Article

A Reconfigurable Convolutional Neural Network-Accelerated Coprocessor Based on RISC-V Instruction Set

Ning Wu *, Tao Jiang, Lei Zhang, Fang Zhou and Fen Ge

College of Electronic and Information Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China; jiangtao08@nuaa.edu.cn (T.J.); zhangleizizi@nuaa.edu.cn (L.Z.); zfnuaa@nuaa.edu.cn (F.Z.); gefen@nuaa.edu.cn (F.G.)

* Correspondence: wunee@nuaa.edu.cn; Tel.: +86-139-5189-3307

Received: 5 May 2020; Accepted: 8 June 2020; Published: 16 June 2020

Abstract: As a typical artificial intelligence algorithm, the convolutional neural network (CNN) is widely used in the Internet of Things (IoT) system. In order to improve the computing ability of an IoT CPU, this paper designs a reconfigurable CNN-accelerated coprocessor based on the RISC-V instruction set. The interconnection structure of the acceleration chain designed by the predecessors is optimized, and the accelerator is connected to the RISC-V CPU core in the form of a coprocessor. The corresponding instruction of the coprocessor is designed and the instruction compiling environment is established. Through the inline assembly in the C language, the coprocessor instructions are called, coprocessor acceleration library functions are established, and common algorithms in the IoT system are implemented on the coprocessor. Finally, resource consumption evaluation and performance analysis of the coprocessor are completed on a Xilinx FPGA. The evaluation results show that the reconfigurable CNN-accelerated coprocessor only consumes 8534 LUTs, accounting for 47.6% of the total SoC system. The number of instruction cycles required to implement functions such as convolution and pooling based on the designed coprocessor instructions is better than using the standard instruction set, and the acceleration ratio of convolution is 6.27 times that of the standard instruction set.

Keywords: convolutional neural network (CNN); coprocessor; RISC-V; IoT

1. Introduction

With the rapid development of artificial intelligence (AI) technology, more and more AI applications are beginning to be developed and deployed on internet of things (IoT) node devices. The intelligent internet of things (AI + IoT, AIoT), which integrates the advantages of AI and IoT technology, has gradually become a research hotspot in IoT-related fields [1]. Traditional cloud computing is not suitable for AIoT, because of its high delay and poor mobility [2]. For this reason, a new computing paradigm, edge computing, is proposed. In order to reduce the computing delay and network congestion, the computing is migrated from the cloud server to the device [2]. Edge computing brings innovation to the IoT system, but also challenges the AI computing performance of the IoT node processor. It is necessary to improve its AI computing performance under the condition of meeting the power consumption and area limitation of IoT node devices [3]. In order to improve the AI computing power of IoT node processors, some IoT chip manufacturers provide some artificial intelligence acceleration libraries for their IoT node processors, but only from the software level optimization and tailoring algorithm, just a stopgap. It is necessary to design the AI algorithm calculator suitable for the IoT node processor from the hardware level.

Among all kinds of AI algorithms, the convolutional neural network (CNN) algorithm is widely used in various IoT systems with image scenes due to its excellent performance in image recognition. Compared to traditional signal processing algorithms, it has a higher recognition accuracy, can avoid complex and tedious data feature extraction, and has stronger adaptability to different image scenes [4]. The common hardware of CNN acceleration is a GPU or TPU, but this kind of hardware accelerator is mainly used for high-performance servers, which are not suitable for the use of IoT node devices. In Reference [5,6], most of the constituent units in the CNN network are implemented, which need to consume more hardware resources while obtaining higher computing acceleration performance, and cannot meet the resource-limited needs of nodes in the IoT systems. In Reference [7], the matrix multiplication in the CNN algorithm is reduced by the FFT, but the FFT itself consumes more hardware resources. The sparse CNN (SCNN) accelerator proposed in Reference [8] uses the data sparsity of neural network pruning operation to design a special MAC computing unit, but it has a highly customized structure and a single application scenario. In Reference [9], an acceleration structure based on in-memory computing is proposed. By shortening the distance between computing and storage, the access to memory is reduced and the computing efficiency is improved. However, the cost of the chip based on in-memory computing is high, so it is not suitable for large-scale application in the IoT systems. The work in Reference [10] proposes several alternative reconfiguration schemes that significantly reduce the complexity of sum-of-products operations but do not explicitly propose a CNN architecture. In Reference [11], an FPGA implementation of a CNN for addressing portability and power efficiency is designed, but the proposed architecture is not reconfigurable. In Reference [12], a CNN accelerator on the Xilinx ZYNQ 7100 (Xilinx, San Jose, CA, USA) hardware platform that accelerates both standard convolution and depthwise separable convolution is implemented, but the designed accelerator cannot be configured for other algorithms in the IoT system. Several research works [13–15] use the RISC-V ecosystem to design an accelerator-centric SoC or multicore processor, but do not configure the design accelerator in the form of a coprocessor and design the corresponding custom coprocessor instructions to speed up the algorithm processing speed.

Based on Reference [16], this paper further optimizes the structure of the CNN accelerator. The basic operation modules in the acceleration chain designed by Reference [16] are interconnected through a crossbar, which makes the flow direction of input data more diversified, thus enriching the function of the acceleration unit and forming a reconfigurable CNN accelerator. Based on the expansibility of the RISC-V instruction set architecture, the reconfigurable CNN accelerator is connected to the E203 core (Nucleisys, Beijing, China) in the form of a coprocessor, and the corresponding custom coprocessor instructions are designed. The compiler environment and library functions are established for the designed instructions, the hardware and software design of the coprocessor is completed, and the implementation process of common algorithms in the IoT system on the coprocessor is described. Finally, resource evaluation of the coprocessor is completed based on the Xilinx FPGA. The evaluation results show that the coprocessor only consumes 8534 LUTs, accounting for 47.6% of the total SoC system. The RISC-V standard instruction set and custom coprocessor instruction set are used to implement the four basic algorithms of convolution, pooling, ReLU, and matrix addition. The acceleration performance of the coprocessor is evaluated by comparing the operation cycles of each algorithm in these two ways. The results show that the implementation of the coprocessor instruction set has a significant acceleration effect on these four algorithms, and the acceleration of the convolution has reached 6.27 times that achieved by the standard instruction set.

The rest of this paper is organized as follows: Section 2 introduces some background knowledge, including the RISC-V instruction set and E203 CPU. In Section 3, the hardware design of the reconfigurable CNN-accelerated coprocessor is introduced. Section 4 introduces the software design of the reconfigurable CNN-accelerated coprocessor. Section 5 introduces the implementation of commonly used IoT algorithms on the coprocessor. Section 6 is the experiment and resource analysis. Section 7 summarizes the conclusions of this work.

2. Background

2.1. RISC-V Instruction Set

The RISC-V instruction set is a new type of instruction set that was born only in 2010. It summarizes the advantages and disadvantages of the traditional instruction set, avoids the unreasonable design of the traditional instruction set, and its open-source features ensure that it can adjust the existing design at any time. This guarantees the simplicity and efficiency of the instruction [17]. RISC-V is a kind of modular instruction set, which contains three basic instruction sets and six extended instruction sets. This modular feature enables the designer to select one basic instruction set and several extended instruction sets for combination according to the actual design requirements so that flexible processor functions can be realized. More information on RISC-V can be found in [17].

Instruction extensibility is a significant advantage of the RISC-V architecture. It has reserved instruction coding space for the special domain architecture in the processor design, and users can easily expand their instruction subsets. The RISC-V architecture reserves 4 groups of custom instruction types in 32-bit instructions, as shown in Figure 1.

Inst[4:2]	000	001	010	011	100	101	110	111
Inst[6:5]								(>32b)
00	LOAD	LOAD-FP	<i>custom-0</i>	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	<i>custom-1</i>	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	<i>custom-2/rv128</i>	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	<i>custom-3/rv128</i>	≥80b

Figure 1. RISC-V architecture instruction space.

According to RISC-V architecture [17,18], *custom-0* and *custom-1* instruction spaces are reserved for user-defined extension instructions and will not be used as standard extension instructions in the future. The instruction space marked as *custom-2/rv128* and *custom-3/rv128* is reserved for the future *rv128*, which will also avoid being used by standard extensions, so it can also be used for custom extension instructions.

2.2. E203 CPU

In order to realize a reconfigurable CNN-accelerated coprocessor based on the RISC-V instruction set, we need to choose a suitable RISC-V processor as the research object. The E203 core is a 32-bit RISC-V processor core with a 2-level pipeline [19]. It extends integer multiplication and integer division instructions, atomic operation instructions, and 16-bit compression instructions on the basis of the *rv32i* instruction set. The E203 kernel is the closest to the Arm Cortex-M0+ (Arm, Cambridge, UK) in the design goal. Its structure is shown in Figure 2.

The E203 kernel adopts a single transmit sequential execution architecture, and the first stage of the pipeline is the fetch instruction. It performs simple decoding of instructions and branch prediction of branch instructions. The second stage of the pipeline completes decoding, execution, memory access, and write back.

The first stage of the pipeline mainly includes a simple decoding unit, a branch prediction unit, and a PC generation unit. The simple decoding unit (as shown by label 1 in Figure 2) performs simple decoding on the retrieved instruction to obtain the type of instruction and whether it belongs to a branch and jump instruction. For branch and jump instructions, a branch prediction unit (as shown by label 2 in Figure 2) is required to perform branch prediction to obtain the predicted jump address of the instruction. The PC generation unit (as shown by label 3 in Figure 2) generates the next PC value to be fetched, and accesses instruction tightly-coupled memory (ITCM) or bus interface unit (BIU) to fetch instructions. The PC value and instruction value are placed in the PC register and IR register and

passed to the next stage of the pipeline. The second level of the pipeline mainly includes a decoding dispatch unit (as shown by label 4 in Figure 2), a branch prediction analysis unit (as shown by label 5 in Figure 2), an arithmetic logic operation unit (as shown by label 6 in Figure 2), a multi-cycle multiplier and divider (as shown by label 7 in Figure 2), an access memory unit (as shown by label 8 in Figure 2), and an extension accelerator interface (EAI) (as shown by label 9 in Figure 2). The decoding and dispatching unit implements the decoding and dispatching of instructions. It dispatches instructions to different execution units for execution according to the specific types of instructions. Ordinary arithmetic logic operation instructions are dispatched to the arithmetic logic.

Figure 2. E203 core structure.

The operation unit for execution includes instructions such as logic operations, addition, subtraction, and shift. The branch and jump instructions are dispatched to the branch prediction analysis unit to perform branch prediction analysis. For instructions that fail to predict, pipeline flushing and instruction fetching are required. The multiplication and division instructions are dispatched to a multi-cycle multiplier and divider to execute the multiplication or division operation through multiple cycles. Load and store instructions read and write to the memory through the fetch unit. The coprocessor instructions are dispatched to the EAI interface and executed by the coprocessor. The reconfigurable CNN acceleration coprocessor designed in this paper is connected to the E203 processor through the EAI interface. The EAI interface has four channels: Request channel, feedback channel, memory request channel, and memory feedback channel. The request channel is used by the main processor to send custom extended instructions to the coprocessor, including the source operand, dispatch label, and other information. The feedback channel is used for the coprocessor to feedback the execution of custom extended instructions. The feedback information includes the calculation results of the instructions, the dispatch label, etc. The memory request channel is used for memory access by the coprocessor, which reads and writes memory through the main processor. The memory feedback channel is used for the main processor to return the read and write results of the memory to the coprocessor, including the data read from the memory. Please refer to [19] for a detailed introduction of the EAI interface.

The coprocessor extended instructions of the E203 processor follow the instruction expansion rules of RISC-V, and its coprocessor extended instruction encoding format is shown in Figure 3.

Figure 3. E203 coprocessor extended instruction encoding format.

The 25th to 31st bits of the instruction are funct7 intervals, which are used as the sub coding space and for the parsing of instructions in the decoding stage. Funct7 is 7 bits and can encode 128 instructions. rs1, rs2, and rd are source operand 1, source operand 2, and destination operand register indexes, respectively. xs1, xs2, and xd are used to indicate whether read source and write destination registers are required. The 0th to 6th bits of the instruction are opcode encoding segments, which can be opcode encoding values of custom-0, custom-1, custom-2, or custom-3. According to the bit width of funct7, each group of custom instructions can encode 128 instructions, and four groups of custom instructions can encode 512 coprocessor instructions.

3. Hardware Design of CNN-Accelerated Coprocessor

The reconfigurable CNN acceleration coprocessor designed in this paper is a further extension of the compact CNN accelerator designed in Reference [16]. It mainly optimizes the acceleration chain structure and adds the coprocessor-related modules. Other basic components are consistent with those in Reference [16].

3.1. Accelerator Structure Optimization

In the compact CNN accelerator designed in Reference [16], each operation unit in the acceleration chain is connected by a serial structure, and its data flow direction is single. When some algorithms are implemented, the data need to be moved in the memory and accelerator many times, which will reduce the calculation efficiency and increase the power consumption of the processor. In this paper, four basic operation modules, convolution, pooling, ReLU, and matrix plus, are interconnected by a crossbar, and a reconfigurable CNN accelerator is designed. The accelerator mainly comprises four reconfigurable computing acceleration processing elements (PEs). By configuring PE units with different parameters, the hardware acceleration of various algorithms can be realized. The structure is shown in Figure 4.

Figure 4. Reconfigurable convolutional neural network (CNN) accelerator structure diagram.

The accelerator includes a source convolution kernel cache module (COE RAM), a reconfigurable circuit controller (Reconfigure Controller), two ping-pong buffer blocks (BUF RAM BANK), and four PE units. Each PE unit contains four basic computing components and a configurable crossbar (Crossbar).

The configuration of the crossbar allows data to flow to different computing components, which can speed up different algorithms. The entire accelerator can be parameterized and data accessed through an external bus.

The key to the acceleration of various algorithms by the four PE units is the design of the crossbar circuit. The configuration of the crossbar can make the input data flow pass through any one or more calculation modules. Its structure is shown in Figure 5.

Figure 5. The structure of Crossbar.

The crossbar mainly consists of an input buffer (FIFO), a configuration register group (Cfg Regs), and five multiplexers (MUX). The five MUX select the data path to be opened according to the configuration information of the Cfg Regs. Thus, the data stream passes through different calculation modules in different orders. For example, the edge detection operation in image processing usually extracts the input image and then convolves it with the edge detection operator. It is necessary to configure the MUX as the path in the red part of Figure 5. In the convolutional neural network algorithm, it is usually necessary to perform convolution, ReLU, and pooling operations on the source matrix. You need to configure the MUX as the path in the blue part of Figure 5. The crossbar circuit and each calculation module use the ICB bus for data transmission. The ICB bus is a bus protocol customized by the open source CPU E203. It combines the advantages of the AXI bus and the AHB bus. For more information about the ICB bus, please refer to [19].

3.2. Coprocessor Design

In addition to optimizing the accelerator chain module designed in Reference [16], this paper also adds the EAI controller, decoder, and data fetcher, and completes the hardware design of the reconfigurable CNN-accelerated coprocessor, whose structure is shown in Figure 6.

The EAI controller is used to process the time sequence related to the EAI interface, and hand over the instruction information and source operands obtained from the EAI request channel to the decoder for decoding. The memory data obtained from the EAI memory response channel are handed over to the data fetcher for allocation to the corresponding cache. The decoder is used to decode custom extended instructions. For configuration instructions, the instructions are handed over to the reconfiguration controller to realize the configuration of each functional module. For the memory access instruction, the instruction is handed over to the data fetcher for processing, and the memory access is realized. The data fetcher implements the processing of the memory access instruction. It reads data from external memory to the corresponding cache through the memory response channel of the EAI interface, the convolution kernel coefficient is loaded to the COE CACHE unit, and the source matrix is loaded to CACHE BANK1 or CACHE BANK2. The calculation results are written back to the external memory through the memory request channel of EAI interface.

Figure 6. Hardware structure of coprocessor.

When the main processor executes an instruction, the decoding unit of the main processor first determines whether the instruction belongs to the custom instruction group according to the opcode of the instruction. For the instructions belonging to the custom instruction group, determine whether to read the source operand according to the xs1 and xs2 bits in the instruction. If to read, read the source operand from the register group according to rs1 and rs2. The main processor also needs to maintain the data correlation of instructions. If the instructions have data correlation, the processor will pause the pipeline until the correlation is removed. At the same time, if the instruction needs to be written back to the target register, the target register rd will also be used as one of the bases for the subsequent instruction data correlation judgment. Then, the instructions are sent to the coprocessor through the EAI request channel for processing. After the coprocessor receives the instructions, the instructions are further decoded and allocated to different units for execution according to the types of instructions. The coprocessor will execute instructions in the form of blocking. Only when one instruction is executed can the main processor dispatch the next instruction. Finally, after the instruction is executed, the result of instruction execution is returned to the main processor through the response channel. For the instruction to be written back, the result of instruction execution needs to be written back to the target register.

The designed reconfigurable CNN acceleration coprocessor and the compact CNN accelerator designed by Reference [16] are used to process multimedia data such as voice and image. The data flow is shown in Figure 7.

Figure 7a shows the data flow direction of the compact CNN accelerator designed in Reference [16]. It is an external device mounted on the SoC bus, which requires the CPU core to handle the data transfer. First, it needs to move the multimedia data from the external interface to the data memory through the CPU core, and then to the accelerator for processing. After the processing, it needs the CPU core to move the calculation results from the accelerator to the data memory and then to the

external interface to send them to the network interface. It needs to move data between the accelerator and the data memory twice.

Figure 7. Data flow comparison: (a) Data flow direction of the compact CNN accelerator designed in Reference [16]; (b) data flow direction of the reconfigurable CNN accelerated coprocessor designed in this paper.

Figure 7b shows the data flow direction of the reconfigurable CNN-accelerated coprocessor designed in this paper. Compared to the accelerator designed in Reference [16], the coprocessor can read multimedia data directly from the external interface, and after processing, it can be directly sent to the network interface by the external interface, which does not need the data moving between the coprocessor and the data memory.

The coprocessor method reduces the data moving, further speeds up the algorithm processing speed, and saves power consumption. In addition, the coprocessor can provide coprocessing instruction support, with higher code density and simple programming.

4. Software Design of CNN-Accelerated Coprocessor

4.1. Instruction Design of Coprocessor

The coprocessor instructions designed for the reconfigurable CNN-accelerated coprocessor are shown in Table 1.

Table 1. Custom coprocessor instruction list.

Instruction	Funct7	Rd	Xd	Rs1	Xs1	Rs2	Xs2
acc.rest	1	-	0	-	-	-	0
acc.load.cd	2	-	0	CoeMemoryAddress	1	Length CacheBaseAddress	1
acc.load.bd	3	BANKID	0	BankMemoryAddress	1	Length BankBaseAddress	1
acc.cfg.ldl	4	-	0	SrcDataBaseAddress	1	Location	1
acc.cfg.sdl	5	-	0	StoreDataBaseAddress	1	Location	1
acc.cfg.sms	6	PEID	0	Width	1	Height	1
acc.cfg.ldo	7	-	0	Offset1 Offset2	1	Offset3 Offset4	1
acc.cfg.sdo	8	-	0	Offset1 Offset2	1	Offset3 Offset4	1
acc.cfg.cp	9	PEID	0	CoeCacheBaseAddress	1	CoeHeight CoeWidth	1
acc.cfg.cb	10	-	0	PEID	1	CoeBasis	1
acc.cfg.pw	11	-	0	Width	1	Height	1
acc.cfg.adl	12	-	0	SrcAddDataAddress	1	Location	1
acc.cfg.ado	13	-	0	Offset1 Offset2	1	Offset3 Offset4	1
acc.cfg.wm	14	-	0	PEID	1	Order	1
acc.sc	15	-	0	-	0	-	0

There are 15 coprocessor instructions in total, which are divided into two categories: coprocessing configuration instruction and coprocessing data loading instruction. Configuration instructions are used to configure each function module of the coprocessor, such as PE working mode, cross-switch data flow direction, etc. The data loading instruction loads the convolution coefficient and the source matrix and loads them into the respective coprocessor cache. The assembly format of coprocessor instructions is shown in Figure 8.

acc.rest			
acc.load.cd		rs1	rs2
acc.load.bd	rd	rs1	rs2
acc.cfg.wm		rs1	rs2
acc.cfg.ldl		rs1	rs2
acc.cfg.ldo		rs1	rs2
acc.cfg.sdl		rs1	rs2
acc.cfg.sdo		rs1	rs2
acc.cfg.sms	rd	rs1	rs2
acc.cfg.cp	rd	rs1	rs2
acc.cfg.cb		rs1	rs2
acc.cfg.pw		rs1	rs2
acc.cfg.adl		rs1	rs2
acc.cfg.ado		rs1	rs2
acc.sc			

Figure 8. Assembly format of coprocessor instructions.

The use of the designed coprocessor instructions is generally divided into 5 steps:

1. Reset the coprocessor Before using the coprocessor, you need to use the **acc.rest** instruction to reset the coprocessor.
2. Load the convolution coefficient If you need to use the convolution module in the coprocessor, you need to use the **acc.load.cd** instruction to load the convolution coefficient into the COE cache.
3. Load the source matrix The coprocessor supports reading the source matrix directly from memory for calculation, and also supports moving the source matrix to CACHE BANK and reading the data from CACHE BANK for calculation. The instruction to achieve this function is **acc.load.bd**.
4. Configure coprocessor parameters
 - (1) Working mode Configure the working mode of each PE of the coprocessor, that is, configure the crossbar in each PE, select the acceleration unit that participates in the calculation, and configure the direction of the data flow according to the algorithm. The instruction to achieve this function is **acc.cfg.wm**.
 - (2) Calculate data source location Configure the coprocessor calculation data source. The coprocessor can load data directly from memory or load data from CACHE BANK. Before starting the coprocessor, you need to configure the coprocessor calculation data source according to the storage location of the source matrix. The instruction to achieve this is **acc.cfg.ldl**. Configure the offset from the base address of the calculation data required by each PE in the data source. The calculation data required by the four PEs is continuously stored in the data source, so the relative offset of the calculation data required by each PE needs to be determined. The instruction to achieve this is **acc.cfg.ldo**.
 - (3) Storage location of calculation results The configuration coprocessor calculation result storage location is the same as the configuration calculation data source. It supports saving calculation results to memory or CACHE BANK, and the configuration instruction is **acc.cfg.sdl**. Configure the relative offset of each PE calculation result in the storage location. The configuration instruction is **acc.cfg.sdo**.

- (4) The width and height of source matrix Configure the width and height of the source matrix in each PE, and the instruction to achieve this function is acc.cfg.sms.
 - (5) Convolution kernel coefficient parameter Configure the offset address of the convolution kernel in the COE CACHE and the size of the convolution kernel. The configuration instruction is acc.cfg.cp. Configure each convolution kernel bias. The configuration instruction is acc.cfg.cb.
 - (6) Pooling parameters Configure the height and width of the maximum pooling method. The configuration instruction is acc.cfg.pw.
 - (7) Matrix plus parameters Configure the loading position of another input matrix participating in the matrix addition calculation, support the input matrix to be loaded from memory or CACHE BANK, and the configuration instruction is acc.cfg.adl. Configure the relative offset of another input matrix in each PE. The configuration instruction is acc.cfg.adl.
5. Start calculation Using the acc.sc instruction calculation, it enables the coprocessor to start the calculation according to the configuration parameters.

4.2. Establishment of Instruction Compiling Environment for Coprocessor

The RISC-V foundation open-source compilation tool chain includes GCC and LLVM. The most commonly used compilation tool chain is GCC. By modifying the Binutils toolset in the GCC compilation tool chain, a compilation environment for coprocessor instructions is established.

The RISC-V foundation open-source GCC compilation tool chain mainly includes riscv32 and riscv64 versions. The E203 kernel only supports the rv32IMAC instruction set in RISC-V, so the riscv32 version of the GCC compilation tool chain is used. The GCC compilation tool chain mainly includes the GCC compiler, Binutils binary toolset, GDB debugging tools, and C runtime libraries. The Binutils binary toolset is a set of tools for processing binary programs. Some commonly used tools and functions in the 32-bit Binutils toolset are shown in Table 2.

Table 2. Binutils toolset list.

Tool Name	Function
riscv32-unknown-elf-as	Assembler, convert assembly code to executable ELF file
riscv32-unknown-elf-ld	Linker, linking multiple target and library files as executables
riscv32-unknown-elf-objcopy	Converting ELF format files to bin format files
riscv32-unknown-elf-objdump	Disassembler, which converts binary files into assembly code

By calling each Binutils tool in the table, the assembly code can be assembled and linked, and an object file can be generated. In order to enable the Binutils toolset to support the custom coprocessor instructions in Section 4.1, the source code of Binutils needs to be modified. After adding all custom coprocessor instructions to the Binutils source code, compile the Binutils source code to generate the toolset in Table 2. To test the functionality of the toolset, write the following test assembler test.s as shown in Figure 9.

Use riscv32-unknown-elf-as assembler to assemble the program to generate the test.out binary object file, and use riscv32-unknown-elf-objdump to disassemble test.out to obtain the disassembly information shown in Figure 10.

The first column is the binary information of the instruction, and the second column is the assembly code obtained by disassembling the instruction. Analyze the binary information of the instruction, which is consistent with the design of Section 4.1, and the assembly code obtained by disassembly is also consistent with the test assembly code ($\times 5$, $\times 1$, $\times 2$, t0, ra, and sp represent the same register). This test indicates that the designed custom coprocessor instructions have been successfully added to the Binutils toolset.

```

.text
acc.rest
acc.load.cd    x1, x2
acc.load.bd    x5, x1, x2
acc.cfg.ldl    x1, x2
acc.cfg.sdl    x1, x2
acc.cfg.sms    x5, x1, x2
acc.cfg.ldo    x1, x2
acc.cfg.sdo    x1, x2
acc.cfg.cp     x5, x1, x2
acc.cfg.pw     x1, x2
acc.cfg.cb     x1, x2
acc.cfg.adl    x1, x2
acc.cfg.ado    x1, x2
acc.cfg.wm     x1, x2
acc.sc

```

Figure 9. Test assembler.

test.out: file format elf32-littleriscv		
Disassembly of section .text:		
00000000	<.text>:	
0:	0200000b	acc.rest
4:	0420b00b	acc.load.cd ra,sp
8:	0620b28b	acc.load.bd t0,ra,sp
c:	0820b00b	acc.cfg.ldl ra,sp
10:	0a20b00b	acc.cfg.sdl ra,sp
14:	0c20b28b	acc.cfg.sms t0,ra,sp
18:	0e20b00b	acc.cfg.ldo ra,sp
1c:	1020b00b	acc.cfg.sdo ra,sp
20:	1220b28b	acc.cfg.cp t0,ra,sp
24:	1420a00b	acc.cfg.pw ra,sp
28:	1620b00b	acc.cfg.cb ra,sp
2c:	1820b00b	acc.cfg.adl ra,sp
30:	1a20b00b	acc.cfg.ado ra,sp
34:	1c20b00b	acc.cfg.wm ra,sp
38:	1e00000b	acc.sc

Instruction
binary

Instruction
assembly code

Figure 10. Assembly code disassembly information.

4.3. Coprocessor-Accelerated Library Function Design

After completing the instruction design of the custom coprocessor and the establishment of the instruction compilation environment, the use of the coprocessor can be completed by writing assembly code. However, the method of writing assembly code has a low development efficiency. In order to facilitate the use of the coprocessor, the common functions of the coprocessor are packaged into the form of a C language function interface, and the coprocessor acceleration library functions are designed.

The designed library function interface is shown in Table 3.

The library functions use the C language inline assembly to implement the call to the coprocessor instructions. Taking the CfgPoolWidth function interface as an example, the specific implementation is shown in Figure 11.

The CfgPoolWidth function calls the acc.cfg.pw instruction in the form of inline assembly and passes in the values of the width and height variables to implement the height and width configuration of the pooling operation.

Table 3. Library function interface list.

Function Interface	Function
void CoprocessorRest ()	Reset coprocessor
int LoadCoeData (unsigned int CoeMemoryAddress, unsigned int CacheBaseAddress, unsigned int Length)	Load convolution kernel coefficients into COE CACHE
int LoadBankData (unsigned int BankMemoryAddress, unsigned int BankBaseAddress, unsigned int Length, int Bankid)	Load source matrix into CACHE BANK
int CfgLoadDataLoc (int Location, unsigned int SrcDataBaseAddress, unsigned int Offset1, unsigned int Offset2, unsigned int Offset3, unsigned int Offset4)	Configure the location of the calculation data source and the relative offset of the calculation data of each PE unit in the data source
int CfgStoreDataLoc (int Location, unsigned int StoreDataBaseAddress, unsigned int Offset1, unsigned int Offset2, unsigned int Offset3, unsigned int Offset4)	Configure the calculation result storage location and the relative offset of the calculation result of each PE unit in the storage location
int CfgAddDataLoc (int Location, unsigned int SrcAddDataAddress, unsigned int Offset1, unsigned int Offset2, unsigned int Offset3, unsigned int Offset4)	Configure the loading position of another input matrix participating in the matrix addition calculation and the relative offset of the other input matrix in each PE unit in the loading position
int CfgSrcMatrixSize (int Width, int Height, int PEID)	Configure the width and height of the source matrix
int CfgCoeParm (unsigned int CoeCacheBaseAddress, int CoeHeight, int CoeWidth, int PEID)	Configure convolution kernel size and offset address in COE CACHE
int CfgCoeBias (float CoeBasis1, float CoeBasis2, float CoeBasis3, float CoeBasis4)	Configure the convolution kernel bias for each PE unit
int CfgPoolWidth (int Width, int Height)	Configure pooling width and height
int CfgWorkMode (unsigned int Order1, unsigned int Order2, unsigned int Order, 3 unsigned int Order4)	Configure the working mode of 4 PE units, that is, select the acceleration unit that participates in the calculation and the direction of the data flow
int CoprocessorStartCal ()	Start calculation

```

int CfgPoolWidth (int Width, int Height)
{

    int source1= Width;
    int source2= Height;

    asm volatile (
        "acc.cfg.pw  %[ src1], %[ src2]"
        :[src1]"r"(source1), [ src2] "r"(source2)
    );

    return 1;
}

```

Figure 11. The specific implementation of the CfgPoolWidth function.

5. Implementation of Common Algorithms on Coprocessor

The reconfigurable CNN-accelerated coprocessor designed in this paper can accelerate not only the CNN algorithm but also some other commonly used algorithms in the IoT system. This article describes the implementation of the LeNet-5 network, Sobel edge detection, and FIR filtering algorithms on the coprocessor.

5.1. LeNet-5 Network Implementation

In order to verify the acceleration of the coprocessor to the CNN algorithm, the classical LeNet-5 network is used in this paper. The structure of the LeNet-5 network is shown in Figure 12.

Figure 12. The structure of LeNet-5 network.

The LeNet-5 network mainly comprises six hidden layers [20]:

- (1) Convolution layer C_1 . Convolutions are performed using six 5×5 size convolution kernels and 32×32 original images to generate six 28×28 feature maps, and the feature map is activated using the ReLU function.
- (2) Pooling layer S_2 . The S_2 layer uses a 2×2 pooling filter to perform maximum pooling on the output of C_1 to obtain six 14×14 feature maps.
- (3) Partially connected layer C_3 . The C_3 layer uses 16 5×5 convolution kernels to partially connect with the six feature maps output by S_2 and calculates 16 10×10 feature maps. The partial connection relationship and calculation process are shown in Figure 13. Take the calculation process of the output 0th feature map as an example: First, use the 0th convolution kernel to convolve with the 0, 1, 2 feature maps output by the S_2 layer, and then add the results of the three convolutions, Plus a bias, and finally activate it to obtain the 0th feature map of the C_3 layer.
- (4) Pooling layer S_4 . This layer uses a 2×2 pooling filter to pool the output of C_3 into 16 5×5 feature maps.
- (5) Expand layer S_5 . This layer combines the 16 5×5 feature maps output by S_4 into a one-dimensional matrix of size 400.
- (6) Fully connected layer S_6 . The S_6 layer fully connects the one-dimensional matrix output from the S_5 layer with 10 convolution operators, and obtains 10 classification results as the recognition results of the input image.

From comprehensive analysis of the operating characteristics of each layer of the network, the calculation of each layer of LeNet-5 is summarized in Table 4.

Table 4. LeNet-5 calculation steps of each layer of LeNet-5.

Layer	Calculation Formula	Explanation
C_1	$[C_1^0-C_1^5] = \text{ReLU}[S \times [K_1^0-K_1^5]]$	Convolution and ReLU
S_2	$[S_2^0-S_2^5] = \text{Pool}[C_1^0-C_1^5]$	Pooling
C_3	$[C_3^0-C_3^{15}] = \text{ReLU}[\sum_{i=0}^5 S_2^i \times [K_3^0-K_3^{15}]]$	Convolution, matrix addition, and ReLU
S_4	$[S_4^0-S_4^{15}] = \text{Pool}[C_3^0-C_3^{15}]$	Pooling
S_5	$S_5 = [S_4^0-S_4^{15}]$	Expand
S_6	$[S_6^0-S_6^9] = S_5 \times [K_6^0-K_6^9]$	Fully connected

Figure 13. Schematic diagram of the C_3 layer calculation process of LeNet-5 network.

As can be seen from the above table, the implementation of the LeNet-5 network on the coprocessor can be performed in four steps:

- (1) First, map the C_1 layer and the S_2 layer. Configure the coprocessor to use the convolution, ReLU, and pooling modules of the PE unit, and configure the parameters of these three modules. Configure the crossbar to enable the data flow in the sequence shown in Figure 14a, start the coprocessor, and calculate the C_1 and S_2 layers.
- (2) Map C_3 layer and S_4 layer. Configure the coprocessor to use the convolution, matrix addition, ReLU, and pooling modules of the PE unit, and configure the parameters of these four modules. Configure the crossbar to enable the data flow in the order shown in Figure 14b, start the coprocessor, calculate the C_3 layer and S_4 layer, and cache the calculation result in BUF RAM BANK1.
- (3) The S_5 layer uses CPU calculations. Use a software program to expand the calculation result in (2) into a 1×400 one-dimensional matrix, which is buffered in BUF RAM BANK2.
- (4) Map the S_6 layer. Configure the coprocessor to use the convolution module of the PE unit so that the convolution module supports one-dimensional convolution. Configure the crossbar so that the data flow follows the sequence in Figure 14c and uses only the convolution module. Configure the data source as BUF RAM BANK2, start the coprocessor, and calculate the S_6 layer.

Figure 14. Data flow mapping diagram of each layer of LeNet-5 network: (a) Mapping of C_1 layer and S_2 layer; (b) mapping of C_3 layer and S_4 layer; (c) mapping of S_6 layer.

5.2. Sobel Edge Detection and FIR Algorithm Implementation

In image processing algorithms, edge detection of images is often required. Edge detection based on Sobel operators is a commonly used method. Sobel operators are first-order gradient algorithms that can effectively filter out noise interference and extract accurate edge information [21]. Sobel edge detection uses two 3×3 matrix operators and the input image to convolve to obtain the gray values of the horizontal and vertical edges, respectively. If A is the original image and G_x and G_y are the horizontal and vertical edge images, respectively, the calculation formulas of G_x and G_y are shown in Figure 15.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \times A \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \times A$$

Figure 15. G_x , G_y calculation formula.

The complete edge gray value of the image can be approximated by Equation (1).

$$|G| = |G_x| + |G_y| \quad (1)$$

Before image edge detection, downsampling is usually performed to reduce the image size and the amount of data for operations.

In summary, the implementation of Sobel edge detection on the accelerator can be performed in three steps:

- (1) Calculate G_x and G_y . Configure the coprocessor to use two PE units. Each PE unit uses pooling and convolution modules, and the convolution kernels used by the convolution modules in the two PEs are the two convolution kernels of the Sobel operator. Configure the crossbar so that the data flow follows the sequence in Figure 16a. Start the coprocessor, calculate G_x and G_y , and cache the calculation result in BUF RAM BANK1.
- (2) Use the CPU to calculate $|G_x|$ and $|G_y|$. Use a software program to take the absolute value of each element in the cached result of (1) to obtain $|G_x|$ and $|G_y|$, and cache the calculation result in BUF RAM BANK2.
- (3) Calculate $|G|$. Configure the coprocessor to use the matrix addition module of the PE unit and configure the crossbar so that the data flow follows the order in Figure 16b and only the matrix addition module is used. Configure the data source as BUF RAM BANK2, start the coprocessor, and calculate $|G|$.

In speech signal processing, FIR filters are often used for denoising. FIR filtering is a one-dimensional convolution operation. You only need to configure the PE to use a convolution module. Configure the crossbar so that the data flow follows the sequence shown in Figure 17.

The convolution module is configured as a one-dimensional convolution, loads the convolution kernel into COE RAM, starts the coprocessor, and calculates the FIR filtering result.

Figure 16. Data flow mapping diagram of Sobel edge detection algorithm: (a) Calculate G_x and G_y ; (b) Calculate $|G|$.

Figure 17. Data flow mapping of FIR filter algorithm.

6. Experiment and Resource Analysis

This section completes resource analysis and performance analysis of the designed reconfigurable CNN acceleration coprocessor based on Xilinx FPGA. The FPGA model is Xilinx xc7a100tftg256-1 (Xilinx, San Jose, CA, USA), and the synthesis tool is Vivado 18.1 (Xilinx, San Jose, CA, USA). FPGA circuit synthesis is performed on the E203 SoC connected to the coprocessor. Table 5 shows the resource consumption of each main functional unit.

Table 5. E203 SoC resource consumption list.

Module	Slice LUTs	Slice Registers	DSPs
E203 kernel	4338	1936	0
coprocessor	8534	7023	21
CLINT	33	132	0
PLIC	464	411	0
DEBUG	235	568	0
Peripherals	3199	2281	0
Others	1092	1325	0
Total	17,913	13,676	21

As shown in Table 5, the E203 kernel and coprocessor account for most of the resource consumption in the E203 SoC. The E203 core accounts for 24.2% of the total LUT consumption, while the designed coprocessor accounts for 47.6% of the LUT resource consumption in the SoC.

To evaluate the performance of the coprocessor, the four basic arithmetic algorithms of convolution, pooling, ReLU, and matrix addition are implemented in two ways. One is implemented by the I and M instruction sets of the standard instruction set of RISC-V. The other is implemented using the coprocessor instructions designed in this paper, comparing the number of cycles of algorithm execution in these two implementations. By writing a testbench file to load the compiled binary file as an input stimulus for the entire E203 SoC, it is simulated by modelsim simulation software to count the execution cycles of each algorithm in two implementations. The experimental results are shown in Table 6.

Table 6. Number of execution cycles of each algorithm implemented by different instructions.

Algorithm	Rv32IM Instruction	Coprocessor Instruction
Convolution (4×4 matrix and 3×3 convolution kernel)	12,982	2070
Pooling (6×6 matrix and 2×2 filter)	2452	1706
ReLU (6×6 matrix)	1249	1101
Matrix Addition (6×6 matrix)	2755	2005

As can be seen from Table 6, using the coprocessor can accelerate all four algorithms, and the acceleration ratio of each algorithm is shown in Figure 18.

Figure 18. Acceleration ratio of each algorithm.

It can be seen from Figure 18 that the coprocessor has the most obvious acceleration effect on the convolution algorithm, which is 6.27 times faster than the standard instruction set. This is because, on the one hand, the coprocessor realizes convolution calculation by the special hardware unit, while the RISC-V-based main processor realizes convolution calculation by software; on the other hand, the coprocessor architecture reduces the data moving, further speeding up the algorithm processing speed.

7. Conclusions

In this paper, we optimize the compact CNN accelerator designed by Reference [16], and interconnect the operation units in the acceleration chain with crossbar, to realize the reconfigurable design of a CNN accelerator. Furthermore, the accelerator is connected to the E203 core in the form of a coprocessor, and a reconfigurable CNN acceleration coprocessor is designed. Based on the EAI interface, the hardware design of the reconfigurable CNN-accelerated coprocessor is completed. The designed custom coprocessor instructions are introduced, and the open-source compilation tool chain GCC is modified to complete the compilation environment. Coprocessor instructions are packaged into a function interface in the C language inline assembly mode, and coprocessor acceleration library functions are established. The implementation process of common algorithms on the coprocessor is described. Finally, the resource evaluation of the coprocessor was completed based on the Xilinx

FPGA. The evaluation results show that the coprocessor shown in the design consumes only 8534 LUTs, accounting for only 47.6% of the entire SoC system. By comparing the RISC-V standard instruction set and the custom coprocessor instruction set to achieve the four basic algorithms of convolution, pooling, ReLU, and matrix addition, the number of running cycles is used to evaluate the acceleration performance of the coprocessor. The results show that the implementation of the coprocessor instruction set has a significant acceleration effect on these four algorithms, and the acceleration of the convolution has reached 6.27 times that achieved by the standard instruction set.

Author Contributions: Concept and structure of this paper, N.W.; Resources, F.G. and T.J.; Supervision, L.Z.; Writing—original draft, N.W.; Writing—review and editing, F.G. and F.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (No. 61774086), and the Fundamental Research Funds for Central Universities (No. NP2019102, NS2017023).

Acknowledgments: The authors would like to thank Lei Zhang for his beneficial suggestions and comments.

Conflicts of Interest: The authors declare no conflict of interest.

References

- Sheth, A.P. Internet of Things to Smart IoT Through Semantic, Cognitive, and Perceptual Computing. *IEEE Intell. Syst.* **2016**, *31*, 108–112.
- Shi, W.S.; Cao, J.; Zhang, Q.; Li, Y.H.; Xu, L.Y. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [\[CrossRef\]](#)
- Varghese, B.; Wang, N.; Barbhuiya, S.; Kilpatrick, P.; Nikolopoulos, D.S. Challenges and Opportunities in Edge Computing. In Proceedings of the 2016 IEEE International Conference on Smart Cloud (SmartCloud), New York, NY, USA, 18–20 November 2016.
- Chua, L.O. *CNN: A Paradigm for Complexity*; World Scientific: Singapore, 1998.
- Sainath, T.N.; Kingsbury, B.; Saon, G.; Soltau, H.; Mohamed, A.R.; Dahl, G.; Ramabhadran, B. Deep Convolutional Neural Networks for Large-scale Speech Tasks. *Neural Networks* **2015**, *64*, 39–48. [\[CrossRef\]](#)
- 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, Monterey, CA, USA, 15 February 2015.
- Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* **2017**, *105*, 2295–2329. [\[CrossRef\]](#)
- Parashar, A.; Rhu, M.; Mukkara, A.; Puglielli, A.; Venkatesan, R.; Khailany, B.; Emer, J.; Keckler, S.W.; Dally, W.J. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017.
- Chi, P.; Li, S.; Xu, C.; Zhang, T.; Zhao, J.; Liu, Y.; Wang, Y.; Xie, Y. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea, 18–22 June 2016.
- Hardieck, M.; Kumm, M.; Möller, K.; Zipf, P. Reconfigurable Convolutional Kernels for Neural Networks on FPGAs. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 24–26 February 2019.
- Bettoni, M.; Urgese, G.; Kobayashi, Y.; Macii, E.; Acquaviva, A. A Convolutional Neural Network Fully Implemented on FPGA for Embedded Platforms. In Proceedings of the 2017 New Generation of CAS (NGCAS), Genova, Italy, 6–9 September 2017.
- Liu, B.; Zou, D.; Feng, L.; Feng, S.; Fu, P.; Li, J.B. An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution. *Electronics* **2019**, *8*, 281. [\[CrossRef\]](#)
- Kurth, A.; Vogel, P.; Capotondi, A.; Marongiu, A.; Benini, L. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. In Proceedings of the Workshop on Computer Architecture Research with RISC-V (CARRV), Boston, MA, USA, 14 October 2017.

14. Ajayi, T.; Al-Hawaj, K.; Amarnath, A. Experiences Using the RISC-V Ecosystem to Design an Accelerator-Centric SoC in TSMC 16nm. In Proceedings of the Workshop on Computer Architecture Research with RISC-V (CARRV), Boston, MA, USA, 14 October 2017.
15. Matthews, E.; Shannon, L. Taiga: A Configurable RISC-V Soft-Processor Framework for Heterogeneous Computing Systems Research. In Proceedings of the Workshop on Computer Architecture Research with RISC-V (CARRV), Boston, MA, USA, 14 October 2017.
16. Ge, F.; Wu, N.; Xiao, H.; Zhang, Y.; Zhou, F. Compact Convolutional Neural Network Accelerator for IoT Endpoint SoC. *Electronics* **2019**, *8*, 497. [[CrossRef](#)]
17. Patterson, D.; Waterman, A. *The RISC-V Reader: An Open Architecture Atlas*; Strawberry Canyon: Berkeley, CA, USA, 2017.
18. Available online: <https://riscv.org/> (accessed on 26 February 2020).
19. Available online: https://github.com/SI-RISCV/e200_opensource (accessed on 26 February 2020).
20. El-Sawy, A.; El-Bakry, H.; Loey, M. CNN for Handwritten Arabic Digits Recognition Based on LeNet-5. *Springer Sci. Bus. Media* **2016**, *533*, 566–575.
21. Bhogal, R.K.; Agrawal, A. Image Edge Detection Techniques Using Sobel, T1FLS, and IT2FLS. In Proceedings of the ICIS 2018, San Francisco, CA, USA, 13–16 December 2018.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).