IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 25 September 2023, accepted 17 October 2023, date of publication 23 October 2023, date of current version 26 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3326714

RESEARCH ARTICLE

Automatic Compilation of CNN on Reconfigurable Array Processor Based on Scalable Instruction Group Library

YUANCHENG LI, NA WANG, MINGWEI SHENG, AND JIAQI SHI
College of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an 710054, China

Corresponding author: Yuancheng Li (yuanch_li@126.com)

This work was supported in part by the National Key Research and Development Program of China under Grant 2022ZD0119005, in part by the Key Project of the National Natural Science Foundation of China under Grant 61834005, and in part by the Natural Science Basic Research Plan in Shaanxi Province of China under Grant 2020JM-525.

ABSTRACT Although reconfigurable architecture is an inevitable choice for dealing with high-intensive applications, the opacity of hardware structure programming and the fuzziness or imprecision of many dependencies in applications make it difficult to develop applications and take full advantage of reconfigurable architecture. In order to address these challenges, this paper proposes an automatic compilation framework based on a scalable instruction group library, using convolutional neural network (CNN) applications as an example. The proposed framework first identifies functions such as data extraction, data distribution, data summary, and data processing, and divides the reconfigurable processors into multiple logical types. Next, the calculation modes are extracted by analyzing the CNN calculation process. Based on these calculation modes, efficient assembly instruction groups are designed, constituting the scalable instruction group library. Using this library, efficient mapping for CNN applications written in high-level languages can be easily realized on reconfigurable array processors. The experimental results demonstrate that the proposed method reduces the difficulty of programming and achieves better speedup performance compared with OpenMP and LLVM parallel compilation models. Moreover, the scalability of the instruction group library provides helpful guidance for the reconfigurable implementation of domain-oriented applications.

INDEX TERMS Reconfigurable architecture, reconfigurable computing, compilation techniques, CNN.

I. INTRODUCTION

With the flourishing development of CNN, more and more application fields have benefited from its advancement. However, the typical high-density characteristics of CNN, including intensive computation and memory access, impose higher requirements on processor efficiency, flexibility, and resource utilization [1], [2], [3]. Because of the high flexibility of general-purpose processors and the high energy efficiency of specialized hardware, as shown in Figure 1, reconfigurable architecture has become an inevitable choice for addressing high-intensity applications [4], [5], [6], [7]. However, due to the complexity of programming the hardware structure, the fuzziness or imprecision of many application dependencies, and the resulting conservative parallel task partitioning strategy, it is challenging to develop applications and fully harness the advantages of reconfigurable architecture. Reconfigurable processor compilation technology [8], [9], [10] can generate configuration information by automatically mining parallelism, reducing storage delays, and employing other optimization techniques. This enables all hardware resources on the array processors to work efficiently together, offering the potential to overcome the usability limitations of reconfigurable architecture [11]. However, due to the significant differences in hardware structure between reconfigurable array processors and general-purpose processors, the traditional compilation methods cannot be directly applied to reconfigurable array processors,

The associate editor coordinating the review of this manuscript and approving it for publication was Ali Kashif Bashir.

© 2023 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

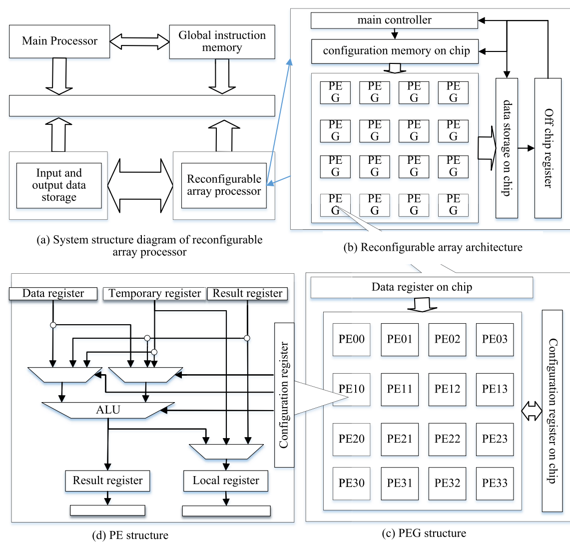


FIGURE 1. Typical architecture of a reconfigurable array processor.

making the compilation process for reconfigurable array processors extremely challenging [12], [13].

Many existing methods based on compilation have achieved good performance in reconfigurable structures, primarily focusing on two aspects: compilation optimization through software and hardware co-design and various high-level synthesis tools (HLS). In contrast to traditional CGRA static allocation schemes, where different paths are allocated to non-overlapping spatial and temporal dimensions, [14] proposed a 4D-CGRA solution that incorporates two spatial dimensions, time, and branching based on collaborative compilation architecture. This approach allows mutually exclusive branch paths to overlap and be mapped to the same target processing unit, effectively reducing resource waste. To achieve faster execution and shorter compilation times, [15] abandoned the time-consuming and costly hardware measurements of manual optimization libraries or traditional heuristic methods. Instead, they proposed a solution that can quickly adapt to previously unseen design spaces for code optimization by designing an adaptive sampling algorithm with faster search speeds and higher performance. By implementing partial reconfiguration of a packet-switched fat-tree, [16] introduced a divide-and-conquer approach to reduce compilation times. Since the partially reconfigured leaves are independent of each other and can be compiled separately in parallel, only the corresponding leaves need to be incrementally compiled for generating the final bitstream.

While the quality of results produced by high-level synthesis (HLS) tools has tended to lag behind those of manual register-transfer level (RTL) flows, the adoption of HLS from languages such as C++ has significantly improved programmer productivity [17]. To address the limited parallelism resulting from excessive reliance on manual experience for application partitioning and mapping in existing reconfigurable processor compilation systems, the Polysa compilation framework was introduced in [18]. It achieved the end-to-end fully automatic translation of high-level languages to FPGA for the first time. This framework allows for the generation of the best design scheme while considering multiple constraints by amalgamating previous manual design approaches and conducting performance analysis. To reduce the effort and expertise required for customizing CNN design models, [19] proposed an RTL-level CNN compiler. RTL modules are developed to correspond to different operations in each CNN layer. As a result, the compiler can automatically generate customized FPGA hardware with consistent performance for various inference tasks and CNNs. Reference [20] introduced the overlay processor OPU for accelerating CNN networks. OPU's software-like programmability allows well-designed OPU instructions to offer ample flexibility and performance while simplifying microarchitecture and compiler development complexity. Reference [21] presented a compilation framework named AutoSA for generating systolic arrays on FPGA. This end-to-end compiler alleviates programmers from the complex manual trial-and-error iterative process, which typically arises from the high knowledge requirements of both low-level hardware structure details and high-level application features. To rapidly and efficiently implement CNN inference accelerators on FPGAs, [22] proposed a compilation flow that can integrate a pre-trained model into OpenCL kernels using the TVM compiler. Although its performance lags behind hand-optimized methods, this approach is valuable for rapid FPGA prototyping design without requiring extensive hardware design expertise.

Although the existing methods based on compilation support have achieved good results in reducing programming difficulty and improving execution performance, they are facing more and more complicated CNN applications caused by higher classification accuracy and the application of CNN in more fields. Consequently, it is becoming more difficult to develop CNN applications and map FPGA hardware with high execution performance. In this paper, we propose an automatic compilation framework for CNN based on a scalable instruction group library. Firstly, we divide the reconfigurable processors into logical multiple types based on different functions such as data extraction, data distribution, data summary, and data processing. Next, we extract the calculation modes by analyzing the CNN calculation process. Then, by consulting the logical multiple types and the calculation modes, we design an efficient assembly instruction group, which constitutes the scalable instruction group library. Finally, for CNN applications written in high-level languages, the efficient mapping on reconfigurable array processors can be easily achieved based on the scalable instruction group library. The experimental results demonstrate that the proposed method can reduce the difficulty of programming and achieve better speedup performance compared to OpenMP and LLVM parallel compilation models. Specifically, the three main contributions of our work are as follows:

- A scalable instruction group library is constructed, in which each efficient assembly instruction group

is designed according to the logical multiple types of configurable processors and the calculation modes extracted from CNN. This library deeply reflects the mapping relationship between calculation modes and configurable processors.

- Based on the scalable instruction group library, we propose an automatic compilation framework that can be used to map CNN applications written in high-level languages onto reconfigurable processors. No manual intervention is required in the whole compilation and mapping process, and CNN programming does not require programmers to understand the underlying hardware structure.
- The scalable instruction group library can be easily expanded, iterated, and optimized according to the library construction method. For specific field applications, it can be easily applied to configurable processors simply by extending the instruction group library, which is why we call it a scalable instruction group library. Therefore, our work can provide helpful guidance for the reconfigurable implementation of domain-oriented applications.

The subsequent chapters are organized as follows: Section II introduces the overview of the compilation framework, Section III presents the calculation modes extracted from CNN, Section IV describes the construction method of the scalable instruction group library in detail, Section V proposes the automatic compilation method for CNN, Section VI provides the experiment and results analysis, and finally, Section VII concludes this work.

II. OVERVIEW OF THE COMPILATION FRAMEWORK

A. BASIC IDEA

CNN applications for reconfigurable processors can be realized by either writing assembly programs, which require knowledge of low-level hardware structure details for programmers, or writing high-level programs based on domain-specific languages (DSL). However, adapting high-level programs to more complex application scenarios is increasingly difficult, leading to a struggle in balancing efficiency and ease of programming for programmers [23]. Nevertheless, for specific repeated operations such as multiplication, addition, and comparison, the compiled representation under the reconfigurable structure remains the same. Therefore, if we can establish a mapping relationship between high-level application programs and compiled representations, similar to LLVM [24] Intermediate Representation (IR), and then map the IR to reconfigurable processors, we can maintain good performance while keeping programming easy. Based on the above discussion, we deeply analyze and extract the calculation mode of the CNN algorithm to abstract it. Next, through manual optimization, we explore the optimal mapping relationship on the reconfigurable structure and build the instruction group according to the logical reconfigurable processors. Finally, using the instruction groups that constitute the instruction group library, it becomes easy to map CNN applications written in high-level languages to reconfigurable processors.

B. DIVIDING OF LOGICAL CONFIGURABLE PROCESSORS

Reconfigurable processors can achieve efficient calculations, nearly matching the performance of Application-Specific Integrated Circuits (ASICs), by adapting their hardware functions based on the hardware configuration during operation. Figure 2 displays an array of processing element groups (PEGs) within the reconfigurable array processor employed in our paper, with each PEG containing 16 configurable processing elements (PEs). Each PE can execute data processing functions or control functions depending on the specific configuration. Within the reconfigurable structure, the ALU in each PE can flexibly perform various common operations, including addition, subtraction, logical AND, OR, NOT, and more.

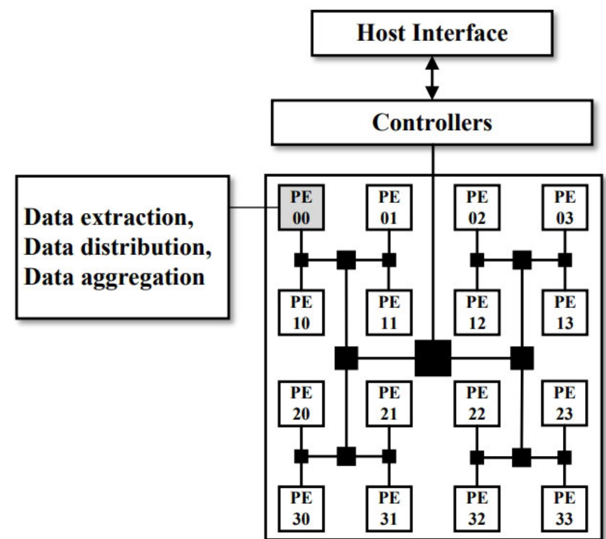


FIGURE 2. Schematic diagram of logical partitioning for reconfigurable processors.

To better realize and optimize the mapping of CNN calculation modes onto reconfigurable processors, it is necessary to logically divide or partition reconfigurable processors based on their functions. These functions primarily encompass data extraction, data distribution, data summarization, and data processing. Furthermore, specific processors handle data extraction, distribution, and aggregation tasks. In this paper, to facilitate the implementation and demonstrate the effectiveness of the proposed automatic compilation method, we present the logical partitioning scheme for the 16 PEs, as depicted in Figure 2. Each PE00 in every PEG is logically partitioned or configured as the execution control functional unit responsible for completing data extraction, distribution, and aggregation. The remaining PEs in each PEG are logically partitioned or configured as data processing functional units responsible for completing data processing tasks.

C. COMPILATION FRAMEWORK

The overall compilation framework structure and its process flow is depicted in Figure 3.

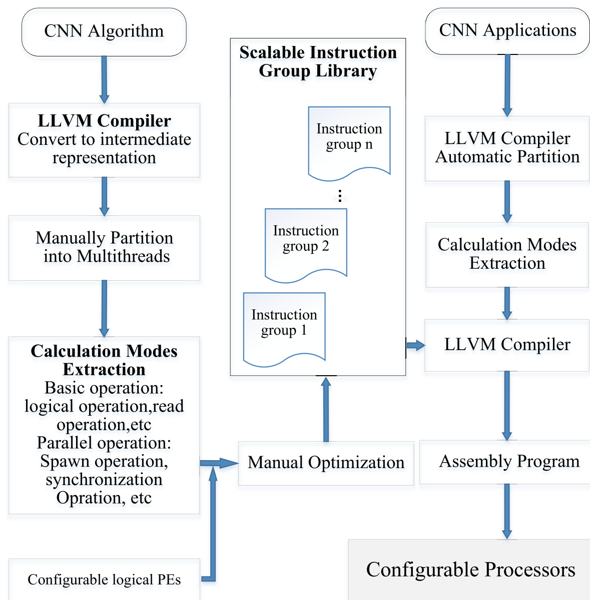


FIGURE 3. Overall compilation flow of the proposed compilation framework.

The input CNN algorithm, written in a high-level language, is first converted into LLVM IR code through the LLVM compiler. Then, the IR codes are partitioned into multiple threads (meaning multiple program segments that can be executed in parallel) using manual mode, which is considered the best optimization method based on the configurable processor. For simplicity, we assume that reconfigurable resources are sufficient. At the IR level, the calculation modes are extracted, which include basic operations and parallel operations. Finally, based on the configurable logical processors, these operations are classified and combined into different instruction groups described in the assembly program.

It should be emphasized that there are some design principles for instruction groups: 1)The extracted calculation modes are typical operations of the CNN algorithm, which occur repeatedly in the CNN process, such as addition operations and comparison operations. 2)When constructing the assembly instruction group, it is necessary to ensure that no calculation errors are caused by the previous operations during program execution. This can be achieved, for example, by clearing the registers involved in the operation in advance. 3)The assembly instruction group and the calculation mode have a one-to-one mapping, meaning that a calculation mode corresponds to a specific assembly instruction group, and the same assembly instruction group corresponds to a specific calculation mode. For a specific CNN application, it is first converted into LLVM IR by the LLVM compiler, and then it is partitioned into multiple threads on the reconfigurable processor at the IR level based on the paralleling partitioning method [25]. Next, the calculation modes are extracted and compiled into an assembly program suitable for configurable processors, according to the scalable instruction groups provided by the LLVM compiler.

III. CALCULATION MODES EXTRACTED FROM CNN

CNN is a typical high-intensive application with distinctive features such as one-way direction of data dependencies and a high proportion of data reuse. Therefore, the main issues to be considered when extracting the calculation mode are data storage space and processor execution order. Based on the execution process of CNN on a reconfigurable processor and combined with the feature information of CNN extracted by the LLVM compiler, the extraction process of the calculation mode is shown in Figure 4.

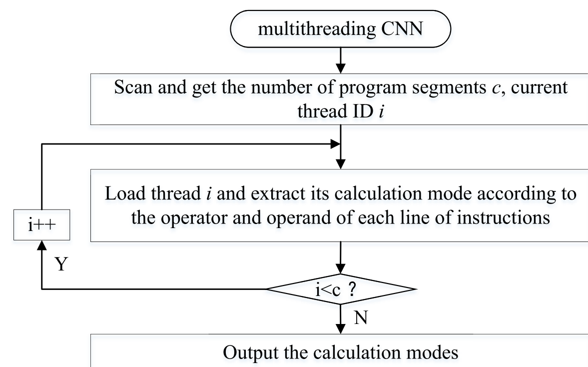


FIGURE 4. The extraction process of calculation mode.

It can be observed that the calculation modes are mainly extracted from the operators, operands, and the parallel execution information of the multithreads. Therefore, the extracted calculation modes mainly include basic operations and parallel operations, which are discussed below.

A. EXTRACT BASIC OPERATIONS

Basic operations are designed based on operator types, encompassing four fundamental arithmetic operations, logical operations, combination operations, addressing operations, and read or write operations. The four arithmetic operations encompass addition, subtraction, multiplication, and division operations, with their typical mode outlined in (1). When utilizing operations of this kind, it is imperative to ensure correctness and timeliness.

$$a = b\ op_sign\ c. \tag{1}$$

Logical operations mainly occur during the conversion between layers and statement jump, and the general mode of such operations is shown in (2). The operator *logjc_sign* is used to control the operation mode, and the operation result determines whether statement d1 or d2 is executed based on the operands a and b.

$$c = a\ logic_sign\ b\ ?\ d_1 : d_2 \tag{2}$$

For combination operations such as convolution and pooling, as shown in (2), it is necessary to decompose the operations to obtain specific operands, operators, and repetition times in order to ensure the correctness of the calculation results.

$$c = \sum_{i=0}^{1} \sum_{j=0}^{1} a_{ij} \times b_{ij} \tag{3}$$

The addressing operation is often used for reading data. For example, in the convolution operation, a large amount of data needs to be read, and the value of each element in the convolution kernel must be obtained to extract the value of the pixel in the image. The characteristic of the addressing operation is that the operands are addresses, and correct addressing operations can prevent data errors, calculation errors, etc., caused by overstepping the address boundary. Due to the fact that load and store instructions are mainly related to address operations, these instructions will be classified as address operations.

B. EXTRACT PARALLEL OPERATIONS

Parallel operation refers to the synchronous execution of multithreads, as shown in Figure 5. For example, thread 1, 2, 3, and 4 are synchronously executed to shorten the program execution time. There are two main operation types in parallel execution: spawning operation (Sp_Op) and synchronization operation (Syn_Op). The spawning operation is used to start the parallel execution of a thread, and the synchronization operation is used to keep a thread waiting for exchanging data or guaranteeing the logical execution sequence.

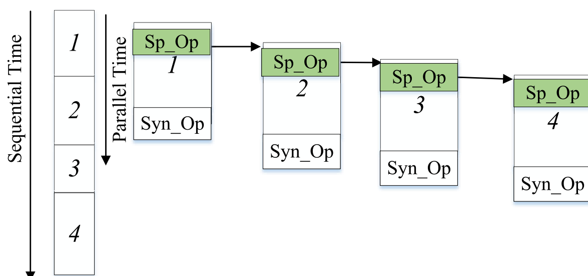


FIGURE 5. Schematic diagram of parallel operation execution.

IV. CONSTRUCTION OF SCALABLE INSTRUCTION GROUP LIBRARY

The reconfigurable processor instruction set adopted in this paper is a simplified version based on MIPS instructions, as shown in Table 1. All other complex instructions can be completed by combining these instructions with registers. For example, the register clearing operation can be accomplished by the instruction *ADDI R1, R0, # 0*. During the process of implementing complex instructions, it is crucial to ensure the correct execution order of instructions to guarantee accurate operations. When designing an assembly instruction group, the first consideration is the functions that the assembly instruction group needs to realize. Specific assembly instructions, registers, and storage units are designed accordingly based on these functions. Next, to maintain the consistency of data, each instruction group is designed with only one input and one output, following the characteristics of the reconfigurable structure. Finally, the execution order of the instruction group must align with the execution order of the original calculation function to ensure the correct functioning of the processor. When designing the corresponding assembly instruction group according to the specific calculation mode, strict adherence to the execution order of the calculation mode and the division of processor positions is essential. The assembly instruction group design for basic operations mainly focuses on specific operations. On the other hand, the assembly instruction design for parallel operations is primarily based on the dependencies between processors and the partition information of multithreads.

TABLE 1. Assembly instruction set of reconfigurable processor.

Type	Assembly Instruction	Description
Arithmetic operation and shift	ADD	RD ← RS + RT
	ADDI	RD ← RS + imme
	SUB	RD ← RS − RT
	AND	RD ← RS AND RT
	SRL	RD ← RS ≫ RT
	NOP	NO operation
Data transfer	LD	RD ← memory[RS]
	ST	memory[RD] ← RS
Jump	BEQ	if RS = RT, then branch
	BNE	if RS ≠ RT, then branch
	SLT	if RS < RT, then RD = 1
	J	PC = #immediate
Immediate operation	LDI	RD ← memory[#imme]
	STI	memory[imme] ← RS
	SUBI	RD ← RS − imme

A. INSTRUCTION GROUP DESIGN OF BASIC OPERATION

The basic operation instruction group is primarily composed of compound operations, address operations, and access operations. By analyzing the operation rules, the number of participating operations is determined, appropriate registers and memory cells are selected, the number of instructions and the order of instruction execution are determined. Finally, the basic operation assembly instruction group can be constructed. This paper mainly focuses on describing how to construct instruction groups.

For a more intuitive explanation, we will use addition and multiplication operations as examples to provide specific instruction groups separately. For instance, based on (1), the addition operation requires 3 operands, which means three storage units are needed to store the operands and their addresses for the design of the corresponding instruction group. The instruction groups for the addition operation are shown in Table 2.

The instruction sequence can be divided into three parts: fetch operation, add operation, and write operation result. First, the address corresponding to the first operand is obtained through the fetch operation, as seen in the first

TABLE 2. The instruction groups of addition operation operaton.

Serial number	Instruction sequence	Address	Data
1	ADDI R1,R0,#1	1	25
2	LD R2,R1	2	26
3	ADDI R1,R1,#1	3	27
4	LD R3,R1	25	12
5	ADDI R4,R3,R2	26	13
6	ADDI R1,R1,#1	27	-
7	ST R4,R1	-	-

and second instruction sequences in Table 2. The instruction *ADDI* retrieves the address for operand 1, and the instruction *LD* fetches the data from the storage area based on the address obtained in register *R1*, writing the data into register *R2*. Similarly, the address of the second operand can be obtained through the 3rd and 4th instruction sequences. Next, for the addition operation, as shown in the 5th instruction sequence, *R2* and *R3* are added, and the result is written into *R4*. Finally, the write operation is executed in the 6th instruction sequence. The instruction *ADDI* obtains the address of operand 3 in the 3rd sequence, and the instruction *ST* in the 7th instruction sequence writes the operation result from *R4* into the storage unit corresponding to the address in *R1*.

The subtraction operation is similar to the addition operation. During the calculation process, the operands need to be operated using the instruction *SUB*. For example, the instruction "sub *R4*, *R3*, *R2*" indicates that the result of subtracting *R2* from *R3* is stored in *R4*. By using a similar method as described above, we can construct the subtraction instruction groups.

Since there is no multiplication instruction in the reconfigurable assembly instruction used in this paper, a set of multiplication instruction groups suitable for the reconfigurable processor is designed based on the operation rules of multiplication, as shown in Table 3. From Table 3, it can be observed that the multiplication operation is similar to the addition operation. Similarly, for the division operation, two points need to be considered when designing the instruction groups: 1)The divisor cannot be zero. After reading operand 2, it is necessary to determine whether the value of operand 2 is zero. If it is zero, the execution will either directly jump to the last instruction or continue to execute. 2)After obtaining the operands, the division operation is processed using the arithmetic right shift operator to obtain the division instruction groups.

Logical operations are similar to four arithmetic operations and require reading operands through read operations during the calculation process. However, they differ from the four arithmetic operations in two main points: 1)Operation instructions are different, with logical operation instructions including AND, OR, and NOT. 2)Logical operations are accompanied by the loop structure. After the operation is completed, the corresponding jump label needs to be added.

During the addressing operation, the data in the designated address can be transferred to the register by the *LD* instruction. However, it should be emphasized that since the

TABLE 3. The instruction groups of multiplication operation.

Serial number	Instruction sequence	Address	Data
1	ADDI R1,R0,#1	1	25
2	LD R2,R1	2	26
3	ADDI R1,R1,#1	3	27
4	LD R3,R1	25	12
5	ADDI R5,R0,#1	26	13
6	MUL:ADDI R5,R5,#1	27	-
7	LFT R2,R2,R5	-	-
8	RFT R3,R3,R0	-	-
9	BNE R3,R0,#MUL	-	-
10	ADDI R1,R1,#1	-	-
11	ST R4,R1	-	-

address is dynamically generated, the address data for the addressing operation needs to be calculated in advance when designing the addressing operation instruction groups. For memory access operations, they can be designed using the *ST* instruction, which includes two registers. The data to be stored is placed in register 1, and the address where the data needs to be stored is placed in register 2. Since the storage address cannot be directly loaded manually, we can write the data into the specified area by determining the area where the data may exist according to the linear search method. The design of the storage zone is similar to the design of the multiplication and addition operation instruction groups.

Additionally, for combination operations which refers to several basic operations, it has been decomposed into multiple single operations through LLVM compiler. Taking a * B + C * D as an example, the combination optimization problem can be solved by decomposing it into the form % 1 = a * B, % 2 = C * D, % 3 = % 1 +% 2.

B. INSTRUCTION GROUP DESIGN FOR PARALLEL OPERATION

The parallel operation is a type of operation that results from parallel execution, as shown in Figure 5, which includes thread spawning operations and thread synchronization operations. Thread spawning means that the activation status for thread running has been met, and the thread can be spawned to execute. The spawning instruction groups are described in Table 4. The processor judges whether the activation status is satisfied by reading the data value of the specified location (such as address 1). If it is satisfied, the next thread will be executed; otherwise, it continues to wait. Thread synchronization operation is the process of exchanging data or ensuring the logical execution order among the parallel multithreads. The instruction groups of synchronization operation are shown in Table 5.

TABLE 4. The instruction groups of spawning operation.

Serial number	Instruction sequence	Address	Data
1	ADDI R1,R0,#1	1	999
2	ADDI R2,R0,#999	-	-
3	Active: LD R1,R1	-	-
4	BNE R1,R2,#Active	-	-
5	ADDI R1,R0,#1	-	-

TABLE 5. The instruction groups of synchronization operation.

Serial number	Instruction sequence	Address	Data
1	ADDI R2,R0,#128	128	999
2	ADDI R3,R0,#999	-	-
3	Para:LD R2,R2	-	-
4	BNE R3,R2,#Para	-	-
5	ADDI R1,R0,#512	-	-
6	ADDI R2,R0,#999	-	-
7	ST R2,R1	-	-

By setting a specific Process Element (PE) (e.g., PE00) used to record the data information exported by each thread, the specific PE will send the mark of calculation completion to all the PEs (e.g., send the synchronization signal 999 to the No. 1 storage unit of PE01). When all the instructions have been completed, each PE will immediately read this mark. Based on this mark, the thread synchronization instruction is then executed.

Based on the above methods of instruction group construction, all instruction groups for CNN can be constructed, forming the instruction group library. Moreover, for other specific applications, the instruction group library can be easily expanded, iterated, and optimized following the construction method provided in this paper. Additionally, in specific fields of applications, the instruction group library can be easily applied to configurable processors by extending it.

V. AUTOMATIC COMPILATION

Based on the processor division results presented in Section II, the execution tasks are divided into four parts, which include data extraction, data distribution, data processing, and data transmission. For data extraction, PE00 writes the data into the storage area. In data distribution, PE00 distributes the extracted data to each PE. Data processing is mainly achieved through the mapping scheme of calculation mode and assembly instruction groups. Data transmission, on the other hand, is realized using *ST* and *LD* instruction groups. Taking convolution and pooling as examples, the implementation process is illustrated in Figure 6. The left subfigure represents the conversion of convolution into multiplication and addition operations, while the right subfigure represents the conversion of pooling into comparison operations.

Taking multiplication operation and addition operation as examples, we define the multiplication and addition operation zones, result output zones, data zones, and result zones. The multiplication and addition operations zone refers to the data addresses involved in multiplication and addition operations. The result output zone refers to the addresses where the calculation results need to be exported. The data zone refers to the area where the data involved in the operation is stored. The result zone refers to the area where the operation result is stored after multiplication and addition operations are completed. The comparison operation is similar to multiplication and addition operations, except that only the specific operation differs. The difference lies

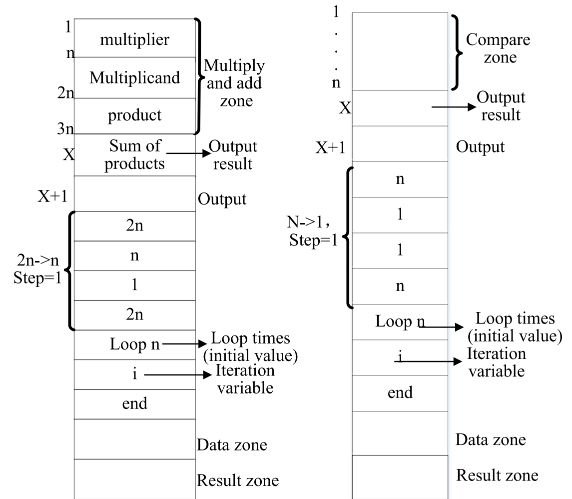


FIGURE 6. Schematic diagram of multiplication and addition operations, and compare operations: multiplication and addition operations(left), compare operations(right).

in simply changing the calculation method for operations: multiplication and addition operations involve calculating the product of two sets of numbers and then summing them up, while comparison operations require finding the maximum value among n numbers. The mapping rules of the automatic compilation method are constructed based on the association rules between the calculation modes and the assembly instruction groups of the CNN algorithm. The calculation modes are extracted according to the operators and operands in the CNN algorithm. The calculation mode contains all operations in the operation process of the CNN algorithm. The assembly instruction groups in the mapping rule are generated one-to-one according to the calculation modes, and the extracted calculation modes are consistent with the type and number of operations in the CNN algorithm. Therefore, the mapping rules are complete and comprehensive. In the next section, Section VI, we will perform simulation and FPGA verification to prove the correctness and effectiveness of our proposed method.

VI. EXPERIMENT AND RESULTS ANALYSIS

A. EXPERIMENTAL ENVIRONMENT

The compiler adopts LLVM 9.0, and the simulation platform adopts modulesim10.1. C and Intel (R) Xeon (R) CPU e5-2678 V3 @ 2.50ghz which has 96 isomorphic multicores. Additionally, the reconfigurable array processor adopts BEE4 which includes Four virtex-6 series xc6vlx550t FPGA development boards from Xilinx Company. The simulation parameters and data sets are provided in Table 6.

B. SIMULATION AND FPGA VERIFICATION

To verify the correctness of the assembly instruction groups, functional simulation is carried out using modulesim10.1c. During the simulation process, the basic operations and parallel operations are simulated separately.

TABLE 6. The instruction groups of spawning operation.

Parameter	description
Number of PE	1024
Number of register	R0 - R15
load/store	RISC
Instruction storage capacity	512*32 (bit)
Data storage capacity	512*16 (bit)
Clock cycle	20ns
activation function	ReLU
Data set	MINIST[26]

To verify the accuracy of the automatic compilation method, the assembly instructions generated by the automatic compilation method are tested on the FPGA platform. First, the assembly instructions generated by the automatic compilation method are obtained, converted into machine-recognizable binary instruction groups by the assembler, and placed in the processor file corresponding to the instructions shown in Figure 7. Then, the generated files are imported into the reconfigurable array processor, and the calculation results are obtained. Finally, the results are compared with the simulation results to verify their correctness.

```
#LoopS01:NOP                00110100000000000000000000000000
MUL0:ADDI R1,R0,#1          00001000010000000000000000000001
LD R1,R1                    01000000010001000000000000000000
ADDI R1,R1,#0               00001000010001000000000000000000
LD R1, R1                   01000000010001000000000000000000
ADDI R2,R0,#16              00001000100000000000000000010000
LD R2,R2                    01000000100010000000000000000000
ADDI R2,R2,#0               00001000100010000000000000000000
LD R2,R2                    01000000100010000000000000000000
BEQ R2, R0, #MUL_E00        10000000100000000000000000001111
ADDI R3,R0,#0               00001000110000000000000000000000
ADDI R4,R0,#0               00001001000000000000000000000000
MUL00:ADD R4,R4,R1          00000101000100000100000000000000
SUBI R2,R2,#1               10100000100010000000000000000001
BNE R2,R0,#MUL00            10000100100000000000000000001100
MUL_E00:ADDI R3,R0,#0       00001000110000000000000000000000
ADDI R3,R3,#32              00001000110011000000000000100000
ST R4,R3                    01000101000001100000000000000000
MUL1:ADDI R1,R0,#2          00001000010000000000000000000010
LD R1,R1                    01000000010001000000000000000000
ADDI R1,R1,#0               00001000010001000000000000000000
LD R1, R1                   01000000010001000000000000000000
ADDI R2,R0,#17              00001000100000000000000000010001
LD R2,R2                    01000000100010000000000000000000
```

FIGURE 7. Assembly instructions and corresponding machine instructions generated by our automatic compilation method: Assembly instructions(left), Machine instructions(right).

C. PERFORMANCE ANALYSIS

To further prove the effectiveness of the automatic compilation method proposed in this paper, we introduce two typical parallel compilation models, OpenMP and LLVM, for performance comparison. LLVM stands as an advanced compilation system with several obvious advantages, including extensibility, extensive language support, a robust toolset and ecosystem, and wide applicability. Meanwhile, OpenMP represents a set of programming APIs facilitating multi-threaded concurrent programming through cross-platform shared-memory mechanisms, and adopts a portable and extensible model, offering developers a straightforward and flexible development platform for parallel applications. Given their broad representativeness and excellent application performance, as well as their similarity to the scenarios applicable to reconfigurable processors, we compared our speedup performance with OpenMP and LLVM parallel compilation models. Using the trained Lnet-5 network as the input network, we conduct performance comparisons based on OpenMP, LLVM, and the proposed method in this paper for different scales of test data, such as 1, 5, 10, 20, 30, 50, 100, and 200, respectively. The comparison of execution performance is shown in Figure 8. Additionally, in the experiment, the number of reconfigurable processors used is 16, and the number of multicore utilized for OpenMP and LLVM is also 16, based on the server with Intel (R) Xeon (R) CPU e5-2678 V3 @ 2.50GHz.

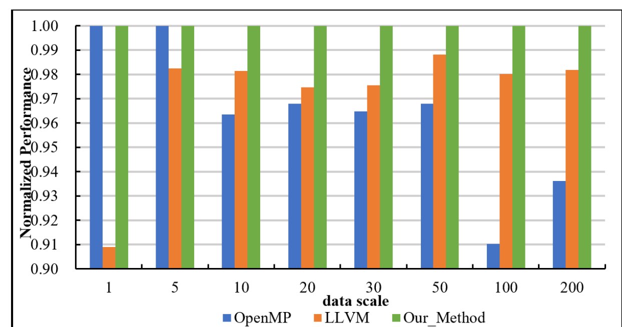


FIGURE 8. The comparison of the execution performance based on OpenMP, LLVM and the proposed method.

From Figure 8, it can be observed that our method outperforms the OpenMP and LLVM parallel compilation models in terms of execution time, achieving speedup improvements of 3.63% and 2.84%, respectively. These results clearly demonstrate that the proposed automatic compilation method performs even better than the typical parallel compilation models. Moreover, as the scale of input test data gradually increases, our method consistently maintains a performance advantage over the other two methods, indicating its strong generalization ability.

To analyze the deep-seated reasons, it is mainly because of the following factors: 1) CNN applications have complex data and control dependencies, making it challenging to extract all parallel information solely from high-level semantics. Unfortunately, both the OpenMP and LLVM compilation models heavily rely on high-level semantics for parallel partitioning. Moreover, to handle these complex dependencies, both compilation models often adopt conservative parallel strategies, leading to limited exploitation of parallelism. 2) In contrast to OpenMP and LLVM, our compilation method focuses not only on high-level semantic information but also on parallel information closer to the lower level (hardware structure). By manually constructing assembly instruction groups based on high-level parallel information, including parallel partition and mapping knowledge, our method can effectively reflect the advantages of software and hardware co-design and uncover the parallelism of the application.

Consequently, our method can better leverage the advantages of configurable hardware.

To further evaluate the effectiveness of the proposed method, we conducted experiments on an FPGA-based experimental platform. The reconfigurable array processor utilized BEE4, which incorporates four Virtex-6 series XC6VLX550T FPGA development boards from Xilinx Company. We conducted a performance comparison of the CNN application using our compilation method and a manual optimization method (where the assembly program is directly written by hand). The results are shown in Figure 9. Compared with the CNN assembly application that is meticulously developed by hand and considered to have near-optimal performance, the algorithm execution time of the automatic compilation method is longer. But at the same time, from the trend shown in Figure 9, it can be seen that as the scale of test data expands, the performance of our proposed automatic compilation method is very stable. On average, the proposed automatic compilation method achieves 62.56% of the optimal performance. On the one hand, for the automatic compilation method proposed in this paper, while greatly reducing the programming difficulty, it still has comparability with the optimal manual method and has excellent generalization ability which strongly demonstrates that our method is feasible and effective. Additionally, it also highlights that there is still room for further improvement and optimization of this method.

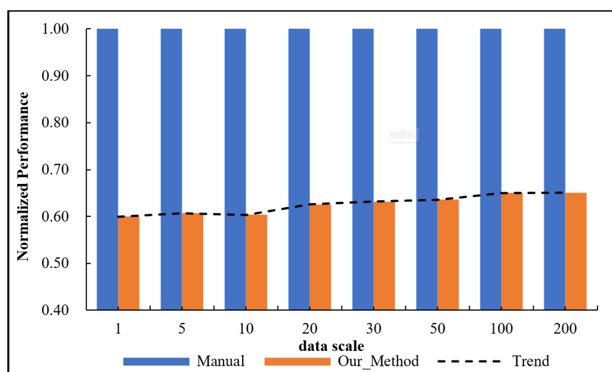


FIGURE 9. Comparison of execution performance based on automatic compilation method and manual optimization method.

In a deeper analysis, it can be observed that although the assembly instruction groups constructed by hand can effectively reflect the parallel partition and mapping knowledge of both high-level semantic information and low-level hardware structure, they often adopt a more conservative compilation strategy. This may include inserting redundant waiting cycles during simultaneous operations to strictly ensure absolute correctness of timing. As a result, the manual optimization method may not fully exploit the available parallelism, leading to suboptimal performance. However, with the continuous optimization and improvement of relevant compilation strategies in the future, these adverse effects can be gradually reduced or even eliminated, potentially leading to better performance.

VII. CONCLUSION

More and more application fields are reaping the benefits of artificial intelligence technologies like CNN. However, these typical high-density applications present greater challenges for processors in terms of their processing capabilities and flexibility. While reconfigurable architecture is a necessary choice for addressing high-intensity applications, it still encounters difficulties due to the complexity of hardware structure programming and the vagueness or imprecision of many dependencies in applications. Consequently, developing applications and fully capitalizing on the advantages of reconfigurable architecture can be quite challenging. In response to these challenges, we propose an automatic compilation of CNN on a reconfigurable array processor, based on a scalable instruction group library. This library consists of meticulously designed instruction groups, which encompass parallel partition and mapping knowledge of both high-level semantic information and low-level hardware structure. This compilation approach facilitates the straightforward mapping of CNN applications written in high-level languages onto reconfigurable array processors without manual intervention.

The Experimental results show that the proposed method can reduce the difficulty of programming and achieve better speedup performance compared with OpenMP and LLVM parallel compilation models, and these also fully show that our method has good generalization ability. Meanwhile, in our method, the delay between PEs and other issues arising from the questions such as the limited power, outage probability, latency, unauthorized access etc., are mainly limited by the interconnection and transmission structure of the array processor itself, and are not directly related to the automatic compilation method. Therefore, the proposed method has good adaptability. Additionally, along with the optimization and improvement of relevant compilation strategies in the future, the scalability of the instruction group library also can provide a helpful guidance for the reconfigurable implementation of domain oriented applications with better performance.

REFERENCES

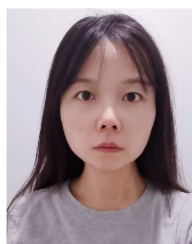
[1] Y. Niu, R. Kannan, A. Srivastava, and V. Prasanna, "Reuse kernels or activations? A flexible dataflow for low-latency spectral CNN acceleration," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 2020, pp. 266–276.

[2] Y. Zhang, N. Zhang, T. Zhao, M. Vilim, M. Shahbaz, and K. Olukotun, "SARA: Scaling a reconfigurable dataflow accelerator," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 1041–1054.

[3] C. Tan, C. Xie, T. Geng, A. Marquez, A. Tumeo, K. Barker, and A. Li, "ARENA: Asynchronous reconfigurable accelerator ring to enable data-centric parallel computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 12, pp. 2880–2892, Dec. 2021.

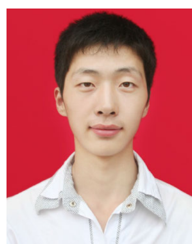
[4] M. Brand, F. Hannig, O. Keszocze, and J. Teich, "Precision- and accuracy-reconfigurable processor architectures—An overview," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 6, pp. 2661–2666, Jun. 2022.
[5] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surveys*, vol. 52, no. 6, pp. 1–39, Nov. 2020.
[6] Y. Lu, L. Liu, J. Zhu, S. Yin, and S. Wei, "Architecture, challenges and applications of dynamic reconfigurable computing," *J. Semiconductors*, vol. 41, no. 2, 2020, Art. no. 021401.
[7] G. Brignone, M. U. Jamal, M. T. Lazarescu, and L. Lavagno, "Array-specific dataflow caches for high-level synthesis of memory-intensive algorithms on FPGAs," *IEEE Access*, vol. 10, pp. 118858–118877, 2022.
[8] A. Ohwada, T. Kojima, and H. Amano, "An efficient compilation of coarse-grained reconfigurable architectures utilizing pre-optimized subgraph mappings," in *Proc. 30th Euromicro Int. Conf. Parallel, Distrib. Network-Based Process. (PDP)*, Mar. 2022, pp. 1–9.
[9] Y. Xiao, E. Micallef, A. Butt, M. Hofmann, M. Alston, M. Goldsmith, A. Merczynski-Hait, and A. DeHon, "PLD: Fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Feb. 2022, pp. 933–945.
[10] M. Weinhardt, M. Messelka, and P. Käsgen, "CHiPReP—A compiler for the HiPReP high-performance reconfigurable processor," *Electronics*, vol. 10, no. 21, p. 2590, Oct. 2021.
[11] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 2019, pp. 242–251.
[12] R. Zhao, S. Liu, H.-C. Ng, E. Wang, J. J. Davis, X. Niu, X. Wang, H. Shi, G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Hardware compilation of deep neural networks: An overview," in *Proc. IEEE 29th Int. Conf. Application-Specific Syst., Architectures Processors (ASAP)*, Jul. 2018, pp. 1–8.
[13] N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Improving performance and energy consumption in embedded systems via binary acceleration: A survey," *ACM Comput. Surveys*, vol. 53, no. 1, pp. 1–36, Jan. 2021.
[14] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4D-CGRA: Introducing branch dimension to spatio-temporal application mapping on CGRAs," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8.
[15] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh, "Chameleon: Adaptive code optimization for expedited deep neural network compilation," in *Proc. Int. Conf. Learn. Represent.*, Apr. 2020, pp. 1–17.
[16] D. Park, Y. Xiao, N. Magnezi, and A. DeHon, "Case for fast FPGA compilation using partial reconfiguration," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 2350–2353.
[17] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 898–911, May 2019.
[18] J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.
[19] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Automatic compilation of diverse CNNs onto high-performance FPGA accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 2, pp. 424–437, Feb. 2020.
[20] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-based overlay processor for convolutional neural networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 1, pp. 35–47, Jan. 2020.
[21] J. Wang, L. Guo, and J. Cong, "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 2021, pp. 93–104.
[22] S.-H. Chung and T. S. Abdelrahman, "A compilation flow for the generation of CNN inference accelerators on FPGAs," 2022, *arXiv:2203.04015*.
[23] E. Mutlu, R. Tian, B. Ren, S. Krishnamoorthy, R. Gioiosa, J. Pienaar, and G. Kestor, "COMET: A domain-specific compilation of high-performance computational chemistry," in *Proc. Int. Workshop Lang. Compil. Parallel Comput.*, Feb. 2022, pp. 87–103.
[24] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2004, pp. 75–86.
[25] X. Lin, S. Yin, F. Tu, L. Liu, X. Li, and S. Wei, "LCP: A layer clusters paralleling mapping method for accelerating inception and residual networks on FPGA," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.
[26] L. Yann, C. Cortes, and C. J. C. Burges. *The MNIST Database Of Handwritten Digits*. [Online]. Available: http://yann.lecun.com/exdb/mnist/

YUANCHENG LI is currently pursuing the Ph.D. degree with the Xi'an University of Science and Technology, Xi'an, China. His current research interests include reconfigurable compilation and artificial intelligence.



NA WANG is currently pursuing the master's degree with the Xi'an University of Science and Technology, Xi'an, China. Her current research interest includes reconfigurable computing.

MINGWEI SHENG is currently pursuing the master's degree with the Xi'an University of Science and Technology, Xi'an, China. His current research interest includes reconfigurable computing.



JIAQI SHI is currently pursuing the degree with the Xi'an University of Science and Technology, Xi'an, China. His current research interest includes reconfigurable computing.