RV-CNN: Flexible and Efficient Instruction Set for CNNs Based on RISC-V Processors

Wenqi Lou, Chao Wang(✉), Lei Gong, and Xuehai Zhou

School of Computer Science, University of Science and Technology of China, Hefei, China
{louwenqi,leigong0203}@mail.ustc.edu.cn, {cswang,xhzhou}@ustc.edu.cn

Abstract. Convolutional Neural Network (CNN) has gained significant attention in the field of machine learning, particularly due to its high accuracy in character recognition and image classification. Nevertheless, due to the computation-intensive and memory-intensive character of CNN, general-purpose processors which usually need to support various workloads are not efficient for CNN implementation. Therefore, a great deal of emerging CNN-specific hardware accelerators is able to improve efficiency. Although existing accelerators are significantly efficient, they are often inflexible or require complex controllers to handle calculations and data transfer. In this paper, we analyze classical CNN applications and design a domain-specific instruction set of 9 matrix instructions, called RV-CNN, based on the promising RISC-V architecture. By abstracting CNN into instructions, our design possesses a higher code density and provides sufficient flexibility and efficiency for CNN than general-purpose ISAs. Specifically, the proposed instructions are extended to RISC-V ISA as custom instructions. Besides, we also introduce micro-architectural optimizations to increase computational density and reduce the required memory bandwidth. Finally, we implement the architecture with the extended ISA and evaluate it with LeNet-5 on the datasets (MNIST, Caltech101, and Cifar-10). Results show that compared with the Intel Core i7 processor and Tesla k40c GPU, our design has 36.09x and 11.42x energy efficiency ratio and 6.70x and 1.25x code density respectively.

Keywords: CNN · RISC-V · Domain-specific instructions · FPGA

1 Introduction

Convolutional neural network (CNN), a category of feed-forward artificial neural networks, is well known for its high precision in the fields of character recognition, image classification, and face detection [10,13,14]. Inspired by the visual cortex of the brain, CNN is typically composed of multi-layer networks. In recent years, with the improvement of recognition accuracy, the depth of the network

P.-C. Yew et al. (Eds.): APPT 2019, LNCS 11719, pp. 3–14, 2019.
https://doi.org/10.1007/978-3-030-29611-7_1

has been considerably increased. However, a deeper network structure means more computation and more weight data access, which makes the low efficiency of general-purpose processors in performing CNN calculations intolerable. Therefore, various accelerators based on FPGA [8,15,16], GPU [9], and ASIC [2] have been proposed, which gain better performance than general-purpose processors. However, these accelerators are often optimized only for some layers of the neural network, with less flexibility. To address this problem, Chen's team proposes Cambricon [11], a domain-specific Instruction Set Architecture (ISA) for NN accelerators, which supports ten network structures and has higher code density and performance than traditional ISA. Nevertheless, it is not specific to CNNs, thereby overlooking the reusability and parallelism of data. Flexible for CNNs, DaDianNao [3] can support the Multi-Layer Perceptrons (MLPs), but it demands substantial reconfigurable computing units and complex control. In [5,6], Luca et al. propose a Hardware Convolution Engine (HWCE) based on RISC-V architecture and achieve significant results, but they only design instructions for convolution operation, without considering other layers. Thus, an efficient and flexible CNN-specific instruction set is still demanding.

In this paper, we present a novel lightweight ISA for CNN reference, called RV-CNN. It consists of 9 instructions based on RISC-V, thereby to support the current mainstream CNN technologies. The main work of this paper is summarized as follows:

- Though studying computational patterns of the popular CNNs, we propose a small and easy-to-implement CNN-specific instruction set, which can flexibly support a variety of CNN structures.
- Breaking the traditional peripheral accelerator pattern, we extend CNN-specific instructions into RISC-V five-stage pipeline architecture and particularly optimize the implementation of instructions.
- As a case study, we implement our design and evaluate it from code density, performance, and power consumption, which demonstrates our design possess promising energy efficiency.

The rest of this paper is organized as follows. Section 2 briefly introduces our motivations and a few design preferences. Section 3 describes the details of the new ISA. Section 4 illustrates the overall architecture. Section 5 displays the experiment setup and evaluation results. Section 6 is the conclusion.

2 Motivations and Preferences

2.1 Motivations

Flexibility. Although compared with general-purpose processors, application scenarios that hardware accelerators for CNNs will deal with are much decreased and more certain, there are still many network structures which are different but have similarities, possessing different strengths. Nevertheless, common hardware accelerators normally deploy the whole or part of the neural network on the

FPGA, in which the network structure is often not reconfigured due to time expense. Thus, we expect to provide more flexibility for CNN techniques by abstracting the intensive operations in CNNs into dedicated instructions. Users can write assembly instructions to build a particular CNN.

Efficiency. Typically, an accelerator serves as a peripheral to the host CPU. Hence, the host CPU is in charge of transferring data from the main memory to the accelerator over a bus. It is obviously not a negligible overhead because of additional processing in the operating system and massive data. Besides, bus bandwidth also limits the performance of these accelerators. Therefore, instead of the previous pattern, we deploy the acceleration unit into the processor's pipeline, and then optimize the memory access of the acceleration unit to satisfy its data bandwidth requirements, thereby improving efficiency.

2.2 Design Preferences

RISC-V Extension. Designing a completely novel CNN-specific ISA usually involves plenty of factors, but the part that restricts the speed of calculation is what matters precisely. In view of the extensibility of RISC-V, we extend it with our dedicated instructions which are crucial to accelerating the CNN computing, while maintaining the basic kernel and each standard extension unchanged. In this way, we can concentrate on designing our CNN instructions, as well as directly using scalar and logical control instructions that the RISC-V provides. Moreover, we can also utilize the toolchain provided by RISC-V to speed up the development process.

Data-Level Parallelism. Taking the topological structure of convolutional neural networks (layer-by-layer) and the independence of weight matrices between layers into account, it is a more efficient way that utilizes data-level parallelism by applying matrix instructions than exploring instruction-level parallelism in NN operations. Furthermore, when dealing with calculations involving large amounts of data, matrix instructions can explicitly specify the independence between the data blocks, which can significantly reduce the size of dependency detection logic, compared to the conventional scalar instructions. What's more, matrix instructions also possess higher code density, so we chiefly focus on data-level parallelism here.

Scratchpad Memory. Vector registers commonly appear in the vector architecture, each of which is a fixed-length bank holding a single vector, and allow processors to operate all elements in a vector at one time. Scratchpad memory [1], a high-speed internal memory used for the temporary storage of calculations, can be accessed by direct addressing, costs low power, and supports variable-length data accesses. Considering that dense, continuous, variable-length data access often occurs in CNNs, and weight data rarely reused, we replace vector registers with the scratchpad memory in our design.

3 Details of Custom Instructions

3.1 Custom Instructions

We design the RV-CNN architecture, including both data transfer and computational instructions, as shown in Table 1. Cooperating with the base ISA, RV-CNN can perform typical CNN calculations. RV-CNN has 32 32-bit General-Purpose Registers, which can be used to store scalar values, as well as in register-indirect addressing of the on-chip scratchpad memory. Additionally, due to the 32-bit instruction length limit, we set up a vector length register (VLR) to ascertain the length of the vector, similar to the vector architecture. RV-CNN still access memory only through the corresponding MLOAD/MSTORE, obeying a load-store architecture of RISC-V. The instructions already in RISC-V are not described here.

Table 1. An overview of RV-CNN.

Instruction type	Example
Data Transfer	MLOAD/MSTORE
Computational	MMM/MSIG/MSFMX/MRELU
Logical	MAXPOOL/MINPOOL/APOOL

Data Transfer Instructions. To flexibly support matrix operations, data transfer instructions can load/store variable-size data blocks (an integer multiple of VLR value) from/to main memory. Specifically, the stride field of instructions can designate the stride of adjacent elements, avoiding expensive matrix transpose operations in memory. Figure 1 illustrates the matrix load (MLOAD) instruction, where Reg0 specifies the destination address; Reg1, Reg2, and Reg3 respectively specify the source address of matrix, the size of the matrix, and the stride of adjacent elements. Matrix store (MSTORE) instruction is similar to that of MLOAD, while regularly ignoring the stride fields.

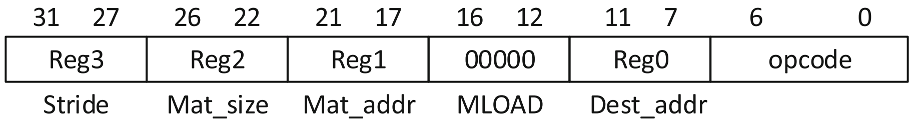

Fig. 1. Matrix Load (MLOAD) instruction.

Matrix Computational Instructions. CNNs are mainly composed of convolutional layers, pooling layers, and fully-connected layers, where the most computation concentrates in convolutional layers [4]. In the convolutional layer, convolution kernels move continuously on input feature maps and do a dot-product operation with the coincidence region to generate the input data of the next layer. Nevertheless, in this process, the operation between different feature maps and corresponding convolution kernels is independent of each other. To make full use of data parallelism, we adopt mapping technology (im2col algorithm) to transform 3-D convolution operation to MM operation (see Fig. 2 for an illustration). Moreover, the computing unit can be reused by fully-connected layers on account of analogous MM (row = 1) computing pattern. Note that instead of storing the entire input feature data, the rearrangement is performed before when we store them in the FPGA on-chip memory.

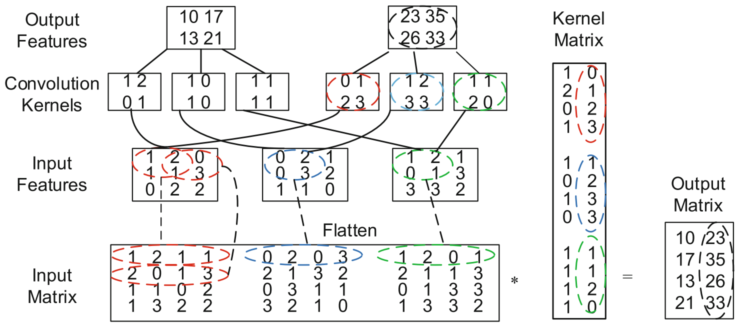

Fig. 2. Matrix multiplication version of convolution.

After mapping the 3D convolution to MM (matrix multiplication) operation, we use the Matrix-Mult-Matrix instruction to perform it. It is illustrated in Fig. 3, where Reg0 specifies the base scratchpad memory address of the matrix output (Destination address); Inst[16:12] is the function field of the instruction, indicating the MM operation. Reg1 and Reg2 specify the base address of the matrix1 and matrix2 respectively. The upper 16 bits and the lower 16 bits in Reg3 specify the number of rows of matrix 1 and the number of columns of matrix 2 (Rows + Cols), respectively. Accordingly, the size of matrix1 and matrix2 can be ascertained by the value in VLR and Reg3. To utilize greater extent of data locality as well as reduce concurrent read/write requests to the same address, we choose to adopt dedicated the MMM instruction to perform matrix multiplication instead of decomposing it into finer-grained instruction (e.g., matrix-mult-vector and vector dot products) here.

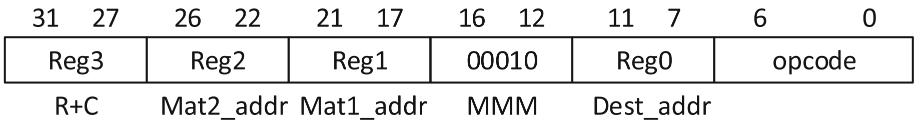

Fig. 3. Matrix Multiply Matrix (MMM) instruction.

The Matrix Sigmoid (MSIG) and the Matrix Softmax (MSFMX) instruction are essential to complete the entire computation. By default, we employ the MSIG instruction to activate the input data through the sigmoid function, to define the output of neurons. Alternatively, users can choose the MRELU or MTANH instruction to implement the relu or the tanh function by modifying the Inst [31:27] field (see Fig. 4). Correspondingly, to obtain the prediction results, the MSFMX instruction is used to normalize the output data.

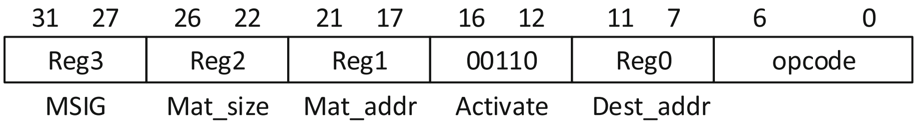

Fig. 4. Matrix Sigmoid (MSIG) Instruction.

Matrix Logical Instruction. The formats of maximum pooling (MAXPOOL) instruction (see Fig. 5) are similar to those of MLOAD, where reg0, reg1, and reg2 possess the same meaning as MLOAD instruction, respectively presenting destination address of output data, source address, and size of the input matrix. Whereas, as to the GPOOL instruction, the upper 16 bits and the lower 16 bits in Reg3 respectively designate the size and sliding step of kernels. Also, the MINPOOL (minimum pooling) or APOOL (average pooling) instruction, only differing from the function field of the MAXPOOL instruction, can be taken to determine the minimum or average pooling.

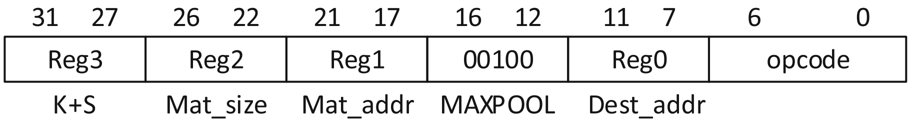

Fig. 5. Maximum Pooling (MAXPOOL) instruction.

3.2 Code Examples

To illustrate the usage of our proposed instructions, we implement two simple yet representative components of CNN, a convolutional layer and a pooling layer. The example code of the fully-connected layer is similar to that of the convolutional layer, except that the output should pass through the activation function.

Convolutional Layer Code:

```
//$1:input mat1 address, $2:input mat2 address
//$3:temp variable address, $4:output size
//$5:row of mat1, col of mat2, $6:mat1 address, $7:mat1 size
//$8:mat2 address, $9:mat2 size, $10:output matrix address
    LI       $5,    0x0310_0006  //row of mat1(784),col of mat2(6)
    LI       $VLR,  0x1A         //set vector length(26)
    MLOAD    $1, $6,  $7         //load kernel matrix
    MLOAD    $2, $8,  $9         //load feature map matrix
    MMM      $3, $2,  $1, $5     //mat1 x mat2
    MSTORE   $3, $10, $4         //store output to address($10)
```

Pooling Layer Code:

```
//$1:input mat address, $2:feature map size
//$3:loop counter, $4~$6:temp variable address
//$8:mat address, $7:output mat address, $9:output size
    ADDI     $4, $6, #0
    LI       $10,   0x0004_0002  //kernel size:2x2,step=2
    LI       $VLR,  0x1C         //set vector length(28)
    LI       $3,    0x06         //set loop counter(6)
L0:MLOAD     $1, $8, $2          //load feature map
    APOOL    $5, $1, $2, $10     //subsample
    MSIG     $4, $5, $2          //activate
    ADDI     $4, $4, 0xC4        //update temp address(add 14x14)
    ADD      $8, $8, $2          //next feature map
    SUB      $3, $3, #1
    BGE      $3, #0, L0          //if(loop counter>0) goto label0
    MSTORE   $6, $7, $9          //store mat to address($7)
```

4 Implementation Details

4.1 Overall Architecture

A simplified block diagram of the RV-CNN core architecture displaying its pipeline stages and major functional blocks is illustrated in Fig. 6. It includes five stages: fetching, decoding, execution, memory access, and write-back, in which the matrix computing unit is in the execution stage of the pipeline. Since the matrix unit can directly interact with the scratchpad memory, matrix operations do not go through the memory access and write-back stage. Correspondingly, after the fetching and decoding stage, the instructions in the base ISA will enter the ALU then go to the next stage, while the custom instructions will perform corresponding operations such as data transmission, convolution, and activation through the matrix unit. The address space of the scratchpad memory is mapped to the main memory by global mapping, and the remaining memory address space is still accessed through the cache. We can readily embed Direct Memory Access (DMA) in the matrix unit for the data transfer.

Furthermore, through appending a data buffer between the matrix unit and the scratchpad memory, we can effectively reduce the data delay. In a nutshell, the scratchpad memory and cache are relatively independent, and the control module will detect the data dependence and decide whether to stall the pipeline or not. By default, data involved in the execution of matrix computational logical instructions should already exist in the scratchpad memory, which requires strict control from the program.

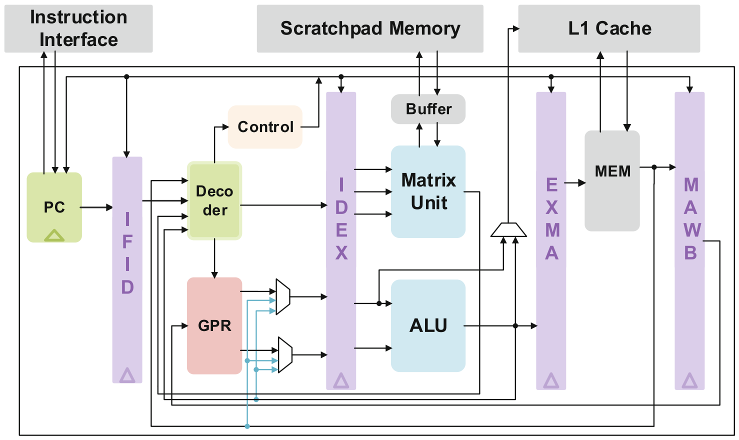

Fig. 6. A simplified block diagram of the RV-CNN core.

4.2 Matrix Unit's Architecture

The overall structure of the matrix unit is illustrated in Fig. 7, where the orange and black arrows represent the control flow and the data flow, respectively. As we can see, the internal controller, which works as a finite state machine, is the center of the matrix unit. After receiving control signals from the previous stage, the controller arouses the sub-component to complete the corresponding task if it is available. Otherwise, it will generate a feedback signal to indicate that the task is busy. The matrix unit contains four sub-units: Matrix multiplication, Sigmoid, Pooling, and Softmax. The buffer module is essentially an on-chip memory that buffers the input and output matrices and temporarily stores intermediate results. Lastly, the input/output module is responsible for receiving/sending the elements of the matrix in order.

4.3 Optimization

Since we reuse the MM unit to complete the computation-intensive convolutional layers and memory-intensive fully-connected layers, the performance of the MM unit exerts a significant impact on that of the whole matrix unit. Therefore, we adopt an adder tree and data reuse to optimize the MM unit in terms of computation and data access (see Fig. 7).

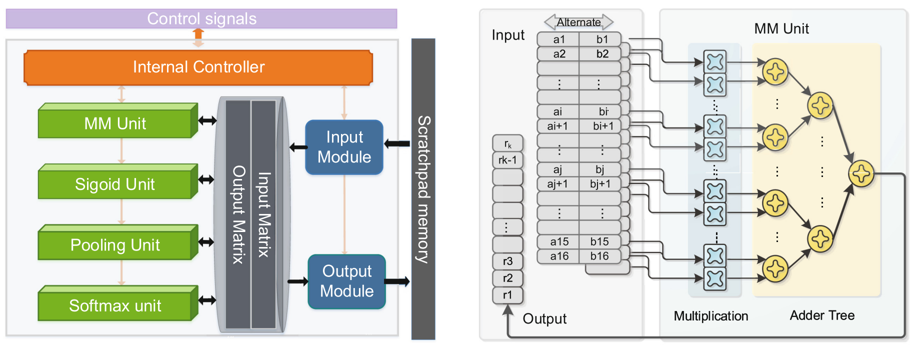

Fig. 7. The architecture of matrix unit (left); Optimizing details for MM Unit (right).

Adder tree. Matrix multiplication composes of multiple vector dot products where the length of the vector is variably regulated by VLR. In our design, the length of two vectors that MM unit to handle is fixed at 16. Therefore, vector length greater than 16 will be fragmented, and less than 16 will be padded with zeros. We deploy several DSP48Es to complete the float-point addition and multiplication and optimize the accumulation process of the dot product by adopting a binary tree, which significantly reduces the time complexity, from O(N) to O(log N). At the same time, utilizing pipeline technology, we can get sum or partial sum in every clock cycle.

Data reuse. After the data required for matrix multiplication has been transferred from the main memory to the scratchpad memory, the matrix unit continuously acquires two vectors through the input unit to complete computing. Considering the potential data locality in multiplication calculation, we lessen data bandwidth requirements through data reuse. Clearly, the reuse distance of the feature data is quite shorter than that of the weight matrix. Hence, we give priority to the reuse of the feature matrix. Meanwhile, we set two vectors to read the weight data, alternately used in current iteration computation.

5 Experiment and Results

5.1 Experiment Method

Platform. We synthesize the prototype system, then place, route the synthesized design in Vivado 2017.4, and evaluate the core by deploying it in Nexys4 DDR development board.

Baselines. We compare the performance of our system with existing implementations on general-purpose CPU and GPU.

- CPU. The CPU baseline is i7-4790K. We compare the processing time and power of our design with those of the CPU version of the program, utilizing the PAPI (Performance Application Programming Interface) tool which is an open source project provided by Intel, and gettimeofday() function.

– GPU. For comparison, we implement a GPU solution on Tesla k40c, and measure the processing time and running power by applying the cuEventElapsedTime() function and nvprof command respectively (provided by NVIDIA).

Benchmarks. Based on LeNet-5, we take three common image classification data sets (MNIST, CalTech 101, and CIFAR-10) as our benchmarks.

5.2 Results

In this subsection, we first report the resource utilization and power consumption of the system in the FPGA board, and then compare our design on the FPGA with CPU, GPU, and existing FPGA-based accelerators in three aspects respectively.

Area and power. We obtain the utilization of resources and the power consumption of FPGA by checking the implementation report in Vivado tools (LUT: 39.09%, 24780; FF: 26.49%, 33594; BRAM: 21.85%, 29.5; DSP: 50.42%, 121; and Power: 0.331 W).

Flexibility. The dedicated ISA we propose is not only suitable for accelerating CNN applications but also provides support to other deep learning algorithms with similar computing patterns, like DNNs. We implement the popular CNNs (Lenet-5, Alexnet, VGG) by using the specific instructions and measure the average code size of three CNNs. Compared with the GPU, x86, and MIPS, RV-CNN achieves 1.25x, 6.70x, 9.51x reduction of code length respectively.

Energy Efficiency. We compare the energy consumption of our system with CPU and GPU in the CNN reference process. As shown in the Fig. 8, the power consumption of CPU and GPU is 91.03x and 228.66x that of our design, and the energy consumption is 36.09x and 11.42x that of our design, respectively. Experimental results indicate that our design is significantly better than CPU and GPU in terms of energy consumption.

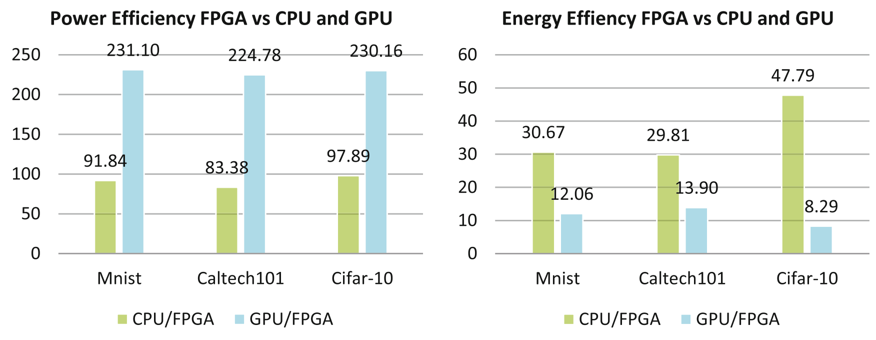

Fig. 8. Power ratios (left) and Energy ratios (right) vs CPU and GPU (Based on LeNet-5).

Performance. We also compare our design with the existing accelerators. Since different work adopts different quantization strategies and platforms, it is hard to choose a precise and effective comparison method. As we can see, taking Giga Operations Per Second (GOPS) as the evaluation standard, previous works can achieve better performance than ours. However, higher performance is the consumption of more resources, such as DSP blocks. In view of efficiency, we finally chose a relatively fair comparison standard - performance/w, which is defined as average GOPS per watt. As shown in Table 2, compared to previous works, our design achieves the highest performance efficiency.

Table 2. Comparison with other FPGA accelerators.

	[7]	[17]	[12]	Ours
Platform	Zynq XC7Z045	Virtex7 VX485T	Zynq XC7Z045	Artix7 XC7A100T
Frequency(MHz)	150	100	150	100
Precision	16-bit fixed	32-bit float	Q15	32-bit float
Power(W)	8	18.61	10	0.331
Perf(GOPS)	23.18	61.62	38.4	3.14(CONV)
DSP util	N/A	2240	391	121
Perf/w(GOPS/w)	2.90	3.31	3.84	9.48

6 Conclusion

In this work, we present an easy-to-implement CNN-specific instruction set, called RV-CNN, to provide more flexibility for CNN structures. Through studying computational patterns of the popular CNN techniques, we design nine coarse-grained matrix instructions in RV-CNN and extend the base RISC-V ISA with it. Then, we embed the corresponding acceleration unit in the classic five-stage pipeline architecture. Using Xilinx Artix7 100T to implement our design, compared with the Intel Core i7 processor and Tesla k40c GPU, it holds 36.09x and 11.42x energy efficiency ratio and 6.70x and 1.25x code density respectively. Besides, compared with the existing accelerators, it also achieves a promising energy efficiency.

Acknowledgments. This work is partially supported by the National Key Research and Development Program of China (under Grant 2017YFA0700900), National Science Foundation of China (No. 61772482), Jiangsu Provincial Natural Science Foundation (No. BK20181193), Youth Innovation Promotion Association CAS (No. 2017497), and Fundamental Research Funds for the Central Universities (WK2150110003).

References

1. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In: International Symposium on Hardware/software Codesign (2002)

2. Chen, T., et al.: DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In: ACM SIGPLAN Notices, vol. 49, pp. 269–284. ACM (2014)
3. Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., 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)
4. Cong, J., Xiao, B.: Minimizing computation in convolutional neural networks. In: Wermter, S., et al. (eds.) ICANN 2014. LNCS, vol. 8681, pp. 281–290. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11179-7_36
5. Conti, F., Rossi, D., Pullini, A., Loi, I., Benini, L.: PULP: a ultra-low power parallel accelerator for energy-efficient and flexible embedded vision. J. Signal Process. Syst. **84**(3), 339–354 (2016)
6. Flamand, E., et al.: GAP-8: a RISC-V SoC for AI at the edge of the IoT. In: 2018 IEEE 29th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 1–4. IEEE (2018)
7. Gokhale, V., Jin, J., Dundar, A., Martini, B., Culurciello, E.: A 240 G-ops/s mobile coprocessor for deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pp. 682–687 (2014)
8. Gong, L., Wang, C., Li, X., Chen, H., Zhou, X.: MALOC: a fully pipelined fpga accelerator for convolutional neural networks with all layers mapped on chip. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **37**(11), 2601–2612 (2018)
9. 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)
10. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
11. Liu, S., et al.: Cambricon: an instruction set architecture for neural networks. In: ACM SIGARCH Computer Architecture News, vol. 44, pp. 393–405. IEEE Press (2016)
12. Moini, S., Alizadeh, B., Ebrahimpour, R.: A resource-limited hardware accelerator for convolutional neural networks in embedded vision applications. IEEE Trans. Circuits Syst. II: Express Briefs **64**(10), 1217–1221 (2017)
13. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
14. Sun, Y., Chen, Y., Wang, X., Tang, X.: Deep learning face representation by joint identification-verification. In: Advances in Neural Information Processing Systems, pp. 1988–1996 (2014)
15. Wang, C., Gong, L., Yu, Q., Li, X., Xie, Y., Zhou, X.: DLAU: a scalable deep learning accelerator unit on FPGA. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **36**(3), 513–517 (2016)
16. Wang, C., Li, X., Chen, Y., Zhang, Y., Diessel, O., Zhou, X.: Service-oriented architecture on FPGA-based MPSoC. IEEE Trans. Parallel Distrib. Syst. **28**(10), 2993–3006 (2017)
17. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 161–170. ACM (2015)