Design of a Convolutional Neural Network Instruction Set Based on RISC-V and Its Microarchitecture Implementation

Qiang Jiao[1(✉)], Wei Hu[1], Yuan Wen[2], Yong Dong[1], Zhenhao Li[1], and Yu Gan[1]

[1] Wuhan University of Science and Technology, Wuhan, China
qjtxjq@gmail.com
[2] Trinity College Dublin, Dublin, Ireland

Abstract. The success of Convolution Neural Network (CNN) in computer vision presents a continuing challenge on performance requirement in both training and inference processes. Various software optimization has been examined towards existing hardware devices such as CPU and GPU to meet the computation needs; however, the performance gap between ideal and reality will keep going if there is short of hardware support. In this paper, we propose a customized CNN processor by extending the RISC-V instruction set. We have added six primary instructions by analyzing and abstracting the characteristics of conventional CNN models. The target micro-architecture has been upgraded accordingly to exploit the parallelism in the massive data access. We evaluated our work on the broadly used CNN model, LeNet-5, on Field Programmable Gate Arrays (FPGA) for the correctness validation. Comparing to traditional x86 and MIPS ISAs, our design provides a higher code density and performance efficiency.

Keywords: CNN · RISC-V architecture · RISC-V processor · Custom instruction · FPGA

1 Introduction

Convolution Neural Network (CNN) has been one of the most attractive technique in the past decade due to its outstanding prediction capability in a wide range of computer vision applications [5,10,11,20,22]. Such a high precision comes from extensive training over numerous training samples by well-structured models which usually have a large number of trainable weights to capture the characteristics of the model inputs. An example is a model with a larger size generally outperforms its smaller counterparts in terms of prediction accuracy [17]. But, at the same time, it also introduces a significantly greater number of operations which requires more time and resources to perform the calculation [6]. The computation burden can prohibit a comprehensive CNN model's deploying in practice, particularly for those applications with performance requirement.

The mainstream machine framework selects the most appropriate form of convolution among the options allowed by the runtime for the given platform. For CPU processors, a general matrix multiplication (GEMM) routine is a popular replacement of direct convolution [8,23], as most machines already have a fast implementation of the Basic Linear Algebra Subprograms (BLAS), while for the GPU counterpart, it often relies on the runtime library provided by the device vendors, such as Nvidia CuDNN and AMD ROCm. Because neither CPU nor GPU is initially designated for performing machine learning applications, it requires a significant effort for software to optimize code towards those processors. As a result, the generated code is often cumbersome, and its performance falls into suboptimal if there is a short of customized hardware support.

GPU (Graphics Processing Unit) is the broadest used hardware in training machine learning models because of its high throughput in parallel computing together with the massive bandwidth [9,13,21]. The GPU programming model allows a single instruction to perform on a group of independent data on separate PEs (Computing Elements) in a synchronized manner, which called SIMD (Single Instruction Multiple Data), to upgrade the computation throughput. The programmability of GPU makes it an easy-to-use platform that is popular among data scientists and CNN designers. However, the GPU also comes with obvious limitations that constrain its usage in a narrower range of scenarios. First, workload performance is not portable. The program has to be tuned before migrating from one GPU to another if the two processors have different architectures or various hardware resources. Second, GPU is not transparent to the programmer, and it requires explicit management of data and the functionality of processing the data. Transforming legacy code and library to the GPU is not a trivial effort. Finally, GPU requires the code to be compiled on-the-fly by a Just-in-Time compiler which shipped by the device vendor. Such a process introduces a nonnegligible runtime overhead.

In this paper, we propose hardware optimization for Convolutional Neural Networks. Instead of implementing a discrete accelerator, we extended the RISC-V instruction set by characterizing and abstracting the working pattern of mainstream CNN models. The proposed instruction set is transparent to the programmer and can be easily supported by compilers. It enhances the readability of the assembly program by decreasing the size of the code. In our experiment, we validated our design by implementing in on FPGA (Field Programmable Gate Arrays) fabrics. We have also examined and optimized the corresponding micro-architecture to improve the data parallelism.

The paper makes the following contributions:

– Design an instruction set based on RISC-V to optimize classic CNN operations
– Validate the extended instruction set on the FPGA fabric
– Optimize the micro-architecture to upgrade data access parallelism.

2 Background

2.1 Convolution Neural Network

Convolution Neural Network (CNN) has won considerable attention due to its outstanding performance in many sub-domains in computer vision, including image classification, object detection, motion estimation, video tracking, and so on. It outperforms the human being's capability in many of the above aspects via extensive training over a vast amount of samples for well-designed model structures. A typical CNN model consists of an input layer, an output layer and many hidden layers in between. Each layer performs a distinct operation on its input which is the output from its leading layer and outputs the processing outcome to its following layer. The most remarkable operation is the convolution that applies the multiply-accumulate operation for each element of the input (usually a 4-dimensional array called tensor) by a group of trainable variables, called weights. Behind each convolution, a CNN model usually presents an activation and pooling layer to perform the non-linear transformation and downsampling. The typical sub-structure of CNN shows in Fig. 1. Assisted by the activation layer, the convolution captures the feature hidden in the input by adjusting its weights and passes the abstracted information to a deeper layer for further analysis. Due to the intensive operation, the convolution layers dominant the overall execution time of a CNN model. Most of the optimization is targetting this type of layer because of its overwhelming runtime cost.

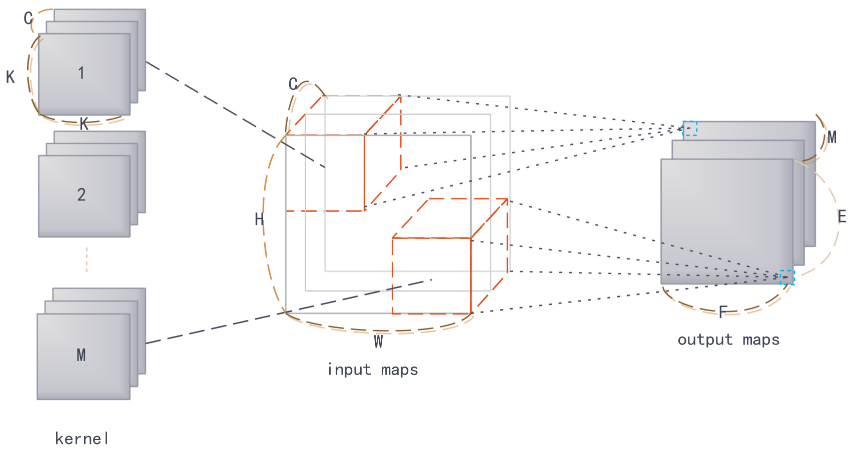


Fig. 1. Typical substructure of convolution neural network.

2.2 RISC-V

RISC-V is an open standard instruction set architecture (ISA) under open source licenses that allow hardware researchers to study and develop on top of it without having to pay fees to use it [12,24]. The design of RISC-V follows reduced instruction set computer principles that made up by a small but highly-optimized set of instructions. To the contrary of x86 and ARM, the modular nature of RISC-V promises an incremental ISA design. The core of RISC-V is called RV32I which is the primary ISA that never changes; therefore, it provides a stable hardware target for the compiler and operating system designers. The principle specifies the basic ISA in terms of instruction code, the number of registers, memory addressing mode and so on. Following the principle, architecture researchers are able to extend ISA by adding new instructions to perform customized functionalities. Through incrementally modifying the compiler to support the extended instruction, such enhanced service can be delivered to the software. Besides the upgrade brought by the customization, the whole process is transparent to the programmer and no explicit program changing is required.

3 RISC-V CNN Processor Design

Developing a dedicated ISA and implementing the corresponding processor with Verilog is a complex process that requires a proper abstracting of the computation and efficient implementation. However, for CNN, such a process can be simplified due to its regular building structures. A group of operations, such as convolution, pooling, and activation, are the building block for standard CNN networks and are often the type of compute-intensive. Hence in this paper, we designate to expend RISC-V standard towards these operations. We also optimize the storage structure because of the massive data access required by CNNs. In this section, we present the details of accelerating CNN model by combining scalar, logic, jump, control and other instructions provided in rv32i with CNN specialized instructions. In order to simplify instruction decoding, all instructions are 32 bits. A data buffer memory is implemented, and peripheral devices such as flash, SRAM and SDRAM are added, including 32 32-bit registers.

3.1 CNN Oriented Instruction Set

We design our customized instruction set based on top of RV32I. The instructions include matrix loading and storing, together with other operations such as activation, pooling and so on. The customized instructions are designed to share common opcode bits as many as possible for those ones that perform on the same data path to simplify the control logic implementation. The combination of standard RV32I instructions and CNN oriented customization fulfills the functionality of CNN computation. Thereby, CNN can be implemented in assemblies with the extended ISA.

We describe the customized commands in this section. The `inst[6:0]` is the opcode part of the instruction. We use `0001011` of this part to identify the instruction is customized for a special operation. It used together with the function code (funt), which is `inst[14:12]`, to decode the functionality of a specific instruction. The bits in `inst[11:7]`, `inst[19:15]`, and `inst[24:20]`, contains the register address respectively.

Matrix Load and Store Instructions

The instruction format is shown in Fig. 2. The function code 0000 indicates matrix load, MLOAD. It loads matrix, such as feature map and convolution kernel from the main memory to the on-chip cache. The `m_addr`, `m_size` and `dest_addr` specifies the memory address, size of the fetching data and the on-chip memory address, respectively. The bits in `ker_str` indicates the rs3 register with its upper 16 bits describe the size of convolution window while the lower 16 bits represent the stride of the convolution. The loaded matrix is kept in the on-chip scratchpad. For the computation efficiency reason, we reformat the matrix while storing it. The layout rearrangement changes with the size of convolution, and we will show the details in the following section.

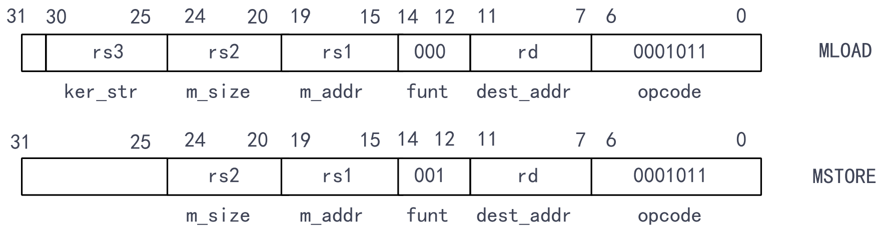


Fig. 2. Instruction for matrix load and store.

The function code `001` stands for matrix store command, `MSTORE`. It writes a matrix from the on-chip scratchpad to the off-chip memory. The decoding parts are functionally similar to `MLOAD`. The mnemonic of load and store is described as:

```
MLOAD rd, rs1, rs2, rs3
MSTORE rd, rs1, rs2
```

Matrix Operation Instruction

The instruction format is shown in Fig. 3: Function codes `010` and `011` define the operation of matrix multiplication (`MCONV`) and addition (`MADD`). Intrinsically, both instructions perform multiply and accumulations. For `MCONV` and `MADD`, `dest_addr` is the address to store the computation result, while `m_addr1` and `m_addr2` keep the address to the two operands separately. Different from `MLOAD` and `MSTORE`, the `inst[30:25]` indicates register `rs3`. In `MCONV`, the upper `16` bits of `rs3` register keep the number of rows for the first matrix while the lower `16` bits store the number of columns for the second one. This mechanism is related to the matrix rearrangement and details is shown in the following section. In `MADD`, the value of the `rs3` is the size of the matrix.

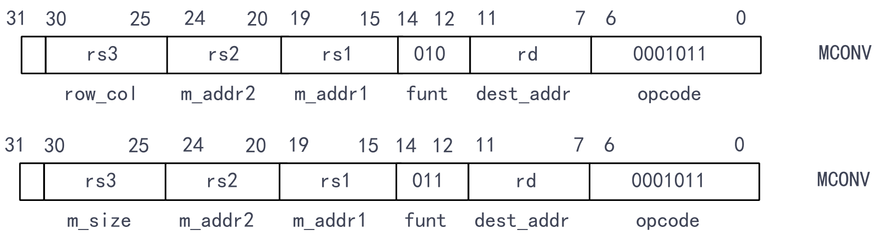

31 30 25 24 20 19 15 14 12 11 7 6 0
rs3 rs2 rs1 010 rd 0001011
row_col m_addr2 m_addr1 funt dest_addr opcode
MCONV
31 30 25 24 20 19 15 14 12 11 7 6 0
rs3 rs2 rs1 011 rd 0001011
m_size m_addr2 m_addr1 funt dest_addr opcode
MCONV

Fig. 3. Instruction of Matrix operation.

The instruction mnemonic is described as:

```
MCONV rd, rs1, rs2, rs3
MADD rd, rs1, rs2, rs3
```

Pooling and Activation Instructions
The instruction format is shown in Fig. 4. The function code of 100 means the maximum pooling instruction `MPOOL`. The maximum pooling is selected here because, in CNN, the effects of maximum pooling, minimum pooling and average pooling will not be much different. But the minimum and maximum pooling have advantages in implementation simplicity. The `dest_addr` represents the starting address of the result after pooling. The `m_addr` and `m_size` respectively represent the starting address and size of the matrix to be pooled. The `ker_str` identifies the rs3 register. The upper `16` bits of the register store the size of the pooling window, and the lower `16` bits stores the size of the stride.

The function code `101` indicates the activation command `MRELU`. The `ReLU` activation has been broadly used since the appearance of `AlexNet`. Comparing to the function of `Sigmod` and `tanh`, `ReLU` has a much simpler structure and easy to implement, particularly in hardware. The `dest_addr` specifies the starting address to the result after activation. The `m_addr` and `m_size` specify the address and size of the matrix feeding to the activation function.

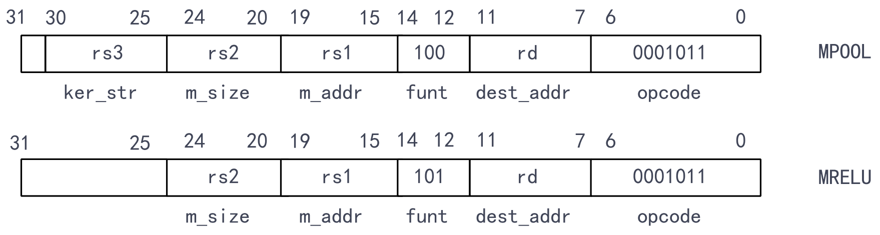

31 30 25 24 20 19 15 14 12 11 7 6 0
rs3 rs2 rs1 100 rd 0001011
ker_str m_size m_addr funt dest_addr opcode
MPOOL
31 25 24 20 19 15 14 12 11 7 6 0
rs2 rs1 101 rd 0001011
m_size m_addr funt dest_addr opcode
MRELU

Fig. 4. Instruction of pooling and activation.

The instruction mnemonic is described as:

```
MPOOL rd, rs1, rs2, rs3
MRELU rd, rs1, rs2
```

3.2 Processor Micro-architecture

The simple structure diagram of the processor is shown in Fig. 5, which includes five stages of operations: fetch, decode, execute, access to memory and write back. The execution unit is the main part of the design, which includes a general-purpose computing unit (ALU unit) and a CNN (M_ALU unit) computing unit, in which the data processed by the convolution computing operation interacts indirectly with the off-chip memory. Apart from the execution unit, there is no significant difference between the other four levels and the classic five-level pipeline. Therefore, we focus on introducing the CNN computing unit and data access optimization method in detail.

After the basic instructions are decoded, the logic, arithmetic, shift and other instructions are executed in the general computing unit. Our CNN instructions, from matrix loading, convolution, to pool activation, and the final matrix storage will be completed in CNN computing unit. All phases of CNN processing are taken place in the chip, including matrix loading, convolution, pooling, activation and writing the results. Such a process does not involve the main memory accessing. Data exchange and management are performed within the in-chip buffer (`rerang_buffer`).

Memory Access Optimization

Given the fact that the speed of computation comes faster than data loading for convolution, we have to optimize data access.

As we can see from Fig. 1, for classic convolution layer, multiple multi-channel convolution kernels convolve input feature map by stride. The result of one layer is the input of the next layer. Based on this truth, we optimize data access in two directions.

A: Because the same group of weight kernels convolves different feature maps, we can reuse the weight data with `M_LOAD` instruction and perform the convolution in parallel. The convolution kernel and the feature map from the off-chip `SDRAM` will be sent to the on-chip memory (`rerang_buffer`), from where we can change their layout. We take a $2 * 2$ convolution with stride 2 as an example, which is shown in Fig. 6.

Here, we use the same position convolution window of different input feature maps in parallel with the calculation of the corresponding convolution kernel. Using the `M_LOAD` instruction requires multiple cycles to load the convolution kernel matrix, and each convolution kernel only needs to be accessed once. The convolution kernel is fully reused, and the input feature map may need to be loaded multiple times or completed in a single time depending on its size. The `inst [31:25]` field of `M_LOAD` specifies the size of the convolution window.

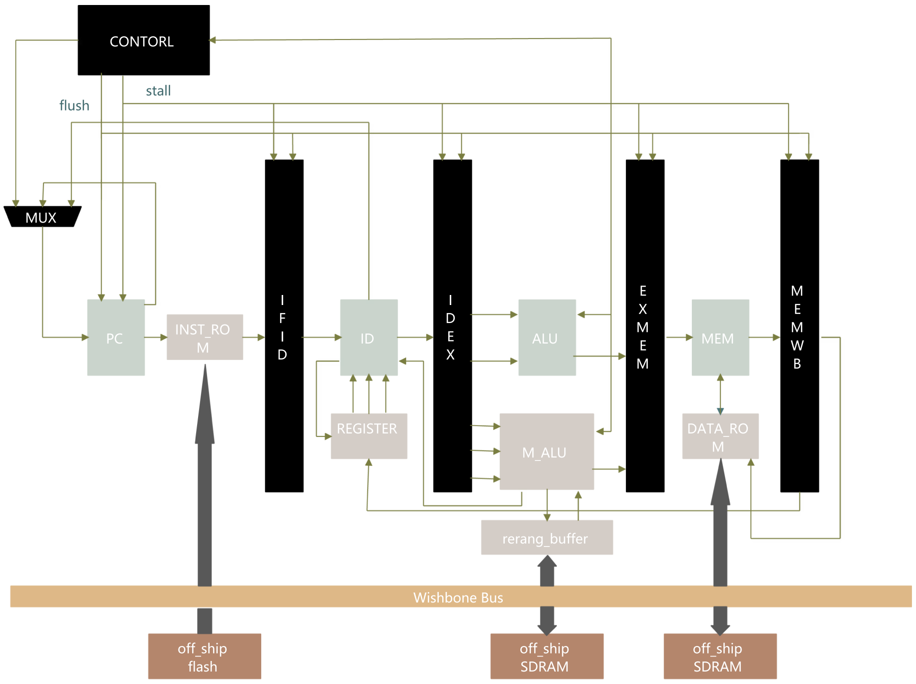


Fig. 5. Structure diagram of the processor.

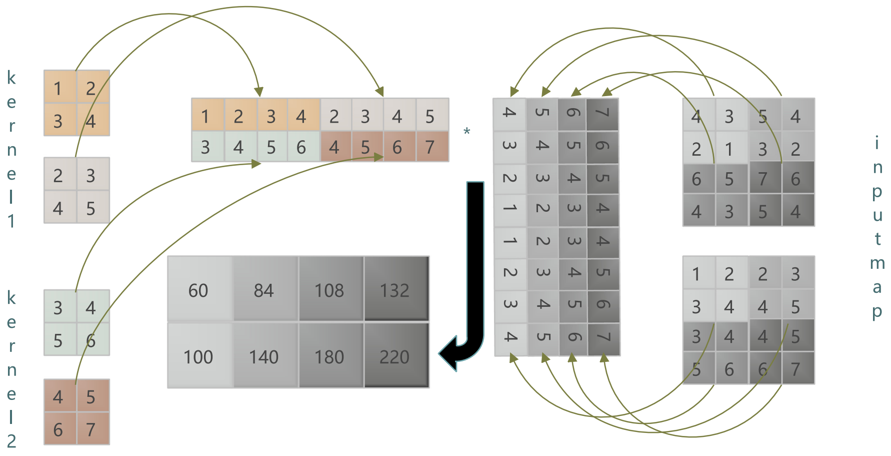


Fig. 6. Matrix layout rearrangement.

This value is also used in the convolution calculation, so it will be kept in a register. In this way, when the convolution instruction performed, the data in the `rerang_buffer` can be accessed according to the size of the convolution window, the number of rows of one matrix, and the number of columns of another matrix to fulfill the calculation. Though the rearrangement requires extra hardware overhead, the efficiency of convolution calculation is hugely improved.

B: The convolution takes a few cycles to finish after the two matrices loaded. Hence, we need to suspend the pipeline until the two matrices loading completed. Because there is no dependency between the convolution kernel and the feature map, we can add the matrix loading instructions after matrix computation. The efficiency is improved as a result of hiding the delay of data access. In such a scheme, data can be loaded while the computing unit is performing convolution.

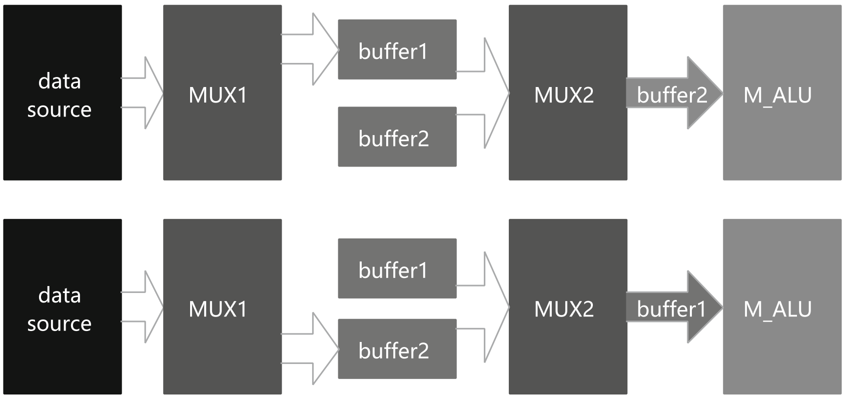


Fig. 7. Matrix layout rearrangement.

We have two buffers in `rerang_buffer`. In specific, when loading data from off-chip, the multiplexer is used to select the data buffer module that needs to store the data. For instance, if the matrix operand is delivered from the buffer 1, then the loaded data is sent to the buffer 2, and vice versa. Such a process takes place alternately to use the idle time caused by data access effectively. Its structure is shown in Fig. 7.

Calculation Unit

In our `M_ALU`, there are 32 tree-shaped multipliers and adders. Hence, it can support eight $2 * 2$ convolution kernels, three $3 * 3$, two $4 * 4$ or one $5 * 5$ kernel's parallel processing. We need compiler supports for generating the correct code. Figure 8 presents the structure.

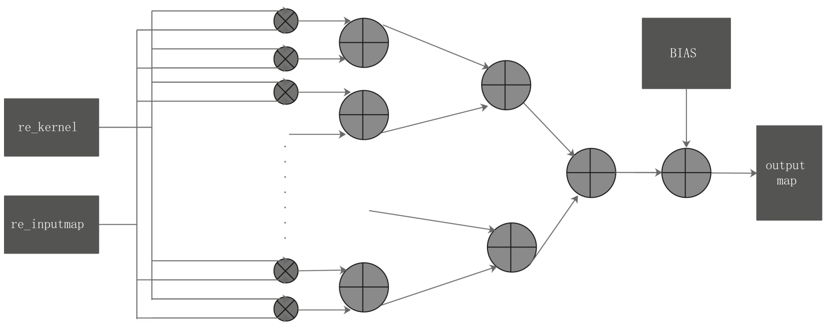


Fig. 8. Diagram of the calculation unit.

To give a clearer example, we present the pseudocode convolutional layer as follows:

```
 LI $1,#imm1        //size of kernel ($1 high 16 bits)
                    //sliding step ($1 low 16 bits)
MLOAD $3,$4,$5,$0       //LOAD kernel map
                        //$3 dest M1 addr
                        //$4 kernel addr
                        //$5 kernel size
 ...
MLOAD $6,$7,$8,$1       //LOAD feature map
                        //$6 dest M2 addr
                        //$7 input map addr
                        //$8 input map size
 ...
 LI $2,#imm2            //row of M1 ($2 high 16 bits)
                        //cow of M2 ($2 low 16 bits)

MCONV $9,$3,$4,$2       //$9 temporary output addr1
 ...
MADD $10,$9,$11,$2      //$10 temporary output addr2
MSTORE $13,$10,$2       //$13 store output addr
```

4 Experiment

We have implemented the hardware micro-architecture and customized instructions by Verilog. The design is synthesized by Xilinx toolset on a Artix-7 FPGA. To validate the hardware implementation, we have tested and simulated it by Vivado2019.1.

Vivado report regarding occupancy and power consumption is shown in Table 1.

Table 1. Resources occupancy and power consumption

Resource	Utilization	Available	Utilization (%)
FF	29015	202800	14.3
LUT	19720	101400	19.4
BRAM	31	135	22.9
DSP48	158	840	18.8
I/O	83	400	20.8
BUFG	11	32	34.8
Multi-column	0.362 W		

According to the synthesis report, the implementation is hardware friendly. It requires 19.4% LUT and 14.3% Flip-Flop of the FPGA chip. The memory consumption dominants the area usage of the design. As we can see from the result, the RISC-V extension uses 22.9% of the RAM and 34.8% of the BUFG. This high memory occupancy is determined by the nature of the CNN network, which introduces a large amount of data processing. The power consumption is 0.362 W. Combining the result, we can expect our implementation accommodates three CNNs on a single chip under a 1-W power budget.

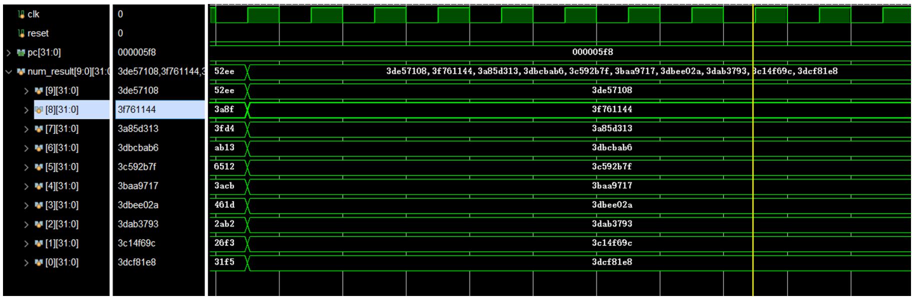
We use LeNet-5 on MNIST dataset to validate our design for the reason of simplicity. LeNet-5 is a well-known machine learning model that has been reported with outstanding image classification accuracy. It comes with a simple structure that contains three sets of convolutional and pooling layers, and two sets of fully connected layers. As a CNN model, LeNet-5 has all classic structural features while requires minimum effort to implement, which is particularly critical for hardware design and validation. For the same reason, we select MNIST dataset, which consists of a group of handwriting images of a consistent size by 28 * 28 pixels. All these selections aim at testing the correctness of hardware with the simplest settings.

We use risc32-unknown-linux-gun-series cross-compilation tool chain to generate the code. The customized instructions are implemented with embedded instruction codes.

We used the MNIST data set to test our design. Limited by the length of the paper, here we present the simulation result of classifying digit `8` and `9` in this section. We observe the similar result while doing all the rest handwriting classifications.

Figure 9 shows the simulation of predicting handwriting digit `8` in MNIST. The result buffer `num_result [0:9]` carries the confidence of the prediction and it is a vector of real number in hexadecimal that indicate the probability of digit `0` to `9`. In Fig. 9, the values in `num_result [0:9]` are: `3dcf81e8`, `3c14f69c`, `3dab3793`, `3dbee02a`, `3baa9717`, `3c592b7f`, `3dbcbab6`, `3a85d313`, `3f761144`, `3de57108`. Once transformed in to the decimal, the corresponding values are: 0.101322, 0.009092, 0.0836021, 0.093201, 0.005206,0.013255, 0.092153, 0.001021, 0.961201, 0.112032.

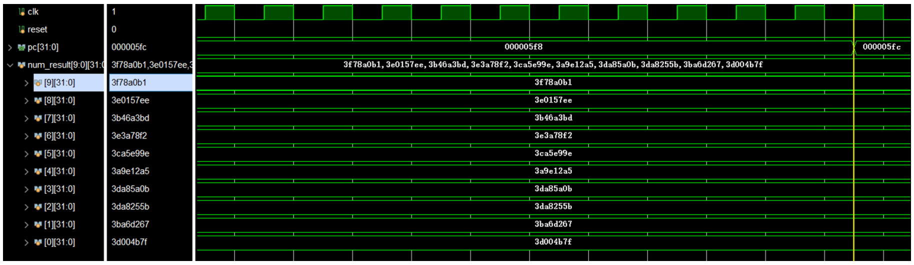
As `num_result [8]` has the highest value (0.961201), we know that the digit 8 has been correctly classified.



```
clk                     0
reset                   0
pc[31:0]                000005f8         000005f8
num_result[9:0][31:0    3de57108,3f761144,3   52ee  3de57108, 3f761144, 3a85d313, 3dbcbab6, 3c592b7f, 3baa9717, 3dbee02a, 3dab3793, 3c14f69c, 3dcf81e8
  [9][31:0]             3de57108         52ee  3de57108
  [8][31:0]             3f761144         3a8f  3f761144
  [7][31:0]             3a85d313         3fd4  3a85d313
  [6][31:0]             3dbcbab6         ab13  3dbcbab6
  [5][31:0]             3c592b7f         6512  3c592b7f
  [4][31:0]             3baa9717         3acb  3baa9717
  [3][31:0]             3dbee02a         461d  3dbee02a
  [2][31:0]             3dab3793         2ab2  3dab3793
  [1][31:0]             3c14f69c         26f3  3c14f69c
  [0][31:0]             3dcf81e8         31f5  3dcf81e8
```

Fig. 9. Simulation of classifying MNIST digit 8

Similarly, Fig. 10 provides the result in predicting digit 9. The hexadecimal values in `num_result [0:9]` are 3d004b7f, 3ba6d267, 3da8255b, 3da85a0b, 3a9e12a5, 3ca5e99e, 3e3a78f2, 3b46a3bd, 3e0157ee, 3f78a0b1, which are 0.031 322, 0.005091, 0.0821025, 0.082203, 0.001206, 0.020253, 0.182102, 0.003031, 0.126312, 0.971202 in decimal. The `num_result [9]` which has the largest probability 0.971202, is the correct prediction. Hence, we can conclude that our RISC-V extension are performing correctly.



```
clk                     1
reset                   0
pc[31:0]                000005fc         000005f8   000005fc
num_result[9:0][31:0    3f78a0b1,3e0157ee,3   3f78a0b1, 3e0157ee, 3b46a3bd, 3e3a78f2, 3ca5e99e, 3a9e12a5, 3da85a0b, 3da8255b, 3ba6d267, 3d004b7f
  [9][31:0]             3f78a0b1         3f78a0b1
  [8][31:0]             3e0157ee         3e0157ee
  [7][31:0]             3b46a3bd         3b46a3bd
  [6][31:0]             3e3a78f2         3e3a78f2
  [5][31:0]             3ca5e99e         3ca5e99e
  [4][31:0]             3a9e12a5         3a9e12a5
  [3][31:0]             3da85a0b         3da85a0b
  [2][31:0]             3da8255b         3da8255b
  [1][31:0]             3ba6d267         3ba6d267
  [0][31:0]             3d004b7f         3d004b7f
```

Fig. 10. Simulation of classifying MNIST digit 9

5 Related Work

Convolution Neural Networks is a typical compute-intensive application that involves massive tensor operations. The performance of CNN impacts significantly on its deployment, particularly for those models used in inference. Various methods have been examined to boost computation efficiencies, such as using matrix multiplication, Fast Fourier Transform (FFT) [1,3], or Winograd [7,25] to replace the direction convolution. Such type of algorithm generally have a better data locality and therefore capable of shortening the execution by strengthened data reuse.

Mainstream CNN frameworks intrinsically support GPU as a training and inference accelerator. Because of the SIMD model, the GPU can process a GEMM-based (General Matrix Multiplication) convolution in massive parallelism [14]. Though GPU has high performance in computation, it is tricky to release its full horsepower because of the architectural difference within processors by various vendors. Hand tuning code towards different platforms is not feasible. Hence, automatic tools, like TVM [2,18], has been developed to perform auto-optimization for diverse hardware platforms.

The automatic tools also open the window to explore the territory of software define hardware (SDH). Accelerators implemented on FPGA fabrics have been widely studied because of the wide bandwidth, high performance and low energy consumption. For the moment, SDH still requires collaboration from machine learning experts and hardware designers, because of the complexity.

RISC-V is an open standard instruction set architecture (ISA) that follows the reduced instruction set computer (RISC) principles [24]. It provides a framework to facilitate microprocessor and accelerator design. For CNN accelerating, solutions such as custom processor [15,16,19], integrated/discrete accelerator, SoC for edge IoT computing [4] have been proposed built on top of RISC-V. The design works correctly and efficiently according to our FPGA simulation.

6 Conclusion

In this work, we designed an instruction set to accelerate the CNN algorithm based on the computational characteristics of the CNN network algorithm. We propose a micro-architecture and two schemes in reformatting data layout to optimize data access. The design has been implemented in FPGA fabric by Verilog. In our experiment, we validate our hardware design by implementing LeNet-5 on MNIST dataset. The simulation result by Vivado shows that our RISC-V extension for CNN performs efficiently with a small hardware consumption.

Acknowledgement. This work was supported by Science Foundation Ireland grant 13/RC/2094 to Lero - The Irish Software Research Centre.

References

1. Abtahi, T., Kulkarni, A., Mohsenin, T.: Accelerating convolutional neural network with FFT on tiny cores, pp. 1–4 (May 2017). https://doi.org/10.1109/ISCAS.2017.8050588
2. Chen, T., et al.: TVM: end-to-end optimization stack for deep learning. CoRR abs/1802.04799 (2018)

3. Chitsaz, K., Hajabdollahi, M., Karimi, N., Samavi, S., Shirani, S.: Acceleration of convolutional neural network using FFT-based split convolutions. CoRR abs/2003.12621 (2020)
4. 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 (2018)
5. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR, pp. 770–778. IEEE Computer Society (2016)
6. Iandola, F.N., Moskewicz, M.W., Ashraf, K., Han, S., Dally, W.J., Keutzer, K.: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1 mb model size. CoRR abs/1602.07360 (2016)
7. Kala, S., Jose, B.R., Mathew, J., Nalesh, S.: High-performance CNN accelerator on FPGA using unified Winograd-GEMM architecture. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **27**(12), 2816–2828 (2019)
8. Kala, S., Mathew, J., Jose, B.R., Nalesh, S.: UniWiG: unified Winograd-GEMM architecture for accelerating CNN on FPGAs. In: 2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID), pp. 209–214 (2019)
9. Kim, H., Nam, H., Jung, W., Lee, J.: Performance analysis of CNN frameworks for GPUs. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 55–64 (2017)
10. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: NIPS, pp. 1106–1114 (2012)
11. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
12. Lee, Y.: An agile approach to building RISC-V microprocessors. IEEE Micro **36**(2), 8–20 (2016)
13. Li, C., Yang, Y., Feng, M., Chakradhar, S., Zhou, H.: Optimizing memory efficiency for deep convolutional neural networks on GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 633–644 (2016)
14. Li, X., Liang, Y., Yan, S., Jia, L., Li, Y.: A coordinated tiling and batching framework for efficient GEMM on GPUs. In: PPoPP, pp. 229–241. ACM (2019)
15. Li, Z., Hu, W., Chen, S.: Design and implementation of CNN custom processor based on RISC-V architecture. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 1945–1950 (2019)
16. Lou, W., Wang, C., Gong, L., Zhou, X.: RV-CNN: flexible and efficient instruction set for CNNs based on RISC-V processors. In: Yew, P.-C., Stenström, P., Wu, J., Gong, X., Li, T. (eds.) APPT 2019. LNCS, vol. 11719, pp. 3–14. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29611-7_1
17. Luo, J., Zhang, H., Zhou, H., Xie, C., Wu, J., Lin, W.: ThiNet: pruning CNN filters for a thinner net. IEEE Trans. Pattern Anal. Mach. Intell. **41**(10), 2525–2538 (2019)
18. Moreau, T., Chen, T., Jiang, Z., Ceze, L., Guestrin, C., Krishnamurthy, A.: VTA: an open hardware-software stack for deep learning. CoRR abs/1807.04188 (2018)
19. Porter, R., Morgan, S., Biglari-Abhari, M.: Extending a soft-core RISC-V processor to accelerate CNN inference. In: 2019 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 694–697 (2019)

20. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: ICLR (2015)
21. Strigl, D., Kofler, K., Podlipnig, S.: Performance and scalability of GPU-based convolutional neural networks. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 317–324 (2010)
22. Szegedy, C., et al.: Going deeper with convolutions. In: CVPR, pp. 1–9. IEEE Computer Society (2015)
23. Vasudevan, A., Anderson, A., Gregg, D.: Parallel multi-channel convolution using general matrix multiplication. In: 2017 IEEE 28th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 19–24 (2017)
24. Waterman, A., Lee, Y., Avizienis, R., Cook, H., Patterson, D.A., Asanovic, K.: The RISC-V instruction set. In: Hot Chips Symposium, p. 1. IEEE (2013)
25. Yu, J., et al.: Instruction driven cross-layer CNN accelerator with Winograd transformation on FPGA. In: 2017 International Conference on Field Programmable Technology (ICFPT), pp. 227–230 (2017)