



Accelerating Sparse Convolutional Neural Networks Based on Dataflow Architecture

Xinxin Wu^{1,2(✉)}, Yi Li^{1,2}, Yan Ou^{1,2}, Wenming Li¹, Shibo Sun³,
Wenxing Xu³, and Dongrui Fan^{1,2}

¹ State Key Laboratory of Computer Architecture,
Institute of Computing Technology, Chinese Academy of Sciences,
Beijing 100190, China

{wuxinxin, liyi, ouyan, liwenming, fandr}@ict.ac.cn

² School of Computer and Control Engineering, University of Chinese Academy
of Sciences, Beijing 100190, China

³ Beijing Institute of Petrochemical Technology, College of Information Engineering,
Beijing 100190, China
771586376@qq.com, xuwenxing@bipt.edu.cn

Abstract. Convolutional Neural Networks (CNNs) achieve state-of-the-art performance in a wide range of applications including image recognition, speech recognition, and natural language processing. Large-scale CNNs generally have encountered limitations in computing and storage resources, but sparse CNNs have emerged as an effective solution to reduce the amount of computation and memory required. Though existing neural networks accelerators are able to efficiently process sparse networks, the strong coupling of algorithms and structures makes them inflexible. Dataflow architecture can implement different neural network applications through flexible instruction scheduling. The dataflow architecture needs to be initialized at execution time to load instructions into the computing array. Running a dense convolutional layer only needs to be initialized once due to regular calculations. However, running a sparse convolutional layer requires multiple initializations, which takes a long time to fetch instructions from memory, resulting in the computing array being idle and degrading performance. In this paper, we propose an instruction sharing strategy based on the field content in the instruction, which can reduce initialization time and improve performance. Moreover, we use an extended instruction sharing strategy based on the static nature of filters to remove filters-related instructions, further reducing initialization time. Experiments show that our strategies achieve 1.69x (Alexnet), 1.45x (VGG-16) speedup and 37.2% (Alexnet), 34.26% (VGG-16) energy reduction compared with dense networks. Also, they achieve on average 2.34x (Alexnet), 2.12x (VGG-16) and 1.75x (Alexnet), 1.49x (VGG-16) speedup over Titan Xp GPU and Cambricon-X for our benchmarks.

Keywords: Dataflow architecture · Sparse convolutional neural networks · Instruction scheduling · Instruction sharing

1 Introduction

Application-specific accelerators [4, 16] have proposed as a high performance and low power alternative to traditional CPUs/GPUs due to stringent energy constraints. However, CNNs continue to evolve towards larger and deeper architectures as applications diversify and complicate, which leads to a heavy burden on processing computation, memory capacity, and memory access of accelerators [6]. To address these challenges, many methods for reducing model parameters are proposed to turn dense networks into sparse networks using the redundant characteristics of model parameters [7], such as pruning [13], low rank [15]. In order to make full use of the advantages of sparse network computing and storage, many accelerators have emerged to accelerate sparse networks [1, 11, 12, 29, 30] due to the low efficiency of CPU/GPU for sparse network acceleration.

However, these application-specific accelerators suffer from inflexible characteristics due to the tightly coupled nature of algorithms and structures. For example, DianNao family [4], a series of customized accelerators without sparsity support can not benefit from sparsity. Cnvlution [1] has completely modified DaDianNao's [5] microstructure to support sparse networks. EIE [12] and ESE [11] accelerators supporting sparse networks are no longer suitable for dense networks.

Dataflow architecture has advantage in good flexibility and high data parallelism and power efficiency for today's emerging applications such as high performance computing [2], scientific computing [21, 27] and neural networks [25, 28]. It implement different applications through flexible instruction scheduling. Compared with the traditional control flow (controlled by PC), it consists of a simple control circuit composed of processing unit array (PEs) that can communicate directly avoiding frequent memory access. Based on codelet model (a dataflow-inspired parallel execution model) [9], an application is represented as a Codelet Graph (CDG) which is a directed graph consisting of codelets, which are composed of instructions, and arcs, which represent data dependencies among codelets. At the same time, a codelet is fired once all data is available and all resource requirements are met, which maximizes instruction codelet-level and data-level parallelism. Its natural parallel characteristics fit perfectly with the inherent parallelism of neural network algorithms. Based on this architecture, we study the data characteristics and instruction characteristics of the neural network to optimize the mapping and execution of the network to maximize the structural benefits.

In the dataflow architecture, codelets instructions in the CDG need to be loaded from the memory into the instruction buffer in the PE through initialization process when the PE array is running. A codelet in the buffer waits for the condition to be satisfied and then is emitted. For dense convolutional layer of CNN, convolution operation of each channel is mapped on the PE array in the form of a CDG diagram. The codelets instructions formed by different channels are the same due to the regular calculation mode. In this case, codelets instructions need only be loaded once to implement convolution operations in all channels.

However, for a sparse convolutional layer of CNN obtained using pruning method such as [13], the following problem will occur when the PE array is running. The codelet instructions in the CDG of different channels are no longer the same, due to the pruning operation makes the convolution calculation irregular. It must load the codelets instructions of different channels from different memory address when running, which causes frequently memory access and the idleness of the PE array. Eventually results in a significant performance degradation.

Based on the above problem, in this paper we use two strategies to accelerate sparse CNNs based on dataflow architecture. Our contributions are as follows:

- By analyzing the data and instruction characteristics of the sparse convolution layer, we share same instructions in the convolutional layer based on the field contents in instructions, and proposed an instruction sharing strategy (ISS), which can reduce initialization time and computing array idle time.
- Based on the ISS strategy, we use the extended instruction sharing Strategy (EISS) to delete filter-related instructions by using the static nature of filters, further reducing the initialization time.
- These two strategies achieve on average 1.45x (Alexnet), 1.24x (VGG-16) and 1.69x (Alexnet), 1.45x (VGG-16) speedup and 36.57% (Alexnet), 33.05% (VGG-16) and 37.2% (Alexnet), 34.26% (VGG-16) energy reduction respectively compared with dense convolutional layers. In addition, they achieve on average 2.34x (Alexnet), 2.12x (VGG-16) and 1.75x (Alexnet), 1.49x (VGG-16) speedup over Titan Xp GPU and current state-of-the-art Cambricon-X accelerator.

This paper is organized as follows. In Sect. 2, we introduce the background of dataflow architecture and CNN. In Sect. 3 we analyze the existing problems in implementing sparse CNN based on a dataflow architecture. In Sect. 4, we describe two strategies in detail to accelerate sparse convolution. In Sects. 5 and 6, we present our evaluation methodology and experimental results respectively. In Sects. 7 and 8, we provide related work and a conclusion to this work.

2 Background

2.1 Dataflow Architecture

This section explains microarchitecture, execution model, and instruction format of a dataflow process unit (DPU), that resembles coarse-grained instruction level dataflow architecture, such as Runnemedede [2], TERAFLUX [10].

Microarchitecture. Figure 1 is an instantiated dataflow accelerator, which includes a micro controller (Micc), processing element (PE) array and a network on chip (NoC). The structure of the DPU is similar to the traditional many-core architecture [8], and each PE inside the array is a control-flow core. The micro controller manages the overall execution of the PE array and is responsible for communicating with the host. Each PE contains a buffer for storing instructions,

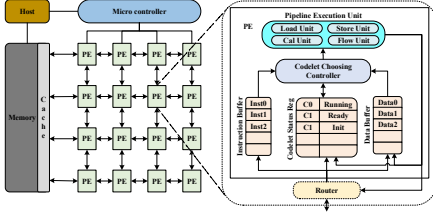


Fig. 1. DPU: An instantiated dataflow accelerator.

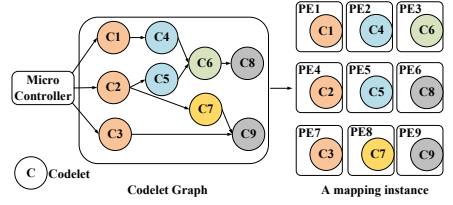


Fig. 2. Codelet execution model.

a buffer for storing data, a codelet choosing controller unit, a codelet status register unit and a pipeline execution unit.

Before running each PE, it needs to go through an initialization phase to load all required codelets in the memory into the instruction buffer by routers. Then the codelet choosing controller selects ready codelet instructions and sends them to the pipeline execution unit according to the codelet status register. The pipeline execution unit contains load, store, calculation and flow unit to execute the corresponding instructions.

Opcode	F0	F1	F2	Fuction
Load	Operand index0	Address offset	Base address index0	$Value(F0)=DRAM(Base\ address(F2) + F1)$
Store	Operand index0	Address offset	Base address index0	$DRAM(Base\ address(F2) + F1)=Value(F0)$
Madd	Operand index0	Operand index1	Operand index2	$Value(F2)=Value(F0)*Value(F1)+Value(F2)$
Mul/Add/Sub	Operand index0	Operand index1	Operand index2	$Value(F2)=Value(F0)*+/-Value(F1)$
Flow	Operand index0	Operand index1	PE index1	$Value(F1,F2)=Value(F0)$

Fig. 3. Instruction format.

Execution Mode. In order to achieve high utilization of computing elements in each PE, an application is written for a codelet model, which is a dataflow-inspired parallel execution model [9]. All code are partitioned into codelets containing a sequence of instructions, which are linked together based on codelets dependencies to form a Codelet Graph (CDG) and then are mapped in the PE array, as shown in the Fig. 2. A codelet will only fire when all data is available and all resource requirements are met, which maximizes instruction codelet-level and data-level parallelism.

Instruction Format. The instruction format of DPU is shown in Fig. 3. And the instruction set is fixed-length. Each instruction is composed of instruction code, source operand index and destination operand index. DPU contains basic arithmetic instructions (Add, Sub, Mul, Madd) and memory access instructions (Load, Store) and direct communication instructions (Flow) between PEs.

2.2 Neural Network

Convolutional Neural Network. CNN is mainly composed of multiple convolutional layers, which performs high-dimensional convolutions computation and occupies about 85% computational time of the entire network processing [19,24]. A convolutional layer applies filters on the input feature maps (ifmaps) to generate the output feature maps (ofmaps). The Fig. 4a shows a convolution calculation that the dimensions of both filters and ifmaps are 3D. In order to reduce memory access and save data movement energy cost, data needs to be reused [3]. In the PE array, convolution can be reused in each PE, and ifmap and filter are reused between PEs through flow operations. Figure 4b shows the process of convolution reuse in PE. To generate first row of output, three rows (*row1, row2, row3*) of the ifmap and the filter are mapped in PE. The PE implements convolution reuse by sliding window. Figure 4c shows ifmap and filter reuse between PEs. Filter weights are reused across PEs vertically. Rows of ifmap values are reused across PEs vertically and horizontally. In Fig. 4c, through flow operation, PE3 reuses filter1 of PE1 and *row2–row3* of PE1 ifmap. PE2 reuses *row1–row3* of PE1 ifmap. PE4 reuses filter2 of PE2, and also reuses *row2–row3* of PE2 ifmap, and *row4* of ifmap of PE3.

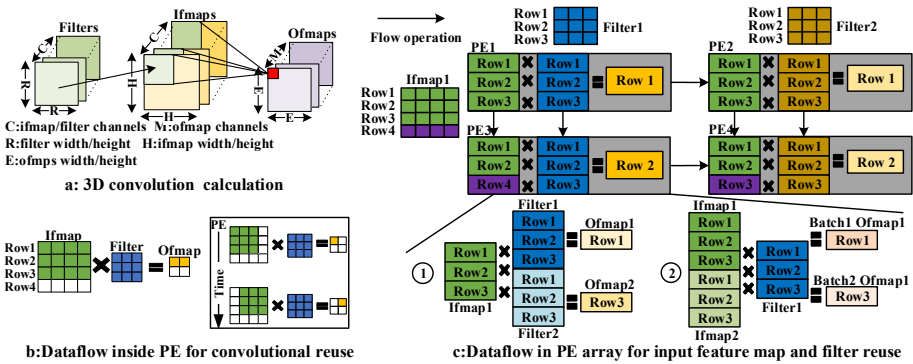


Fig. 4. Opportunities for convolution reuse.

Sparse Neural Network. Due to the challenge of large-size CNN models on hardware resources, researchers have proposed many methods to compress CNN models (e.g. pruning [13], low rank [15], short bit-width [14]) that reduce models size without loss of accuracy or slight loss. Among them, using the pruning method to generate a sparse network is one of the effective methods. The state-of-art pruning method [13] using three-step method (First, network is trained to learn which connections are important. Second, unimportant connections are pruned based on a pre-set threshold. Third, network with remaining connections is retrained to get final weights) achieves a sparsity of 11% for Alexnet [17] and 7% for VGG-16 [23].

3 Motivation

Dense networks can easily achieve high performance and energy efficiency on application-specific accelerator platforms due to dedicated hardware support [4]. However, these accelerators lack dedicated hardware support for sparse networks due to strong coupling of algorithms and structures. Even though the sparse network greatly reduces the amount of calculation and memory access, its performance and energy efficiency gains are small. For data flow architecture, flexible instruction scheduling provides the possibility for different algorithm implementations.

Based on the execution model of the dataflow, the convolution operations of different channels of the dense network have the same instruction codelets due to the regular calculation of features. Therefore, it only needs one initialization operation, that is, loading instructions of codelets from memory, and then all channels convolution operations can be implemented. The Fig. 5 shows the instructions required to calculate the partsum value of two channels. The operand indexes of ifmap, filter, and ofmap are respectively represented by $IF0$ – $IF3$, $F0$ – $F3$, and $OF0$. The base address indexes of ifmap, filter, and ofmap are 0, 1, and 2, respectively.

In Fig. 5, channel 1 and 2 perform the same convolution operation by using different data, which is obtained by channel offset, and address offset and base address index (points to base address) in the instruction. For the data in the same position of each channel, their channel offsets are different (Ifmap are 0×0 , 0×400 . Filter are 0×0 , 0×100 respectively), but the instructions are the same due to the operand index, the address offset and the base address index are the same. For example, the base address, address offset, operand index0 of the *inst1* instruction of two channels are respectively 0×0 , 0, and $IF0$. Therefore, convolution can be done without interruption by one initialization, which ensures the full utilization of the computing resources of the PE array.

However, compared to dense CNN, the execution way of sparse CNN has changed. As shown in the Fig. 5, the pruning operation removes the instructions required for the zero weight. The load instructions (*inst6*, *inst7*), madd instructions (*inst10*, *inst11*), flow instructions (*inst19*, *inst20*) of channel 1 are removed. For channel 2, the load instructions (*inst5*, *inst8*), madd instructions (*inst9*, *inst12*), flow instructions (*inst18*, *inst21*) are removed. It can be seen that the instructions of different channels are no longer exactly the same, which makes it necessary to reinitialize the PE array when performing the convolution operation. That is, new instructions of codelet are loaded from memory into the instruction buffer. When instructions are loaded, the PE array is in an idle state, making the computing resources underutilized, and eventually causing a serious performance degradation of sparse convolution.

The Fig. 6 shows the time taken by the DPU to execute several sparse convolutional layers of Alexnet and VGG-16. Since dense convolution requires only one initialization operation, the initialization time is almost negligible compared to the execution time (accounting for 1.04% of the total time). However, for sparse convolution, it can be seen that the execution time of the PE array is

significantly reduced, but multiple initialization operations take a long time, and they account for an average of 50.2% of the total time. Obviously, the total time of sparse networks does not decrease much compared with dense networks, and some are even longer than dense networks, which makes the sparse network unable to accelerate at all. The multiple initialization seriously hindered the benefits of sparse networks.

The reason for sparse convolution initialization multiple times is that the pruning operation removes filter-related instructions and data, which makes the instructions of different channels no longer the same. In order to reduce the initialization time of sparse networks, an intuitive idea is to use one load instruction to load multiple values from memory. This requires adding multiple operand indexes in the instruction and increasing the data transmission bandwidth to support the operation. However, all instructions are fixed-length, and the data transmission bandwidth is also fixed, so this method is undesirable.

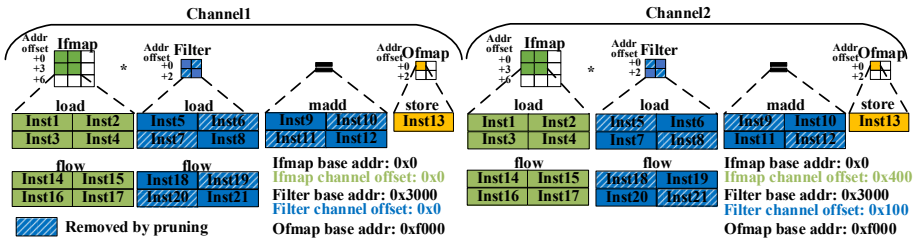


Fig. 5. The instructions required for the two channels to perform a convolution operation in PE1.

By analyzing these instructions and data, it is found that the pruning operation only removes the instructions related to the zero weight in filters, these instructions include load, flow, and madd. However, the relevant instructions of ifmap and partsum are not affected. These instructions are still the same between different channels. For example, in the Fig. 5, the instructions *inst1*–*inst4*, *inst13*, and *inst14*–*inst17* of channel 1 and channel 2 are still the same. These same instructions can be shared between different channels. Multiple accesses of these instruction addresses increase the cache hit rate during each initialization, which reduces instruction load time, also reduces the idle time of the PE array, and increases the full use of computing resources.

The above observation and analysis motivate us using highly efficient strategies to take advantage of the sparsity of neural networks.

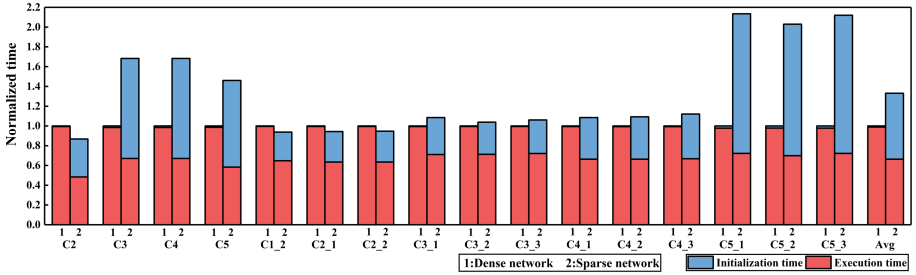


Fig. 6. Execution time breakdown of convolutional layers normalized to dense networks.

4 Acceleration Strategy for Sparse Networks

In this section, we present the detailed strategies of our proposed, including the instruction sharing strategy (ISS) and extended instruction sharing strategy (EISS). We use the advanced pruning method [13] to obtain sparse networks. This method resets the weights below the threshold to 0 and retains the weights above the threshold based on a preset threshold.

Using Eyeriss [3] analysis framework can achieves energy saving mapping of convolutional layers statically based on the reuse opportunities of the above dataflow in the Fig. 4. It folds multiple logical PE arrays, which are required for full convolutional layer operations, into the physical PE array for data reuse and ofmaps accumulation. For example, it folds logical PE from different channels at the same position onto a single physical PE to achieve ofmaps accumulation. It also folds multiple ifmaps and multiple filters to a same physical PE for ifmaps and filters reuse, as shown in the Fig. 4c. In this paper, the ISS and EISS strategies are implemented on the logical PE array to realize the instruction sharing of the convolutional layer.

The reason for instruction sharing is that the convolution calculates characteristics regularly, that is, the same multiply-accumulate operation is performed using different data. Pruning method destroys part of the characteristic of regular calculation. Therefore, our instruction sharing idea is still applicable to other dataflow methods, such as WS, OS, NLR, RS [3], whose purpose is to maximize data reuse to achieve excellent performance and energy efficiency.

4.1 Instruction Sharing Strategy

Multiple initializations of the sparse convolution cause the PE array to be idle, which severely hinders performance improvement. By analyzing these instructions, we have found that the same instructions exist in the convolution operation between different channels, which include instructions related to ifmaps and partsums. More aggressively, the same instruction also exists in the convolution operation in a channel, because the PE array reuses filters in the vertical direction and reuses ifmaps in the vertical and horizontal directions. The Fig. 7

shows the instructions required to calculate an ofmap value in each of the four PEs (corresponds to channel 1 in Fig. 5).

According to the instruction format in the Fig. 3, the load instructions ($inst3, inst4$) of PE1 and PE3 are the same. The $inst3/inst4$ operand index, address offset and base address index are $IF2/IF3, 3/4, 0/0$ respectively. Similarly, the $inst9, inst12, inst16, inst17$ in PE1 and PE3, the $inst22, inst23$ in PE1 and PE2, and the $inst11, inst12$ in PE2 and PE4 are also same. Also, the $inst13$ is the same for all PEs. To reduce initialization time, we use the instruction sharing algorithm (ISS) to set same instructions to same address.

We divide these instructions into three categories, ifmaps instructions (IF_type), which include load instructions and flow instructions with ifmaps. Filters instructions (F_type), these include load instructions, flow instructions, and madd instructions with filters. Partsum/Ofmap (P_type) instructions, including store instructions with partsums/ofmaps. For the same instruction, we use Algorithm 1 and Algorithm 2 to achieve the instruction sharing of different channels and the same channel respectively.

For convolution instructions of different channels, as previously analyzed, the pruning operation makes the filters instructions of different channels different, but has no effect on ifmaps instructions and partsums instructions. These instructions are still the same in different channels. Algorithm 1 achieves the sharing of the same instructions in different channels. Based on the instruction of channel 1 ($channel = 1$), for each instruction in other channels ($channel > 1$), the instruction address is updated to the instruction address in channel 1 if the type of this instruction is an ifmap (IF_type) or a partsum type (P_type), and it is the same as an instruction in channel 1. In the Fig. 5, the load instructions $inst1-inst4$, flow instructions $inst14-inst17$ in channel 2 belong to the ifmap type (IF_type), and store instruction $inst13$ belongs to partsum type (P_type). These instructions are the same as those in channel 1. Using algorithm 1, their addresses are updated to the instructions in channel 1.

Algorithm 2 implements instruction sharing between different PEs in a channel. According to the convolution reuse rules above, we divide the PE array into groups by columns because PEs in the vertical direction are mapped to the same filters. For each PE in the group, the filters type instructions are partially the

Algorithm 1: Instruction sharing algorithm for different channels.

Input: Codelet instruction, $Inst$; Convolution layer channels, C ; Instruction type, F_type ; IF_type ; P_type ;

Output: Address of instruction: $Addr$;

```

1 for each channel  $\in [2, C]$  do
2   for each  $Inst \in channel$  do
3     if ( $Inst$  type belongs to  $IF\_type$  or  $P\_type$ ) && ( $Inst$  is the same as
4       |   |  $Inst'$  of channel1) then
         |   |  $Addr(Inst) = Addr(Inst')$ 

```

Algorithm 2: Instruction sharing algorithm for the same channel.

Input: Codelet instruction, $Inst$; First PE, $PE1$; PE array group number, G ; PE number in group, P ; First PE in the group, $P1$; Instruction type, F_type ; IF_type ; P_type ;

Output: Address of instruction: $Addr$;

```

1 for each  $g \in [1, G]$  do
2   for each  $p \in [1, P]$  do
3     for each  $Inst \in p$  do
4       if ( $p > 1$ )  $\&\&$  ( $Inst$  type belongs to  $F\_type$ )  $\&\&$  ( $Inst$  is the same
5         as  $Inst'$  of  $P1$ ) then
6            $Addr(Inst) = Addr(Inst')$ 
7       if ( $inst$  type belongs to  $IF\_type$  or  $P\_type$ )  $\&\&$  ( $Inst$  is the same
           as  $Inst'$  of  $PE1$ ) then
              $Addr(Inst) = Addr(Inst')$ 

```

same. Based on the first PE in the group ($p = 1$), for each instruction in the other PEs in the group ($p > 1$), the address is updated to the instruction address in $P1$ if the instruction type is a filter type (F_type) and is the same as one in $P1$. In the Fig. 7, Algorithm 2 updates the madd instructions $inst9$ and $inst12$ addresses in PE3 to the $inst9$ and $inst12$ addresses in PE1, and also updates the $inst11$ and $inst12$ addresses in PE4 to the $inst11$ and $inst12$ addresses in PE2. For instructions of ifmaps (IF_type) and partsums type (P_type), using PE1 as the benchmark, an instruction address in another PE is updated to the instruction address in PE1 if it is the same as an instruction in PE1. In the Fig. 7, the $inst13$, $inst22$, $inst23$ of PE2, $inst3$, $inst4$, $inst13$, $inst16$, $inst17$ of PE3 and $inst13$ of PE4 are all updated to the addresses in PE1.

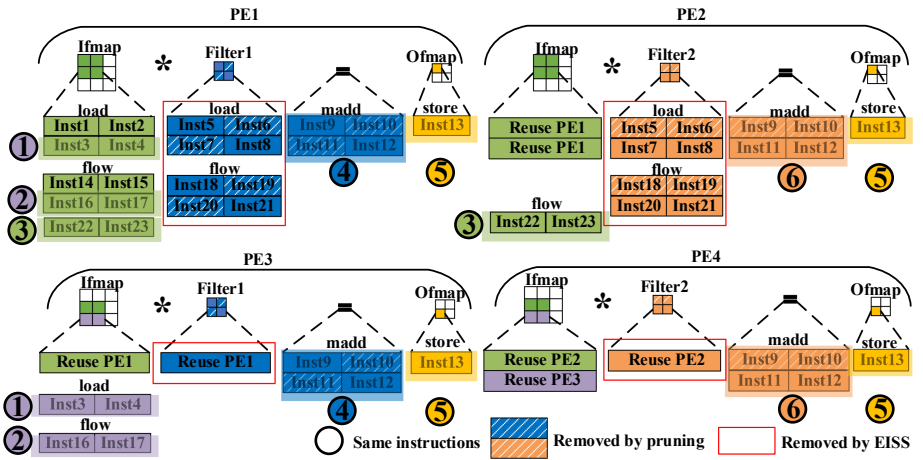


Fig. 7. Instructions required to perform a convolution operation in one channel. The PE array, filter, ifmap and ofmap size are $2 * 2$, $2 * 2$, $4 * 4$ and $2 * 2$, respectively.

4.2 Extended Instruction Sharing Strategy

To further reduce the initialization time, we use the static sparsity of weights to extend the instruction sharing strategy. For static sparsity, zero value weights are permanently removed from the network. And the sparseness of weights does not change with the input data. Once they are mapped on the PE, their related instructions will not change.

As mentioned above, the same instructions come from the ifmap and partsum related instructions, and the instructions where PEs are mapped to the same filters in the same channel. Due to the pruning operation, the related instructions of different filters are different, and they cannot benefit from the instruction sharing strategy. Based on its static characteristics, we use immediate multiply-accumulate instructions instead of multiply-accumulate instructions, which are not needed for dense instructions.

Opcode	F0	F1	F2	Fuction
Imm-madd	Operand index0	Imm	Operand index2	Value(F2)=Value(F0)*Imm+Value(F2)

Fig. 8. Imm-madd instruction format.

The Fig. 8 shows the format of the imm-madd instruction, which directly carries a value. For all filters, it eliminates the need to load non-zero weights from memory using load instructions. At the same time, flow instructions are no longer required to pass non-zero weights to other PEs. Therefore, it reduces the loading of filter-related instructions and non-zero weights, which is beneficial to memory access and performance. In structure, we added the control logic of immediate multiply-accumulate instruction decoding and transmission to achieve this operation. It only adds a little hardware overhead. In the Fig. 7, by using the extended instruction sharing strategy, the load instructions (*inst5*, *inst8*) and flow instructions (*inst18*, *inst21*) are removed in PE1. And the load instructions (*inst7*, *inst8*) and flow instructions (*inst20*, *inst21*) are removed in PE2. For ifmap and partum instructions, they are consistent with the instruction sharing strategy.

5 Experimental Methodology

In this section, we introduce the experimental methodology. We evaluated the our strategies using a DPU simulator based on the cycle-accurate and large-scale parallel simulator framework SimICT [26]. Table 1 lists the configurations of the simulator. We also implement DPU with RTL description in Verilog, synthesize it with Synopsys Design Compiler using TSMC 12 nm GP standard VT library. We calculate energy consumption of the applications according to circuit-level of atomic operations with Synopsys VCS using PrimeTime PX.

Figure 1 shows the structure of the simulation system which consists of all components of the DPU. The DPU in the simulator consists of 8×8 PE array,

and PE nodes are connected by 2D mesh networks. Each PE contains one 16-bits MAC (fix point multiply accumulate) in the SIMD8 model. There are 8 KB instruction buffer and 32 KB data buffer in each PE. To provide high network bandwidth, the on-chip networks consist of multiple independent physical 2D mesh networks. And to provide fast memory access, a 1 MB cache is added to the structure.

Table 1. Configuration of the instantiated dataflow accelerator.

Module	Configuration	Module	Configuration
Host	ARM core	Micc	Control messages, data transmission
PE	8 * 8, SIMD-8, 16-bits MAC, 8 KB Instruction Buffer, 2 KB Data Buffer	NoC	2D mesh, 1 control network, 1 data/memory-access network, 1 inter-PE network
Memory subsystem	DDR3 1333 MHz with 1 MB cache	Peak performance	1.887 TOPS

Benchmarks. To evaluate our strategies, we use representative neural networks AlexNet [17] and VGG-16 [23] convolution layers as benchmarks which have different sizes and parameter scales. Table 2 lists the corresponding sparsity of different kinds of layers. Each layer is translated into a CDG through our designed compiler based on LLVM [18] platform. With the limited space, the details of the compiler are not included in this paper.

Table 2. Configuration of convolutional layer parameters and sparsity for Alexnet and VGG-16.

	Layer	Ifmap	Filter	Sparsity	Layer	Ifmap	Filter	Sparsity
Alexnet	C2	$31^2 * 96$	$5^2 * 256$	0.62	C3	$15^2 * 256$	$3^2 * 384$	0.65
	C4	$15^2 * 384$	$3^2 * 384$	0.63	C5	$15^2 * 384$	$3^2 * 256$	0.63
VGG-16	C1_2	$224^2 * 128$	$3^2 * 128$	0.78	C2_1	$112^2 * 64$	$3^2 * 128$	0.66
	C2_2	$112^2 * 128$	$3^2 * 128$	0.64	C3_1	$56^2 * 128$	$3^2 * 256$	0.47
	C3_2	$56^2 * 256$	$3^2 * 256$	0.76	C3_3	$56^2 * 256$	$3^2 * 256$	0.58
	C4_1	$28^2 * 256$	$3^2 * 512$	0.68	C4_2	$28^2 * 512$	$3^2 * 512$	0.73
	C4_3	$28^2 * 512$	$3^2 * 512$	0.66	C5_1	$14^2 * 512$	$3^2 * 512$	0.65
	C5_2	$14^2 * 512$	$3^2 * 512$	0.71	C5_3	$14^2 * 512$	$3^2 * 512$	0.64

Evaluation Metric. In this paper, we refer to the instruction sharing strategy, and extended instruction sharing strategy as ISS, EISS, respectively. By applying these benchmarks, we evaluate our strategies in different ways. For the ISS and EISS, we compare with dense networks and verify the effectiveness of our method in terms of instruction execution times, execution time (performance) and energy. At the same time, we report the speedup of our DPU and Cambricon-X (peak sparse performance 544 GOP/s) over NVIDIA Titan Xp GPU (12 TFLOP/s peak performance, 12 GB GDDR5x, 547.7 GB/s peak memory bandwidth), a state-of-the-art GPU for deep learning. To run the benchmark, We use cuSparse library based on CSR indexing implement sparse network (GPU-cuSparse) [20]. We use nvidia-smi utility to report the power.

6 Experimental Results

6.1 Instruction Execution Times

Compared to dense networks, pruning operation removes redundant filter weights, thereby removing related filters instructions, which reduces the number of instruction executions. As shown in the Fig. 9, the load, calculation, and flow instructions execution times of Alexnet (VGG-16) sparse network are reduced by 10.38% (5%), 63.38% (66.95%), and 61.69% (48.06%), respectively, and the total instructions execution times are reduced by an average of 55.03% (54.24%). The EISS uses imm-madd instructions to further remove load instructions and flow instructions of non-zero weights and does not affect the execution times of other instructions, which reduces the load and flow instructions execution times by an average of 0.4% (1.33%) and 12.18% (8.77%) based on the pruning operation. Compared with dense networks, the EISS reduces load, calculation, and flow instructions execution times of Alexnet (VGG-16) by 10.78% (6.33%), 63.38% (66.95%), and 73.87% (56.83%), respectively, and reduces the total instructions execution times by an average of 55.8% (55.8%).

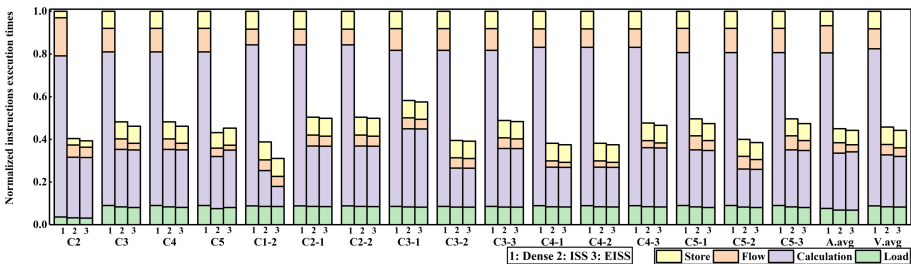


Fig. 9. Instruction execution times breakdown of convolutional layers normalized to dense layers.

6.2 Execution Time and Performance

The pruning method reduces the number of instruction executions and memory access to redundant data. It also reduce the execution time of the sparse network. By implementing ISS and EISS strategies, the initialization time of the sparse network is reduced, which reduces the total time and improves performance. As can be seen from the Fig. 10, compared with the dense network, the total execution time of ISS was reduced by 30.84% (Alexnet) and 19.65% (VGG-16) on average, respectively. In terms of performance, the average performance of the sparse network under the ISS is 1.45x (Alexnet) and 1.24x (VGG-16) that of the dense network and the maximum is 1.84x (Alexnet C2 layer). The EISS uses imm-madd instructions to replace the madd instructions of the dense network, eliminating non-zero weights instructions and data memory access, which further reduces the initialization time, execution time and improves performance. Compared with the dense network, the total execution time of EISS has been reduced by an average of 40.74% (Alexnet) and 31.35% (VGG-16), respectively. For performance, the EISS improves average performance by 14% on the basis of the ISS, which is 1.69x (Alexnet) and 1.45x (VGG-16) that of the dense network.

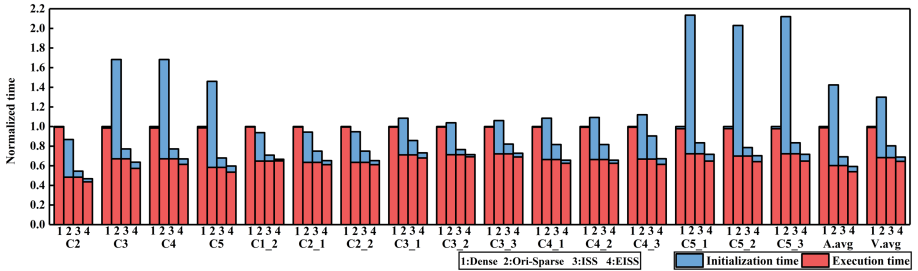


Fig. 10. Execution time breakdown of convolutional layers normalized to dense networks.

6.3 Energy

The ISS method improve the performance of sparse convolution, and also reduce energy consumption due to the reduction in instruction execution times, instruction and data memory access. We show the energy breakdown for convolutional layers in Fig. 11. Compared to dense convolution, the ISS reduces the total energy on average by 36.57% (Alexnet) and 33.05% (VGG-16). The EISS reduces the total energy by an average of 37.20% (Alexnet) and 34.26% (VGG-16). The EISS only slightly reduces the energy consumed by the data buffer, instruction buffer, and transmission due to the removal of non-zero weight instructions (load and flow) and data loading. Although it reduces the memory access of instructions and data, the energy proportion of memory access is small. Based on the ISS, the total energy is reduced by less than 1% on average.

6.4 Compare with Other Accelerators

In addition to comparing with dense networks, we also compare the performance and energy efficiency with GPU and Cambricon-X accelerator. In Fig. 12, we report the speedup comparison of DPU and Cambricon-X over GPU baseline across all the benchmarks. Compared to the GPU platform, on average, DPU achieves 2.34x (Alexnet), 2.12x (VGG-16) speedup for the sparse convolution. Cambricon-X achieves 1.34x (Alexnet), 1.42x (VGG-16) speedup. Therefore, DPU speedup 1.75x (Alexnet) and 1.49x (VGG-16) over Cambricon-X. Table 3 reports the energy efficiency comparison of the GPU, Cambricon-X and our DPU. Our energy efficiency is 12.56x of GPU, while Cambricon-X is 18.9x

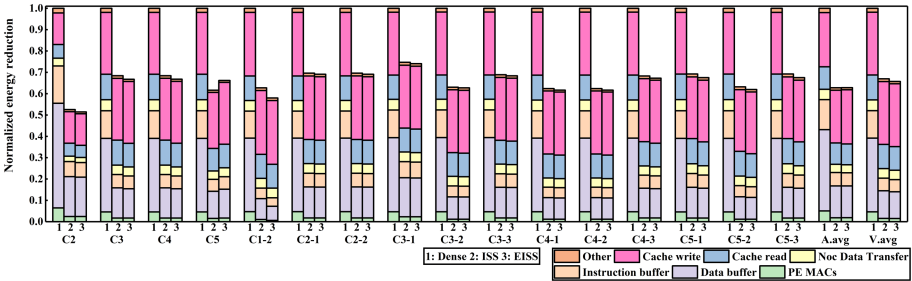


Fig. 11. Energy breakdown for convolutional layers normalized to dense layers.

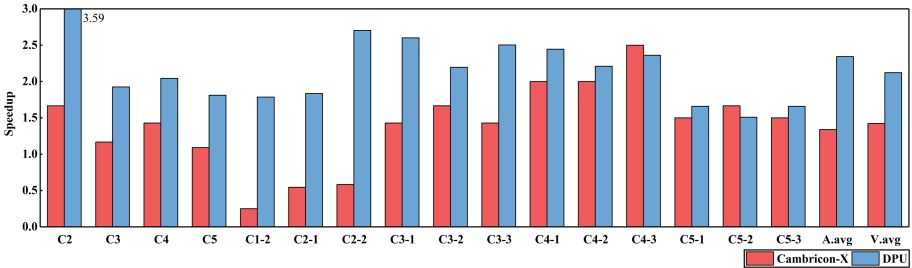


Fig. 12. Speedup of DPU and Cambricon-X sparse convolution over GPU baseline.

Table 3. GPU and Cambricon-X compared to DPU.

	GPU (Titan Xp)	Cambricon-X	This work
Technology	16 nm	65 nm	12 nm
Area	-	6.38 mm ²	13.82 mm ²
Power	243 W	954 mW	1.93 W
Throughput	7320 GFLOPS	544 GOPS	730 GOPS
Energy efficiency	30.1 GFLOPS/W	570.2 GOPS/W	378.2 GOPS/W

of GPU. It can be seen that the energy efficiency of DPU is not as good as Cambricon-X, but the performance is higher than it.

7 Related Work

Sparse neural networks have emerged as an effective solution to reduce the amount of computation and memory required. Many application-specific accelerators have been proposed to accelerate sparse networks.

Although DaDianNao [5] uses wide SIMD unit with hundreds of multiplication channels to achieve efficient processing of the neural network, it cannot take advantage of the sparsity of modern neural networks. CNV [1] decouples these lanes into finer-grain groups, and by using activation sparseness, only non-zero activation values are passed to the arithmetic unit, which accelerates the convolutional layer. However, it does not take advantage of the sparsity of weights. Eyeriss [3] implements a data gating logic to exploit zeros in the ifmap and saves processing power. However, it only has energy saving effect without acceleration effect. At the same time, it cannot accelerate the network with sparse weights. EIE [12], ESE [11] were proposed for leveraging the sparsity of full-connected layers in neural networks with CSC sparse representation scheme. However, EIE and ESE are no longer suitable for dense networks and sparse convolutional layer. Compared with these accelerators, we use the sparsity of the weights to accelerate the convolutional layer of the neural network.

Cambricon-X [29] designs Indexing Module (IM) efficiently selects and transfers no-zero neurons to connected PEs with reduced bandwidth requirement. Finally, it accelerates the convolutional layer and the fully connected layer, including dense networks and sparse networks. Cambricon-S [30] uses a coarse-grained pruning method which reduces the irregularity drastically sparse network. And designs a hardware structure to leverage the benefits of pruning method. It also achieves acceleration of convolutional layers and fully connected layers. SCNN [22] uses Cartesian product-based computation architecture which eliminates invalid calculations. Moreover, it designs the hardware structure to efficiently deliver weights and activations to a multiplier array. However, those application-specific accelerators sacrifice the flexibility of the hardware architecture to achieve the highest performance and energy efficiency, and cannot adapt to new applications. In this paper, applications can be implemented through flexible instruction scheduling based on the dataflow architecture.

8 Conclusion

In this paper, we propose two strategies to accelerate sparse CNN in the dataflow architecture. These strategies improve the performance of sparse networks and reduces energy consumption. We are the first to propose instruction sharing method to accelerate sparse networks based on a dataflow architecture. Our method is only applicable to networks with sparse weights, not to activate sparse networks because it is dynamically generated by using Relu function after each

operation. In the future, we will continue to study sparse acceleration at the hardware level.

Acknowledgement. This work was supported by the National Key Research and Development Program (2018YFB1003501), the National Natural Science Foundation of China (61732018, 61872335, 61802367, 61672499), Austrian-Chinese Cooperative R&D Project (FFG and CAS) Grant No. 171111KY5B20170032, the Strategic Priority Research Program of Chinese Academy of Sciences, Grant No. XDC05000000, and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (2019A07).

References

1. Albericio, J., Judd, P., Hetherington, T., Aamodt, T., Jerger, N.E., Moshovos, A.: Cnvlutin: ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Comput. Arch. News* **44**(3), 1–13 (2016)
2. Carter, N.P., et al.: Runnemed: an architecture for ubiquitous high-performance computing. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 198–209. IEEE (2013)
3. Chen, Y.H., Krishna, T., Emer, J.S., Sze, V.: Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-State Circuits* **52**(1), 127–138 (2016)
4. Chen, Y., Chen, T., Xu, Z., Sun, N., Temam, O.: Dianao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM* **59**(11), 105–112 (2016)
5. Chen, Y., et al.: DaDianNao: a machine-learning supercomputer. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 609–622. IEEE Computer Society (2014)
6. Dean, J., et al.: Large scale distributed deep networks. In: Advances in Neural Information Processing Systems, pp. 1223–1231 (2012)
7. Denil, M., Shakibi, B., Dinh, L., Ranzato, M., De Freitas, N.: Predicting parameters in deep learning. In: Advances in Neural Information Processing Systems, pp. 2148–2156 (2013)
8. Fan, D., et al.: SmarCo: an efficient many-core processor for high-throughput applications in datacenters. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 596–607. IEEE (2018)
9. Gao, G.R., Suetterlein, J., Zuckerman, S.: Toward an execution model for extreme-scale systems-runnemed and beyond. CAPSL Technical Memo 104, Department of Electrical and Computer Engineering, University of Delaware (2011)
10. Giorgi, R.: TERAFLUX: harnessing dataflow in next generation teradevices. *Microprocess. Microsyst.* **38**(8), 976–990 (2014)
11. Han, S., et al.: ESE: efficient speech recognition engine with sparse LSTM on FPGA. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 75–84. ACM (2017)
12. Han, S., et al.: EIE: efficient inference engine on compressed deep neural network. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 243–254. IEEE (2016)
13. Han, S., Pool, J., Tran, J., Dally, W.: Learning both weights and connections for efficient neural network. In: Advances in Neural Information Processing Systems, pp. 1135–1143 (2015)

14. Holi, J.L., Hwang, J.N.: Finite precision error analysis of neural network hardware implementations. *IEEE Trans. Comput.* **42**(3), 281–290 (1993)
15. Jaderberg, M., Vedaldi, A., Zisserman, A.: Speeding up convolutional neural networks with low rank expansions. arXiv preprint [arXiv:1405.3866](https://arxiv.org/abs/1405.3866) (2014)
16. Jouppi, N.P., et al.: In-datacenter performance analysis of a tensor processing unit. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), pp. 1–12. IEEE (2017)
17. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: *Advances in Neural Information Processing Systems*, pp. 1097–1105 (2012)
18. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, p. 75. IEEE Computer Society (2004)
19. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3431–3440 (2015)
20. Naumov, M., Chien, L., Vandermersch, P., Kapasi, U.: Cuspars library. In: *GPU Technology Conference* (2010)
21. Oriato, D., Tilbury, S., Marrocu, M., Pusceddu, G.: Acceleration of a meteorological limited area model with dataflow engines. In: *2012 Symposium on Application Accelerators in High Performance Computing*, pp. 129–132. IEEE (2012)
22. Parashar, A., et al.: SCNN: an accelerator for compressed-sparse convolutional neural networks. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), pp. 27–40. IEEE (2017)
23. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint [arXiv:1409.1556](https://arxiv.org/abs/1409.1556) (2014)
24. Szegedy, C., et al.: Going deeper with convolutions. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9 (2015)
25. Xiang, T., et al.: Accelerating CNN algorithm with fine-grained dataflow architectures. In: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 243–251. IEEE (2018)
26. Ye, X., Fan, D., Sun, N., Tang, S., Zhang, M., Zhang, H.: SimICT: a fast and flexible framework for performance and power evaluation of large-scale architecture. In: *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, pp. 273–278. IEEE Press (2013)
27. Ye, X., et al.: An efficient dataflow accelerator for scientific applications. *Fut. Gener. Comput. Syst.* **112**, 580–588 (2020)
28. Ye, X.: Applying cnn on a scientific application accelerator based on dataflow architecture. *CCF Trans. High Perform. Comput.* **1**(3–4), 177–195 (2019)
29. Zhang, S., et al.: Cambricon-X: an accelerator for sparse neural networks. In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 20. IEEE Press (2016)
30. Zhou, X., et al.: Cambricon-S: addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 15–28. IEEE (2018)