# A RISC-V Extended Infrastructure for CNNs through Pipelined Computing and Data Dependence Optimization

Teng Luo*, Tengfei Xia*, Jiayuan Chen†, Zhihua Fan†, Wenming Li, Yudong Mu, Xuejun An, Xiaochun Ye, Dongrui Fan

***Abstract*—With the rapid development of artificial intelligence (AI), convolutional neural networks (CNNs) have been widely applied in fields like computer vision and recommendation systems. This growth has intensified the demand for hardware acceleration of CNNs. Existing accelerators are either designed as co-processors or improve performance through extended instructions. While these methods can significantly improve performance, they often result in limited programming and execution flexibility. In this paper, we design custom RISC-V instructions specifically for CNNs to maximize data reuse and exploit parallelism. Then, to efficiently execute CNNs instructions, we extend a Pipelined Vector Computing Unit (PPVCU). Finally, we incorporate Pattern Detection Logic (PDL) to identify common data dependence patterns in CNNs, enabling the Data Dependence Computing Unit (DDCU) to process instructions within each pattern in parallel. Experimental results show that our approach achieves, on average, 9.54× performance improvement and 6.7× energy efficiency improvement compared to our baseline, 8.34× performance improvement and 3.1× energy efficiency improvement compared to state-of-the-art designs.**

***Index Terms*—CNNs acceleration, RISC-V extended instructions, pipelined computing, dataflow optimization.**

## I. Introduction

Convolutional Neural networks (CNNs) represent a pivotal class of artificial intelligence models primarily focused on convolutional computation. Their architecture mimics the way the human brain processes information, allowing for hierarchical feature extraction. Over recent years, these models have gained immense popularity in various applications, including image processing, computer vision, and pattern recognition, where they have consistently demonstrated superior efficacy compared to traditional methods. However, the inference process of CNNs is resource-intensive, requiring substantial computational power and storage capacity to handle complex calculations and large model sizes. To address these challenges, numerous specialized accelerators have been developed, including GPUs [1], [2], TPUs [3], [4] and so on. These accelerators optimize CNNs performance by fully exploiting parallelism of CNNs, thereby enabling more efficient processing while maintaining the accuracy required for critical applications.

CNN accelerators can broadly be categorized into two main types. The first category includes Application-Specific Integrated Circuits (ASICs) [5], [6] designed specifically for CNN tasks. These ASICs typically operate as external co-processors to the host system, enhancing computational efficiency and speed by executing CNN operations in parallel. The second approach involves leveraging general-purpose processors by extending their instruction sets and incorporating specialized hardware. This strategy enables tightly integrated acceleration, allowing the processors to efficiently manage CNN workloads while maintaining their general-purpose functionality [7]–[9].

Although ASICs offer significant performance and energy efficiency improvements, rapid evolution of AI algorithms often outpaces their adaptability. Additionally, various accelerators typically come with distinct programming models and interfaces, challenging the creation of a unified ecosystem. Current researches focus on achieving substantial performance gains while retaining the generality of processors. The emergence of the RISC-V ISA presents a promising solution, as its open-source, flexible, and extensible nature facilitates the customization of instructions tailored for CNNs, all while maintaining the generality of the processor. Extensive researches have been conducted to accelerate CNNs with RISC-V extended instructions [10]–[12]. This paper explores the RISC-V extension technique and designs an accelerator aimed at achieving high execution efficiency in accelerating various components of CNNs, without sacrificing generality. In detail, our work includes two key techniques:

First, we design RISC-V extended instructions to

Teng Luo* and Tengfei Xia* contribute equally to this paper.

Teng Luo, Tengfei Xia, Zhihua Fan, Wenming Li, Yudong Mu, Xuejun An, Xiaochun Ye and Dongrui Fan are with the SKLP, Institute of Computing Technology, Chinese Academy of Sciences and also with the School of Computer Science, University of Chinese Academy of Sciences, Beijing, China. Jiayuan Chen is with China Mobile Research Institute.

† Corresponding authors: Zhihua Fan (fanzhihua@ict.ac.cn), Jiayuan Chen.

accelerate CNNs, combined with a Pipelined Vector Computation Unit (PPVCU). These extended instructions are precisely formatted to comply with the RISC-V ISA and are integrated into existing networks using inline assembly, enabling seamless incorporation. The extended instructions effectively exploit data reuse and computational parallelism intrinsic to CNNs. The PPVCU, responsible for executing these instructions, is integrated into the base processor's pipeline and operates similarly to a conventional ALU.

Second, we propose a method for identifying and accelerating data dependencies between instructions. This approach analyzes specific data dependency patterns and, together with the Data Dependence Computing Unit (DDCU), enables the entire data dependency sequence to execute without internal stalls. In conclusion, the main contributions of this paper are as follows:

- We propose a set of RISC-V instruction sets for efficient CNN inference.
- We design a CNN accelerator which highlights pipelined execution for multi-level operations and dynamic detection of data dependence.
- We created PPVCU, an approach that utilizes multi-level pipelines and parallelism to improve CNN performance.
- We introduce DDCU, a method that dynamically recognizes data dependency patterns between instructions and accelerates them.
- Experiment results show that, compared with the state-of-the-art works, our approach achieves significant performance and energy efficiency improvement and preserves the generality.

## II. Background and Related Work

### A. CNNs and Related Accelerators

CNNs typically consist of alternating convolutional layers, pooling layers, and activation functions. Convolutional operations, which involve numerous matrix multiplications and additions, constitute the majority of the computational burden of CNNs inference. In detail, the convolutional layer takes in $C$ input feature maps and applies $M$ three-dimensional filters to produce $M$ output feature maps. Each three-dimensional filter operates across all $C$ input feature maps to generate a single output feature map. Table I shows the parameter description of the convolutional layer operation. The calculation of convolutional operations is formulated as shown in Equation 1:

$$A^{l+1}[z][u][x][y] = \sum_{k=0}^{C-1} \sum_{i=0}^{S-1} \sum_{j=0}^{R-1} A^{l}[z][k][Ux+i][Uy+j] \times W^{l}[u][k][i][j] \quad (1)$$

$$0 \le z \le N,\ 0 \le u \le M,\ 0 \le x \le Q,\ 0 \le y \le P$$

Where $A^{l} \in \mathbb{R}^{NCXY}$, $W^{l} \in \mathbb{R}^{MCRS}$ and $A^{l+1} \in \mathbb{R}^{NMPQ}$ is the input of the {l+1}th layer, the filter between the {l}th layer and the {l+1}th layer and the output of the {l+1}th layer, respectively.

TABLE I: Convolutional layer parameter description

| Para | Description | Para | Description |
|---|---|---|---|
| N | Batch size of 3D input | U | Stride size |
| C | Input/Filter channels | X/Y | Input height/width |
| R/S | Filter height/width | P/Q | Output height/width |
| M | Filter numbers/Output channels | - | - |

Currently, widely adopted convolution acceleration algorithms in deep learning frameworks include Winograd algorithm [16], FFT (Fast Fourier Transform) [14] and im2col [15]. These algorithms primarily focus on enhancing convolution efficiency within the software layer, offering improvements in computational speed by restructuring or transforming convolution operations. However, purely software-based acceleration remains limited, especially for large models and real-time inference, due to memory bandwidth and parallelism constraints. To overcome these limitations, there has been an increasing focus on custom hardware accelerators. For example, TPUs [3], [4], FPGAs [17], [18], and ASICs [5], [6], [19], [20] have been designed to substantially improve the efficiency of CNNs inference through highly optimized architectures. A representative example is the Gemmini generator [21], which offers a flexible architectural template enabling the design of efficient ASIC accelerators across a wide design space.

However, these custom hardware accelerators come with several limitations. One of the major challenges is compatibility. Since these accelerators are designed with fixed functionality, it is difficult for them to adapt to the rapid evolution of AI algorithms. ASICs, in particular, lack flexibility and require significant redesigns or new hardware to support emerging models and frameworks. Another significant limitation is the lack of a unified ecosystem. Custom accelerators often depend on specific frameworks and come with unique programming models and interfaces [22]. This leads to fragmentation in the development process, requiring developers to modify their code to suit different hardware architectures. Such fragmentation not only complicates development but also hinders portability across platforms, making it more challenging to build a seamless, unified AI development environment. Finally, the cost of custom accelerators remains a major barrier. Designing and fabricating ASICs involves substantial investment, which is inaccessible for smaller companies or research groups. Although FPGAs offer some reconfigurability, they remain expensive and still lack the flexibility of general-purpose processors in efficiently handling diverse AI tasks. Despite the performance gains, these limitations in cost, compatibility, and ecosystem support can slow the widespread adoption of custom hardware.

TABLE II: RISC-V opcode map

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

Current research increasingly focuses on balancing both generality and performance in hardware accelerators. RISC-V stands out as a promising solution due to its open-source nature, enabling broad participation and development [23], [24]. Its extensible instruction set is adaptable to evolving AI workload demands,while its modular and flexible architecture of RISC-V enables efficient customizations while maintaining general-purpose functionality. As the RISC-V ecosystem continues to grow, it offers enhanced support for AI applications, from machine learning inference to deep neural networks. Given these strengths, this paper opts to build upon the RISC-V platform, extending its capabilities for AI acceleration to achieve high performance and flexibility.

### *B. CNN Accelerators Based on RISC-V Extension*

RISC-V ISA follows the design principle of modularity, involving a basic instruction subset (*I* instruction subset) which includes addition, subtraction, shift and logic operations. Apart from the necessary *I* instruction subset, several optional standard extended instruction subsets are proposed. For example, *M* instruction subset stands for integer multiplication and division instructions. *C* instruction subset provides compressed instructions with the length of 16 bits. If a 32-bit RISC-V ISA supports the subset of *I*, *M* and *C*, it is conventionally named as *RV32IMC* [25].

In addition to standard extended instruction subsets, the RISC-V ISA is highly convenient for extending new instructions. It reserves four opcodes for users to define additional custom instructions in the opcode map, as shown in Table II, which enables users to select suitable opcodes for custom instruction set design. Researchers have designed various custom instructions to achieve computational acceleration in specific domains such as information security [26], [27], digital signal processing [28], [29], high-performance computing [30], and artificial intelligence [31]–[33]. Most of the related work has been carried out based on RISC-V general-purpose processors. Our baseline is the open-source CVA6 [34], a 6-stage, single-issue, in-order CPU that fully supports I, M, A, and C extensions, along with three privilege levels (M, S, U) for Unix-like operating systems. This architecture is designed to be versatile and robust, suitable for various applications.

In recent years, numerous works utilizing RISC-V extended instructions for CNNs acceleration have also been proposed. Except for methods targeting low-precision data [10], [11], the majority of the efforts have focused on directly improving the computational speed of CNNs. Both Yu [7] and Li [8] achieve convolution acceleration by designing multiply-accumulate (MAC) instructions. Other works [35]–[37] focus on the architecture design of coprocessors and corresponding optimizations in memory access to accelerate CNNs. The extended instructions they design primarily serve as activation control signals for the accelerators, but they do not flow through the CPU pipeline. To optimize convolution, activation, and pooling operations in CNNs inference, Wang et al. [9] propose seven SIMD extension instructions. Their main contribution includes the CONV23 instruction, which accelerates computation speed using the Winograd algorithm.

Our design aims to improve computational efficiency by focusing on computation operations rather than data precision. Overall, similar acceleration strategies can be divided into the following two categories:

The first approach involves using external coprocessors, where extended instructions serve as initiation signals for performing complex operations (e.g., convolution of an image block). The drawback of this method lies in the complexity of the operations that a single instruction must handle, often requiring a stack of computational components and intricate control logic for the coprocessor. This approach does not fully exploit the potential of extended instructions. Furthermore, complex instructions can result in a decrease in clock frequency. In contrast, our design uses multiple computation and memory access instructions to collaboratively perform complex convolution operations. Although these instructions are more complex than regular ones, they can flow through the pipeline like other instructions and be executed in a pipelined manner, improving computational efficiency without reducing clock frequency and significantly increasing the number of computing units.

The second acceleration strategy employs a tightly-coupled approach, integrating corresponding decoding logic and computational units into the processor's existing pipeline. A common practice is to extend multiply-accumulate instructions for sliding-window convolution computations, enhancing computational efficiency through parallelized multiplication and combined multiply-add operations. However, compared to methods using im2col or the Winograd algorithm, this approach is significantly less effective. The latter two methods are generally more efficient in practical computational scenarios. In our design, we leverage the Winograd convolution algorithm and specifically design extended instructions and acceleration strategies to further enhance computational performance, achieving more practical and effective acceleration.

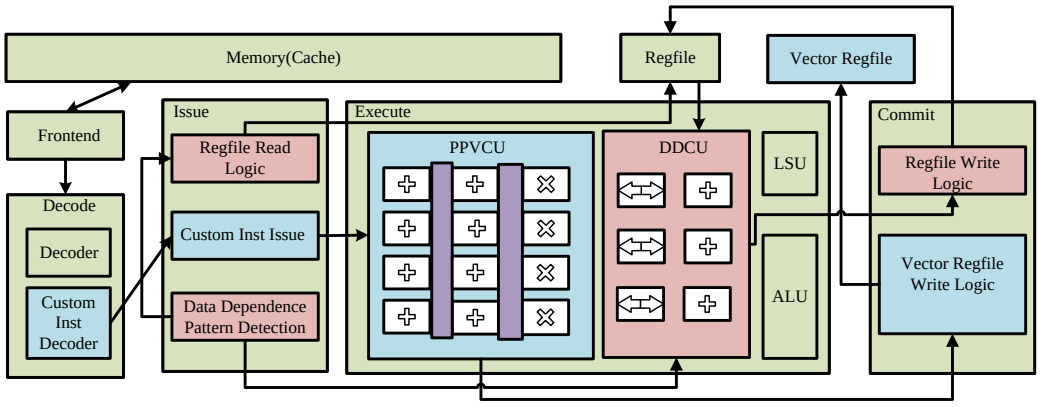


Fig. 1: Extended architecture of CVA6 RISC-V processor

## III. Our Architecture

This section introduces our extended computational components and control logic based on the original architecture, implementing optimized designs for extended instructions and data dependence handling, as illustrated in Figure 1. The key features of our architecture are as follows:

1) **CNNs extension instructions**: To enhance the computational performance of CNNs convolution and max-pooling operations, we propose an acceleration scheme based on RISC-V CNN extension instructions. This scheme includes the design of the encoding and semantics of the extension instructions, as well as an acceleration algorithm implemented using these instructions, which can fully leverage the data parallelism in convolution and max-pooling computations.
2) **Pipelined computing unit**: To optimize the execution efficiency of extended multi-level instructions, we design a Pipelined Computing Unit (PPVCU). This computing unit features three computing components arranged sequentially in a pipeline, enabling the pipelined execution of extended instructions with multi-level operations, effectively hiding their execution latency.
3) **Data dependence computing unit**: To efficiently execute data dependence patterns in general program sections and CNN activation functions, we propose a solution that automatically identifies data dependence patterns and executes them in parallel. This solution includes the design of a Pattern Detection Logic (PDL) and a Data Dependence Computing Unit (DDCU), which work together to eliminate stalls within the data dependence patterns.

### A. RISC-V Custom Instruction Extensions

*1) Encoding and Execution of Custom Instructions:* We choose opcode 0x0b to implement custom instructions, and employ funct7 to distinguish each instruction. The vector register operands used by these instructions have a width of $4 \times 32$ bits. The encoding of all custom instructions is summarized in Table III. Next, we will detail the specific meanings of each instruction.

**Data Transfer Instructions** include LDTILE and WBTILE. These instructions is used to enable data exchange between the main memory and vector registers, allowing the use of vector registers as operands for computation. Compared to standard memory access instructions, these instructions access wider memory regions using custom vector registers as source or destination. To handle large, contiguous data unsuitable for Dcache, we extend the CVA6 LSU (Load Store Unit) with burst transfer mechanisms and bypass Dcache for read/write operations, specifically designed for these instructions.

LDTILE read data from memory and store them into vector registers. Operand rs1 represents base address in memory, funct3 indicates the data size and rd indicates the first destination vector register number.

WBTILE stores a vector from the source vector register (rd) into memory at the address specified by rs1. The equivalent expression is $vreg[rd]\text{->}mem[rs1]$.

**Data Calculation Instructions** include AAMUL, TRIADD, OACC and MAXF. The first three accelerate convolution and are executed within the PPVCU, as detailed in the Table IV, while MAXF accelerates max-pooling operations, as detailed below.

The equivalent expression for instruction AAMUL is $cal(vreg[src0] + vreg[src1]) \times vreg[src2]\text{->}vreg[dest]$: First, vector addition is performed in the first stage to obtain a four-element vector $x$. Then, the 'cal' operation (which involves four additions: $x[0] - x[2]$, $x[1] + x[2]$, $-x[1] + x[2]$, $x[1] - x[3]$) is performed in the next stage. Finally, vector multiplication is executed in the third stage.

The equivalent expression for instruction TRIADD is $vreg[src0] + vreg[src1] + vreg[src2]\text{->}vreg[dest]$, where two vector additions are performed in the first and second stages, respectively.

OACC produces only two valid output elements: First, 4 additions between four elements from src1 and two elements from src0 are performed in the first stage. Then, in the second stage, the first and second elements of res1 are added together, as well as the third and fourth elements.

TABLE III: The encoding of custom instructions

| Instruction | Funct7[31:25] | Rs2[24:20] | Rs1[19:15] | Funct3[14:12] | Rd[11:7] | Opcode[6:0] |
|---|---|---|---|---|---|---|
| LDTILE | 0 | / | addr. of input | size of input | vector dest | 0x0b |
| AAMUL | 1 | vector src1 | vector src0 | dest indicator | vector src2 | 0x0b |
| TRIADD | 2 | vector src1 | vector src0 | dest indicator | vector src2 | 0x0b |
| OACC | 3 | vector src1 | vector src0 | dest indicator | / | 0x0b |
| WBTILE | 4 | vector src | addr. of ofmap | / | / | 0x0b |
| MAXF | 5 | vector src1 | vector src0 | index | vector dest | 0x0b |

The write-back addresses of these three instructions are jointly determined by rs1 and funct3, specifically as waddr = (funct3 = 0)? rs1 : funct3. This encoding design aims to flexibly encode the target vector registers within the constraints of limited encoding bits.

TABLE IV: Operations of AAMUL, TRIADD and OACC

| Instruction | 1st Level | 2nd Level | 3rd Level |
|---|---|---|---|
| AAMUL | src0+src1 | 4 adds within res1 | res2×src2 |
| TRIADD | src0+src1 | res1+src2 | / |
| OACC | src0+src1* | 2 adds within res1 | / |

* 4 additions between 4 elements from src1 and 2 elements from src0 (not typical vector addition)

MAXF compares four values across two adjacent vector registers, where the first bit of funct3 selects the source data index, as specified in Table V, while the last two bits determine the output position in the destination register, represented as $max(sourceData)->vreg[rd]_{funct3[1:0]}$. Three comparisons are needed: two to identify the larger value in each pair and one to find the maximum of these. The entire operation completes in one cycle.

TABLE V: The usage of the first bit of funct3 in MAXF

| funct3[2] | data0 | data1 | data2 | data3 |
|---|---|---|---|---|
| 0x0 | $vreg[rs1]_0$ | $vreg[rs1]_1$ | $vreg[rs2]_0$ | $vreg[rs2]_1$ |
| 0x1 | $vreg[rs1]_2$ | $vreg[rs1]_3$ | $vreg[rs2]_2$ | $vreg[rs2]_3$ |

To support extended instructions, we modify RISC-V SDK toolchain to allow the compiler to recognize them. Building on this, we write inline assembly functions for these instructions and design corresponding acceleration strategies using the functions.

*2) Convolution Acceleration Strategy:* We implement the Winograd algorithm with designed extended instructions to accelerate convolution operation.

The two-dimensional Winograd convolution $F(2\times 2, 3\times 3)$ completes the operation of convolving a $4\times 4$ input matrix with a $3\times 3$ convolution kernel to obtain a $2\times 2$ output matrix. Let $Y$ be the $2\times 2$ two-dimensional convolution output matrix, $d$ be the $4\times 4$ input matrix, and $g$ be the $3\times 3$ convolution kernel, then define several auxiliary matrices:

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \tag{2}$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{3}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Then the two-dimensional Winograd convolution can be computed using the following equation:

$$Y = A^T \left[ \left( GgG^T \right) \odot \left( B^T dB \right) \right] A \tag{5}$$

In the above equation, $GgG^T$ is typically precomputed before the main loop. So equation 5 can be further expressed using equation 6, where $U$ represents the transformed convolution kernel matrix.

$$Y = A^T \left[ U \odot \left( B^T dB \right) \right] A \tag{6}$$

Figure 2 (b) shows the common implementation (for origin CVA6), while Figure 2 (c) presents our implementation of the Winograd convolution algorithm. In Figure 2 (c), memory access instructions are marked in blue (LDTILE $addr, size, dest$ and WBTILE $src, addr$), while computation instructions are highlighted in red, with the destination register as the last operand. Our strategy accelerates the process by optimizing both memory access and computation through a blend of software and hardware design.

For memory access, the input of $F(2\times 2, 3\times 3)$ includes 16 rearranged input matrix elements and 16 transformed convolution weights, totaling 128Bytes. In CVA6, memory access instructions make requests to the Dcache. In CVA6, the Dcache line size is 16Bytes, which leads to frequent cache block replacements due to lack of data reuse, requiring four burst transfers per loop iteration.

To optimize this, we utilize instruction LDTILE for burst transfers, storing data directly into expanded vector registers. This enables completing the data access for the input matrix in a single burst transfer per iteration, eliminating cache block replacements and reducing latency. Additionally, we implement prefetching by explicitly calling LDTILE to load the next iteration's data before the current iteration's computation, overlapping memory access and computation latencies.

As for the computation part, $F(2\times 2, 3\times 3)$ involves matrix computations with high data parallelism, but scalar instructions and limitations of computation units of CVA6 prevent full parallel execution. For example, one iteration requires 56 additions, with only one ALU in CVA6, these instructions cannot be effectively executed in parallel.

To exploit data parallelism, we use expanded vector

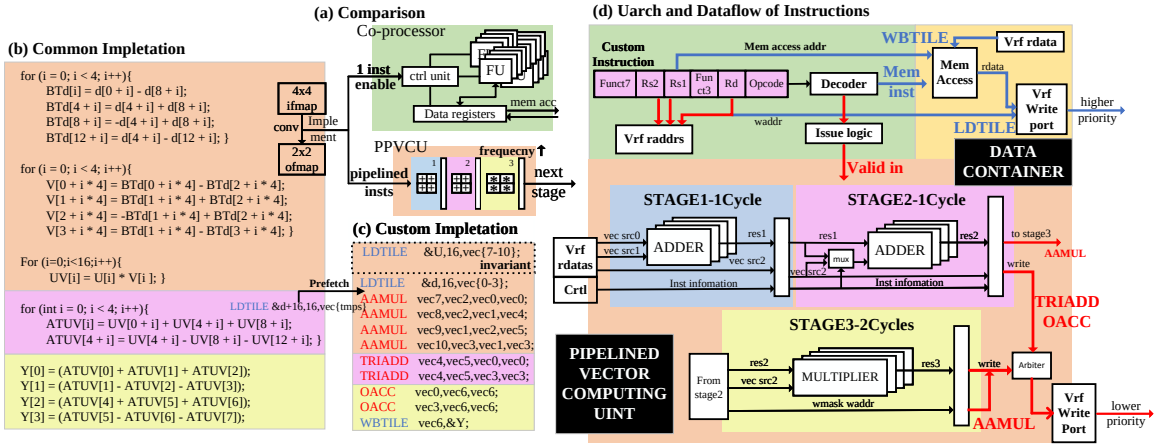


Fig. 2: Convolution accleration strategy and micro-architecture of PPVCU

registers and computation units. In one iteration, a LDTILE instruction loads the input matrix and kernel into vector registers, followed by 4 AAMUL, 2 TRIADD, and 2 OACC instructions for computation. Finally, the WBTILE instruction writes the results back to memory. All these instructions use 4-element vectors as operands, which aligns with the source program's structure. In addition, these instructions combine operations with data dependencies, effectively reducing the instruction count. For instructions with data dependencies, we adjust the instruction sequence to hide the latency, enabling them to be executed back-to-back in a pipelined manner in the PPVCU, achieving efficient acceleration.

*3) Max Pooling Acceleration Strategy:* Max pooling operation outputs the maximum number from a specified matrix. We optimize the design for the $2 \times 2$ size case using instruction MAXF.

As shown in Figure 3 (a), the original max pooling layer calculates the maximum value of a $2 \times 2$ matrix by traversing its 4 elements, performing boundary checks, and comparing values. Each boundary check requires 4 branch predictions, leading to 16 predictions per output. This variability in input sizes challenges branch predictors, and mispredictions significantly degrade pipeline performance.

In fact, when there is no padding in max pooling layer, every index corresponds to a value in memory, these boundary checks can be omitted and the original loop-based computation method can be replaced by a vectorized computation approach. Therefore, we develope a corresponding computation strategy based on the LDTILE, WBTILE, and MAXF instructions.

Our acceleration strategy is shown in Figure 3 (b). First, two LDTILE instructions load 8 elements from adjacent two rows into vector registers. MAXF is then used to perform max pooling on the two $2 \times 2$ matrices, with results written to the designated vector register. Meanwhile, the next set of elements is loaded with LDTILE. This process continues until the pooling layer is complete. When the vector register is full, WBTILE writes the results back to memory.

For an $8 \times 8$ input matrix and $2 \times 2$ pooling size, the

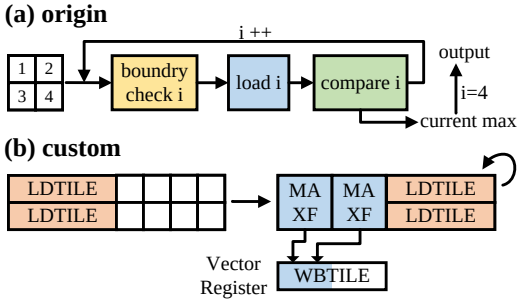


Fig. 3: Max pooling acceleration strategy

original implementation requires 64 load, 16 store, and 48 comparison operations. Even without considering the performance loss due to branching, the latency is significant. In contrast, our strategy uses only 8 LDTILE, 4 WBTILE, and 16 MAXF instructions. The MAXF instruction completes in one cycle, and the latency of LDTILE can be hidden by adjusting their sequence. This approach improves performance through vectorized computation and optimized instruction sequencing.

## B. Pipelined Vector Computing Unit

The extended instructions designed are not executed in an external accelerator connected to the CPU. Instead, they can flow through the stages of the pipeline like other instructions in the instruction set. Comparing our design to using a single complex instruction to start the co-processor and execute more intricate operations (such as conducting convolution operations on matrix blocks), the advantage lies in the facilitation of pipelined execution for instructions.

As shown in Figure 2 (a), coprocessors typically require a large number of computational units, and the utilization of these units and the complexity of control logic are mutually constraining factors. Moreover, complex instructions often have a significant impact on clock frequency, which poses a bottleneck for acceleration performance. The internal structure of the designed pipelined vector computing unit (PPVCU) is shown in Figure 2 (d). As shown in the figure, PPVCU is a three-level pipeline computational unit, where each

level is: 4 addition components, 4 addition components and 4 multiplication components. The delays of the first two stages of the computational unit are each 1 cycle, while the delay of the final stage is 2 cycles. Each stage employs interlocking control strategies to achieve pipelining. Additionally, the multiplication component in the final stage is internally pipelined. Therefore, theoretically, PPVCU can simultaneously process 4 instructions, achieving complete pipelining.

Due to the use of custom vector registers as source operands in extended instructions, additional issue checking are added to the issuing module: assigning a 1-bit flag for each vector register to indicate whether there are unfinished instructions in the computing unit that need to write back to that vector register (*"1"* indicates yes, *"0"* indicates no). Before issuing extended instructions, this table is checked to determine operand readiness. Once an instruction is ready for issue, the corresponding flag in the table (and the valid signal of PPVCU) is set to *"1"*, and it is reset to *"0"* once the instruction completes.

Similar to other existing computational units such as ALU and FPU, PPVCU is tightly coupled and integrated into the pipeline. It interacts with execution pipeline stages through handshake signals. When the corresponding valid signal is asserted (valid = 1), the first stage of PPVCU's pipeline component is initiated. Control signals carried by instructions govern their movement through the pipelined computational unit. Upon completion of the instruction, PPVCU sets its external valid signal high, indicating that an instruction is ready for writeback. The tightly coupled connection allows PPVCU to function similarly to other computational components, allowing instructions from different computational units (such as regular ALU instructions and extended instructions) to execute in parallel.

In Figure 2 (d), data path of instruction AAMUL, TRIADD and OACC within the PPVCU is illustrated using red lines. As shown in the figure, AAMUL passes through all three stages, while TRIADD and OACC only passes through the first two stages and do not enter the third stage. When AAMUL and TRIADD/OACC simultaneously reach their respective final execution stages, there will be contention for write-back resources. Our strategy to address this is to prioritize the write-back of the AAMUL. This prioritization is based on the fact that AAMUL generally appears earlier in the source program, at the top of the data dependence chain before the corresponding TRIADD and OACC. Prioritizing the completion of AAMUL facilitates the efficient activation of more data-dependent instructions.

To achieve vectorized computation, a vector register file (vrf) has been extended. The vector register file has 3 read ports and 1 write port, and includes 15 vector registers, each of which has a width of 128 bits and can store 4 32-bit data. Data from the memory is fetched and stored in a portion of these vector registers. When computation instructions are called, the data in these registers is sent as source operands to the pipelined vector computing unit for corresponding calculations. The results obtained are then written back to specific vector registers. The design of vector registers and vector computing unit makes vectorized computation possible, which aligns well with the characteristics of convolution calculations. Additionally, these registers can also flexibly serve as buffers, reducing unnecessary memory access operations.

### C. Data Dependence Optimization

In the general program sections of CNNs, there exists numerous read-after-write data dependence patterns between multiple consecutive RISC-V instructions. For example, as illustrated in Figure 4, the second instruction addw uses a4 register as source register while the first instruction mulw yields it. The data dependence relationship among the following instructions is the same. As a result, the latter instruction must wait for the operation of the previous instruction to be completed before it executes. In the semantic, this set of instructions is just a collection of multiply, addition and shift operations which takes the blue a5,a6 and a7 registers as the real source registers and the green a4 register as the real destination register, while a4 registers in the red are only for temporary data storage. Data dependence patterns like this are yielded due to the defect of the open-source compilers and undue simplicity of RISC-V ISA, which cannot be solved by classic instruction extension, since it is quite difficult to find the corresponding location of one instruction section in the source program. For this reason, we are trying to optimize the executing procress by hardware extension which dynamically looks like instruction extension technique.

To solve the data dependence issue, the concept of tight dependence relationship is defined by us as below: If the destination register of the *n*th instruction is one of the source registers as well as the destination register of the $(n+1)th$ instruction, there exists tight dependence relationship between these two instructions. Based on this concept, three data dependence patterns is sampled and abstracted from the general program sections and activation functions:

*1) Basic Optimization Pattern:* In this pattern, each pair of continuous instructions conforms to tight dependence relationship, as the example instruction section in Figure 4.In fact, computation result of the first mulw instruction is yielded in Execute stage, and the second instruction addw is supposed to execute immediately. However, the second instruction addw is still in Issue stage and the subsequent instructions are in the earlier stages. They do not have the access to the ALU and can only be executed until entering Execute stage. As a result, we compare the register numbers of the instruction in Execute stage with those in the earlier stages, and judge the tight dependence

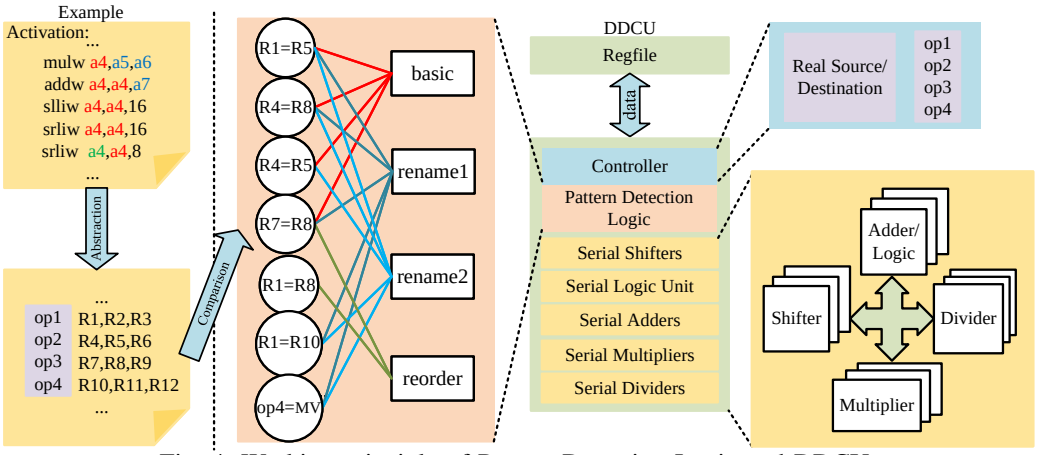


Fig. 4: Working principle of Pattern Detection Logic and DDCU.

relationship between them using Pattern Detection Logic (PDL). If the tight dependence relationship does hold, execution information of the instructions(like the real source or destination registers and the concrete operations) will be forwarded to the controller of the Data Dependence Computing Unit (DDCU) in the Execute stage immediately. The forwarding process is similar to the plain data forwarding in the pipeline, with minimal timing impact and resource cost. Then, under the guidance of the controller, DDCU will complete the operations of all instructions included in this pattern within the shortest possible time. In order to avoid repeated calculations, the instructions which are forwarded must be removed from the pipeline, making room for the following instructions. Based on this, these five instructions in the example take 6 cycles from entering to exiting the pipeline, compared to 10 cycles needed before.

*2) Rename Optimization Pattern:* Rename technique is usually referred to as an effective way to solve write-after-read and write-after-write data dependence in the out-of-order processors [39], while in rename optimization pattern mentioned here, some registers in the instruction sections can be renamed to become the basic optimization pattern. There are two patterns that can be renamed to the basic optimization pattern which are shown in Figure 4 with purple and brown lines. In both cases, register R1/R4/R5/R8 is only used to store the middle results of the instructions and the value of register R1 will be reloaded after the pattern, so renaming register numbers to construct basic optimization pattern will not influence the correctness of the program. In the fact, we do not need to explicitly rename the register number in hardware as out-of-order processors commonly do. After recognizing rename optimization pattern, a tag corresponding to the pattern is used to indicate the real source and destination registers. Then, value of the real source register will be sent to DDCU from regfile and the computing result will be directly written back to the real destination register. This completes the implicit pattern conversion and computation. The timing and hardware overhead of pattern recognition and conversion is minimal and acceptable.

*3) Reorder Optimization Pattern:* Reordering instructions is a common optimization method of compilers [40], while here it refers to using hardware to adjust the order of instructions, thus converting some instruction sections into basic optimization pattern. The detection logic of the reordering pattern is shown in Figure 4 with yellow lines, the first instruction in the sequence does not have the tight dependence relationship with the second instruction, but it does have the tight dependence relationship with the third instruction. Moreover, the third instruction does not depend on the second. Therefore, it is meaningful to adjust the order of the second and the third instruction that constructs the basic optimization pattern. The actual conversion into the basic optimization mode is similar to the rename pattern, both of which are implicit processes.

*4) Pattern Detection Logic (PDL):* This module is designed to recognize the optimization patterns. It compares the register numbers of the instruction in the Execute stage with those in the earlier stages using bypass technique. The detection logic is in the line with the description of the three optimization patterns mentioned above and also shown in the Figure 4. To make it clear, conditions are connected with the corresponding optimization pattern using colored lines. For example, if R1==R5 and R4==R5, tight dependence relationship does hold between the first and the second instruction. Similarly, there is also tight dependence relationship between the second and the third instruction if R4==R8 and R7==R8. As a result, pattern detection logic will recognize the data dependence pattern as the basic optimization pattern.

To ensure the correctness of the PDL, the conditions for each pattern are mutually exclusive which means an instruction section will never be recognized as two optimization patterns. Moreover, when it comes to branch instruction, whether it will branch and where it will jump to are highly uncertain. Therefore, when the detection logic finds that a branch instruction and its following instructions conform to data dependence patterns, no optimization pattern will be recognized. It should be noted that in the RISC-V ISA, a branch instruction is actually the combination of one subtraction

and one write to the program counter. When a branch instruction is the last instruction in a program section that meets the optimization patterns, data dependence optimization still applies.

*5) Data Dependence Computing Unit (DDCU):* The vast majority of the optimization patterns are yielded due to the defect of RISC-V open-source compiler and the undue simplicity of RISC-V ISA. That's why the calculation process of optimization patterns are relatively fixed as the collection of addition/subtraction and shift operations. In addition, multiplication-accumulation and branch are also frequently observed in data dependence patterns. The specific operations are abstracted as op1-op2-op3-op4, as illustrated in Figure 4. The controller of DDCU is responsible for getting the operation sequencce and controlling the computing process. To meet the computing need, the main body of DDCU are adders, shifters and multipliers. It can complete the tasks of shift-shift-shift, shift-shift-add, shift-add, shift-add-add, add-add-add, add-branch in only one cycle and multiply-add, multiply-add-add in two cycles. For example, shift-shift-add operation is one of the most common operations in basic optimization pattern. After the controller gets operation sequence and real source/destination registers, two operands of the first shift operation enter the shifter1, after that, the result comes out and enters shifter2 together with the shift length of the second shift operation. Similarly, the middle result becomes the input of the adder2, the final result is computed here. Data dependence patterns not only exist in operations such as shifts and additions, but also frequently occur in combinations of memory access and computation. As a result, the memory access inference is connected to DDCU so that computation can be completed as soon as memory access is finished, just like register-memory instructions in X86.

Different from the instruction extension technique, data dependence optimization is in the charege of hardware. The optimization patterns are yielded by compiler while it's difficult to find the corresponding source code in C/C++. As a result, inline assembly technique cannot be used to make compile support instruction extension. In this sense, data dependence optimization makes up for the defect of instruction extension technique by hardware extension.

## IV. Methodology

### A. Setup

In our experiments, we utilize the CVA6 CPU [34], whose open-source repository can be found here: CVA6. CVA6 features a configurable architecture with separate TLBs, a hardware page table walker (PTW), and advanced branch prediction mechanisms, including a branch target buffer and branch history table. These configurations are critical for optimizing performance in specific workloads, as detailed in Table VI. Further configuration settings for the hardware platform are outlined in Table VII.

TABLE VI: Configurations of the original CVA6.

| | |
|---|---|
| Bits | 32/64 |
| Pipeline | 6-stage in-order |
| Extensions | IMAFDC |
| Funct Units | 6 |
| Icache | 16KB, 4-way set-associative, 16B line sizes |
| Dcache | 32KB, 8-way set-associative, 16B line sizes |
| Branch Pred | bimodal predictor with BTB, BHT and RAS |

Our acceleration design based on CVA6 is completed using SystemVerilog. The extended CVA6 introduces decoding, issuing, and write-back logic for custom instructions, as well as a computing unit called PPVCU, which contains 8 adders and 4 multipliers. Additionally, for the data dependence optimization, pattern detection logic and a computing unit called DDCU have been added.

We use Vivado 2020.2 to synthesize the design and assess the impact on FPGA hardware resources. The target FPGA board is XCVU9P Xilinx Virtex UltraScale+, with a clock frequency of 112MHz. For ASIC implementation, we synthesize both CVA6 and our design in 22 nm FDX technology down to ready-for-silicon layout to reliably estimate their operating frequency, power, and area. The tools we use include: (i) Synopsys Design Compiler 2019.03 to perform the physical synthesis; (ii) Cadence Innovus 2020.12 for the place & route; (iii) Synopsys PrimeTime 2020.09 for the power analysis and (iv) Siemens Questasim 10.7b to design the parasitics annotated netlist.

TABLE VII: Setup of the hardware platform.

| | |
|---|---|
| Bits | 32/64 |
| Pipeline | 6-stage in-order |
| Extensions | IMAFDC and **our custom instructions** |
| Function Units | 6 original units, **PPVCU**, **DDCU** |
| PPVCU | 8 adders, 4 multipliers |
| DDCU | 3 adders, 2 multipliers, 3 shifters |
| Icache | 16KB, 4-way set-associative, 16B line sizes |
| Dcache | 32KB, 8-way set-associative, 16B line sizes |
| Branch Pred | bimodal predictor with BTB, BHT and RAS |
| FPGA | XCVU9P, 112MHz |
| ASIC | 22nm FDX technology, 800MHz |
| Compiler | riscv64-unknown-linux-gnu-gcc v.13.2.0 -O2 |

### B. Benchmarks

LeNet [41], ResNet18 [42], VGG16 [43], GoogLeNet-v3 [38] and MobileNet [44] are used as benchmarks. We replace the original implementation of convolutional layers with our inline assembly function implementation of the Winograd algorithm. Moreover, the max pooling layers are also revised by our custom instructions. Specially, the general program section and activation functions which are accelerated by hardware remain unchanged.

The first three benchmarks—LeNet, VGG16, and ResNet18—are the same as those used by Wang et al [9]. To ensure fairness, we replace the $5 \times 5$ convolution with $3 \times 3$ kernels and the $3 \times 3$ maxpooling with $2 \times 2$, adjusting the network structure accordingly in

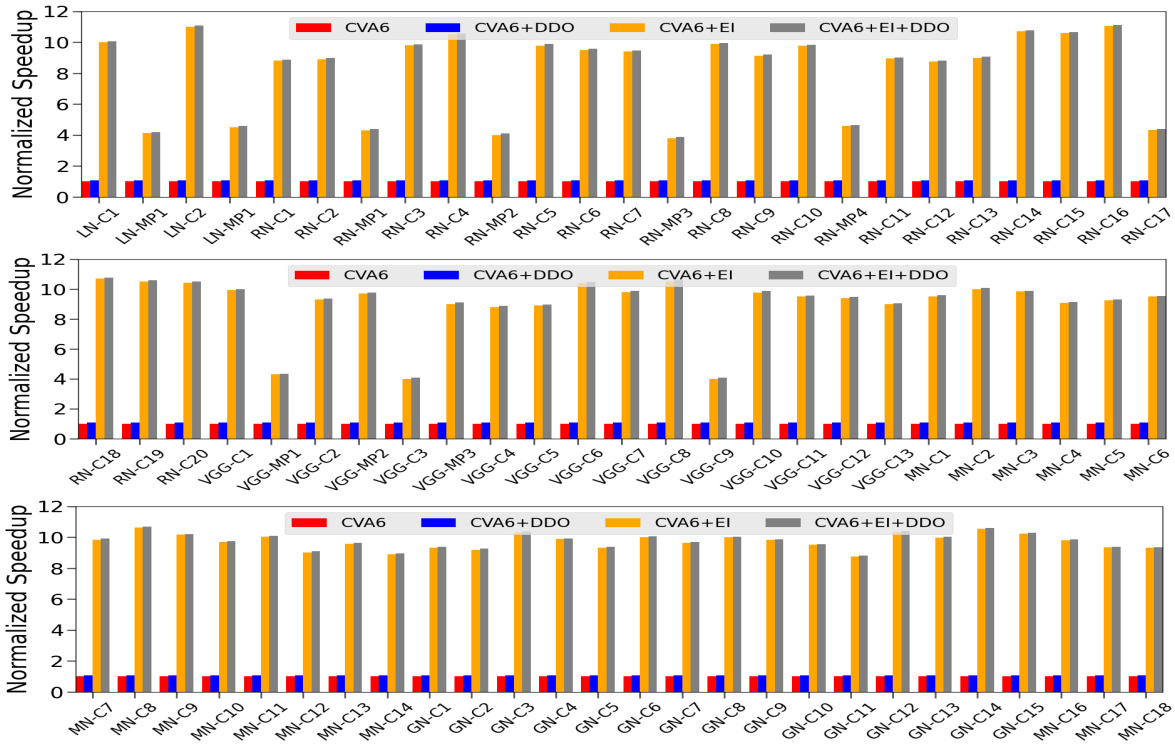


Fig. 5: Performance evaluation results of the ablation study.

line with their work. We adopt the same parameters, network structure, and datasets for training, achieving nearly identical accuracy. Previous studies have shown that larger convolution kernels can be decomposed into 3×3 kernels, improving both performance and efficiency [38], [43]. Moreover, as 3×3 kernels dominate modern CNNs, our optimization for this kernel size ensures strong scalability. Additionally, we select two larger and more complex CNNs, GoogLeNet-v3 and MobileNet, and test them on the ImageNet dataset to further validate the scalability and generality of our acceleration scheme in the ablation study.

TABLE VIII: Benchmarks.

| benchmark | layer | IH, IW, IC | layer | IH, IW, IC |
|---|---|---|---|---|
| Lenet | c1 | 28, 28, 1 | c2 | 14, 14, 6 |
| ResNet18 | c1 | 32, 32, 3 | c2-c5 | 32, 32, 16 |
| | c6 | 16, 16, 16 | c7-c10 | 16, 16, 32 |
| | c11 | 8, 8, 32 | c12-c15 | 8, 8, 64 |
| | c16 | 4, 4, 32 | c17-c20 | 4, 4, 128 |
| VGG16 | c1 | 32, 32, 3 | c2 | 32, 32, 16 |
| | c3 | 16, 16, 16 | c4 | 16, 16, 32 |
| | c5 | 8, 8, 32 | c6-c7 | 8, 8, 64 |
| | c8 | 4, 4, 64 | c9-c10 | 4, 4, 128 |
| | c11-c13 | 2, 2, 128 | | |
| MobileNet | c1 | 224, 224, 3 | c2 | 112, 112, 32 |
| | c3 | 112, 112, 64 | c4-c5 | 56, 56, 128 |
| | c6, c7 | 28, 28, 256 | c8-c13 | 14, 14, 512 |
| | c14 | 7, 7, 1024 | | |
| GoogLeNet-v3 | c1 | 299, 299, 3 | c2 | 149, 149, 32 |
| | c3 | 147, 147, 32 | c4 | 73, 73, 64 |
| | c5 | 73, 73, 80 | c6-c9 | 35, 35, 64 |
| | c10-c13 | 35, 35, 96 | c14 | 35, 35, 288 |
| | c15-c18 | 17, 17, 192 | | |

Detailed parameters of these CNNs are demonstrated in the Table VIII. The selection of benchmarks includes various models and datasets, thereby addressing both scope and scalability considerations, ensuring the validity of our experiments.

### C. Baselines

Our experiments encompass an ablation study and a comparison with state-of-the-art architectures. Additionally, we evaluate the overhead of our proposed methods and engage in a discussion about the results. Our performance data is evaluated in the pre-synthesis stage and measured based on execution time. To obtain accurate energy data, we use the PrimeTime PX tool.

The first part is the ablation study, which aims to demonstrate the effectiveness of our design. In this experiment, we use the original CVA6 processor as the baseline and the runtime speedup as the evaluation criterion to separately prove the effectiveness of our two optimization designs: the extended instructions and the data dependence optimization.

The second part is a comparison with RI5CY+Accel [9] which is state-of-the-art. We conduct the comparison from both runtime and power consumption perspectives to demonstrate the advancement of our design. RI5CY+Accel extends 7 instructions based on the in-order 4-stage CPU RI5CY to accelerate CNNs inference. The core extended instruction is the CONV23 instruction, which accelerates convolution computation using the Winograd algorithm.

## V. Result and Analysis

### A. Effectiveness of Our Methods

We take the original CVA6 processor as the baseline in the ablation study and carry out incremental design to reflect the acceleration effect of our extended instructions and data dependence optimization separately. CVA6+EI represents CVA6 processor with our

designed intructions, CVA6+DDO represents CVA6 processor with data dependence optimization, and CVA6+EI+DDO includes both extended instructions and data dependence optimization. We use speedup as the performance metric and normalize the running speed of each layer on the original CVA6 processor.

Figure 5 presents the results of the ablation study, with performance data obtained from pre-simulation. The results demonstrate that for most convolutional layers, our extended instructions can achieve significant performance improvement. For example, for all the convolutional layers, the performance speedups provided by CVA6+EI are all over 8×. Among the three benchmarks, VGG16 is the best-case scenario with an average performance speedup of 9.63×. Moreover, The maximum speedup for Lenet, ResNet18, VGG16, GoogLeNet-v3 and MobileNet are 10.96×, 10.80×, 10.51×, 10.62× and 10.53× respectively. With respect to the extended instructions designed for pooling layers, we can also observe great performance improvement. For Lenet, there is a best speedup of 4.48× in LN-MP1 layer and the average speedup is 4.31×. For VGG16, the average speedup is 4.21× and the acceleration effect for ResNet18 is almost identical to that of the previous two networks.

Compared to our extended instructions, data dependence optimization can be applied to every layer since it targets all the general program sections and activation function part of the neural network. As shown in Figure 5, CVA6+DDO is much less effective than our extended instructions. The reason is that convolution operations and data loading make up ablationthe main body of the convolutional and pooling layers which have already been optimized by the extended instructions, leaving a small portion of the general program sections. Nevertheless, data dependence optimization still achieves a maximum speedup of 7.8% for the convolutional layers and 6.9% for the pooling layers.

From the Figure 5, it is clear that the overall optimization achieves good performance on networks with small parameters (LeNet), relatively larger parameters (modified versions of VGG16 and ResNet18), and even larger parameters (GoogLeNet-v3, MobileNet). The average performance speedup for LeNet, ResNet18, VGG16, GoogLeNet-v3, and MobileNet are 9.59×, 9.48×, 9.70×, 9.65×, and 9.57× respectively. These results substantiate the effectiveness of our design and highlight its strong scalability across networks of varying sizes. This significant acceleration effect main stems from: 1.Vectorizated design of our instructions, which fully leverages the parallelism inherent in convolutional computations. 2.Pipelined computation characteristic of PPVCU, which hides the latency of the multi-stage operations. 3.Memory access optimization, which reduces the memory access time for dense data. 4.Data dependence optimization, which partially addresses the limitation of the overly simplistic nature of RISC-V instructions.

### B. Comparison with the State-of-the-Art Work

We first compare our design with RI5CY+Accel [9] in terms of performance. We calculate the absolute running time by the formula: Time = Cycles/Frequency and normalize it to get visual speedup. Additionally, since the processor base we use (CVA6) differs from theirs (RI5CY) and our base has higher performance, we reproduce CVA6+Accel using CVA6 as the base to isolate the impact of the performance gap between the base CPU. Figure 6 presents the execution time speedups, where the number of cycles is obtained from the pre-synthesis simulation, and the frequency is determined by the target value meeting timing constraints after synthesis. Based on these measurements, our design outperforms RI5CY+Accel, the state-of-the-art for convolutional layers, reducing the execution time to an average of 11.99%. It is obvious that convolutional layers of VGG16 benefits from our design most, with VGG-C12 only taking only 6.4% (1/15.77) time of RI5CY+Accel. Compared to CVA6+Accel, the average speedup is 1.63×, clearly demonstrating that the performance gains come not only from the CVA6, but also from the additional optimizations in our design. For pooling layers, we also observe great performance improvement compared to RI5CY+Accel. The speedup for RI5CY+Accel and CVA6+Accel are 3.87× and 1.67× respectively. For activation functions, while RI5CY+Accel achieves a 67% speedup for ReLU function, its applicability is highly limited, and it does not provide acceleration for other activation functions. In contrast, as shown in Table IX, our design exhibits lower acceleration effect, with only 25% performance improvement for ReLU, 21% for Sigmoid and 17% for Softmax. Although performance is sacrificed, our design offers better generalizability and maintains the original characteristics of neural networks.

TABLE IX: Optimization effects of RI5CY+Accel and CVA6+DDO on activiation functions

| Activation \ Processor | RI5CY+Accel | CVA6+DDO |
|---|---|---|
| ReLU | 66% | 25% |
| Sigmoid | - | 21% |
| Softmax | - | 17% |

Energy efficiency is another important metric for evaluating the quality of processors. It is calucated by the formula: Energy efficiency= Performance/Power. As shown in Figure 7, for all the pooling layers, we observe a significant energy efficiency advantage, with an average of 6.7× improvement compared to RI5CY+Accel. For convolutional layers, the energy efficiency advantage of our design is not that outstanding, but it still achieves an average of 3.06× speedup. Compared to the speedups of the performance in Figure 5, the speedups of the energy efficiency are not that notable because the original CVA6 processor is much more complex than the original RI5CY processor, leading to higher energy consumption.

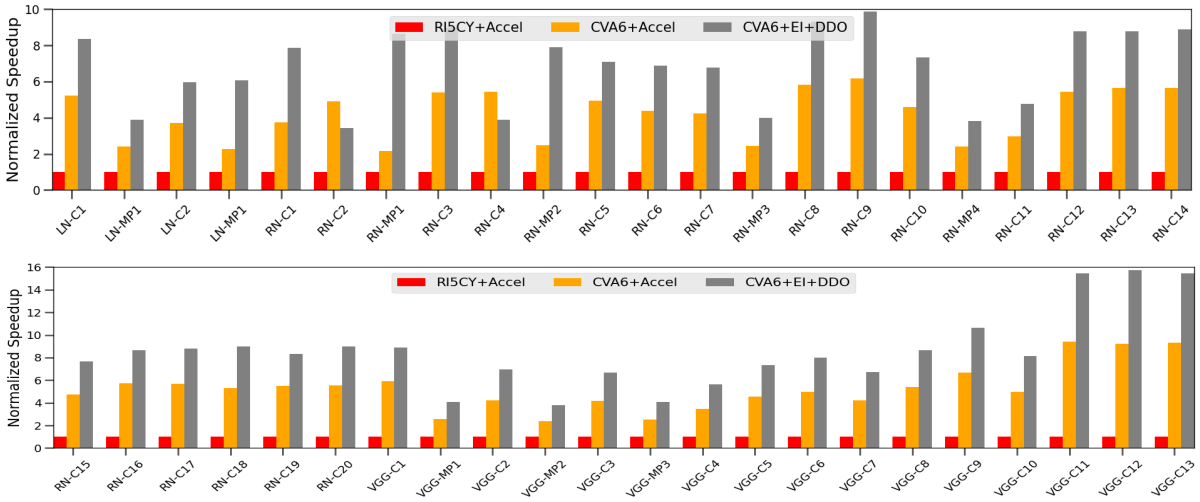


Fig. 6: Performance comparison between RI5CY+Accel, CVA6+Accel, and CVA6+EI+DDO.

### C. Overhead and Discussion

The comparison between our design and the baseline on FPGA and ASIC platforms is shown in Table X. On FPGA, although our design increases the LUTs, Registers, DSPs and power consumption by 2.01%, 5.56%, 3.76% and 5.95% respectively, the energy efficiency improves by approximately 6.95 times. Furthermore, our design also demonstrates significant optimization benefits on ASIC. Specifically, the area and power consumption increase by only 0.38% and 3.21%, respectively, while the energy efficiency improves by approximately 6.71 times. Evaluation on both platforms indicates that our design achieves substantial performance improvements without significantly increasing resource components and power consumption, effectively enhancing energy efficiency.

Additionally, we compare our architecture with other processors, as shown in Table XI. The power and energy efficiency results presented are evaluated using the geometric mean across all benchmarks. For high-performance accelerator, Gemmini (16×16) achieves an energy efficiency of 12.19 GOPS/W, whereas our design reaches 120.87 GOPS/W, which is 9.91 times that of Gemmini. This gain comes from our use of low-power RISC-V custom instructions and integrating the computational units into the processor pipeline. The large spatial array and extensive scratchpad memory of the Gemmini generated accelerator contribute to its higher power consumption. For the low-power CPU ARM Cotex-M0, despite its lower power consumption (which is 18.81% of our design), its peak performance is similarly poor, achieving only 1.65% of our design, resulting in an energy efficiency of just 8.75% of our design. The last three columns of the table show data of the CVA6, the RI5CY+Accel [9], and our design. It can be observed that our baseline CVA6 has lower energy efficiency compared to the RI5CY+Accel. However, our approach achieves 6.9× the peak performance of the CVA6 with only a small increase in power consumption (3.2%), resulting in an energy efficiency that is approximately 6.7 times the baseline. Ultimately, our energy efficiency reaches approximately 3.1 times that of the RI5CY+Accel, representing a significant performance improvement.

TABLE X: Comparison of CVA6 and our design

| **Technology** | **XCVU9P FPGA** | | **22nm FDX** | |
|---|---|---|---|---|
| | **Baseline CVA6** | **Our Design** | **Baseline CVA6** | **Our Design** |
| **Clk Freq. (MHz)** | 112 | 112 | 800 | 800 |
| **Power (mW)** | 5077 | 5374 | 42.03 | 43.38 |
| **Res./ Area ($mm^2$)** | 46K Regs 55K luts 18DSPs | 47.7K Regs 56.1K luts 19DSPs | 0.257 | 0.258 |
| **Energy Efficiency (GOP/s/W)** | 0.020 | 0.139 | 18.01 | 120.87 |

## VI. Conclusion

In this work, we presented a set of extended RISC-V instructions specifically designed to optimize data load/store operations, convolution, and max pooling in CNNs, supported by a Pipelined Vector Computation Unit for efficient execution. We also introduced Pattern Detection Logic to automatically identify common data dependence patterns in CNNs, allowing the Data Dependence Computation Unit to execute multiple operations in parallel without stalls. In future work, we plan to further refine our design to support optimizations for a wider variety of convolution and pooling operations. We also aim to explore optimizations for more complex CNNs to further enhance the scalability and flexibility of our design.

## References

[1] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *ICML-26*, 2009, pp. 873–880.
[2] J. D. Owens, M. Houston, D. Luebke *et al.*, "Gpu computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
[3] N. P. Jouppi, C. Young, N. Patil *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA-44*, 2017, pp. 1–12.
[4] N. Jouppi, G. Kurian, S. Li *et al.*, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *ISCA-50*, 2023, pp. 1–14.
[5] K. Hegde, R. Agrawal, Y. Yao *et al.*, "Morph: Flexible acceleration for 3d cnn-based video understanding," in *MICRO-51*. IEEE, 2018, pp. 933–946.

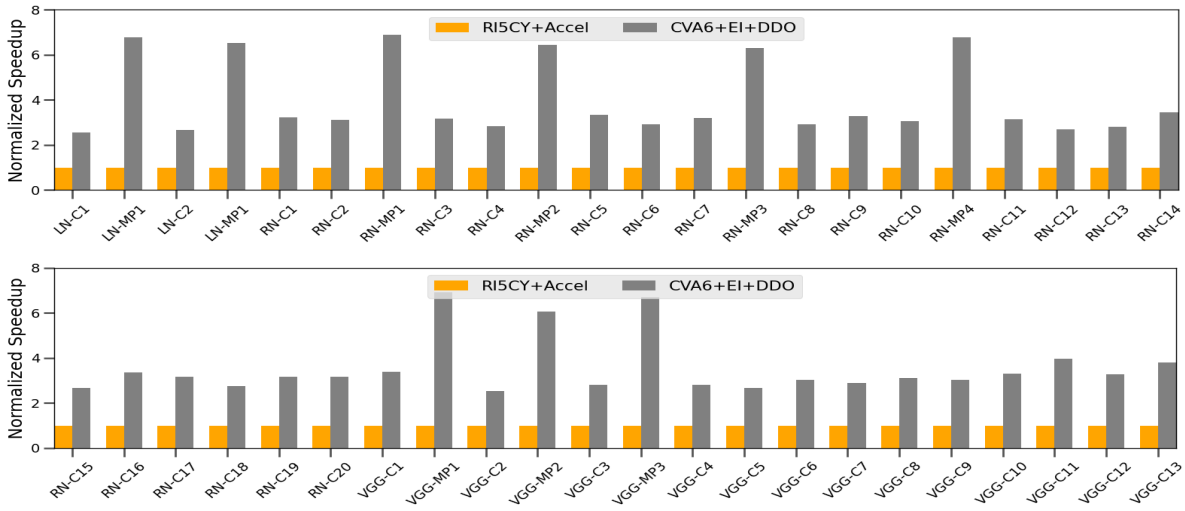


Fig. 7: Energy efficiency (*performance-per-power*) comparisons.

TABLE XI: Comparison with other processors

| Architecture | Gemmini (16×16) | ARM Cotex-M0 | CVA6 | RI5CY+Accel | Ours |
|---|---|---|---|---|---|
| **Platform** | 22nm | 180nm | 22nm | 90nm | 22nm |
| **Clock Freq.** | 52.6MHz | 96MHz | 800MHz | 100MHz | 800MHz |
| **Precision** | 32-bit fixed/float | 16-bit fixed | 32-bit fixed | 32-bit fixed | 32-bit fixed |
| **GOP/s** | 26.93 | 0.0864 | 0.758 (CONV) | 0.225 (CONV) | 5.232 (CONV) |
| **Power** | 2211mW | 8.16mW | 42.03mW | 5.80mW | 43.38mW |
| **Energy Eff. (GOP/s/W)** | 12.19 | 10.58 | 18.01 | 38.79 | 120.87 |


[6] S. Zhang, Z. Du, L. Zhang *et al.*, "Cambricon-x: An accelerator for sparse neural networks," in *MICRO-49*. IEEE, 2016, pp. 1–12.

[7] X. Yu, Z. Yang, L. Peng *et al.*, "Cnn specific isa extensions based on risc-v processors," in *ICCSS-5*, 2022, pp. 116–120.

[8] Z. Li, W. Hu, and S. Chen, "Design and implementation of cnn custom processor based on risc-v architecture," in *HPCC/SmartCity/DSS 2019*, pp. 1945–1950.

[9] S. Wang, X. Wang, Z. Xu *et al.*, "Optimizing cnn computation using risc-v custom instruction sets for edge platforms," *IEEE Trans. Comput.*, vol. 73, no. 5, pp. 1371–1384, 2024.

[10] A. Garofalo, G. Tagliavini, F. Conti *et al.*, "Xpulpnn: Enabling energy efficient and flexible inference of quantized neural networks on risc-v based iot end nodes," *IEEE Trans. Emerging Top. Comput.*, vol. 9, no. 3, pp. 1489–1505, 2021.

[11] H. Jia, H. Valavi, Y. Tang *et al.*, "A programmable heterogeneous microprocessor based on bit-scalable in-memory computing," *IEEE J. Solid-State Circuits*, vol. 55, no. 9, pp. 2609–2621, 2020.

[12] H. Yang and JING, "Design and implementation of cnn acceleration module based on rocket-chip open source processor," *Microelectronics and Computer*, vol. 35, no. 4, 2018.

[13] K. He, G. Gkioxari, P. Dollár *et al.*, "Mask r-cnn," in *ICCV*, 2017, pp. 2961–2969.

[14] P. Heckbert, "Fourier transforms and the fast fourier transform (fft) algorithm," *Computer Graphics*, vol. 2, no. 1995, pp. 15–463, 1995.

[15] M. Dukhan, "The indirect convolution algorithm," *ArXiv*, vol. abs/1907.02129, 2019.

[16] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *CVPR*, June 2016.

[17] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *FCCM-24*, 2016, pp. 40–47.

[18] L. Bai, Y. Zhao, and X. Huang, "A cnn accelerator on fpga using depthwise separable convolution," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 65, no. 10, pp. 1415–1419, 2018.

[19] S. Qin *et al.*, "StreamDCIM: A Tile-based Streaming Digital CIM Accelerator with Mixed-stationary Cross-forwarding Dataflow for Multimodal Transformer," *arXiv preprint arXiv:2502.05798*, 2025.

[20] S. Qin, Z. Fan, W. Li *et al.*, "PANDA: Adaptive Prefetching and Decentralized Scheduling for Dataflow Architectures," in *TACO*, in press.

[21] H. Genc, S. Kim, A. Amid *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *DAC*, 2021.

[22] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking tpu, gpu, and cpu platforms for deep learning," *arXiv preprint arXiv:1907.10701*, 2019.

[23] E. Cui, T. Li, and Q. Wei, "Risc-v instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, pp. 24 696–24 711, 2023.

[24] M. B. Hervé Legenvre, Pietari Kauttu and R. Khawand, "Is open hardware worthwhile? learning from thales' experience with risc-v," *Research-Technology Management*, vol. 63, no. 4, pp. 44–53, 2020.

[25] A. Waterman, Y. Lee, D. A. Patterson *et al.*, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, p. 4, 2014.

[26] E. Tehrani, T. Graba, A. S. Merabet *et al.*, "Classification of lightweight block ciphers for specific processor accelerated implementations," in *ICECS-26*. IEEE, 2019, pp. 747–750.

[27] B. Koppelmann, P. Adelt, W. Mueller *et al.*, "Risc-v extensions for bit manipulation instructions," in *PATMOS-29*. IEEE, 2019, pp. 41–48.

[28] H. B. Amor, C. Bernier, and Z. Přikryl, "A risc-v isa extension for ultra-low power iot wireless signal processing," *IEEE Trans. Comput.*, vol. 71, no. 4, pp. 766–778, 2021.

[29] M. Gautschi, P. D. Schiavone, A. Traber *et al.*, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Trans. VLSI Syst.*, vol. 25, no. 10, pp. 2700–2713, 2017.

[30] S. Qin, W. Li, Z. Fan *et al.*, "Roma: A reconfigurable on-chip memory architecture for multi-core accelerators," in *HPCC/DSS/SmartCity/Dependability*. IEEE, 2023, pp. 49–57.

[31] W. Tang and P. Zhang, "Gpgcn: A general-purpose graph convolution neural network accelerator based on risc-v isa extension," *Electronics*, vol. 11, no. 22, p. 3833, 2022.

[32] S.-Y. Lee, Y.-W. Hung, Y.-T. Chang *et al.*, "Risc-v cnn coprocessor for real-time epilepsy detection in wearable application,"

*IEEE Trans. Biomed. Circuits Syst.*, vol. 15, no. 4, pp. 679–691, 2021.
[33] A. Garofalo, G. Tagliavini, F. Conti *et al.*, "Xpulpnn: accelerating quantized neural networks on risc-v processors through isa extensions. in *DATE*, 2020.
[34] N. Kabylkas, T. Thorn, S. Srinath *et al.*, "Effective processor verification with logic fuzzer enhanced co-simulation," in *MICRO-54*, 2021, pp. 667–678.
[35] Ning Wu, Tao Jiang, F. Ge *et al.*, "A reconfigurable convolutional neural network-accelerated coprocessor based on risc-v instruction set," *Electronics*, vol. 9, no. 6, p. 1005, 2020.
[36] Hansong Liao, Zhaohui Wu, Bin Li *et al.*, "Special instruction set processor for convolutional neural network based on risc-v," *Computer Engineering*, vol. 47, no. 7, pp. 196–204, 2021.
[37] D. Siyuan, "Design and simulation studies of convolutional neural network accelerator based on risc-v," Master's thesis, University of Electronic Science and Technology of China, 2023.
[38] C. Szegedy, V. Vanhoucke, S. Ioffe *et al.*," Rethinking the inception architecture for computer vision," *arXiv preprint arXiv:1512.00567*, 2015.
[39] D. Sima, "The design space of register renaming techniques," *IEEE Micro*, vol. 20, no. 5, pp. 70–83, 2000.
[40] P. S. Rawat, A. Sukumaran-Rajam, A. Rountev *et al.*, "Associative instruction reordering to alleviate register pressure," in *SC18*. IEEE, 2018, pp. 590–602.
[41] Y. LeCun, L. Bottou, Y. Bengio *et al.*, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[42] K. He, X. Zhang, S. Ren *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.
[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[44] A.G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

**Teng Luo** received the B.S. degree from Harbin Institute of Technology in 2024. He is currently a master student in Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include computer architecture, dataflow architecture and RISC-V microarchitecture.

**Tengfei Xia** received the B.S. degree from University of Chinese Academy of Sciences, China in 2024. He is currently a master student in Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include computer architecture, dataflow architecture and reconfigurable architecture.

**Jiayuan Chen** is currently a senior engineer at China Mobile Research Institute. Her main research interests include RISC-V microarchitecture and reconfigurable architecture.

**Zhihua Fan** received his B.S. degree from JiLin University in 2018 and Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences in 2024. He is currently an assistant professor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His current research interests include dataflow architecture, programming model and reconfigurable architecture.

**Wenming Li** received the Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2016. He is currently an associate professor in Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His main research interests include high-throughput processor architecture, dataflow architecture and software simulation.

**Yudong Mu** received the B.S. degree from University of Chinese Academy of Sciences, China in 2023. He is currently a master student in Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include dataflow architecture, dataflow graph mapping and reconfigurable architecture.

**Xuejun An** received his PhD degrees in computer science from the Institute of Computing Technology, Chinese Academy of Sciences. He is a professor with the Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include programming model and processor architecture.

**Xiaochun Ye** received his Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2010. He is currently a professor in Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His main research interests include high-performance computer architecture and software simulation.

**Dongrui Fan** received his Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2005. He is currently a professor and Ph.D. supervisor in Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His main research interests include high-throughput computer architecture and high- performance computer architecture.