International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2021, 11(2): 243–258, doi:10.21655/ijsi.1673-7288.00251

International Journal of Software and Informatics

Research Article

Neural Network Instruction Set Extension and Code Mapping Mechanism

Wenqi Lou (娄文启), Chao Wang (王超), Lei Gong (宫磊), Xuehai Zhou (周学海)

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)

Corresponding author: Chao Wang, cswang@ustc.edu.cn

Abstract In recent years, Convolutional Neural Networks (CNN) have received widespread attention in the field of machine learning due to their high-accuracy performance in character recognition and image classification. Nevertheless, the compute-intensive and memory-intensive characteristics of CNN have posed huge challenges to the general-purpose processor, which needs to support various workloads. Therefore, a large number of CNN-specific hardware accelerators have emerged to improve efficiency. Though significantly efficient, previous accelerators are not flexible enough. In this study, classical CNN models are analyzed, and a domain-specific instruction set of 10 matrix instructions, called RV-CNN, is designed based on the promising RISC-V architecture. By abstracting CNN computation into instructions, the proposed design can provide sufficient flexibility for CNN and possesses a higher code density than the general ISA. On this basis, a code-to-instruction mapping mechanism is proposed. By using the RV-CNN to build different CNN models on the Xilinx ZC702, this paper found that compared to x86 processors, RV-CNN has on average 141 times the energy efficiency and 8.91 times the code density; compared to GPU, it has on average 1.25 times the energy efficiency and 1.95 times the code density. In addition, compared to previous CNN accelerators, the design supports typical CNN models while at high energy efficiency.

Keywords CNN; domain-specific instruction; RISC-V; code mapping; FPGA

Citation Lou WQ, Wang C, Gong L, Zhou XH. Neural network instruction set extension and code mapping mechanism, *International Journal of Software and Informatics*, 2021, 11(2): 243–258. http://www.ijsi.org/1673-7288/00251.htm

Artificial Neural Network (ANN), as an important branch of modern artificial intelligence, has been developing rapidly in recent years. As the continuation and evolution of ANN in the era of big data and deep learning effectively enhances the ability of traditional ANN algorithm to extract data features by increasing the depth of the network model. Convolutional Neural Network (CNN), as an algorithm widely used in deep learning, has been implemented in applications such as facial recognition[1], target detection[2], speech recognition[3], and natural language understanding[4] and has achieved remarkable results. Thanks to its excellent

This is the English version of the Chinese article “一种神经网络指令集扩展与代码映射机制. 软件学报, 2020, 31(10): 3074–3086. doi: 10.13328/j.cnki.jos.006071”.

Funding items: National Key Research and Development Program of China (2017YFA0700900, 2017YFA0700903); National Natural Science Foundation of China (61379040); Natural Science Foundation of Jiangsu Province, China (BK20181193); Youth Innovation Promotion Association CAS (2017497)

Received 2020-02-16; Revised 2020-04-04; Accepted 2020-05-09; IJSI published online 2021-06-22

performance in various application scenarios, CNN has become the focus of researchers and has been widely deployed in data centers and edge embedded devices.

However, the excellent recognition accuracy of CNN derives from the increasingly deeper and more complex network structure and the huge computation and memory. In recent years, the number of parameters in CNN models has reached several millions, and the amount of computation has reached several gigabits[5–7]. The intensive computation and memory of CNN have posed huge challenges to general-purpose processors. Therefore, many accelerators based on FPGA[8–10], GPU[5], and ASIC[11, 12] have emerged in recent years, and achieved higher performance and energy efficiency than CPU. Among the above three types of platforms, FPGA and ASIC generally have lower power consumption than GPU, which consume tens or even hundreds of watts. Although GPU has unique advantages in the training stage of CNN model, the mode of offline training for online prediction makes model-based inferencing more critical, in which the low power consumption of FPGA and ASIC enables them to be applied in a wider range of fields, such as an embedded platform with limited power. Therefore, this paper focuses on the relevant work of CNN accelerators based on these two types of platforms. However, it has been observed that such types of accelerators in previous work usually only accelerated specific network structures and layers, with relatively fixed mode and low flexibility.

In order to solve the above problem, the team of Chen Yunji proposed DianNao[11], a high-throughput ASIC chip for different machine learning applications, and designed the instruction in Very Long Instruction Word (VLIW) style, which supported CNN and Multi-Layer Perceptrons (MLPs). DaDianNao[13], proposed successively, was the SIMD implementation based on DianNao, in which the weight matrix of DaDianNao used for computing was solidified to the local eDRAM, thus reducing the frequency of reading memory. However, the VLIW-style instruction in both DianNao and DaDianNao provided a poor abstraction of computing process, so it was difficult to use them without understanding the underlying hardware. Subsequently, by abstracting the computation in ANN, the team designed a domain-specific instruction set, Cambricon[14], which contained instructions such as scalar, vector, and matrix, supporting a variety of neural networks and possessed higher code density and better performance than traditional ISA. Nonetheless, this instruction set was not dedicated to CNN, and part of CNN-specific data reuse and parallel computing in instructions were canceled for generality. For the application to CNN, Luca *et al*.[15, 16] proposed an extensible multi-core computing platform, namely PULP, and added a hardware convolution engine into PULP to accelerate the convolution operation. The single core of PULP was based on the RISC-V open-source architecture, and extended instructions such as dot product and pack-SIMD, so that the acceleration unit can be driven by using instructions directly, which saved operating system overhead. Since the team did not design instructions for other layers in the network, an efficient, flexible, and easy-to-implement CNN-specific instruction set is still needed.

This paper proposes a small and easy-to-implement CNN-specific instruction set, RV-CNN[17], after studying the computational mode of typical CNN models, which contains 10 matrix instructions and can support the reasoning process of multiple CNN structures flexibly. Then, the mapping process from CNN model description files to specific instructions is introduced. In terms of implementation, this paper extends the instruction set of a RISC-V architecture processor and optimizes the implementation of the instructions specifically. Finally, this paper compares and evaluates the instruction set and its implementation in terms of code density, performance, and energy efficiency by studying typical cases.

Section 1 of this paper describes design preferences for the domain-specific instruction. Section 2 describes in detail the format, the function, and the code mapping process of the domain-specific instructions, and shows some code samples. Section 3 compares the instruction

set qualitatively with a domain-specific instruction set. Section 4 introduces the hardware implementation of the instruction set based on RISC-V kernel. Section 5 shows the experimental procedures and results. Section 6 draws conclusions and introduces the next steps.

1 Design Preferences

This section mainly presents preferences in designing an efficient and easy-to-implement CNN-specific instruction set to design specific instructions.

RISC-V extension: CNN hardware accelerators usually worked as peripheral devices in the past. The host side read and wrote the accelerators through the drivers. Since a large amount of data is copied between user and kernel space, the time and resource overheads of the operating system are obviously unavoidable. The emergence of RISC-V architecture has brought more options for the work patterns of accelerators. The instruction set architecture consists of the basic instruction set and other optional instruction sets and is open-source and instruction customizable, thus providing users with the possibility of customizing the processor microarchitecture[18] and space for designing domain-specific instructions. Therefore, it is simpler and more efficient to control the accelerator module by designing domain-specific instructions based on RISC-V architecture. According to the above analyses, RISC-V is finally chosen as the target ISA, which is extended by CNN-specific instructions on the premise of keeping the basic kernel and each standard extension unchanged. Finally, the domain-specific instruction can complete the reasoning process of CNN in cooperation with the scalar and logic control instructions of RV32.

Data-level parallelism: There are many factors involved in designing a CNN-specific instruction set, among which the performance bottleneck should be paid attention to. Considering the topological structure of CNN stacked layer by layer and the independence of weight data in different layers, it is more effective to design matrix instructions to take advantage of the data-level parallelism rather than instruction-level parallelism in its operation. Studies have demonstrated that the energy consumption of computing of the Intel Xeon processor core only accounted for 37% of that of the whole core[19], and the rest of the energy consumption comes from architecture cost, which is not necessary for computing. In the design of domain-specific instructions, it is therefore advised to increase the granularity of instructions and amortize the overheads of instruction fetching, decoding, and control to the computation of multiple elements to improve operation efficiency effectively. In addition, when solving computations involving large amounts of data, matrix instructions can explicitly specify the independence between data blocks compared to traditional scalar instructions, thus reducing the data dependence detection logic. Moreover, matrix instructions have a high code density, so the data-level parallelism is the main focus here.

Scratchpad memory: Vector register groups are commonly found in vector architectures, where each vector register contains a vector of fixed length and allows the processor to operate on all elements of the vector at once. Scratchpad memory is a high-speed internal memory used to store temporary computing data on a chip, with the characteristics of direct addressing, low cost, and variable-length data access. Due to its low cost, a relatively large scratchpad memory is deployed and a Direct Memory Access (DMA) controller is integrated for fast data transmission. Considering that dense, continuous and variable-length data access often exists in CNN, scratchpad memory instead of the traditional vector register group is used here.

2 Design and Mapping of Domain-Specific Instructions

In this section, the composition of the domain-specific instruction set and the details of the function and format of the domain-specific instructions, under the design preferences presented

in Section 1, are illustrated. On this basis, the mapping process from the CNN model description file based on the deep learning framework to the specific instruction is introduced, and the code of the convolutional layer and the pooling layer realized by the domain-specific instruction is enumerated.

2.1 Extension of domain-specific instruction

The composition of the RV-CNN instruction set is shown in Table 1, i.e., the data transmission instructions, the logical instructions, and the computation instructions. Cooperating with some basic RV-32I instructions (not described repeatedly here), this instruction set can perform typical CNN class computations. The RV-CNN instruction set architecture is still consistent with the RISC-V architecture, belonging to the load-store architecture, and data is transmitted only through domain-specific instructions. Moreover, the instruction set still uses the 32 32-bit general-purpose registers of RV-32, which are used to store scalar values and register indirect addressing for scratchpad memory. In addition, we set up a Vector-Length Register (VLR) to specify the runtime length of the vector. The instructions are described in detail below.

Table 1 Overview of RV-CNN

Instruction types	Samples
Data transmission instructions	MLOAD/MSTORE
Computation instructions	MMM/MMA/MMS/MMSA
Logical instructions	MXPOOL/MNPOOL/APOOL/MACT

2.1.1 *Data transmission instructions*

In order to support matrix operations flexibly, data transmission instructions can complete the transmission of variable-size data blocks between the off-chip main memory and the on-chip scratchpad memory. Figure 1 shows the format of the Matrix LOAD instruction (MLOAD).

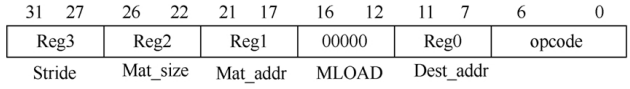


Figure 1 Matrix LOAD (MLOAD) instruction format

In Figure 1, Reg0 specifies the on-chip destination address, and Reg1, Reg2, and Reg3 specify the source address of the matrix, the size of the matrix, and the stride of adjacent elements, respectively. Specifically, the instruction completes the transfer of data from main memory to the scratchpad memory, where the stride field of the instruction can specify the span of adjacent elements, thus avoiding the "expensive" matrix transpose in memory. Accordingly, the Matrix STORE instruction (MSTORE) completes the data transmission from the scratchpad memory to the main memory with a format similar to MLOAD, but the stride field is often ignored to avoid discontinuous off-chip memory.

2.1.2 *Matrix computation instructions*

A CNN is mainly composed of the convolutional layer, the excitation layer, the pooling layer, and the full-connection layer. Most computations are operated in the convolutional layer[20]. In the computation of the convolutional layer, the convolution kernel moves continuously on the input feature image and performs a dot product in the overlap region to generate the input data of the next layer. In this process, the computation of one convolution kernel in different regions of the feature image is independent, so is the computation of different convolution kernels in the same region of the feature image. The Im2col (image to column) algorithm is used to convert 2-D convolution operation into matrix multiplication operation (the algorithm scheme is shown in Figure 2) to make full use of the parallelism in convolution computation.

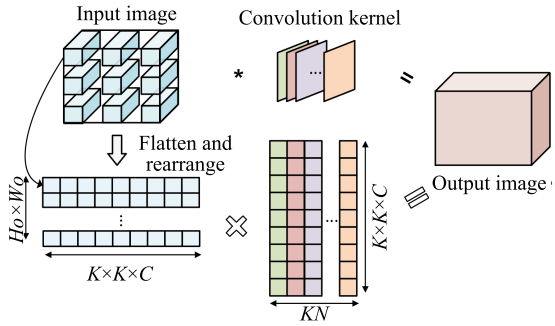


Figure 2 Matrix multiplication version of convolution

Once the 2-D convolution is mapped to a matrix multiplication operation, it is natural to use the MMM (Matrix-Multiply-Matrix) instruction to perform this operation. The instruction format is shown in Figure 3, where Reg0 specifies the destination address of the matrix output in the scratchpad memory; bits 16–12 are the functional fields of the instruction, indicating the matrix multiplication operation. Reg1 and Reg2 specify the source addresses of matrix 1 and matrix 2 in the scratchpad memory, respectively. The four bytes in Reg3 respectively represent the height (H), width (W), convolution kernel size (K), and convolution step (S) of the matrix. Due to the use of tiling techniques in actual operation, a single byte is sufficient here to store the corresponding information. Therefore, the parameter information during the operation of the convolution is packaged into 32' b $\{H, W, K, S\}$, and specified by Reg3. In addition, because the data is loaded in fragments, intermediate results often need to be accumulated. Instead of setting specific matrix addition instruction, this paper designs MMS instruction. After completing matrix multiplication, the instruction is added to the original value at the target address and then stored when part of the results is written into the target address, thus reducing data reloading. The format and meaning of each field of the instruction are consistent with MMM instruction which is specified by the function field, so it will not be illustrated repeatedly here. Moreover, in order to take greater advantage of data locality and reduce concurrent requests for reading or writing to the same address, specifically MMM is chosen for matrix multiplication instead of being decomposed into more fine-grained instructions (such as matrix-vector multiplication and vector dot product).

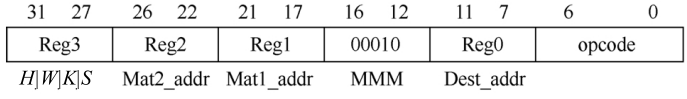


Figure 3 Matrix multiply matrix (MMM) instruction format

The full-connection layer is usually located at the end of the entire convolutional neural network to map the features learned from the previous layers to classify those features. The computation of the full-connection layer can be represented by matrix-vector multiplication, while the MMM instruction with different parameters can also represent the computation of the full-connection layer, so the same computation unit can be multiplexed by the convolutional layer and the full-connection layer.

2.1.3 *Matrix logic instructions*

Fusion[21], as a standard technology in DNN accelerator design at present, minimizes bandwidth limit by fusing partial layers to replace data input and output for a single layer with data transfer once for multiple layers. The advantages of fusion and the feature that the activation layer in CNN is usually immediately after the convolutional layer or the full-

connection layer make it very suitable to design corresponding coarse-granularity instructions to fuse the two layers. Moreover, the activation layer does not change the size of the input tensor, and each element is activated by the function one by one, which requires relatively few parameters. Therefore, the MMMA instruction is designed, so that the partial result of the convolutional layer or full-connection layer obtained by matrix multiplication can be output after activation. The format of MMMA instruction and meaning of each field are consistent with the MMM instruction, which is to specify the instruction by function field. However, the activation instruction is reserved to complete the activation of input data. The format of the activation instruction is shown in Fig. 4, and bits 31–27 of the instruction are used to determine the selection of the activation function, such as ReLU()/sigmoid()/Tanh().

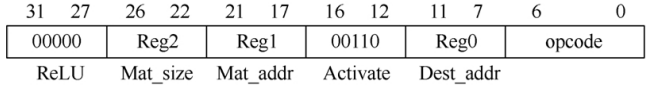


Figure 4 Matrix ACTivation (MACT) instruction format

The pooling layer subsamples each window of the input data into a single pool output by downsampling to reduce the size of the input image. In fact, compared to the convolutional layer and the fully-connected layer, the other layers in the CNN contain little computation and are limited by the time of data access. In some CNN models, it is effective to use fusion technology for the pooling layer and adjacent layers. However, unlike the activation layer, which has a relatively fixed position in CNN and is operated according to elements, the pooling layer is rather flexible. For example, when three convolutional layers are pooled after being stacked, so, here the pooling layer is treated as a separate layer. The instruction format of the MXPOOL for maximum pooling is shown in Figure 5, where Reg0, Reg1, and Reg2, respectively, represent the target address of output data, the source address of input data, and the length of input data. By referring to the idea of designing the MMM instruction, it is observed that the size of the pooling window is usually small, such as 2×2, 3×3, and 5×5, and the input data is usually processed by tiling technology, and thus it is sufficient to use a single byte to represent the height (H), width (W), size of the pool window (K), and stride size (S) of one tile of the input matrix that is processed. Such necessary information is packaged into one 32-bit value, which is specified by Reg3.

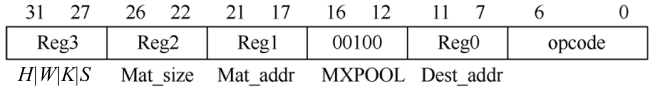


Figure 5 Matrix maximum (MXPOOL) instruction format

2.2 Code mapping mechanism

The RV-CNN instruction generation process is shown in Figure 6. The CNN model description files can be those in popular frameworks that deep learning engineers are familiar with such as Caffe, TensorFlow, or PyTorch. These description files are parsed by the model analyzer to generate parameter information for model building and weight information after rearrangement. On this basis, the data flow diagram should be built and operators are extracted according to the network parameter information, and then the extracted operators are mapped to different instructions in the instruction pool (RV-CNN instruction set) with different fusion strategies. Since the instructions designed here are all coarse-granularity instructions, operators with appropriate granularity should be extracted to be mapped to the target instruction set. After the domain-specific instruction is extracted, the fragmentation size should be determined

according to the parameter values of the hardware, such as the size of the on-chip scratchpad memory and the scale of the hardware computing resources. According to the reuse strategy, such as input reuse or weight reuse, instructions are arranged to generate the final code, and the RV32 basic instruction set should be used to load the parameter information into the register and complete the cycle control.

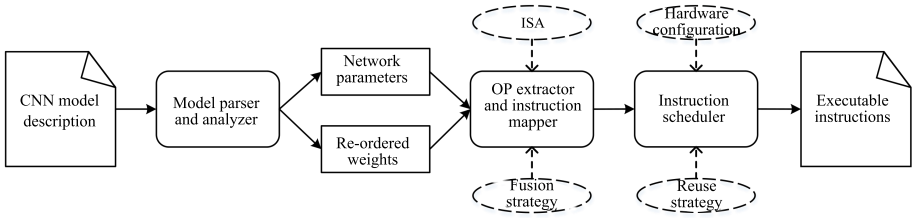


Figure 6 Generation process of RV-CNN instructions

2.3 Code examples

In order to illustrate the usage of the proposed domain-specific instruction set, two representative parts of CNN built by RV-CNN are enumerated, namely the convolutional layer and the pooling layer. The implementation of the convolutional layer includes the operation of the activation layer through fusion instructions. The code implementation of the full-connection layer is similar to that of the convolutional layer, with only slight differences in configuration parameters, so it will not be repeated here.

2.3.1 *Example of convolutional layer code*

The convolutional layer code realized by RV-CNN is shown in Figure 7. The left side is the convolutional layer code written in the Caffe framework (for schematic), which completed the feature extraction process of input feature image ($14 \times 14 \times 512$) by 512 groups of convolution kernel with size of 3×3 and stride of 1. The right side of the figure is the schematic code that performs the same function by specific instructions, assuming that there are sufficient resources on the hardware chip and that the data is arranged in the right order.

Since the instruction obtains the parameter information from the register, the first thing to do is to load the necessary information into the register. Here, the immediate 0x0E0E0301 is firstly loaded to the \$5 register. According to the description of instruction format in Section 2, the four bytes from high to low in the 32-bit data represent the height (14), width (14), convolution kernel size (3), and stride (1) of the input data, respectively, and then the vector register, VLR, is set to 16, which means that the length of the vector processed once is 16. Cycle counters, \$11 and \$12, are set here to complete the traversal of depth direction and different convolution kernels because of the tiling technique. Then the data transmission is completed via MLOAD/MSTORE by loading the physical address of the data in the off-chip DDR in registers, \$6, \$8, and \$10. The information in the above registers should be generated by the instruction generator or user according to the network model and hardware parameter information. On this basis, the actual computation process begins. The data is first loaded from the off-chip 0x30 000 and 0x50 000 to the on-chip destination address specified by \$1 and \$2, respectively. After the data is loaded, the computation is completed by the MMM instruction. During the computation process, the intermediate results are stored in the on-chip scratchpad memory. The following computation is completed by the MMMS instruction and accumulated with the on-chip intermediate results to reduce the unnecessary off-chip memory access. Finally, part of the final result is activated by operating the MMMSA instruction and is transmitted to the off-chip by the MSTORE instruction.

```
layer {
  type: "data"
  name: "data"
  top: "data"
  input_param: {
    shape: {
      dim: 512
      dim: 14
      dim: 14}
  }
}

layers {
 bottom: "data"
 top: "conv1"
 name: "conv1"
 type: CONVOLUTION
 convolution_param {
   num_output: 512
   pad: 1
    kernel_size: 3}
}

layers {
 bottom: "conv1"
 top: "conv2"
 name: "relu1"
 type: RELU
}
```

```
//$1: input mat1 address, $2: input mat2 address
//$3: temp variable address, $4: output size
//$6: mat1 address, $7: mat1 size, $8: mat2 address,
//$9: mat2 size, $10: output matrix address
//$11, $12: loop counter

    LI      $5,  0x0E0E_0301 // H=14, W=14, k=3, s=1
    LI      $VLR, 0x10       // set vector length (16)
    LI      $6,   0x30000
    LI      $8,   0x50000
    LI      $10,  0x70000
    LI      $11,  0x1E       // set loop counter (30)
    LI      $12,  0x20
L0: MLOAD   $1,  $6,  $7     // load tiled weights
    MLOAD   $2,  $8,  $9     // load tiled activations
    MMM     $3,  $2,  $1, $5 // mat1 x mat2
    ADD     $8,  $8,  $9     // update mat2 address
L1: MLOAD   $2,  $8,  $9
    MMMS    $3,  $2,  $1,  $5  // mat2 x mat2 & accumulate
    ADD     $8,  $8,  $9
    SUB     $11, $11, #1
    BGE     $11, #0,  L1     // if(loop counter>0) goto L1
    MLOAD   $2,  $8,  $9
    MMSA    $3,  $2,  $1,  $5  // mat2 x mat2 & accumu. & relu
    MSTORE  $3,  $10, $4     // store  results to address ($10)
    SUB     $12, $12, #1
    ADD     $6,  $6,  $7     // update mat1 address
    ADD     $10, $10, $4     // update output address
    BGE     $12, #0,  L0     // if (loop counter>0) goto L0
```

Figure 7 Example of convolutional layer code implemented by RV-CNN

2.3.2 *Example of pooling layer code*

The pooling layer code implemented by RV-CNN is demonstrated in Figure 8. On the left side, the Caffe framework is used to write the pooling layer code as a function example, which means to sample the maximum value of the input feature image ($14 \times 14 \times 512$) by the pooling window with a size of 2×2 and a stride of 2. The right side of the figure is schematic code that performs the same function using domain-specific instructions, assuming that there are sufficient resources on the hardware chip and that the data is arranged in the right order.

Since the pooling layer does not contain weight data and does not accumulate in the depth direction, the code implemented by the RV-CNN is simpler than the convolutional layer code if the input image size is appropriate. After loading the parameters into the corresponding register, the MLOAD instruction is used to load the data to be processed from the off-chip 0x10 000 ($6) to the on-chip destination address $1. After the input data is loaded, the MXPOOL instruction is used to down sample the data and the results are stored at the temporary address $5 on the chip. After sampling, the results are transmitted by MSTORE to the off-chip address 0x40 000 ($7). During the process, the output does not accumulate on the chip. The loop counter in the code is intended to control the traverse along the depth direction of the input data. A single load, pooling, and loadout complete the sampling of a tile of the input data, and then the load and load out addresses and counter values are updated.

3 Comparison Between RV-CNN and Domain-Specific Instruction Set

At present, the Cambricon instruction set is considered to be one of the most representative instruction sets in the field of neural networks, and the vector instruction set extension (RV-V) based on the RISC-V architecture is also considered as an instruction set to accelerate neural

network computing. Therefore, this section will carry out qualitative analysis and comparison between RV-CNN and the above two typical instruction sets in terms of application scope, granularity, and mapping mechanism of instruction sets.

```
layers {
  name: "pool"
  type: POOLING
  bottom: "data"
  top: "pool"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

```
//$1: input mat address, $2: feature map size
//$3: loop counter, $5: temp variable address
//$6: mat address, $7: output mat address, $8: output size

    LI      $4,    0x0E0E_0202  // H=14,W=14,k=2,s=2
    LI      $VLR, 0x10          // set vector length (16)
    LI      $3,    0x20         // set loop counter (32)
    LI      $6,    0x10000
    LI      $7,    0x40000
L0: MLOAD   $1,    $6,  $2      // load partial activations
    MXPOOL  $5,    $1,  $2, $4  // subsample
    MSTORE  $5,    $7,  $8      // store mat to address ($7)
    ADD     $7,    $7,  $8
    ADD     $6,    $6,  $2      // update output address
    SUB     $3,    $3,  #1
    BGE     $3,    #0,  L0      // if (loop counter>0) goto L0
```

Figure 8 Example of pooling layer code implemented by RV-CNN

Scope of application: The RV-V instruction set is designed to take advantage of data-level parallelism in applications, which can be widely used in scientific computing, data signal processing, machine learning, and other fields. The Cambricon instruction set is designed for more than ten kinds of network models in the field of neural networks, such as CNN, Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. In contrast, the RV-CNN instruction set is mainly designed for CNN in the field of neural network, which involves few types of operations. As a result, the first two instruction sets target a wider range of domains and are more difficult to design, which is also reflected in the type and number of instructions contained in the instruction set. The RV-V instruction set already contains over 60 instructions (draft version 0.8), the Cambricon instruction set contains 47 instructions, while the RV-CNN instruction set contains only 10 instructions.

Coarse or fine granularity: RV-V and RV-CNN instruction sets contain vector and matrix instructions, respectively, while the Cambricon instruction set contains scalar, vector, and matrix instructions. It is because Cambricon is conceptually a complete neural network instruction set that it includes scalar instructions, but vector and matrix instructions are used to speed up the computation. Therefore, the RV-CNN instruction set has the largest instruction granularity among the three instruction sets, followed by the Cambricon instruction set, and the granularity of RV-V is the smallest since all its instructions are vector instructions. This is also related to the application scope of the instruction set. As the application field used for the design of the RV-V instruction set is most extensive, the set needs to extract the common parts from various computing operations. In view of the limitations of hardware scale and power consumption, this process often requires continuously dividing different computing processes to seek computing commonality considering the algorithm characteristics, thus improving the expressivity of the instruction set. Compared with the previous two sets, the RV-V instruction set possesses the smallest granularity, therefore.

Code mapping mechanism: The code mapping of the Cambricon instruction set is based on the framework, providing the popular programming framework with adaptive machine learning high-performance library and software runtime support. It not only provides the framework with rich operators and computational flow diagram methods to construct the entire network, but also controls the hardware by calling the built-in driver to generate instructions. The RV-CNN instruction set's data mapping process begins with the model's description file

based on the framework. Different from Cambricon that modifies the framework, RV-CNN only analyzes the description file of the model in the framework to extract the model structure and weight information, further establishing a mapping between operators and instructions in the model with the fusion strategy, then carries on the instruction arrangement with the reuse strategy, finally forms the executable file by assembling. As the RV-V instruction set is currently ongoing, there is no available compiler to complete the automatic vectorization of the code, and the assembly instructions still need to be written by the users. However, a large number of fine-granularity vector instructions makes register allocation and instruction arrangement more difficult in the programming process.

4 Hardware Implementation of RV-CNN

This section first introduces the overall structure of the open-source processor core containing the RV-CNN instruction extension and details the execution flow of the instructions. Then, the composition of the matrix unit corresponding to the RV-CNN instruction and the functions of its subunits are described, in which the structure of the matrix multiplication unit is shown in detail. Finally, the optimization details of the matrix unit are introduced.

4.1 Overall architecture

The main features of the RISC-V processor core with the RV-CNN extension and a simplified pipelined architecture are shown in Figure 9. It can be seen that it contains five basic pipeline stages: fetching, decoding, execution, access, and writing back. The matrix computing unit is in the execution stage of the pipeline and is used to complete the execution of matrix instructions. After the fetching and decoding, the instructions in the basic instruction set will enter the ALU and then the next phase. When the current instruction is identified as a matrix instruction at the decoding stage, the decoder will obtain the corresponding information from the register and save it, which will be sent into the matrix unit in the next cycle. The matrix unit detects the state of the corresponding functional components according to instruction information received to determine whether to execute. As the address space of the scratchpad register on the chip is visible to users, the matrix unit can interact with the memory through the matrix data transmission instruction. Therefore, matrix instruction will not be accessed or written back after the matrix instruction enters the matrix unit, while the access of the rest instructions goes through the cache without going through the matrix unit. It avoids unnecessary data-dependent detection and automatic data exchange of the hardware. In addition, since the matrix unit contains a large number of computing units, it has its own pipeline structure inside. The decoder decides whether to stop the pipeline according to whether the matrix unit can accept matrix instruction information. Given the continuing and intensive data access matrix instructions, a DMA controller is integrated outside the scratchpad memory to meet the data access requirements of the matrix unit. It should be noted that the data involved in the execution of the computational and logical instructions in the matrix unit need to already exist on the chip, which requires strict control of the program.

4.2 Matrix unit

The overall structure of the matrix unit is shown in Figure 10, which consists of the input and output units, the matrix multiplication unit, the activation unit, the pooling unit, and the internal controller, and the orange and gray arrows represent control flow and data flow, respectively. The matrix unit stores the instruction information and register information received. The internal controller (as a finite state machine) is the control center of the matrix unit, which will wake up the sub-components (if available) to complete the corresponding task based on the control information. Otherwise, it generates a feedback signal indicating that the corresponding functional unit is busy. The buffer module is essentially an on-chip memory from

which the computing core in the matrix unit fetches data and writes the results. Computing units correspond roughly one-to-one with coarse-granularity instructions, except fusion instructions that can start multiple computing cores simultaneously. Finally, the input-output module is responsible for data transmission between the matrix unit and the on-chip scratchpad memory based on the valid address.

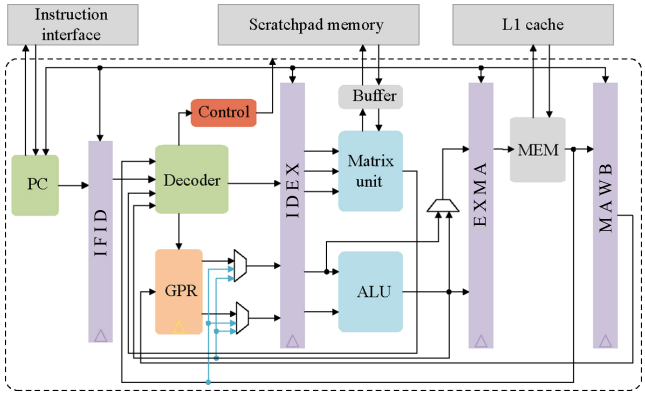


Figure 9 Simplified block diagram of processor core with RV-CNN extension

Since the matrix multiplication unit is shared by both the convolution layer and the fully-connected layer and it performs most of the computation, the implementation of this unit is critical to performance. Here, the matrix multiplication realized by a systolic array structure is adopted, and its structure is shown in the right part of Figure 10. Systolic arrays are an efficient and simple implementation of matrix multiplication by binding MAC (multiply-accumulate) units together in a two-dimensional grid. Except for the computing units in the outermost layer of the array (in this case, the leftmost and uppermost), which are directly connected to the on-chip buffer to obtain data, the rest of the units get input from their neighbors. This approach will significantly reduce the fan-in and fan-out of on-chip buffering when the MAC array has larger size. In addition, the flow of data among MAC units also promotes data reuse. For example, when the MAC array is 12×16, the elements in matrix B are transferred from left to right in the array at different times in rows, and the output data is reused 16 times. Similarly, the elements in matrix A are transmitted from top to bottom in the array at different times, and the output data is reused 12 times. In this process, with pipelining optimization, 192 MAC operations can be performed with only 28 inputs per cycle into the array. In addition, the short local interconnection reduces the difficulty of layout and wiring, so the systolic array is chosen as the implementation of matrix multiplication.

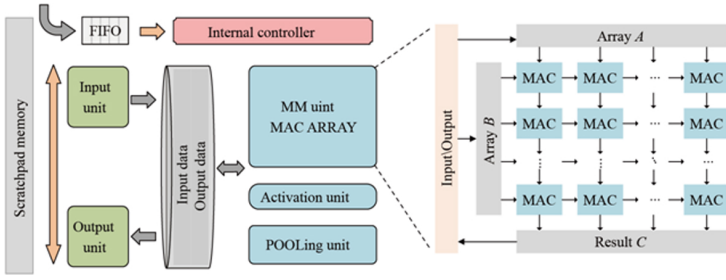


Figure 10 Block diagram of the matrix unit

4.3 Detail optimization

Data reuse: Im2col operations on the input data when the convolutional layer is processed can be costly in terms of memory footprint and bandwidth. For example, when the convolution kernel stride is 1, compared with the original input data, the converted input matrix consumes about $K \times K$ times the original memory. To maintain the benefits of Im2col operations while reducing the additional overhead, this paper implements an on-chip Im2col buffer, and the tiling data is rearranged and expanded on the chip based on the address. The Im2col buffer can effectively increase data reuse. Although it does not reduce the number of accesses to input data, it can replace the expensive off-chip access with the cheap on-chip access, thus greatly reducing the overhead of the additional data access and external bandwidth pressure caused by this operation.

Data quantization: Traditionally, whether CNN is in the training stage or the prediction stage, its data type is the 32-bit single-precision floating-point number, mainly because it is the standard data format of modern GPU. However, it is found that CNN is inherently robust to finite numerical precision, and it is not necessary to use floating-point computation in the prediction stage. Through retraining and specific fine-tuning, the accuracy loss caused by fixed-point numbers for the prediction can be negligible (less than 1%)[22]. For many CNNs, even 8-bit width can provide enough precision. The quantization was taken because the low width representation of weights and activation helps to avoid expensive floating-point computations while significantly reducing bandwidth requirements and memory footprint. DianNao[11] shows that the area and power consumption of the 32-bit multiplier are one order of magnitude higher than that of the 16-bit multiplier through the 65 nm process of Taiwan Semiconductor Manufacturing Company (TSMC). On this basis, 16-bit fixed-point numbers are used for all computation cores of matrix units in this paper.

5 Experiments and Results

A processor core based on the RISC-V architecture containing the instruction set extension is built on the FPGA platform to verify the validity of the proposed domain-specific instruction. On this basis, AlexNet and VGG16, two convolutional neural networks of different scales, are used for evaluation, and the prototype system is compared with Cambricon, CPU, GPU, and other FPGA accelerators for analysis.

5.1 Experimental methods

(1) FPGA implementation of the prototype system

A ZC702 development board is used as the experimental platform, which is an embedded FPGA platform, including an XC7Z020 FPGA chip and 1 GB DDR3 onboard memory, which can provide 4.2 GB/s off-chip data access bandwidth. The basic RISC-V core and matrix unit are controlled by Verilog, the hardware description language, while the subunits of the matrix unit are designed by the Xilinx Vivado HLS 2017.4 high-level synthesis tool, and the complete hardware engineering is implemented by Xilinx Vivado, the integrated development environment.

(2) CPU test baseline

Two target network models are deployed on the CPU platform using the Caffe deep learning framework (CPU-only). The CPU configuration is an Intel i7-4790 K with four physical cores, a maximum thread count of eight, a frequency of 4 GHz, and a DDR3 memory of 16 GB.

(3) GPU test baseline

The GPU version was tested using the Caffe deep learning framework (GPU-only) and the CUDN5.1 accelerator library to deploy two target network models on the GPU platform. The

CPU configuration is an NVIDIA Tesla K40C GPU with a maximum thread count of 2 880, a frequency of 745–875 MHz, and a GDDR5 video memory of 12 GB.

5.2 Results

In this section, the resource consumption and power consumption of the prototype system on the FPGA platform is first reported, and then the design is compared with Cambricon, CPU, GPU, and previous accelerators based on FPGA from three aspects of code density, performance, and energy efficiency.

The resource consumption of the prototype system on the target platform and the power consumption of FPGA are obtained after the deployment report of the prototype system in the Vivado tool is viewed, as shown in Table 2.

Table 2 Hardware resource utilization deployed on Xilinx ZC702 platform

Resource	DSP	BRAM	LUT	FF	Power (W)
Total amount	220	280	53 200	106 400	
Used	200	181	26 177	30 665	2.12
Utilization(%)	91	65	49	29	

(1) Comparison with Cambricon, CPU and GPU platforms in code density, performance, and energy efficiency

Code density: The domain-specific instruction proposed in this paper is not only suitable for accelerating CNN applications, but also provides support for other deep learning algorithms (such as MLPs) with similar computational patterns. The code density between RV-CNN instruction and the basic RV32, ARM, x86, and GPU is compared by using RV-CNN, C, and CUDA-C to implement the popular CNN model and measure its code length. The results are shown in Figure 11, where the length of the code realized by RV-CNN instruction set is taken as the baseline. It can be seen that compared with the original RV32 (IMF) instruction set, the code length after the extension of RV-CNN is reduced by 10.10 times. Compared with GPU, x86, and ARM instruction sets, the code length of RV-CNN is reduced by 1.95 times, 8.91 times, and 10.97 times respectively. With the code length of x86 instruction as the benchmark, the RV-CNN code length is 1.51 times shorter than that of the Cambricon.

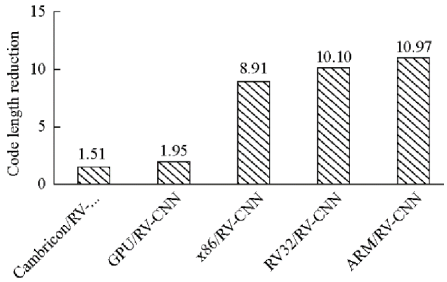


Figure 11 Reduction of code length against Cambricon, GPU, x86, RV32, and ARM

Performance and energy efficiency: Figure 12 shows the performance and energy efficiency of the design compared with CPU and GPU under the inference process test of two neural networks (AlexNet and VGG16). All values are normalized to the experimental results of the CPU. On the Xilinx ZC702 platform, this design performs outperforms the CPU does in implementing the two kinds of networks, accelerating them by 2.64 times and 4.23 times, respectively. However, the platform is embedded, and DSP (91% usage), the hardware resource for computing, is the main performance bottleneck, so the performance of implementing both networks lags behind the GPU. In terms of energy efficiency, with performance per watt

(GOPs/W) as the benchmark in reasoning for AlexNet and VGG-16, the design provides 101.49-fold and 167.62-fold improvement respectively compared with CPU, while 1.06-fold and 1.40-fold improvement respectively, compared with GPU. Since the Cambricon accelerator is designed for ASIC, the performance density (op/multiplier/cycle), namely the number of operands completed by the multiplier per cycle, is compared. With the GPU test data as the baseline, RV-CNN has a 1.16-fold improvement compared with Cambricon.

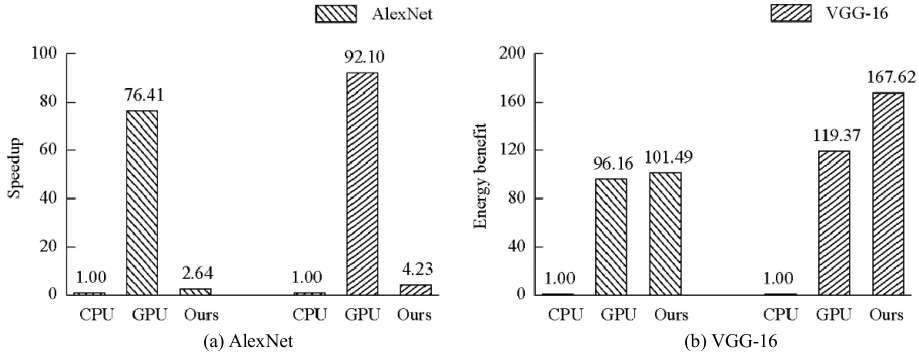


Figure 12 Comparison of prototype system with CPU and GPU in performance and energy efficiency

(2) Comparison with other FPGA accelerators

Table 3 lists the results of comparison between this design and existing typical FPGA accelerators. Due to different quantization strategies and hardware deployments for different work, it is hard to choose an effective and accurate comparison method. With Giga Operands Per second (GOPs) as the performance criterion, previous efforts can achieve better performance than ours, but the higher performance consumes more resources, such as DSP and LUT resources, power consumption increasing accordingly. With performance per watt (GOPs/W) as the energy efficiency criterion, the design in the paper is more efficient than previous accelerators while maintaining flexibility.

Table 3 Comparison between prototype system and previous FPGA-based accelerator deployment

	FPGA 2015[23]	FPGA 2016[24]	FCCM 2017[25]	Ours
Platform	Virtex7 VX485 T	Stratix5 GSD8	Stratix5 GSMD5	Zynq XC7Z020
Frequency (MHz)	100	120	150	100
Model	AlexNet	VGG16	VGG16	AlexNet/VGG16
Width	32-bit float	16-bit fixed	16-bit fixed	16-bit fixed
Performance (GOPs)	61.62	117.8	364.36	21.77/35.95
Power consumption (W)	18.61	25.8	25	2.12
Energy efficiency (GOPs/W)	3.31	4.57	14.57	10.27/16.96

6 Conclusions

CNN is widely used in character recognition and detection of target fields, which makes its performance very significant. By analyzing the computation mode of typical CNN, this paper proposes an efficient and easy-to-implement domain-specific instruction set, called RV-CNN, which contains 10 coarse-granularity matrix instructions and provides support for the reasoning process of the CNN model flexibly. On this basis, the mapping process of CNN model description file to RV-CNN instruction is introduced, and then RV-CNN is compared with typical specific instruction set from different aspects with qualitative analysis. In terms of instruction implementation, the instruction set extends the RISC-V processor core, and the

corresponding matrix units are embedded into the classical five-stage pipeline in a tightly coupled way. Finally, the design is implemented comprehensively on the Xilinx ZC702 platform, and tested with typical neural networks. The results indicate that the prototype system has the highest energy efficiency and code density compared to the Intel i7-4790K processor and the Tesla K40C GPU. In addition, compared to previous accelerators, the prototype system demonstrates superior energy efficiency while maintaining flexibility.

At the moment, new CNN networks are emerging in an endless stream, whereas instruction optimization for operations such as deep separable convolution should be considered to improve efficiency. In addition, the design and implementation of the extension instruction should rely on collaborative optimization for the characteristics of RISC-V. Finally, the process of code mapping based on model and hardware information is not yet automated, and improvements are being planned in the above aspects.

References

[1] Wu F, Kong Y, Dong W, *et al*. Gradient-aware blind face inpainting for deep face verification. Neurocomputing, 2019, 331(FEB.28): 301–311.

[2] Redmon J, Divvala S, Girshick R, *et al*. You only look once: Unified, real-time object detection. Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition. IEEE, 2016. 779–788. [doi: 10.1109/CVPR.2016.91]

[3] Sainath TN, Mohamed A, Kingsbury B, *et al*. Deep convolutional neural networks for LVCSR. Proc. of the IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing. IEEE, 2013. 8614–8618. [doi: 10.1109/ICASSP.2013.6639347]

[4] Collobert R, Weston J, Bottou L, *et a*l. Natural language processing (Almost) from scratch. Journal of Machine Learning Research, 2011, 12(1): 2493–2537. [doi: 10.1016/j.chemolab.2011.03.009]

[5] Krizhevsky A, Sutskever I, Hinton G. ImageNet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems, 2012, 25(2): 1097–1105. [doi: 10.1145/3065386]

[6] Szegedy C, Liu W, Jia Y, *et al*. Going deeper with convolutions. Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition. IEEE, 2014. 1–9. [doi: 10.1109/CVPR.2015.7298594]

[7] He K, Zhang X, Ren S, *et al*. Deep residual learning for image recognition. Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition. IEEE, 2016. 770–778. [doi: 10.1109/CVPR.2016.90]

[8] Gong L, Wang C, Li X, *et al*. MALOC: A fully pipelined FPGA accelerator for convolutional neural networks with all layers mapped on chip. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2018, 37(11): 2601–2612. [doi: 10.1109/TCAD.2018.2857078]

[9] Wang C, Gong L, Yu Q, *et al*. DLAU: A scalable deep learning accelerator unit on FPGA. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2017, 36(3): 513–517. [doi: 10.1109/TCAD.2016.2587683]

[10] Wang C, Li X, Chen Y, *et a*l. Service-oriented architecture on FPGA-based MPSoC. IEEE Trans. on Parallel and Distributed Systems, 2017, 28(10): 2993–3006. [doi: 10.1109/TPDS.2017.2701828]

[11] Chen T, Du Z, Sun N, *et al*. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. Proc. of the Architectural Support for Programming Languages and Operating Systems. ACM, 2014. 269–284. [doi: 10.1145/2541940.2541967]

[12] Moons B, Verhelst M. An energy-efficient precision-scalable ConvNet processor in 40-nm CMOS. IEEE Journal of Solid-state Circuits, 2017, 52(4): 903–914. [doi: 10.1109/JSSC.2016.2636225]

[13] Chen Y, Luo T, Liu S, *et al*. DaDianNao: A machine-learning supercomputer. Proc. of the Int'l Symp. on Microarchitecture. IEEE, 2014. 609–622. [doi: 10.1109/MICRO.2014.58]

[14] Liu S, Du Z, Tao J, *et al*. Cambricon: An instruction set architecture for neural networks. Proc. of the 43rd ACM/IEEE Annual Int'l Symp. on Computer Architecture (ISCA). IEEE, 2016. 393–405. [doi: 10.1145/3007787.3001179]

[15] Conti F, Rossi D, Pullini A, *et al*. PULP: A ultra-low power parallel accelerator for energy-efficient

and flexible embedded vision. Journal of Signal Processing Systems, 2015, 84(3): 339–354. [doi: 10.1007/s11265-015-1070-9]

[16] Schiavone MG, Benini L. A near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. IEEE Trans. on Very Large Scale Integration Systems, 2017, 25(10): 2700–2713.

[17] Lou W, Wang C, Gong L, *et al*. RV-CNN: Flexible and efficient instruction set for CNNs based on RISC-V processors. Proc. of the Int'l Symp. on Advanced Parallel Processing Technologies. Cham: Springer, 2019. 3–14.

[18] Bao Y, Wang S. Labeled von Neumann architecture for software-defined cloud. J. of Computer Science and Technology, 2017, 32(2): 219–223. [doi: 10.1007/s11390-017-1716-0]

[19] Cong J, Ghodrat MA, Gill M, *et al*. Accelerator-rich architectures: Opportunities and progresses. In: Proc. of the Design Automation Conf. IEEE, 2014. 1–6. [doi: 10.1145/2593069.2596667]

[20] Lu LQ, Zheng SZ, Xiao QC, *et al*. Accelerating convolutional neural networks on FPGAs. Science in China (Information Sciences), 2019, 49(3): 277–294. [doi: 10.1360/N112018-00291]

[21] Alwani M, Chen H, Ferdman M, *et al*. Fused-layer CNN accelerators. Proc. of the 49th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO) IEEE, 2016. 1–12. [doi: 10.1109/micro.2016.7783725]

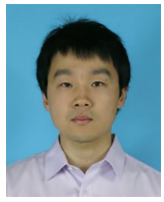
[22] Gysel P, Pimentel J, Motamedi M, *et al*. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. IEEE Trans. on Neural Networks, 2018, 29(11): 5784–5789.

[23] Zhang C, Li P, Sun G, *et al*. Optimizing FPGA-based accelerator design for deep convolutional neural networks. Proc. of the 2015 ACM/SIGDA Int'l Symp. on Field-programmable Gate Arrays. ACM, 2015. 161–170.

[24] Suda N, Chandra V, Dasika G, *et al*. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. Proc. of the 2016 ACM/SIGDA Int'l Symp. on Field-programmable Gate Arrays. ACM, 2016. 16–25.

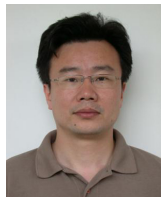
[25] Guan Y, Liang H, Xu N, *et al*. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. Proc. of the 25th IEEE Annual Int'l Symp. on Field-programmable Custom Computing Machines (FCCM). IEEE, 2017. 152–159.

Wenqi Lou, Ph.D. candidate. His research interests include neural network processors and reconfigurable hardware accelerators.



Lei Gong, postdoctor, CCF professional member. His research interests include computer system architecture, reconfigurable hardware accelerators, and neural network processors.

Chao Wang, Ph.D., associate professor, CCF senior member. His research interests include neural network accelerators and deep learning processors.



Xuehai Zhou, Ph.D., professor, doctoral supervisor, CCF senior member. His research interests include computer architecture and embedded systems.