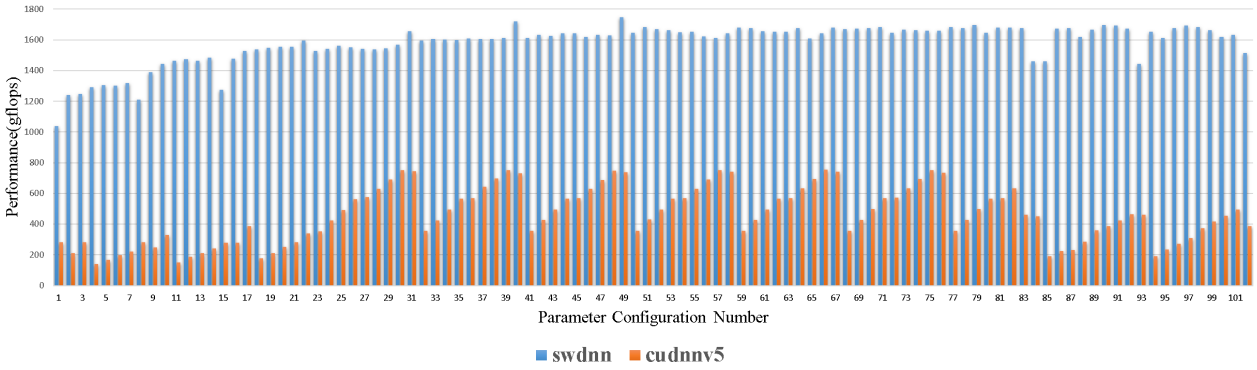

Figure 7: Double-precision performance results of our convolution kernels with different (N_i, N_o) ranging from $(64, 64)$ to $(384, 384)$, compared with the K40m GPU results with cuDNNv5. ($B = 128$, output image =64×64, filter $= 3 \times 3$)

```
do for (( Ni=64; Ni<128; Ni+=32))      do for (( Ni=128; Ni<384; Ni+=32))      do for (( No=128; Ni<256; Ni+=64))
  do for ((No=64; No<=256; No+=32))    do for ((No=128; No<=384; No+=32))     do for ((K=3; K<=21; K+=2))
    ./swdnn $Ni$No; done ; done ;        ./swdnn $Ni$No; done ; done ;          ./swdnn $K$No; done ; done ;
```

Figure 8: Test scripts for swDNN performance evaluations.

$(N_i/8 * 16)/(5 + (N_i/8 - 1) * 17 + 16)$ and larger N_i will get higher execution efficiency.

Assembly Code[1] from [16] shows the final instruction flow, where we apply *register package* (packing 4 *long* or 8 *int* into vector structure) to innermost loop to reduce required register number and unroll the two $if - else$ statements for thread column and row ids in the outer loop to reduce overhead of loop control operations and ids storage.

VII. Performance

To evaluate the performance of our swDNN, we adopt different loop scheduling and blocking strategies according to the performance model for different parameter configurations with code from [16]. Because the current arithmetic architecture does not allow an easy doubling or even quadrupling of the performance by using single or even half precision, we use double-precision for performance evaluation. Figure 7 summarizes the double-precision performance results of our convolution kernels for different input and output image channels parameter configurations, compared with the GPU results measured using cuDNNv5.1 on K40m. The parameter configurations of numbers 1 to 21 are generated from the left script in Figure 8 and configurations of numbers 22 to 101 are generated from the center script of Figure 8. In most cases, we see a convolution performance above 1.6 Tflops and achieve speedup ranging from 1.91x to 9.75x compared with cudnnv5.1.

Figure 9 shows the performance of using different filter kernel sizes ranging from 3×3 to 21×21. The parameter configurations of numbers 1 to 30 are generated from the right script of Figure 8.

[1]https://github.com/THUHPGC/swDNN/tree/master/src/asm

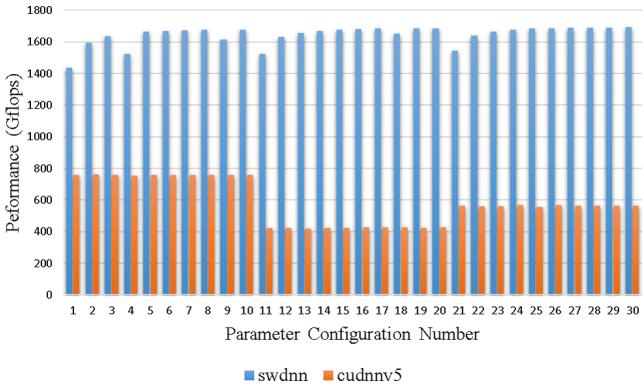

Figure 9: Double-precision performance results of our convolution kernels for different filter sizes ranging from 3×3 to 21×21, compared with the K40m GPU with cuDNNv5. ($B = 128$, output image $= 64 \times 64$)

We achieve over 54% efficiency for most of parameter configurations, while the best efficiency on K40m is around 40% but only for a small set of parameter configurations. Moreover, not like cuDNN, our program is stable under different parameter configurations.

Table III: Performance Model Evaluation

Plan	K_c	b_B	b_{Co}	N_i	N_o	RBW	MBW	mdl	meas
img	3	32	16	128	128	29.0	21.9	**368**	**350**
img	3	32	8	128	256	23.2	18.2	**397**	**375**
batch	3	-	-	256	256	27.1	21.2	**422**	**410**
batch	3	-	-	128	384	25.7	21.2	**407**	**392**

Table III demonstrates the measured performance results (meas) of our CNN kernel design on one CG after applying image-aware (img) and batch-aware (batch) loop transformation strategies, compared to the estimated modeled results (mdl) given by our performance model. The comparison between the measurement and our performance model shows a reasonable match, thus proving that our performance model

has successfully identified the major factors that determine the CNN performance on SW26010, and provided useful guidance in our optimization process.

VIII. Conclusion

This paper reports our efforts on designing and building swDNN, a library that supports efficient DNN implementation on the newly announced Sunway TaihuLight supercomputer. To achieve an efficient mapping of the DNN kernels (specifically focusing on CNN kernels in this work), we derive a performance model that guide us in the design and optimization process that targets on a CNN solution that can maximize the utilization of both computing and memory resources of the SW26010 many-core processor. Based on the performance model, we then apply a series of optimization schemes, including LDM-oriented algorithmic transformations, customized register communication schemes, as well as reordering of the instruction sequence for the two pipelines. The resulting solution is capable of providing double-precision convolution performance around 1.6 Tflops. Compared with the GPU platforms, although the memory bandwidth of SW26010 processor (128 GB/s) is only half of the K40 GPU (240 GB/s), in double precision scenarios, we increase the computational efficiency from 40% (results measured using cuDNNv5) to 54%.

IX. Acknowledgement

This work was supported in part by the National Key R&D Program of China (Grant No. 2016YFA0602200), by the National Natural Science Foundation of China (Grant No. 4137411, 91530323) and by the China Postdoctoral Science Foundation (No. 2016M601031).

References

[1] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.

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

[3] Yi Sun, Xiaogang Wang, and Xiaoou Tang. Deeply learned face representations are sparse, selective, and robust. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2892–2900, 2015.

[4] Geoffrey Hinton, Li Deng, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[5] George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42, 2012.

[6] Volodymyr Mnih, Koray Kavukcuoglu, et al. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] David Silver, Aja Huang, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[8] NVIDIA. Nvidia tegra drive px: Self-driving car computer. http://www.nvidia.com/object/drive-px.html, 2015.

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

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[11] Tianshi Chen, Zidong Du, et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[12] Yunji Chen, Tao Luo, Shaoli Liu, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[13] Daofu Liu, Tianshi Chen, et al. Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 369–381. ACM, 2015.

[14] Zidong Du, Robert Fasthuber, et al. Shidiannao: shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[15] Haohuan Fu, Junfeng Liao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, pages 1–16, 2016.

[16] https://github.com/THUHPGC/swDNN.

[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 675–678. ACM, 2014.

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

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

[20] Stefan Hadjis, Firas Abuzaid, Ce Zhang, and Christopher Ré. Caffe con troll: Shallow ideas to speed up deep learning. In *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, page 2. ACM, 2015.

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

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

[23] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[24] 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 26–35. ACM, 2016.

[25] Naveen Suda, Vikas Chandra, et al. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 16–25. ACM, 2016.

[26] 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 326–331. ACM, 2016.

[27] Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures[J]. Communications of the ACM, 2009, 52(4): 65-76.