PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

WILLIAN ANALDO NUNES

ACCELERATING MACHINE LEARNING USING RISC-V VECTOR EXTENSION IN A MANYCORE PLATFORM

Porto Alegre
2025

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

ACCELERATING MACHINE LEARNING USING RISC-V VECTOR EXTENSION IN A MANYCORE PLATFORM

WILLIAN ANALDO NUNES

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2025

FICHA CATALOGRÁFICA

WILLIAN ANALDO NUNES

ACCELERATING MACHINE LEARNING USING RISC-V VECTOR EXTENSION IN A MANYCORE PLATFORM

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March xx, 2025.

COMMITTEE MEMBERS:

Prof. Dr. Sandro Rigo (IC/UNICAMP)

Prof. Dr. César Augusto Missio Marcon (PPGCC/PUCRS)

Prof. Dr. Fernando Gehm Moraes (PPGCC/PUCRS - Advisor)

ACELERAÇÃO DE APRENDIZADO DE MÁQUINA USANDO EXTENSÃO VETORIAL RISC-V EM UMA PLATAFORMA MANYCORE

RESUMO

O crescente aumento na demanda computacional de cargas de trabalho de Machine Learning (ML), especialmente Redes Neurais Convolucionais (CNNs), exige soluções eficientes de aceleração em hardware. Esta dissertação investiga o uso da Extensão Vetorial do RISC-V (RVV) para acelerar a inferência de CNNs em arquiteturas single-core e manycore. O estudo apresenta o processador RS5, uma implementação RTL de um núcleo baseado em RISC-V aprimorado com um subconjunto de instruções RVV projetado para paralelismo eficiente de dados. Além disso, este processador foi integrado à plataforma manycore Memphis-V, permitindo um escalonamento adicional de desempenho por meio da execução paralela.

Foi realizada uma avaliação abrangente para analisar o impacto da aceleração baseada em RVV no desempenho, consumo de energia, uso de memória e custos de área de hardware. Os resultados demonstram que a implementação vetorizada das operações de CNN no processador RS5 atinge um speedup de até 7,68× (camada 1-D CNN) na execução single-core em comparação com a versão escalar, reduzindo o consumo de energia em até 61% e alcançando ganhos de desempenho de até 16× em uma aplicação de produto escalar (dot-product). Quando implantado no ambiente manycore, foram observados ganhos adicionais de desempenho, com a primeira camada da AlexNet atingindo uma aceleração de até 5,7× sobre a implementação escalar single-core e reduzindo o tamanho do código em até 87,5% na segunda camada. A integração da auto-vetorização e da otimização manual em assembly vetorial destacou ainda mais a eficácia do RVV na aceleração de cargas de trabalho de ML.

Os resultados experimentais demonstram que a integração do RVV melhora significativamente a velocidade de inferência de CNNs, mantendo um baixo overhead de área. A implementação manycore amplifica ainda mais esses benefícios, evidenciando o potencial das arquiteturas vetoriais baseadas em RISC-V para aceleração eficiente de ML. Este trabalho contribui para a área de aceleração em hardware ao apresentar uma solução escalável e de código aberto para aplicações de CNNs.

Palavras-Chave: RISC-V, Processamento Vetorial, Aceleração de Hardware, Manycores, Redes Neurais Convolucionais.

ACCELERATING MACHINE LEARNING USING RISC-V VECTOR EXTENSION IN A MANYCORE PLATFORM

ABSTRACT

The increasing computational demands of Machine Learning (ML) workloads, particularly Convolutional Neural Networks (CNNs), require efficient hardware acceleration solutions. This dissertation investigates the RISC-V Vector Extension (RVV) to accelerate the CNN inference in single-core and manycore architectures. The research presents the RS5 processor, an RTL implementation of a RISC-V-based core enhanced with a subset of RVV instructions designed for efficient data parallelism. Additionally, this processor was integrated into the Memphis-V manycore platform, enabling further performance scaling through parallel execution.

A comprehensive evaluation was conducted to analyze the impact of RVV-based acceleration on performance, energy consumption, memory footprint, and hardware area costs. The results demonstrate that the vectorized implementation of CNN operations on the RS5 processor achieves a speedup of up to 7.68x (1-D CNN layer) in single-core execution compared to a scalar baseline, reducing energy consumption by up to 61% and achieves speed-ups of up to 16x in a dot-product application. When deployed in the manycore environment, additional performance gains were observed, with the first layer of AlexNet achieving up to 5.7× acceleration over the scalar single-core implementation and reducing code size by up to 87% in the second layer. The integration of auto-vectorization and manually optimized vector assembly further highlighted the effectiveness of RVV in accelerating ML workloads.

Experimental results demonstrate that the integration of RVV significantly enhances CNN inference speed while maintaining a low area overhead. The manycore implementation further amplifies these benefits, highlighting the potential of RISC-V-based vector architectures for efficient ML acceleration. This work contributes to hardware acceleration by showcasing a scalable, open-source solution for CNN applications.

Keywords: RISC-V, Vector Processing, Hardware Acceleration, Manycores, Convolutional Neural Networks.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1. INTRODUCTION

Hardware accelerators, specialized hardware components designed for optimized task execution, have regained importance with the growing demand for computation in areas like video processing and Machine Learning (ML) algorithms [Peccerillo et al., 2022]. This resurgence is evident in the use of Graphics Processing Units (GPUs) and the newer Neural Processing Units (NPUs). For instance, Microsoft's recent Copilot+ [Microsoft, 2024] certification for Windows computers requires an NPU with at least 40 TOPS (trillion operations per second) to support advanced Artificial Intelligence (AI) models.

Neural networks (NN) became essential for AI and ML applications, including classification, clustering, and pattern recognition [Goel et al., 2023; Abiodun et al., 2018]. Convolutional Neural Networks (CNNs), specifically designed for structured grid data like images, are particularly effective for image recognition and object detection tasks. However, optimizing CNN inference, which involves extensive data transfers and multiplications, presents challenges in hardware development. Domain-specific accelerators are crucial for enhancing performance in these data-intensive applications [Chang and Culurciello, 2017].

With its open-source instruction set architecture (ISA), the RISC-V [RISC-V Foundation, 2024a] architecture enables the development of processors tailored to specific applications. Due to its rich standard extensions and ease of customization, it has emerged as the favored ISA for domain-specific processor designs. The RISC-V Vector (RVV) Extension [RISC-V Foundation, 2021] enhances the RISC-V ISA by introducing vector processing capabilities. Its primary purpose is to provide efficient and scalable data parallelism tailored for large-scale applications such as scientific computing, ML, multimedia processing, and other high-performance computing tasks.

Manycore architectures incorporate multiple individual processing units, or cores, on a single chip. This design answers the growing demand for enhanced computational power and performance across various applications. The main advantage of manycores lies in their ability to execute multiple tasks concurrently through parallel processing. This results in more efficient utilization of resources and performance. In ML applications, this parallelism enables complex computations to be processed simultaneously, accelerating the inference phase of ML models and leading to faster and more scalable ML solutions.

This work seeks to explore the application of the RISC-V Vector Extension to accelerate CNN applications in single and manycore architectures, exploring parallelism opportunities brought by this extension. It achieved significant application speedups, allowing for much faster CNN inference in single or manycore environments. These results highlight that a processor with the right instruction set extension can play a significant role in deploying modern algorithms.

1.1 Motivation

The primary motivation for this study is the scarce literature on RISC-V CNN accelerator implementations in manycore systems with general-purpose processing elements and with the RVV extension as a tightly coupled accelerator. As discussed in Chapter 3, most manycore studies focus on mapping algorithms or changes in particular architectures; meanwhile, the RISC-V implementations usually focus on high-end accelerators or custom accelerators that do not enable the exploration of different ML tasks.

Another motivation is the increased popularity of ML algorithms, especially CNNs, which require specialized hardware to run efficiently because they are highly computer-intensive. This relates directly to the ratification of the RISC-V Vector extension and the limited work with processors that implement the latest version of the extension. Also, the suitability of the RISC-V Vector extension to accelerate CNN in tandem with the compiler's auto-vectorization has yet to be proven.

1.2 Objectives

This work has two strategic goals. The first is to develop an accelerated RISC-V processor. The second involves exploring a manycore platform that uses this processor to accelerate CNN applications.

Specific goals include:

1. Provide an RTL implementation of a RISC-V processor [RISC-V Foundation, 2024a], namely the RS5 processor [Nunes et al., 2024], with a subset of the RISC-V Vector [RISC-V Foundation, 2021] extension instructions, targeting CNN inference hardware acceleration;
2. Integrate the accelerated processor into a manycore platform and analyze the characteristics of this architecture to enhance the acceleration of ML applications through the extended instruction set implemented in the previous step;
3. Evaluate the effectiveness of the resulting implementation quantitatively using metrics, such as performance and memory footprint for the software component, cycles spent for the application, die area, and energy consumption for the hardware components.

1.3 Contributions

The contributions of this work are:

1. Optimized version of the RS5 processor, including new extensions and mechanisms such as branch prediction, compressed instructions, atomic memory accesses, and cryptography acceleration (Chapter 4).
2. Implementation of a subset of the Vector extension as a tightly coupled accelerator in the RS5 processor (Chapter 5).
3. Extension of the benchmark library of the RS5 processor by adding 1-D and 2-D CNN applications (Chapter 6).
4. New CNN benchmark in the Memphis manycore platform, exploring a manycore mapping technique that further improves parallelism in the CNN inference phase (Chapter 6).
5. Exploration of the accelerated RS5 processor and the chosen manycore mapping technique in CNN applications, analyzing the speedup, suitability, and compiler's efficiency, and providing results and comparisons for single-core and manycore accelerated and non-accelerated versions (Chapter 7).

1.4 Methodology

The work described in this dissertation follows these steps. First, the RS5 processor, an existing open-source implementation of the RISC-V ISA developed by the author, gained specialized instructions conforming to the vector extension of the RISC-V architecture. These instructions accelerate various algorithms and applications, particularly CNNs.

Next, the implemented instructions accelerated specific CNN benchmarks to measure speedup and improvement. Hardware parameters were explored to identify configurations best suited to the target applications, considering the requirements of each benchmark. The compiler's auto-vectorization was analyzed, and manual vectorization was performed to compare the performance gains achieved by each approach.

After obtaining the hardware-accelerated results, the effectiveness of the accelerated implementation was quantitatively compared to non-accelerated versions of the same applications. The evaluation considered software metrics, such as overall code size, and hardware metrics, such as energy consumption and die area. Performance was assessed by comparing the clock cycles required for identical tasks in both implementations. The

die area was evaluated through FPGA implementation and ASIC synthesis, followed by a thorough analysis of the results.

With the hardware accelerator implemented and validated, the accelerated RS5 processor served as the processing element (PE) of Memphis-V [Ruaro et al., 2019], a Network-on-Chip (NoC)-based manycore platform. This integration facilitated exploring a CNN application in this environment, where parallelization techniques were evaluated for applicability. One benchmark application was adapted for execution in the manycore context, enabling metrics extraction for the explored technique.

Finally, the results were compared across all cases: non-accelerated RS5, accelerated RS5, non-accelerated Memphis-V, and accelerated Memphis-V. Metrics were analyzed and evaluated within the different environments, and the final results included system bottlenecks, drawbacks, and trade-offs.

1.5 Document organization

This manuscript is organized as follows.

- Chapter 2 presents fundamental concepts concerning hardware acceleration and the RISC-V architecture, detailing the vector extension. It also presents the manycore platform used in this work, Memphis-V, and concepts related to ML and CNN;
- Chapter 3 presents the state-of-the-art related to hardware acceleration involving RISC-V and ML, and also presents works that target accelerating CNN in manycore platforms;
- Chapter 4 details the RISC-V RS5 processor, corresponding to the first contribution of the Dissertation. The RS5 processor started in the end-of-term work of the Author, being extended to meet the goals of the Dissertation;
- Chapter 5 presents the second contribution of the Dissertation, which is the design of the RISC-V vector extension in the RS5 processor, bringing insights of the implementation and design choices;
- Chapter 6 details the third and fourth contributions of the Dissertation by presenting the benchmark applications and referring to the mapping of a CNN application on the manycore. The Chapter initially presents the selected benchmarks and discusses mapping alternatives on a manycore. The chosen mapping procedure is applied to the manycore, and the CNN deployment is presented;
- Chapter 7 presents the fifth contribution of the Dissertation by presenting the evaluation of the benchmark results in all the developed environments (single and manycore), evaluating die area (FPGA and ASIC), clock cycle reduction, and power consumption;
- Chapter 8 finishes this manuscript with conclusions derived from previous chapters and highlights future work.

2. BACKGROUND KNOWLEDGE

This Chapter introduces fundamental concepts to the understanding of this work:

- Section 2.1 overviews hardware acceleration concepts;
- Section 2.2 introduces the RISC-V instruction set, an open Instruction Set Architecture (ISA);
- Section 2.3 introduces the RISC-V Vector extension, designed to provide efficient and scalable data parallelism;
- Section 2.4 presents the Memphis-V manycore, a framework for the automatic generation and validation of manycores;
- Section 2.5 describe Convolutional Neural Networks (CNNs), a specific type of deep learning architecture within the broader field of Machine Learning;
- Section 2.6 presents final remarks related to the presented concepts.

2.1 Hardware Acceleration

As Gordon Moore stated in 1965, Moore's Law predicted that the number of transistors would double every two years. This trend was upheld for decades, driving exponential growth in computing power. However, the pace has slowed in recent years due to physical and economic challenges. Transistor scaling has become more complex and costly as we approach atomic-scale dimensions [Zhu et al., 2023]. Computing innovation continues through new paradigms. The focus has shifted from transistor count to performance-per-watt, energy efficiency, and new computing architectures.

We are in "A New Golden Age for Computer Architecture", as stated by John Hennessy and David Patterson [Hennessy and Patterson, 2018], which comprises new opportunities for improvements like Domain-Specific Hardware/Software Co-Design. Domain-specific hardware (accelerator) is a specialized hardware component capable of efficiently executing specific tasks (thus domain-specific). The increasing demand for general-purpose processors required hardware specialization across all computing systems, including mobile processors, desktops, and data centers [Kim and Shao, 2018]. Consequently, nearly all leading semiconductor manufacturers and cloud service providers offer chips equipped with accelerators, regardless of their size, for various applications (Cryptography, Graphics and Multimedia Processing, AI and ML, etc.).

Hwu and Patel [2018] define an accelerator as a distinct architectural substructure designed with a different set of objectives than the base processor, originating from the requirements of a specific class of applications. This design approach allows the accelerator to

achieve better performance, reduced cost, lower power consumption, and less development effort, or a combination of these benefits compared to standard base hardware.

One of the most recognized accelerators is the Graphics Processing Unit (GPU), specialized hardware designed for efficiently processing graphical data. Initially developed to render high-resolution images and animations for gaming and visual applications, GPUs are optimized for parallel processing, enabling them to handle multiple tasks simultaneously [Blythe, 2008]. This capability makes them ideal for rendering 3D graphics, performing matrix calculations, and accelerating video processing. Over time, their use has expanded beyond graphics to other domains such as artificial intelligence, machine learning, and scientific computing, where their ability to process large datasets in parallel significantly improves performance. NVIDIA's CUDA and OpenCL (Open Computing Language) are programming frameworks that enable developers to write code for GPUs [Sanders, 2010] using a technique known as General-Purpose computing on Graphics Processing Units (GPGPU). Since 2008, GPUs have seen widespread adoption in the computing market. NVIDIA reports that by 2016, over one billion CUDA-enabled GPUs were sold [Hwu and Patel, 2018].

Field-Programmable Gate Arrays (FPGAs) are reconfigurable hardware accelerators particularly well-suited for real-time, compute-intensive tasks. In contrast to GPUs, which are designed for general-purpose parallel processing, FPGAs are reconfigurable and can be tailored for specific functions, making them highly adaptable to different applications. Their design supports low-latency operations and efficient parallelism, which is crucial for rapid data acquisition and image processing. For instance, FPGAs are effective in real-time image filtering and video processing, achieving high throughput and performance even under strict timing needs [Hudomalj et al., 2020]. Their energy efficiency and hardware-level optimizations enhance their popularity in telecommunications and edge computing. As noted by Hwu and Patel [2018], FPGA solutions are emerging in cloud computing. Microsoft reports that more than one million servers are equipped with FPGA accelerators in Azure Cloud data centers. These FPGA accelerators enhance the performance of network host functions. By offloading the network stack functions to FPGAs, these tasks can be executed more efficiently in terms of energy consumption, freeing up host computing resources for applications and boosting the cost-effectiveness of cloud server provisioning.

Hardware accelerators can be engineered in various configurations. Based on their interaction with the central processor, they may be classified into two main categories: tightly coupled accelerators and coprocessors.

Tightly coupled (or tightly integrated) hardware accelerators are designed to work closely with the main processor's architecture. They allow efficient communication and data transfer with reduced latency and often share resources like the memory bus. These accelerators are often designed to act like an additional execution unit and must be tailored to the specific main processor architecture. However, due to their unique design, they can take longer to develop and have limited reuse.

A coprocessor is a separate, loosely coupled processing unit that accelerates specific tasks while allowing the main processor to focus on general-purpose tasks. A dedicated bus/interface, like AXI or PCI-Express, often connects coprocessors to the main processor. Coprocessors can be added or removed from a system more easily than tightly integrated accelerators. This scalability allows for flexibility in system design and upgrades without requiring significant modifications. Furthermore, coprocessors can be re-used in systems that implement the same dedicated bus/interface.

2.1.1 Parallelism Basic Concepts

Data parallelism distributes data across multiple processing units to perform the same operation concurrently. This optimizes resource use, as all units execute similar tasks simultaneously. It excels in tasks requiring identical operations on several data pieces, ensuring optimal load balance. In this method, computation occurs synchronously, and parallelization is limited by input size and available processing elements.

Task parallelism aims for concurrent execution of independent tasks, breaking larger tasks into smaller ones for different processing units. The extent of parallelization is limited by the number of independent tasks available, and load balancing is influenced by hardware and scheduling algorithms.

Single Instruction, Multiple Data (SIMD) and *Multiple Instruction, Multiple Data (MIMD)* are computing architectures closely linked to data and task parallelism. SIMD architectures naturally enable data parallelism by executing the same instruction across multiple data elements simultaneously. On the other hand, MIMD architectures facilitate task parallelism by allowing the execution of diverse instructions on various data sets.

SIMD is a parallel processing architecture that operates on multiple data elements with one instruction. Unlike scalar architectures, SIMD processes multiple elements simultaneously through a single instruction stream controlling various processing elements. It is ideal for tasks needing the same operation on large datasets.

MIMD is a parallel computing architecture in which multiple processors execute different instructions on different data simultaneously. Each processor has its own control unit and memory, making it flexible for various applications and capable of handling diverse tasks. However, it may require more complex coordination than SIMD architecture.

2.1.2 Vector Architectures

Vector architectures are systems that efficiently process and manipulate vectors or data arrays. Following the SIMD paradigm, they concurrently apply a single instruction to

multiple data elements, enabling high throughput and parallel processing efficiency. Vector instructions are specialized commands that operate on entire vectors in a single step.

Vector processors frequently utilize techniques such as pipelining and parallel execution to enhance performance further. Pipelining facilitates the overlapping of instruction execution stages, whereas parallelism allows for the concurrent execution of multiple vector operations. Vector processors are specifically optimized for consistent and predictable memory access patterns, as they operate on consecutive elements within vectors.

Vector processors usually feature specialized registers designed to store and manipulate vector data. These registers can contain multiple vector elements, enabling their simultaneous processing. A notable advantage of these specialized registers is their reconfigurability, allowing modifications to the element length or size at runtime. This capability allows the vector processor to handle various data types or adjust precision accordingly.

Vector architectures improve computational performance, especially for large-scale data processing tasks. These architectures have been utilized in numerous processors since the early days of computing. A prime example is the ILLIAC IV [Barnes et al., 1968], created in the 1960s at the University of Illinois, one of the first processors with vector architecture. The Cray-1 [Russell, 1978], introduced in 1976, became a renowned vector processor, performing operations at unprecedented speeds and driving advances in climate modeling and molecular dynamics. In the following decades, vector processing was incorporated into many supercomputers, but by the 1990s, it began to be integrated into general-purpose processors, notably through SIMD extensions in CPUs. In 1994, the SPARC ISA, a Reduced Instruction Set Computer (RISC) ISA, introduced the Visual Instruction Set (VIS) that reuses floating-point registers to store multiple values of 8, 16, or 32 bits[1]. Intel's MMX (Multimedia Extensions) in the mid-1990s [Peleg and Weiser, 1996], SSE (Streaming SIMD Extensions) in the early 2000s, and Intel's Advanced Vector Extensions (AVX) [Lomont, 2011], introduced in 2011, represent significant milestones in bringing vector processing to mainstream processors.

2.2 RISC-V ISA

RISC-V is a modern, open, and extensible Instruction Set Architecture (ISA), initially developed at the University of California, Berkeley. It aims to serve as an open standard ISA. RISC-V is managed by RISC-V International, a global non-profit organization (https://riscv.org/about/). The architecture's design principles emphasize simplicity, flexibility, and extensibility, making it scalable for a wide range of applications. This scalability is achieved through a core set of instructions and optional extensions.

[1] https://en.wikipedia.org/wiki/Visual_Instruction_Set

The RISC-V ISA set is defined in its two-volume Instruction Set Manuals [RISC-V Foundation, 2024a,b]. Table 2.1, adapted from RISC-V Foundation [2024a], contains the current list of modules supported by the RISC-V standard. The Base modules correspond to the ISA options available for implementation. The Instruction length defines the ISA and can be 32, 64, or 128 bits, referred to as *XLEN* in the following sections.

Extensions are ways of improving a base module with additional standard instructions. The most relevant extensions are M, A, F and D, which are collectively known as the G extension set. These extensions add integer multiply and division instructions (M), atomic read-modify-write memory instructions (A), and single-precision and double-precision floating point registers (F and D). The extension Zicsr adds a set of instructions that operate atomically in the Control Status Registers (CSRs). Among the other ratified extensions are the Zmmul extension, a subset of the M extension that excludes the division operations, Compressed Instructions extension (C), which adds support to 16-bit instructions reducing memory footprint. The Zicsr extension introduces instructions for accessing and controlling the status of control and status registers (CSRs), and the Zicntr extension adds counters that can be used as performance monitors. Compilation flags inform the compiler of which extensions are available in the target core, indicating the sets of instructions that the generated code can contain.

The RISC-V community is active, with numerous custom extensions being developed and proposed for various applications. Members of this community also have the opportunity to interact with teams responsible for specifying standard extensions currently in development. These extensions fall into three classifications: *(i) Draft*, which are expected to change before ratification; *(ii) Frozen*, not anticipated to change substantially before ratification; and *(iii) Ratified*, which have been officially ratified and are, ideally, permanent.

The RISC-V privileged architecture is a complement of base ISAs. It defines privilege levels and functionalities required to run operating systems and attach external devices. Privilege levels or modes encapsulate different levels of software permissions. There are three defined privileges which are listed in Table 2.2. **Machine (M)** mode is the most privileged level, and its implementation is mandatory in a privileged architecture. **Supervisor (S)** is the second privilege level and is often used by operating systems. **User (U)** mode is the less trustable mode and has the lowest privilege, its operation is restricted. Note that the privilege level 2 is reserved for future use.

All RISC-V hardware implementations must support the M mode, as this is the most privileged mode, which has access to all the Hardware Thread (*hart*) functionalities. All code that runs in machine mode is considered trusted and will not provide any protection against incorrect or malicious application code. The user mode (U) is the most common option besides M since it protects the rest of the system from application code. The hart usually initializes in the M mode, where boot configurations occur. Next, the mode switches to U, if available. The U mode runs applications until a trap occurs, forcing the privilege to change

Table 2.1: The currently defined RISC-V modules are divided into Base and Extensions. Modules can be in the Ratified, Frozen, or Draft status.

Base	Description	Version	Status
RV32I	32-Bits Base Integer Instruction Set	**2.1**	**Ratified**
RV32E	32-Bits Base Integer Instruction Set for embedded systems	**2.0**	**Ratified**
RV64I	64-Bits Base Integer Instruction Set	**2.0**	**Ratified**
RV64E	64-Bits Base Integer Instruction Set for embedded systems	**2.1**	**Ratified**
RV128I	128-Bits Base Integer Instruction Set	*1.7*	*Draft*
Extension	Description	Version	Status
Zifencei	Extension for Instruction-Fetch Fence	**2.0**	**Ratified**
Zicsr	Extension for Control and Status Register (CSR) Instructions	**2.0**	**Ratified**
Zicntr	Extension for Base Counters and Timers	**2.0**	**Ratified**
Zihpm	Extension for Hardware Performance Counters	**2.0**	**Ratified**
Zihintntl	Extension for Non-Temporal Locality Hints	**1.0**	**Ratified**
Zihintpause	Extension for Pause Hint	**2.0**	**Ratified**
Zimop	Extension for May-Be-Operations	**1.0**	**Ratified**
Zicond	Extension for Integer Conditional Operations	**1.0**	**Ratified**
M	Extension for Integer Multiplication and Division	**2.0**	**Ratified**
Zmmul	Extension for Integer Multiplication	**1.0**	**Ratified**
A	Extension for Atomic Instructions	**2.1**	**Ratified**
Zawrs	Extension for Wait-on-Reservation-Set instructions,	**1.01**	**Ratified**
Zacas	Extension for Atomic Compare-and-Swap (CAS) Instructions	**1.0**	**Ratified**
RVWMO	Memory Consistency Model	**2.0**	**Ratified**
Ztso	Extension for Total Store Ordering	**1.0**	**Ratified**
CMO	Extensions for Base Cache Management Operation ISA	**1.0**	**Ratified**
F	Extension for Single-Precision Floating-Point	**2.2**	**Ratified**
D	Extension for Double-Precision Floating-Point	**2.2**	**Ratified**
Q	Extension for Quad-Precision Floating-Point	**2.2**	**Ratified**
Zfh	Extension for Half-Precision Floating-Point	**1.0**	**Ratified**
Zfhmin	Extension for Minimal Half-Precision Floating-Point	**1.0**	**Ratified**
Zfa	Extension for Minimal Half-Precision Floating-Point	**1.0**	**Ratified**
Zfinx	Extension for Floating-Point in Integer Registers	**1.0**	**Ratified**
Zdinx	Extension for double-precision Floating-Point in Integer Registers	**1.0**	**Ratified**
Zhinx	Extension for half-precision Floating-Point in Integer Registers	**1.0**	**Ratified**
Zhinxmin	Extension for Minimal half-precision Floating-Point in Integer Registers	**1.0**	**Ratified**
C	Extension for Compressed Instructions	**2.0**	**Ratified**
B	Extension for Bit Manipulation	**1.0**	**Ratified**
***Zce**	Extension for Code Size Reduction	**1.0**	**Ratified**
P	Extension for Packed-SIMD Instructions	*0.2*	*Draft*
V	Extension for Vector Operations	**1.0**	**Ratified**
***Zbkb**	Bitmanip instructions for Cryptography	**1.0**	**Ratified**
***Zbkc**	Bitmanip instructions for Cryptography	**1.0**	**Ratified**
***Zbkx**	Crossbar permutations	**1.0**	**Ratified**
***Zk**	Standard scalar cryptography extension	**1.0**	**Ratified**
***Zks**	ShangMi Algorithm Suite	**1.0**	**Ratified**
***Zvbb**	Vector Basic Bit-manipulation	**1.0**	**Ratified**
***Zvbc**	Vector Carryless Multiplication	**1.0**	**Ratified**
***Zvkg**	Vector GCM/GMAC	**1.0**	**Ratified**
***Zvkned**	NIST Suite: Vector AES Block Cipher	**1.0**	**Ratified**
***Zvknhb**	NIST Suite: Vector SHA-2 Secure Hash	**1.0**	**Ratified**
***Zvksed**	ShangMi Suite: SM4 Block Cipher	**1.0**	**Ratified**
***Zvksg**	ShangMi Algorithm Suite with GCM	**1.0**	**Ratified**
***Zvkt**	Vector Data-Independent Execution Latency	**1.0**	**Ratified**

Table 2.2: RISC-V privilege levels.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

to a higher level. When a trap occurs, the privilege changes, and the hart branches to a trap handler routine. The trap handler filters the cause of the trap and performs adequate treatment of the trap.

The privileged architecture specifies the CSRs responsible for tracking the hart's privilege and status, and thus requires the RISC-V Zicsr extension. All Zicsr instructions are atomic, meaning they read and write CSRs in a single instruction. Each privilege level has its own set of CSRs, which can be accessed by either its own privilege or by a higher privilege one. If an application tries to access a CSR of a set that belongs to a higher privilege mode, this causes an exception.

2.3 RISC-V Vector Extension

The RISC-V Vector (V) Extension is designed to enhance the RISC-V ISA by introducing vector processing capabilities. Its primary purpose is to provide efficient and scalable data parallelism, which is beneficial for large-scale data processing applications, such as scientific computing, ML, multimedia processing, and other high-performance computing tasks.

The RVV extension adds 32 vector registers (v0-v31) and seven Control and Status Registers (CSRs). Figure 2.1 details the parameters that define an RVV implementation. The parameters are:

- **VLEN**: each vector register has a fixed length, which must be a power of 2 (Figure 2.1 A).
- **ELEN** - element length: also a power of 2, defines the maximum size of a vector element that any operation can produce or consume (Figure 2.1 B). The VLEN and ELEN parameters are fixed and defined at design time. In Figure 2.1, they are represented as arrows above the registers.
- **SEW** - Selected Element Width: the RVV extension allows runtime configuration of the element width, ranging from 8-bit elements up to ELEN-bit elements (Figure 2.1 C). This feature enables the same hardware to support various operations and precision levels.
- **LANE**: design parameter, refers to the hardware capabilities of data-parallel processing, in other words, processing elements. As Figure 2.1 D shows, the implementations can vary in number of Lanes, a unique Lane implies that only one ELEN-wide element will be

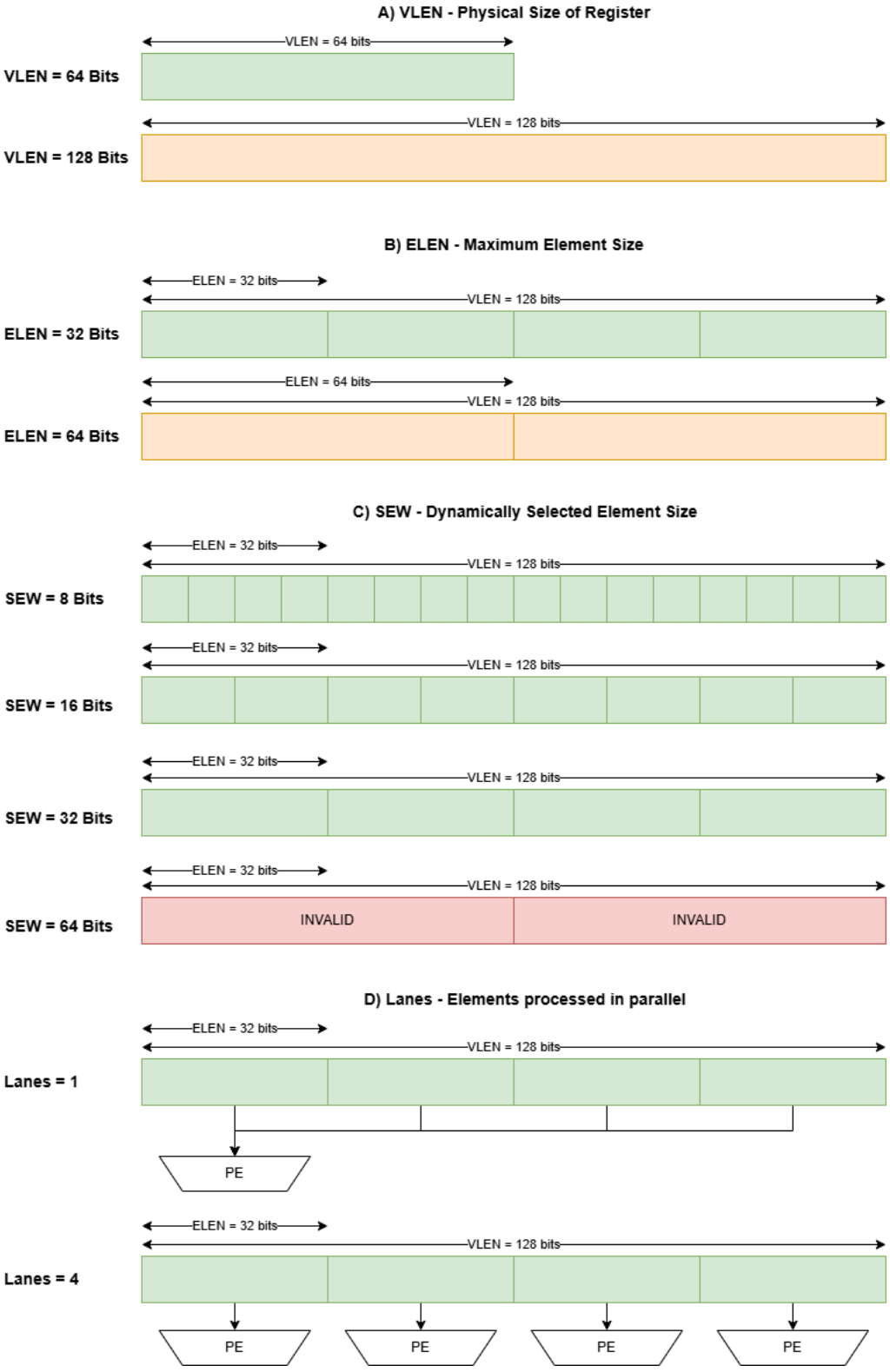


Figure 2.1: RISC-V Vector extension parameters demonstration. The first part explores the VLEN, the second the ELEN, the third the SEW, and the last part introduces Lanes.

processed per cycle. Using fewer Lanes can help keep the area footprint low but requires more multiplexing and/or sliding of operands. More Lanes increase the throughput as the number of elements processed per cycle increases.

The PEs must be capable of processing ELEN-wide elements but can vary in capabilities in different implementations. A PE might support just one element at a time or multiple smaller ones. For example, a PE with 32-bit capabilities might support multi-precision operations, performing 1x 32b operation, 2x 16b, or 4x 8b operations in a single cycle. While other PEs might support just one operation per cycle, independent of element-width.

Equations 2.1 and 2.2 define how VLEN, ELEN, and SEW impact the number of elements a register can hold.

$$\text{Maximum Number of elements in a register} = \frac{\text{VLEN}}{\text{ELEN}} \tag{2.1}$$

$$\text{Number of elements in a register} = \frac{\text{VLEN}}{\text{SEW}} \tag{2.2}$$

The instructions of the RVV extension always start with the “V” prefix followed by the operation name. The operations can have different operands, which are specified by a suffix in the instruction name. The suffix starts with a dot and is followed by the operand types: “V” for vector, “I” for immediate, and “X” for scalar.

The *vector operands* or *results* may occupy one or more vector registers, configuring a register group. A register group is configured using a CSR field called “LMUL”. The CSR field “VL”, which stands for Vector Length, specifies the number of elements the instruction will process in the iteration and depends on the SEW and LMUL values as Equation (2.3) shows:

$$\text{Maximun number of elements in a vector group (VL)} = \frac{\text{VLEN}}{\text{SEW}} \times \text{LMUL} \tag{2.3}$$

A Configuration Instruction is provided to allow rapid configuration of the vector CSRs to match application needs. It changes SEW, LMUL, VL, and tail/mask policies. It is named “vsetvl” and has two variants with immediate operands as Table 2.3 shows. Assembler names for immediate configurations are also shown in Table 2.3.

Code 2.1 demonstrates the shape and use of the vector configuration instructions. The application determines the total amount of elements to process, known as the application vector length (AVL), which serves as a proposed value for VL. In return, the hardware indicates in the *rd* register the (often smaller) number of elements it can manage in each iteration. This value, stored in the VL CSR and referred to as “supported VL", depends on the microarchitectural design and the vtype setting, see Equation (2.3). Code 2.2 presents an example of the returned “supported VL" for a given architecture.

Table 2.3: Vector Configuration Instructions and Assembler names for immediate configurations

Configuration Instruction	**Description**
vsetvl	Vector Configuration Instruction
vsetvli	Vector Configuration Instruction with immediate VL
vsetivli	Vector Configuration Instruction with immediate VL and immediate Configs
Config	**Selected Element Width (SEW)**
e8	SEW=8b
e16	SEW=16b
e32	SEW=32b
e64	SEW=64b
Config	**Register Group (LMUL)**
mf8	LMUL=1/8
mf4	LMUL=1/4
mf2	LMUL=1/2
m1	LMUL=1, assumed if m setting absent
m2	LMUL=2
m4	LMUL=4
m8	LMUL=8
Config	**Tail and Mask Policies**
tu	Tail Undisturbed
ta	Tail Agnostic
mu	Mask Undisturbed
ma	Mask Agnostic

```
1  # t0  = Receives the supported VL (how many elements the hardware supports given the new
       vtype configurations)
2  # t1 = Application VL (AVL)
3  # t2 = new vtype
4  vsetvl t0, t1, t2
5
6  # t0  = Receives the supported VL
7  # 8   = AVL, application vector contains 8 elements
8  # e32 = 32b elements (SEW=32)
9  # m1  = Group size is one register (LMUL=1)
10 # tu  = Tail Undisturbed (Tail elements are untouched)
11 # ma  = Mask agnostic (Masked elements can change)
12 vsetivli t0, 8, e32, m1, tu, ma
```

Code 2.1: Commented assembly with Vector Configuration instructions for demonstration.

The behavior shown in Code 2.2 makes the Vector extension application scalable and independent of VLEN. This means that code compiled for a specific VLEN can run on other platforms that support larger VLENs. Consequently, the RVV extension specification introduces "Minimum Vector Length Standard Extensions" using the "Zvl*" prefix. Thus, code compiled with the extension *Zvl128b* can run on any platform with $VLEN \geq 128$. The value obtained in the destination register (supported VL) can dictate loop control by indicating the number of elements processed in each iteration. Typically, this value is deducted from AVL, and the loop continues until AVL equals 0.

```
# Hardware Configuration:
# VLEN = 128b and ELEN = 32b

# Loads t1 with the AVL (64 elements)
li t1, 64

# t1  = AVL
# m2  = Group size is two registers (LMUL=2)
# e32 = 32b elements (SEW=32)
vsetvli t0, t1, e32, m2, ta, ma

# t0 = Supported VL
# t0 = VLEN/SEW * LMUL
# t0 = 128/32 * 2
# t0 = 8 elements
# With the given hardware configuration, one iteration of a vector instruction will process
    only 8 elements from the 64 requested.
```

Code 2.2: Commented Assembly with Vector Configuration instruction and AVI relationship example.

The CSR field *vstart* specifies the first element to be processed by the instruction, leaving earlier elements in the destination vector undisturbed. Non-zero *vstart* values may cause vector instructions to run substantially slower.

Table 2.4 summarizes the main parameters and configurations used in an RVV implementation and vector operations.

Table 2.4: RISC-V Vector Extension Parameter and Configuration Summary.

Parameter/Configuration	Description	Runtime
VLEN	Vector register size in bits (power of 2)	N
ELEN	Maximum element size (power of 2, $ELEN \leq VLEN$)	N
SEW	Selected element width ($SEW \leq ELEN$)	Y
LMUL	Register group size (1, 2, 4, and 8)	Y
VSTART	Vector's starting index	Y
VL	Vector size in elements	Y

Figure 2.2 exemplifies how two vector registers are used in a group operation. The vector registers *v2* and *v3* are grouped with the *LMUL=2* config. Considering VLEN=128 and the current SEW=32, the amount of elements in each register is 4 ($\frac{128}{32}$, see Equation (2.2)).

Given the previously presented parameters, the register group can hold up to 8 elements (using Equation (2.3): $\frac{128}{32} \times 2 = 8$). The CSR field *vstart=1* setting indicates that the first element to be processed is the element at index 1, which implies that the position 0 of the vector register *v2* will not be touched. The *VL=7* sets the vector length in elements, indicating that seven elements will be processed. As the vector starts in position 1 (*vstart=1*), the elements from 1 to 6 will be processed. Position 7 will not be touched, as the operation will end in position 6.

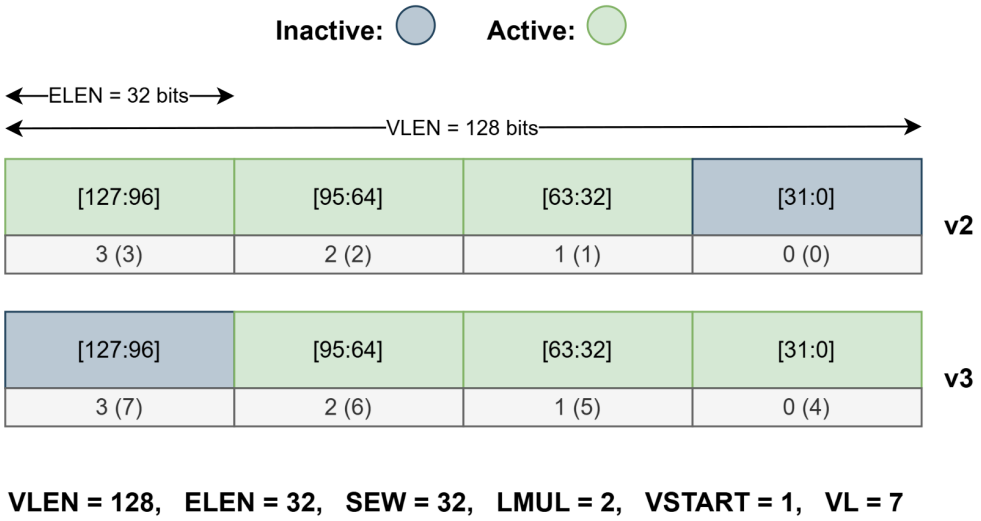


Figure 2.2: Vector registers during operation. Position indexes are represented under the register with parentheses.

Masking is supported on many vector instructions. The mask value used to control the execution of a masked vector instruction is always supplied by the vector register *v0*. Each bit of *v0* represents an element in the register group. Figure 2.3 exemplifies how two vector registers are used in a masked operation, applying a mask to the previously presented example of Figure 2.2.

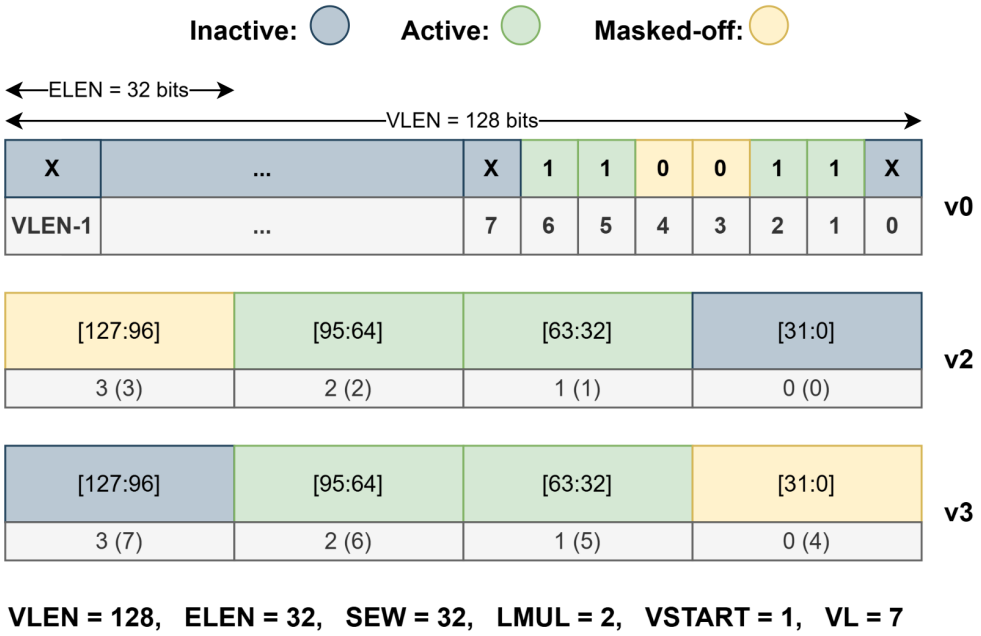


Figure 2.3: Vector registers during operation. Position indexes are represented under the register with parentheses. The "X" values represent don't-cares.

The vector *v0* supplies the mask which will be applied to the vector registers *v2* and *v3*, grouped with the *LMUL=2* config. The selected element width (SEW) is set to 32 bits,

thus each vector register can hold 4 elements (VLEN/SEW). As the vector starts in position 1 (*vstart=1*), the elements from 1 to 6 will be processed. Position 7 will not be touched, as the operation will end in position 6. Two elements are masked-off: the element at the vector index 3, located at the last position of register *v2*, and the element at the vector index 4, located at the first position of register *v3*. The remaining elements are the *active* elements which will generate results.

Loads and stores instructions support three memory addressing modes:

- *unit-stride* - access elements stored contiguously in memory starting from the base effective address.
- *strided* - constant-strided operations access the first memory element at the base effective address and then access subsequent elements at address increments given by the byte offset in the register specified by a scalar register.
- *indexed* - Vector-indexed operations add the contents of each element of the vector offset operand specified by a vector register to the base address to give the effective address of each element.

Subset extensions were proposed to address the numerous instructions defined in the RVV extension. The complete extension, designed for application processors, is named with a single-letter "**V**". It includes all the instructions defined in the specification and requires the scalar core to support floating-point operations with single and double precision.

Smaller extensions intended for embedded use are named with a **"Zve"** prefix, providing varying degrees of vector support. Table 2.5 lists the minimum VLEN for each extension and what floating-point types are needed/supported. They support all vector load and store, integer arithmetic, vector mask, reduction, and permutation instructions.

Table 2.5: Embedded vector extensions.

Extension	Minimum VLEN	Supported SEW	FP32	FP64
Zve32x	32	8, 16, 32	N	N
Zve32f	32	8, 16, 32	Y	N
Zve64x	64	8, 16, 32, 64	N	N
Zve64f	64	8, 16, 32, 64	Y	N
Zve64d	64	8, 16, 32, 64	Y	Y

2.4 Memphis

This Section presents *Memphis-V* (pronounced *memphis-five*), a framework for automatic generation and validation of manycores. Memphis-V stands for Manycore Modeling Platform for Phivers. It allows for designing a NoC-based manycore system surrounded by peripherals. Debugging tools are available to verify hardware and software simultaneously.

The Memphis-V platform is an evolution of Memphis [Ruaro et al., 2019] with a redesigned software stack, RISC-V processors, and the inclusion of a broadcast-capable NoC. Memphis, in turn, originated from Hermes Multiprocessor System (HeMPS) [Carara et al., 2009], but with support for external peripherals. Memphis-V is an open-source framework with code and documentation available at https://github.com/gaph-pucrs/Memphis-5.

Figure 2.4 overviews the Memphis-V platform stack. The bottom of the stack is the hardware, called Phivers, which stands for Processor Hive for RS5 (Section 2.4.1). The middle of the stack is the OS kernel, called MAestro (Section 2.4.2). The software is on the top of the stack, composed of a set of benchmark applications and the *Management Application* (MA). Outside the stack are the generation and debugging tools provided by the Memphis-V framework, which aid in the platform's validation. The Memphis-V platform stack provides a complete manycore model.

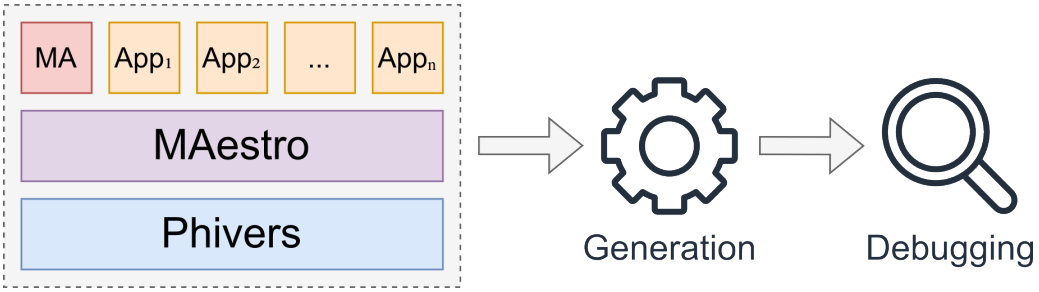


Figure 2.4: Memphis-V platform stack.

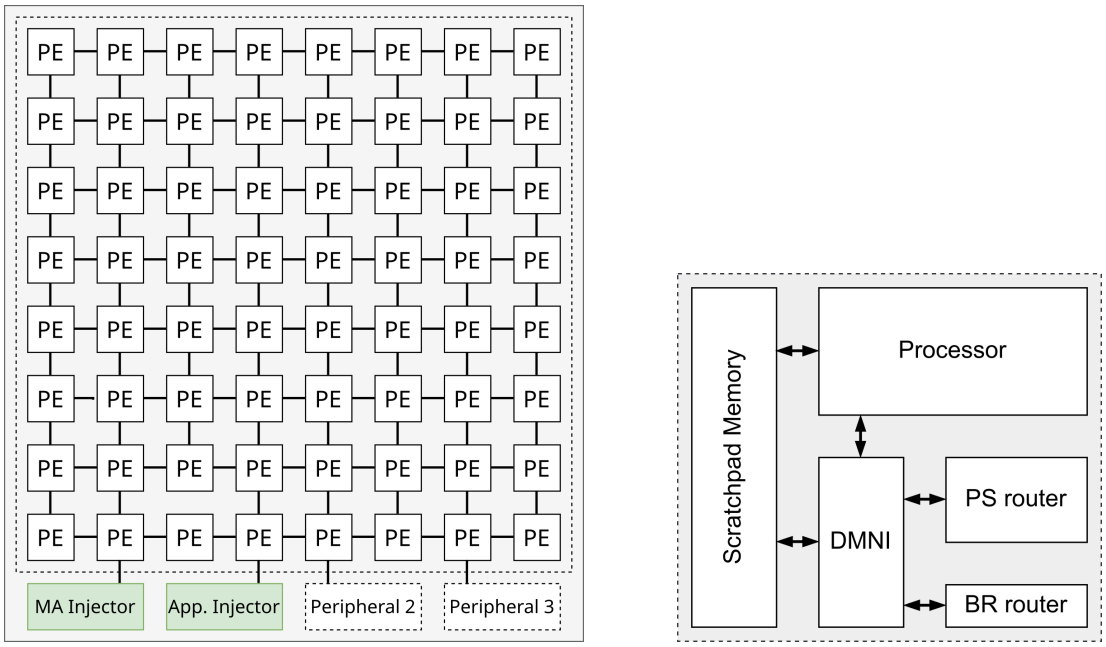
Memphis-V *manages* the system by implementing a Management Application [Dalzotto et al., 2021]. MA transforms the management problem into a distributed application through high-priority tasks, allowing management to benefit from the high parallel power of manycores. *The MA management is out of the scope of this work, with details available in [Dalzotto et al., 2021].*

2.4.1 Phivers: Processor Hive for RS5

Figure 2.5a shows the homogeneous region of Memphis-V with PEs and the external region with peripherals connected to the manycore borders. Two default peripherals are shown in the Figure, *Application Injector* and *MA Injector*. Figure 2.5b pictures the components of each PE, including:

- An RS5 [Nunes et al., 2024] **processor**, detailed in Chapter 4.
- True dual-port **scratchpad memories** for instruction and data.
- A Hermes [Moraes et al., 2004] Packet-Switching (**PS**) **router**. Hermes has XY routing, round-robin arbitration, input buffering, and credit-based control flow.

- A BrLite Broadcast (**BR**) **router**. BrLite is a version of BrNoC [Wachter et al., 2017] without backtracking, thus with a smaller area footprint. BrLite transmits small control and monitoring messages in a single flit. Its broadcast transmission presents low latency and fault tolerance due to its flooding behavior. It occupies approximately 50% of the Hermes router area.
- A Direct Memory Network Interface (**DMNI**) [Ruaro et al., 2016], integrating Network Interface (NI) and Direct Memory Access (DMA) modules.



(a) Memphis-V organization.

(b) Memphis-V PE.

Figure 2.5: Memphis-V manycore overview.

Memphis-V is distributed with two default peripherals: Application Injector and MA Injector. Both peripherals are responsible for dynamic loading tasks into the manycore.

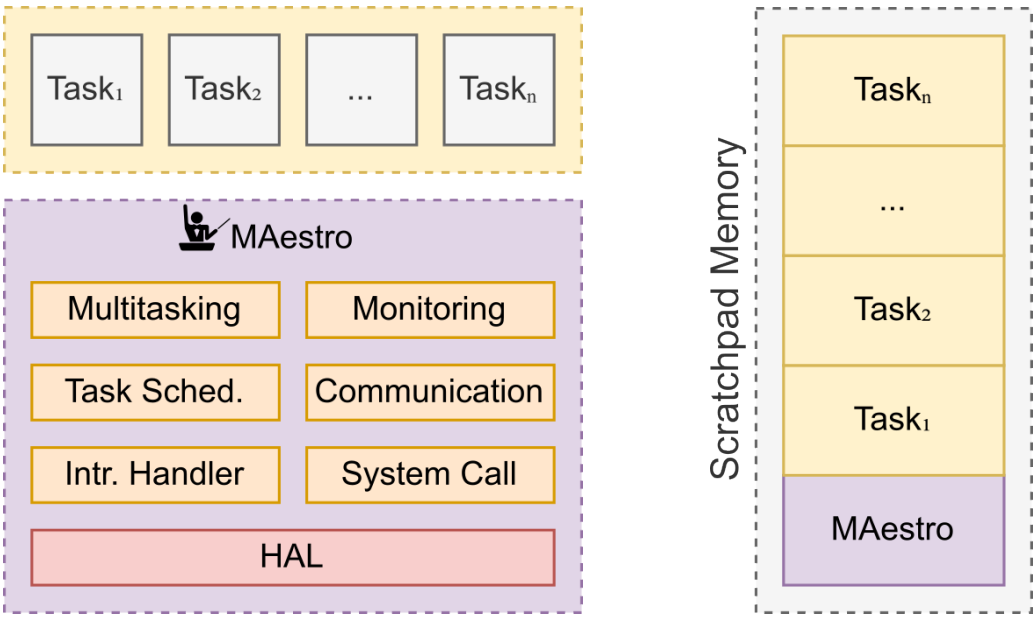
The Application Injector is responsible for deploying applications into the system. The first injection step occurs when the peripheral detects an incoming new application. The Injector sends a message to an MA task responsible for mapping the application, called *Mapper Task*. The message describes the application that will be used by the mapping algorithm implemented by the Mapper Task. After successfully mapping, the Mapper Task answers the peripheral with an array containing the mapped task IDs with their mapped locations. Then, the Application Injector sends the task binaries individually to each mapped PE. The kernel at the target PEs informs the Mapper Task that the task is allocated. When all tasks are allocated, the Mapper Task sends a *task release* message containing all the application task locations back to each allocated PE kernel.

For security reasons, it is necessary to separate the deployment of MA tasks, such as the *Mapper Task*, from user tasks. Thus, the MA Injector is separated from the Application

Injector but uses the same protocol described above. The MA Injector's main functional difference from the Application Injector is injecting the Mapper Task first, sending it to a PE defined at design time.

2.4.2 MAestro: the Management Application Operating System

Figure 2.6 illustrates MAestro, an embedded OS loaded into all PEs. MAestro uses a microkernel design, where all non-core functionalities, such as management, are in separate processes executing at a lower privilege level.



(a) MAestro executes in M-mode, with purple background. User tasks execute in U-Mode, with yellow background.

(b) Memory layout. The memory is organized in equally sided pages.

Figure 2.6: MAestro overview.

The core functionalities of MAestro, highlighted inside the purple rectangle in Figure 2.6a are:

- **Hardware Abstraction Layer (HAL)**: the HAL abstracts the RS5 processor, providing CSR access, memory management, and context switching.
- **System calls**: built on top of newlib-nano [Red Hat, 2024], MAestro provides 6 POSIX system calls to handle standard output, dynamic memory allocation, and process control, along with several manycore-specific system calls.
- **Interrupt handling**: MAestro handles RTC and peripheral interrupts, such as the DMNI that abstracts both Hermes and BrLite routers.

- **Multitasking**: Through Xosvm (Section 4.6), MAestro supports multitasking with dynamic application loading. Figure 2.6b pictures the contiguous memory pages when multiple tasks are allocated into the scratchpad memory.
- **Task scheduling**: MAestro supports preemptive scheduling through timer interrupts provided by the RTC. The built-in schedulers support best-effort tasks through round-robin and soft Real-Time (RT) tasks through Least Slack Time (LST).
- **Communication**: MAestro includes drivers and a protocol stack to enable the communication through the NoC. A Message-Passing Interface (MPI)-like Application Programming Interface (API) provides communication access to user applications. This mechanism supports bidirectional message exchange, including kernel-to-task, peripheral-to-task, and peripheral-to-kernel communication, thereby implementing a message-passing Inter-Process Communication (IPC) system.
- **Monitoring**: MAestro monitors the system through Low-Level Monitors (LLM), which fetch monitored metrics without complex computation. The OS invokes the LLMs periodically, collecting data from, e.g., instruction counters for power and temperature estimation or QoS monitors. Data generated by the LLM is sent through BrLite, and handled by the monitoring framework.

2.4.3 Platform features

Memphis-V provides a set of 14 standard benchmarks to evaluate the manycore. Applications in Memphis are modeled as Communication Task Graphs (CTG). CTG is a model to represent functional parallelism, where an application is composed of independent parts and thus divided into tasks [Rauber and Rünger, 2013]. A graph node represents each task in a CTG, and the graph edges represent the communication between these tasks.

A parallel application is often structured in a pattern effective for different applications [Rauber and Rünger, 2013]. These patterns provide a specific coordination structure for the application. The three main patterns used by Memphis-V applications are:

- **Fork-join**: a task *forks* the workload, splitting the computation between *worker* tasks. Another task awaits for the *workers* termination to *join* the results.
- **Master-slave**: a *master* task controls and distributes the workload to its *slaves*, or *worker* tasks. The *master* is also often responsible for executing the main part of the program and joining the results.

- **Pipeline**: data is forwarded from task to task to perform different processing steps in sequence. Parallelism is achieved by partitioning the data into streams that flow through the pipeline stages.

The OS support library, built on top of newlib-nano, can interface with any user-level application. The Memphis-V library, known as `libmemphis`, offers four procedures to retrieve information about the manycore system, such as the processing element (PE) address where the application is executing, and six message-passing functions. It also includes procedures to assist in the implementation of MA tasks. Additionally, the Memphis-V utilities library, referred to as `libmutils`, provides supplementary data structures to support application development.

Memphis-V has a framework design flow guided by files written in the YAML markup language. A *test case* is a file that describes the features of the platform, and a *scenario* is a file that lists the applications to evaluate and the parameters for the MA. The configuration is interpreted by Python programs that generate the hardware model and build the software stack and applications. The *test case* supports the following parameters:

- **Page size**: the maximum memory a task can occupy for both instructions and data. These parameters will define the scratchpad memory size and the maximum number of tasks in a PE.
- **Maximum number of tasks**: defines the maximum number of tasks per PE. Together with page size, it defines the scratchpad memory size.
- **Manycore dimension**: the number of PEs in the manycore, defined by a 2D mesh.
- **Peripherals**: a list of peripherals connected to the NoC borders.
- **Debugging definitions**: supports seven distinct debugging configurations for simulation, enabling designers to balance debugging granularity and simulation performance.

Memphis-V debugging features allow data extraction at simulation time. This data includes NoC traffic, instructions executed by the processor, memory access log, scheduling status, and task log. The debugging step is aided by a graphical tool that allows visualizing the data extracted and provides several system metrics.

2.5 Convolutional Neural Networks (CNNs)

Machine learning (ML) is a subset of artificial intelligence that enables computers to “learn” and make decisions without being explicitly programmed [Mahesh, 2020]. Essentially, these algorithms use systems to recognize patterns, extract meaningful insights, and adjust

their behavior based on the information they process. This set of algorithms can improve its performance over time, especially during the “training phase” which uses prepared data sets to create the ML model capable of classifying the data. This phase seeks to evolve the model to optimize the results, generally using a metric called “precision”, a value that measures the amount of data recognized/classified correctly. The learning process is iterative and can be done using several techniques, such as supervised or unsupervised learning.

Machine Learning applications require hardware accelerators to handle the considerable computational demands of their algorithms effectively. The increase in computational capabilities, commonly supported by GPUs and dedicated hardware, has enabled the training of complex models like deep neural networks (DNN). This enhanced computing power has speedup both the automatic training phase and the subsequent inference phase of DNN models in their intended applications [Silvano et al., 2023].

Convolutional Neural Networks (CNNs) are a specialized type of DNN designed to process grid-like data, such as images. Unlike traditional neural networks, CNNs use convolutional layers to automatically detect spatial features, reducing the need for manual feature extraction. This architecture makes CNNs highly effective for tasks like image recognition, object detection, and image segmentation.

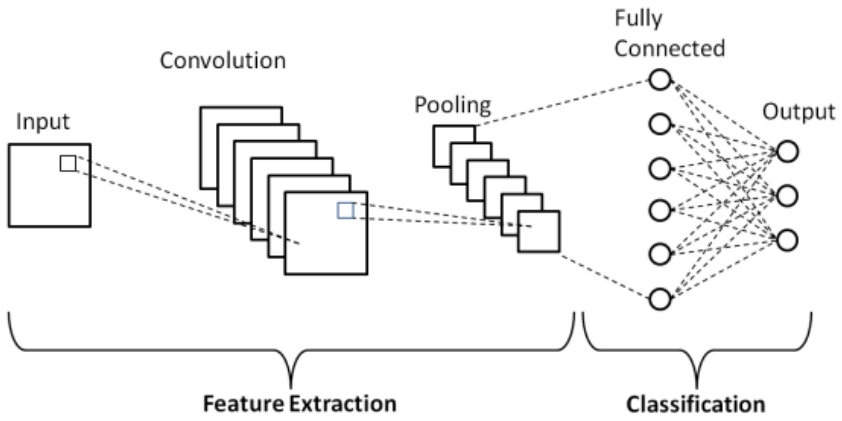


Figure 2.7: Convolutional Neural Network structure(source [Phung and Rhee, 2018]).

Figure 2.7 illustrates the general architecture of a CNN, which contains the following components:

- **Convolution Layer**: the heart of CNNs. It performs multiplications and summations of input values. Convolution employs filters (often referred to as kernels), which restrict these operations to specific matrix windows. Each filter consists of weights, known as parameters. Additionally, convolution incorporates a variable called stride, representing how many positions the filter shifts over the input matrix. Figure 2.8 depicts how a convolution operation is performed using a 3x3 kernel;
- **Activation Function**: a non-linear function that facilitates the classification process in neural networks. This function is applied after a convolution operation to introduce non-

linearity into the model. Commonly used activation functions include the hyperbolic tangent (*tanh*), the exponential function, and the Rectified Linear Unit (ReLU);

- **Pooling Layer**: this layer reduces the amount of data for processing. In contrast to convolutional layers, pooling layers lack parameters and instead consist of operations. The most common operations are Average Pooling and Max Pooling. Average Pooling calculates the average values within a window, whereas Max Pooling identifies the highest value. Figure 2.9 represents a max pooling using 2x2 filter size and stride 2;
- **Padding**: a technique to adjust the input size for a layer with specific configurations. This involves appending rows and columns of zeros around the original input map, resulting in a new map with dimensions that are compatible with the subsequent layer (ensuring the map size aligns with the stride and filter size);
- **Fully Connected (FC)**: the FC layer is used at the end of a CNN, connecting all preceding output layers to each input for the FC. The output from this layer delivers the classification result.

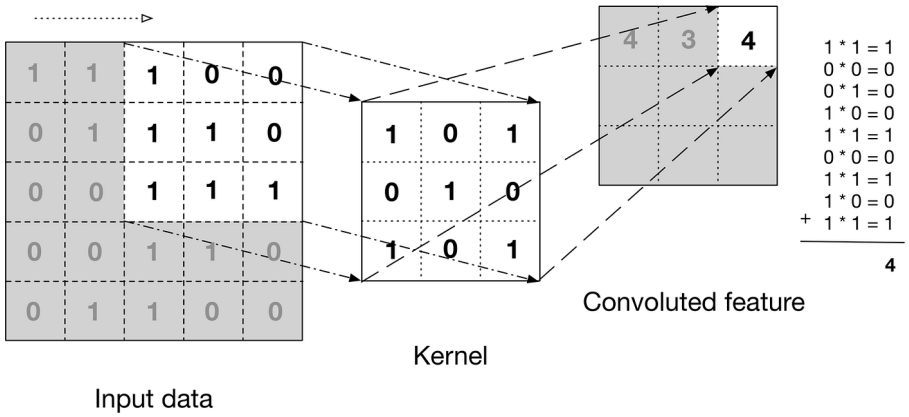


Figure 2.8: Representation of a Convolution Operation using a 3x3 filter/kernel (source [Patterson and Gibson, 2017]). Input data is convoluted using the kernel to generate a point in the convoluted feature.

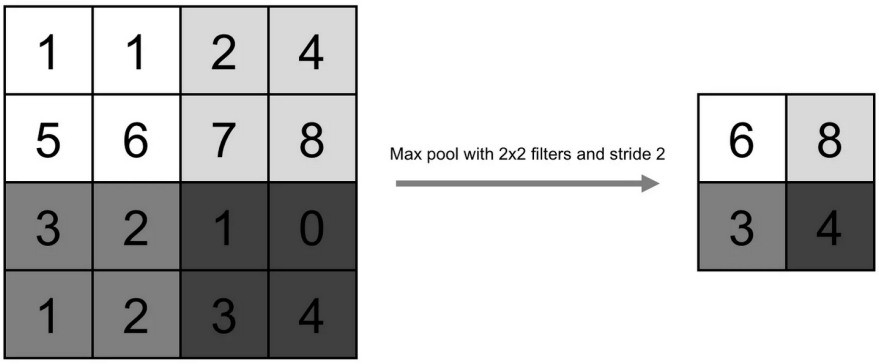


Figure 2.9: Representation of a Max pooling operation using 2x2 filter size and stride 2 (source [Rahman et al., 2022]).

A step of a convolution layer is shown in Figure 2.8. Multiplications are performed between the weights of the kernel and the input feature map (IFMAP), the results of the multiplications are then accumulated to generate a complete convolution value. A complete convolution layer involves convoluting the entire IFMAP to generate the output feature map (OFMAP), using the stride to slide over the IFMAP after each convolution step. Also, to generate the OFMAP, each of the resulting values of each convolution step is added to a bias value and applied to the activation function. The input of a convolutional layer can comprise multiple IFMAPs, which in this case can be called *channels*, and is demonstrated in Figure 2.10. Input channels can differ across networks, typically for RGB images there are 3, with each channel representing a color. Generally, OFMAPs feature several channels, with the quantity specified uniquely for each network and layer, since the OFMAP from one layer serves as the IFMAP for the following layer. For each OFMAP channel, there is an associated set of filters/kernels which are used in the convolution with the IFMAP. So, with support of Figure 2.10 we can conclude:

- An IFMAP with *C* channels uses a set of *C* filters/kernels;
- The amount of *Output Channels* of the OFMAP defines the number of sets (*C-wide*) of filters/kernels, thus *# Filters = Output Channels*;
- Each set of filters/kernels will generate one *output channel* of the OFMAP.

.

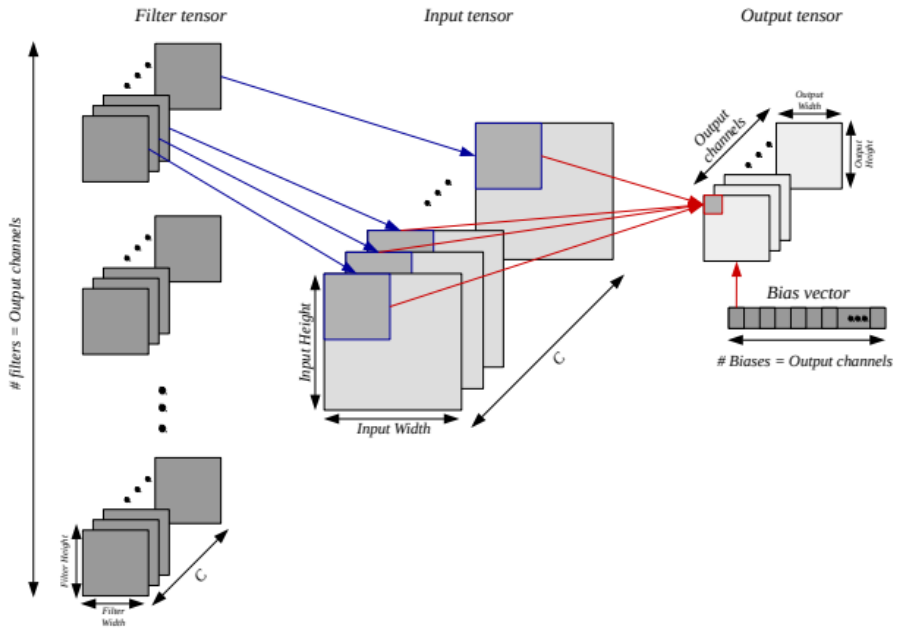


Figure 2.10: The convolution operation in CNNs (Source [Korol, 2019]).

Formally, Equation (2.4) describes the convolution.

$$\mathbf{O}[c_o][x][y] = \mathbf{B}[c_o] + \sum_{k=0}^{C_i-1} \sum_{i=0}^{\text{Width}-1} \sum_{j=0}^{\text{Height}-1} (\mathbf{I}[k][Sx+i][Sy+j] * \mathbf{W}[c_o][i][j]) \quad (2.4)$$

where: c_o is the current output channel; x and y are the horizontal and the vertical positions; C_i is the total number of input and filter channels; Width and Height correspond to the filter size; S is the stride; O is the output; I is the input; W is the filter tensors; and B is the bias vector.

The initial work to utilize a trainable Convolutional Neural Network was LeNet [LeCun et al., 1989]. In 1998, the architecture introduced by LeCun et al. comprised a seven-layer deep structure capable of recognizing handwritten digits using 60,000 parameters.

In 2012, AlexNet won the ImageNet competition [Krizhevsky et al., 2017], marking a significant milestone in neural network development. Its architecture introduced greater depth and a higher number of parameters than LeNet. AlexNet consists of five convolutional layers employing filters of sizes 11x11, 5x5, and 3x3, and includes 60 million parameters. It achieved a Top-5 error rate of 15.3% on the ImageNet dataset, initiating a new era of neural network research. Subsequent works have explored diverse architectures built on similar foundational concepts, including **GoogLeNet** [Szegedy et al., 2015], **ResNet** [He et al., 2016], and **MobileNet** [Howard et al., 2017].

The benefits of CNNs in classification tasks have resulted in the creation of frameworks that assist developers in constructing their models by providing the necessary mechanisms for training and inference [Juracy et al., 2021]. Examples of frameworks include Pytorch [PyTorch, 2024] and TensorFlow [TensorFlow, 2024]. These frameworks offer libraries for developing Machine Learning applications, including CNNs, which simplify the training and inference phases using high-level programming languages like Python.

2.6 Final Remarks

This Chapter introduced key topics related to this Dissertation. The RISC-V Vector (RVV) extension emerged as a promising candidate for a hardware accelerator due to its ability to exploit data parallelism in a SIMD manner. Its popularity stems from the widespread adoption of RISC-V cores and the open-source nature of the RISC-V ecosystem, which has enabled the community to play an active role in its development.

Section 2.5 presented how a CNN is structured and its main components, while Equation (2.4) detailed the operations executed by the convolutional layers, highlighting the Multiply and Accumulate (MAC) operations, which usually are costly to be executed in hardware, taking several cycles if handled through software. This type of operation is well-suited

for hardware acceleration, mainly due to its focus on pure arithmetic tasks. The RVV extension provides numerous variations of MAC instructions, positioning it as a viable option for accelerating CNNs. The RVV extension's runtime programmability enables the core to efficiently execute varying layers by reconfiguring the accelerator based on kernel size and vector length. Masked operations allow operations on data windows, and the load-store addressing modes provide good support for non-contiguous data access.

Due to its modular design, the RS5 processor is well-suited for testing the RVV extension as a CNN accelerator. Its architecture allows users to activate or deactivate various extensions through design-time parameters, enabling them to select an optimal configuration for specific applications and facilitating comparative analyses.

MEMPHIS-V offers a framework for easily configuring manycore platforms and numerous features for launching and debugging parallel applications. Allied with RS5 modular characteristics, the MEMPHIS-V platform becomes a powerful tool for manycore validation.

The CNN channels (depth dimension) can be significantly large (Alexnet's first three layers have 96, 256, and 384 channels, respectively), offering a good opportunity for parallelism because each channel calculation is independent. A CNN running in a scalar core can have a long runtime for a single inference and needs a significant memory size to hold the parameters necessary to run the network. Distributing channels across PEs in a manycore environment effectively addresses both issues: each core manages a segment of the network parameters while parallel processing accelerates the inference process, as shown in the following chapters. When combining parallel processing with vector processing capabilities, both Data Level Parallelism and Task Level Parallelism are explored simultaneously, greatly influencing the time required for inferences.

3. RELATED WORK

This Chapter presents related work in the context of RISC-V hardware accelerators targeting many domains:

- Section 3.1 presents a historical view of SIMD and vector accelerators, presenting the first computers to use this approach and covering INTEL, ARM, and NVIDIA solutions.
- Section 3.2 is dedicated to RISC-V hardware accelerators; it is divided into works that use the standard RVV extension (Section 3.2.1), custom SIMD or vector extensions (Section 3.2.2), and those using dedicated RISC-V processors to control the accelerator (Section 3.2.3).
- Section 3.3 presents the state-of-the-art for manycore hardware acceleration techniques. It is divided into multi-core-based and NoC-based accelerators.
- Section 3.4 concludes the Chapter, positioning this Dissertation w.r.t. the state-of-art.

3.1 SIMD and Vector Accelerators

As Chapter 2 introduced, the use of SIMD and vector acceleration techniques is not new. It started in the early days of computer developments, exploring SIMD techniques in processors like the ILLIAC IV and the CRAY-1. It evolved and arrived at the general-purpose processors through extensions like Intel's MMX (Multimedia Extensions) and Advanced Vector Extensions (AVX).

ILLIAC IV [Barnes et al., 1968] was developed in the 1960s, as a parallel-array computer containing 256 processing elements. Its architecture incorporated SIMD instructions with special features, including multi-array processing, multi-precision arithmetic, and fast data-routing interconnections. The vector processing capabilities of Illiac IV further increased its performance, leading to advancements in scientific and engineering simulations. Despite facing programming complexity and maintenance challenges, its contributions laid the groundwork for subsequent developments in parallel computing architectures.

The first supercomputer to successfully implement a vector processor was **CRAY-1** [Russell, 1978]. It was announced in 1975, and the first unit was installed at Los Alamos National Laboratory in 1976. Eighty Cray-1s were sold, making it one of the most successful supercomputers in history. It could support 138 million floating-point operations per second (MFLOPS) for sustained periods and even higher rates of 250 MFLOPS in short bursts. It features eight 64-element vector registers, each element being 64 bits wide (summing 4096 bits).

In the mid-1990s, Intel [Intel Corporation, 2025] introduced the MMX (Multimedia Extensions), the first set of SIMD instructions implemented in their processors. MMX was de-

signed to accelerate multimedia and communication applications [Peleg and Weiser, 1996], featuring 64-bit registers that overlapped with the floating-point unit registers. In the early 2000s, Intel launched the SSE (Streaming SIMD Extensions), which addressed some of MMX's shortcomings. SSE introduced eight new 128-bit registers. Subsequent versions (SSE2, SSE3, and SSE4) further expanded SSE, adding new instructions and capabilities to enhance performance in multimedia and scientific applications. Intel's Advanced Vector Extensions (AVX) [Lomont, 2011], introduced in 2011, represented a significant leap in SIMD processing. AVX expanded the register size to 256 bits, enabling the simultaneous processing of more data per instruction. AVX also introduced new instructions optimized for complex mathematical computations and scientific workloads, making it highly suitable for data analysis, machine learning, and simulation applications. Subsequent iterations, such as AVX2 and AVX-512, enhanced the extension's capabilities by supporting a wider range of data types and increasing parallelism.

ARM Neon was introduced by ARM [ARM, 2025] as part of the ARMv7-A architecture in 2009, focusing on SIMD operations for multimedia and DSP applications. NEON was ARM's response to the growing demand for efficient processing in mobile and embedded devices, where power efficiency is critical. With the introduction of **SVE (Scalable Vector Extension)** in ARMv8-A, ARM moved towards scalable and flexible vector processing, targeting HPC (high-performance computing) and AI workloads.

The **ARM Neon** extension can accelerate the performance of multimedia applications such as 3-dimensional graphics, image processing, and deep learning. There are 32 64-bit vector registers, D0 – D31, known as Double (D) registers. Two consecutive D registers can be combined into a Quad (Q) register. The bit width of the data in a vector register can be set as 8, 16, 32, or 64, which means a vector register can accommodate multiple data. Zhang et al. [2017] showed that Neon can offer a 5x speed-up for an edge detection algorithm.

ARM Scalable Vector Extension (SVE) [Stephens et al., 2017] is designed for High-Performance Computing (HPC) and shares similarities with the RVV extension, such as making the vector register length implementation-dependent. The minimum word length is 128 bits, and the maximum is 2048 bits. SVE can achieve speedups of up to 3×, even when the vectors are the same size as those in Advanced SIMD.

ARM also features the **M-Profile Vector Extension (MVE)**, known as "Helium", designed to quadruple processing speed with a twofold increase in datapath width. Helium delivers significant performance improvements for ML and digital signal processing applications.

Alternative methods for SIMD techniques find applications in Digital Signal Processing (DSP). The **ARM Cortex-M4** and **Cortex-M7** microcontrollers are known as Digital Signal Controllers (DSC) [ARM, 2016]. The DSC microcontrollers provide a blend of traditional Microcontrollers (MCU) and DSP functionality in a single instruction set using

general-purpose registers to support DSP extensions. This approach can reduce system-level complexity by removing the need for shared memory, MCU and DSP communication, complex multiprocessor bus architectures, and other custom “glue” logic between the MCU and DSP [ARM, 2016]. It also reduces software development costs, as a single compiler/debugger/IDE can support the entire project. The DSP ISE supports 8-, 16-, 32- and 64-bit types. These processors provide SIMD instructions that operate on 8- or 16-bit integers. It offers MAC, saturation additions/subtractions, and other SIMD instructions. All registers are still 32-bit wide, but the SIMD instructions operate simultaneously on 2 x 16-bit values or 4 x 8-bit values within a 32-bit register.

NVIDIA [NVIDIA Corporation, 2025] started its journey in SIMD and vector computing, emphasizing graphics, eventually evolving into a more expansive vision of parallel processing computing. NVIDIA’s CUDA marked a turning point, evolving GPUs from specialized tools into versatile computing powerhouses. NVIDIA utilizes SIMD principles on a large scale, positioning itself as a leader in parallel computing and influencing AI, HPC, and other areas.

NVIDIA CUDA is a proprietary parallel computing platform that allows software to use certain types of GPUs, originally designed for computer video cards, for general-purpose processing. *CUDA cores* are the processing units of the NVIDIA GPUs and are the fundamental blocks for CUDA; they are programmable parallel processors capable of handling a wide range of workloads in a SIMD manner [Sanders, 2010]. CUDA is designed to work with programming languages such as C, C++, and Fortran, making the development of applications to run in this environment much more straightforward than previous tools.

3.2 RISC-V hardware accelerators

The RISC-V vector extension, introduced in Section 2.3, was adopted by several processors, either as a coupled accelerator or as a coprocessor. This extension can target many applications, leading to different implementation approaches. Due to its large instruction set and register bank, the RVV extension can add an important area overhead to the processor core. On the other hand, custom SIMD/Vector accelerators can have more freedom to include only the needed functions without complying with any directive from a standard extension specification. In exchange, custom accelerators need custom software support, often with modified compilers, which can bring extra complexity to the implementation.

This section is organized as follows:

- Section 3.2.1 presents works that use the standard RVV extension as an accelerator. It starts by introducing commercial cores and then presents academic works.

- Section 3.2.2 presents custom SIMD/Vector accelerators that use different approaches to obtain the best domain-specific acceleration results.
- Section 3.2.3 closes the section by presenting works that use RISC-V processors to control complex hardware accelerators.

3.2.1 RVV Accelerators

This section presents related work regarding implementations compliant with the RVV extension. It initially presents commercial cores and then introduces academic works.

Xuantie-910 [Chen et al., 2020] is a commercial high-performance Multi-Core Processor based on the RV64GCV instruction set. Every single core has a 12-stage deep pipeline, out-of-order, multi-issue superscalar architecture depicted in Figure 3.1. It supports all three privilege modes, achieving a maximum clock frequency of 2.5 GHz in TSMC 12nm FinFET, and runs in a Xilinx VU9P FPGA at 200 MHz.

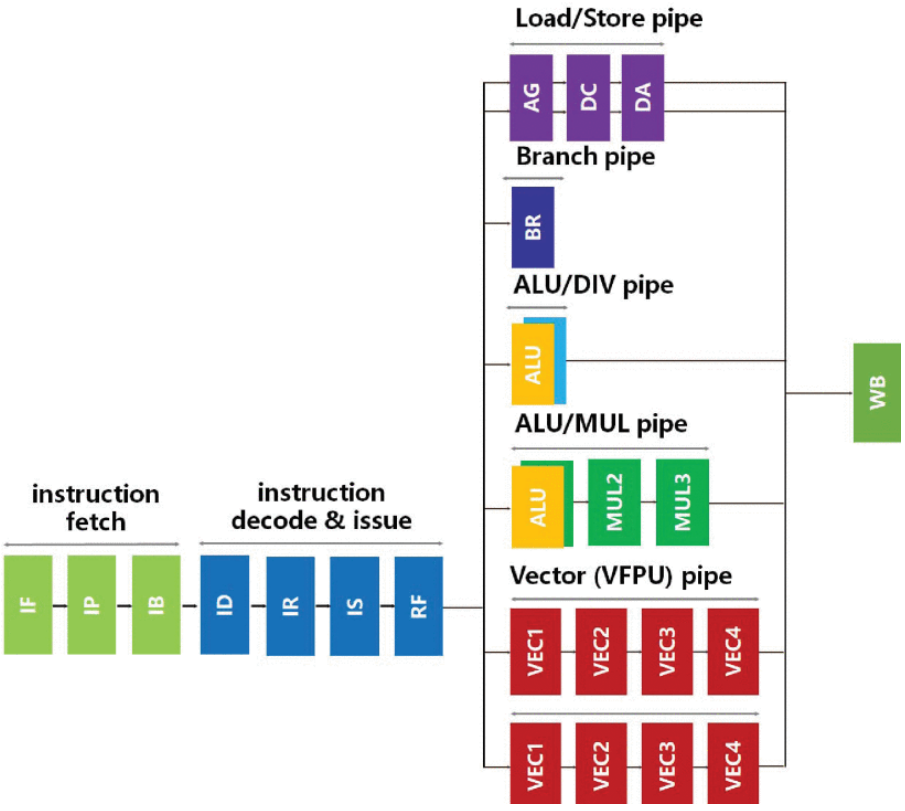


Figure 3.1: 12-stage pipeline in Xuantie-910 core [Chen et al., 2020].

Xuantie-910 implements over 50 non-standard instructions to accelerate various tasks and performs 7.1 CoreMark/MHz. Each core features a 32/64 KB L1 instruction cache and a 32/64 KB data cache. It also supports a standard 8-16 region Physical Memory Protection (PMP) and is compatible with SV39 MMU following the RISC-V Linux specification. The processor includes a Core-Local Interrupt Controller (Clint) and Platform-Level Interrupt

Controller (PLIC), supporting multi-core Symmetric Multi-Processing (SMP) with cache coherence. Each cluster, consisting of 1 to 4 cores, can boot the Linux operating system. Up to four CPU clusters can connect through Ncore, an internal bus with a coherence protocol. They utilize an inclusive L2 cache, which can be configured up to a maximum size of 8 MB. Xuantie was one of the earliest commercial processors to implement the RISC-V Vector Extension. This extension facilitates dual-issue out-of-order vector operation instructions, accommodating operational widths from 64 to 1024 bits. Two 128-bit vector slices are recommended. This configuration allows the XT-910 to execute 256-bit operations in one clock cycle and efficiently perform 128-bit vector load/store operations.

SiFive offers a RISC-V Core IP portfolio with customization options that allow customers to select the best fit for their needs. The portfolio ranges from simpler processors, like *E31* [SiFive, Inc, 2017a] and *E51* [SiFive, Inc, 2017b] to high-end performance cores, like the *P270* [SiFive, Inc, 2022] and *X280*. It is classified into Essential, Performance, Automotive, and Intelligence families. The Intelligence Family includes high-performance AI dataflow processors with scalable vector compute capabilities.

Sifive Performance P270 [SiFive, Inc, 2022], Figure 3.2, is an 8-stage, dual-issue, in-order pipeline compatible with the RISC-V RV64GBCV ISA. It fully supports the RISC V Vector Extension 1.0. It is combined with the SiFive Recode utility, which translates existing SIMD software from popular legacy architectures to RISC-V Vector assembly code. The Vector operations are decoded and queued in the Vector Unit for parallel operation of Scalar and Vector units. The vector processing supports 256-bit VLEN and 64-bit ELEN. The Vector ALU supports integer and floating point operations and is 128b-wide, capable of performing 2x64b, 4x32b, 8x16b, or 16x8b operations per cycle. Multicore and multi-cluster processor configurations are supported for up to 8 cores.

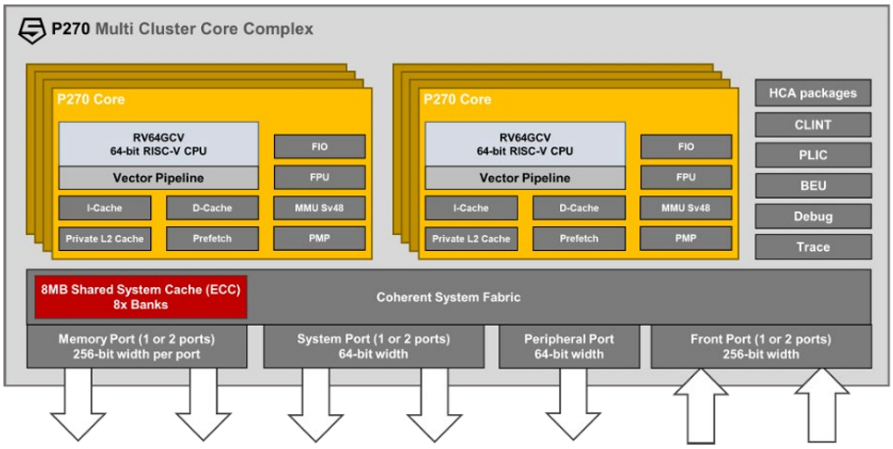


Figure 3.2: Sifive P270 multi-cluster core block diagram.

The **SiFive Intelligence X280** [SiFive, Inc, 2023], Figure 3.3, and **SiFive Intelligence X380**, are multicore RISC-V processors with RISC V Vector Extension 1.0 and custom SiFive Intelligence Extensions optimized for AI/ML compute at the edge. It is a 64-bit

RISC-V, 8-stage, dual-issue, in-order pipeline, Linux capable. It uses tailored instructions to enhance Neural Network calculations and features an Optimized TensorFlow Lite implementation with numerous Neural Network models adapted. It has a 512-bit vector register length processor with variable length operations in a decoupled vector pipeline supporting integer and floating point data types. Multicore and multi-cluster processor configurations are supported for up to 8 cores.

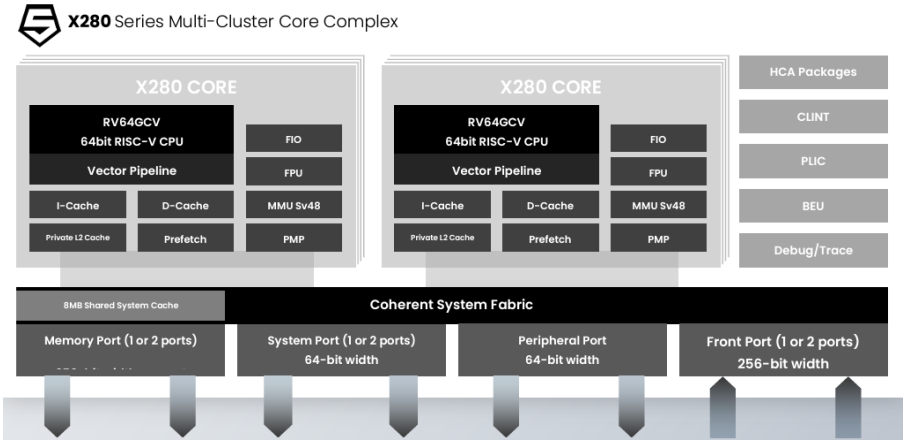


Figure 3.3: Sifive X280 multi-cluster core block diagram.

Arrow [Assir et al., 2021] is a configurable hardware accelerator architecture (**co-processor**) that implements a subset of the v0.9 RVV extension.- Its block diagram is shown in Figure 3.4. Arrow is implemented in a Xilinx XC7A200T-1SBG484C FPGA. The architecture of Arrow supports dual-lane, a VLEN of 256 bits and an ELEN of 64 bits. It can execute two independent vector instructions in parallel and is pipelined to increase execution throughput. Its pipeline stages include decoding, operand fetching, execution or memory accessing, and write-back. When the destination register is within registers 0 to 15, the vector instruction is executed in the first lane; otherwise, it is executed in the second lane. This lane-dispatching scheme simplifies the design and removes the need for complex arbitration hardware. Arrow can support wider lane implementations, but the compiler or assembly programmer must expose vector instruction parallelism through register allocation, similar to statically scheduled superscalar processors. Each bank has two read ports and one write port, with all memory accesses being 64 bits. The accelerator utilizes the ARM AMBA AXI bus interface for communication.

ARA is a 64-bit vector processor initially developed based on version 0.5 of the RVV extension [Cavalcante et al., 2019] and was later updated to utilize version 1.0 [Perotti et al., 2022]. Both versions were implemented targeting Global Foundries 22FDX FD-SOI technology. Ara serves as a **coprocessor** for CVA-6 (Ariane), as depicted in Figure 3.5. In this setup, vector instruction decoding occurs within ARA, rendering Ariane almost agnostic to vector instructions. Each lane in ARA comprises three execution units: an integer ALU, an integer MUL, and an FPU, all operating on a 64-bit datapath. It supports multi-precision ALU and MUL operations, producing 1×64, 2×32, 4×16, and 8×8-bit signed or unsigned

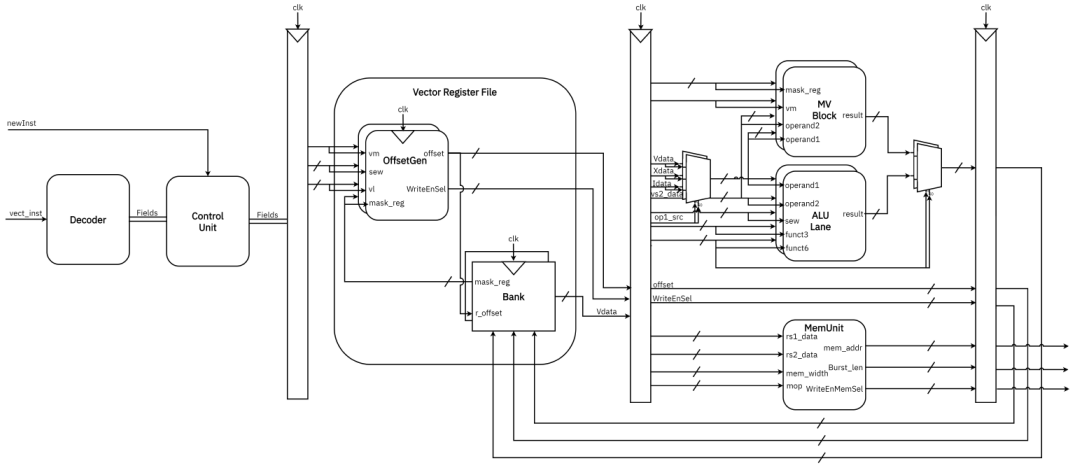


Figure 3.4: Arrow datapath block diagram [Assir et al., 2021].

operands. A series of benchmarks, including matrix multiplication and convolutions, were executed. The accelerators demonstrated optimal performance with two lanes, but some performance degradation was observed with sixteen lanes.

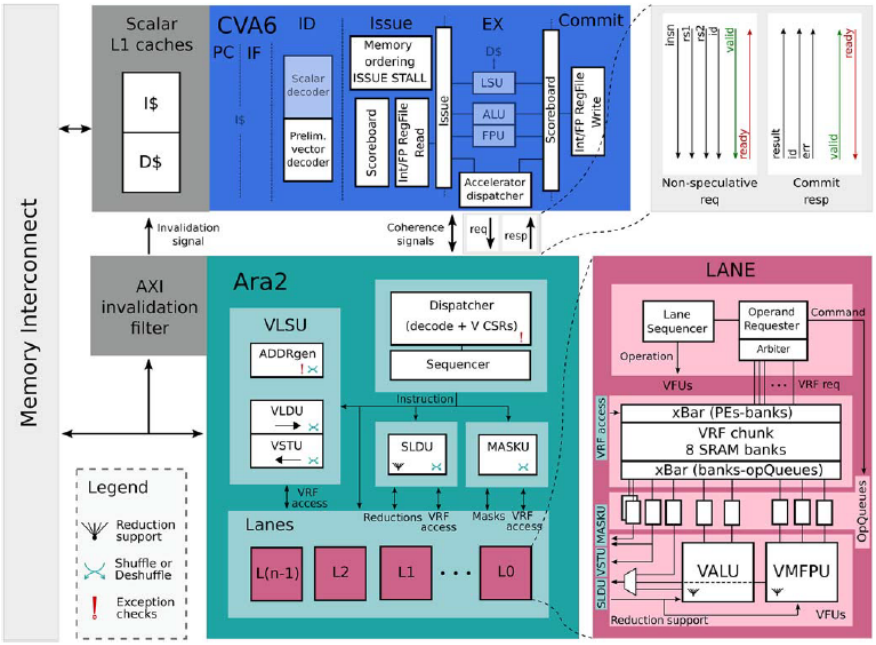


Figure 3.5: ARA block diagram and integration with Ariane [Perotti et al., 2024].

ARA2 [Perotti et al., 2024] extend ARA's previous work. They implemented support for all the RVV 1.0 instructions not supported in the previous version and described the micro-architectural optimizations that allow the architecture to scale up. There were improvements in decoding, slide-unit, and CVA-6 interfaces. The mask unit was introduced to control the mask over all the lanes, which leads to routing complexity, especially when scaling up the number of lanes. Doubling its number of lanes also doubles its AXI bus data width and the parallelism of the all-to-all internal units (MASKU, SLDU, Vector Load/Store Unit (VLSU)).

Pircher et al. [2021] explores the RVV extension version 0.9 to accelerate the quantum-resistant code-based Classic McEliece scheme. This study primarily focuses on the Gaussian Elimination Algorithm (GEA), a critical component in the key generation process of the McEliece scheme. GEA exhibits significant potential for acceleration through the vector instructions provided by the RVV extension. An instruction set simulator, ETISS, augmented with a SoftVector library, was employed to support simulations of cores with the RVV extension. The reference architecture for this research was the RTL model of OpenHW Group's CVA6 ARIANE. The processing of vector instructions is facilitated by a dedicated four-stage vector pipeline equipped with resources such as a vector load/store unit (VLSU) and a vector arithmetic logic unit (VALU). This setup allows loading one row of a 1664 x 8192-bit matrix into a vector register group with a single load instruction. The overall performance gain is influenced by the memory interface of the vector unit. However, the performance gains achieved for the GEA algorithm with RVV are significantly constrained by memory bottlenecks.

Some works seek to explore the RVV extension in an embedded environment, where resource utilization must be minimized. To that end, subsets of the V extensions are implemented. Another strategy for reducing the area footprint is to reduce the number of vector registers in the Vector Register File (VRF). This strategy is inspired by the RV32E ISA, which reduces the number of integer registers in the base ISA from 32 to 16. Jacobs et al. [2024] use smaller VRFs with 16 and 8 registers. One example of the impact of the VRF size is that a VRF with eight 64-bit registers has 512 bits, the same number of bits as the RV32E register file. They observed that the full RVV 1.0 VRF (32x 64-bit registers) could take up about 30% of the overall area in small processor designs. Halving the vector register file to 16x 64-bit registers or reducing it to a quarter size of 8x 64-bit registers can lead to savings of around 15% to 23% of the processor's area. Their analysis indicates that operations such as dot-product and matrix multiplication can be effectively executed using fewer than 32 vector registers without requiring a new programming model.

Johns and Kazmierski [2020] present the implementation of a minimal RISC-V vector processor targeting embedded systems, which also includes only 16 vector registers. The design targets microcontrollers using the RISC-V Vector extension specifically for small embedded devices. The core of the RISC-V processor is an in-order, single-stage RV32E. It implements a subset of the vector extension, comprising only the essential arithmetic, memory, move, and slide instructions for vector functionality. The vector processing unit is **tightly-coupled** with the processor as scalar functionality, as illustrated in Figure 3.6. The design introduces a smaller solution based on variable-width processing units (PUs), where each PU functions is an independent scalar integer ALU. In a single cycle, it can execute one 32-bit, two 16-bit, or four 8-bit operations. This efficient design does not require four 32-bit lanes; instead, it utilizes only a single 32-bit PU, one 16-bit PU, and two 8-bit PUs operating

on one register per cycle. Tests, written in assembly and optimized for this architecture, were implemented on an Intel Cyclone V 5CSEMA5F31C6 FPGA at 50MHz.

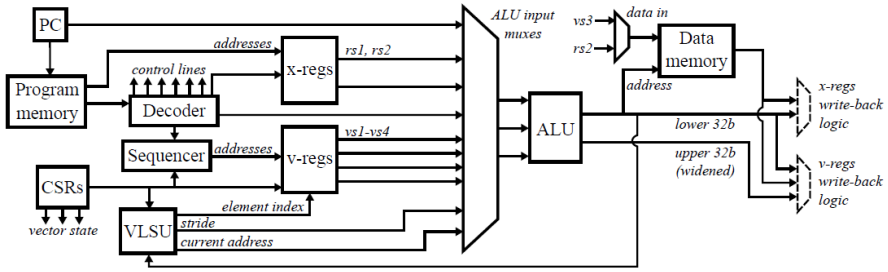


Figure 3.6: Block diagram of minimal vector processor [Johns and Kazmierski, 2020].

Ali et al. [2021] present a Vector Processing Unit (VPU): a RISC-V based SIMD **co-processor** for embedded processing integrated with the CV32E40P. It is a parameterizable VPU that implements a subset of the v0.1 of the RVV extension for embedded processing. The VLEN and number of lanes are configurable. The VPU uses an auxiliary processing unit (APU) interface, which can be easily integrated into other APU-enabled processors. ELEN was set to be 32-bit. The VPU architecture, shown in Figure 3.7, is a two-stage pipeline: instruction fetch and decode stage (fd-stage) and instruction execution and write-back stage (ew-stage) with two main components: execution block and load store unit. Each lane owns a part of the vector register file, and its length is called lane length (LLEN). The lane operates on LLEN/ELEN and requires at least many cycles to complete an instruction. The VALU design is similar to CV32E40P ALU. Load and store serializes the VLEN register data to 32-bit data chunks. It was implemented on a Xilinx Zynq xc7z020clg484-1, each lane utilizing three DSP blocks.

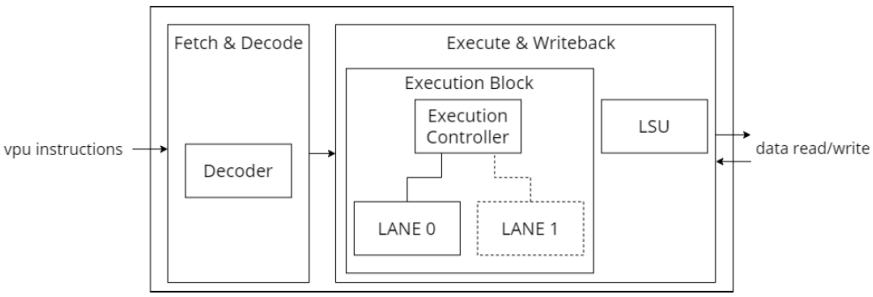


Figure 3.7: Ali et al. [2021] VPU pipeline overview.

ZeroVex [Zhao and Ye, 2024] is a scalable and high-performance RVV processor core. The architecture employs a portion of the RVV v1.0 extension coupled to a 32-bit processor implemented in SystemVerilog, which is used for integer operations. It can accommodate a power of 2 lanes, each with 32x 32-bit vector registers. Each lane has a SIMD ALU and a SIMD multiplier. Figure 3.8 depicts the organization of ZeroVex. Incorporating the vector function results in a substantial and directly proportionate area growth and a propor-

tional growth in power consumption. As a trade-off, the vector core exhibits approximately 30 times more energy efficiency than the scalar core.

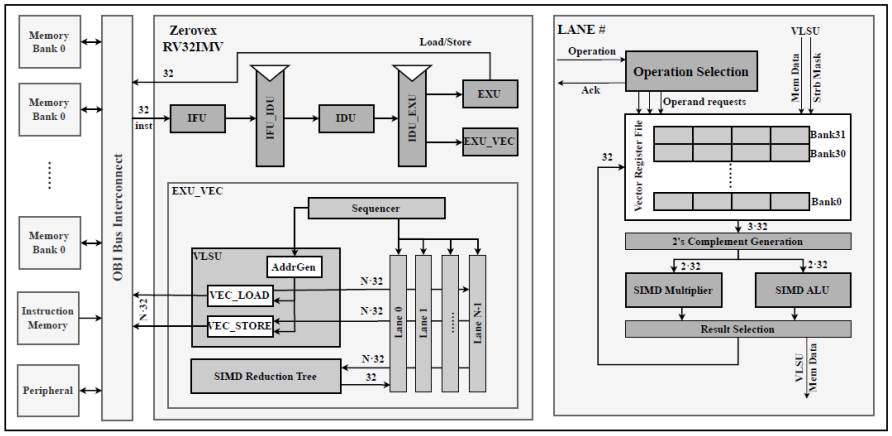


Figure 3.8: ZeroVex [Zhao and Ye, 2024] top-level diagram.

SPEED Wang et al. [2024] is a scalable RVV processor optimized for multi-precision inference in DNNs, featuring customized instructions, hardware design, and dataflow mapping for efficiency. It is tightly coupled to a RISC-V scalar core for programmable instructions and an external memory for fetching necessary data. Each PE consists of sixteen 4-bit multipliers that can be dynamically combined to perform MAC operation with 16-bit precision, four sets of MACs at 8-bit precision, or sixteen sets of MACs at 4-bit precision

Table 3.1 summarizes the RVV accelerators presented in this section. The last Table row presents de features of our work w.r.t the state-of-the art.

3.2.2 RISC-V Custom Accelerators

Hwacha [Schmidt et al., 2018] served as a base architecture for the definition of the RVV Extension. It is depicted in Figure 3.9. Hwacha is an open-source RISC-V **co-processor** that implements a custom ISA extension for parallel data processing. It has evolved over several versions, the most recent being V4. Developed at the University of California, Berkeley, its RTL is written in Chisel. Its goal is to explore energy-efficient implementations of vector architectures. Hwacha also features a runtime-variable vector length facilitated by a configurable register file. It supports unit-stride, constant-stride, and indexed load/stores, as well as a comprehensive set of operations, including half, single, double floating-point, integer, and predicate operations.

Yu et al. [2022] present a RISC-V ISE that includes a data operation instruction and a data transfer instruction to boost the computational efficiency of CNNs on edge devices. The ISE is implemented on zero-riscy (IBEX) microarchitecture, illustrated at Figure 3.10, and the proposed extended instructions are integrated into the GCC toolchain. The proposed

Table 3.1: Comparison between RISC-V Vector accelerators.

	Scalar Core	Coupling	RVV Version	Multi-Lane	Pipelined	VLEN/ELEN	Target Application	Language	Bus
Xuantie-910 [Chen et al., 2020] (Commercial)	Xuantie-910 (RV64GCV + custom ISEs)	Coupled	v0.7.1	✓	✓	64-1024/256	High-Performance Computing	N/S	N/S
Sifive Performance P270 [SiFive, Inc, 2022] (Commercial)	SiFive Performance P270 (RV64GBCV)	Coprocessor	v1.0	✓	✓	256/64	High-Performance Computing	N/S	N/S
Sifive Intelligence X280 [SiFive, Inc, 2023] (Commercial)	SiFive Intelligence X280 (RV64GCV + custom)	Coprocessor	v1.0	✓	✓	512/64	High-Performance AI and ML	N/S	N/S
ARROW [Assir et al., 2021]	MicroBlaze v11.0 (not RV)	Coprocessor	v0.9 subset	✓	✓	256/64	edge ML	VHDL	AXI
ARA [Cavalcante et al., 2019]	CVA6 (ARIANE) (RV64IMAC)	Coprocessor	v0.5	✓	✓	128-2048/64	General Purpose	SystemVerilog	AXI
ARA [Perotti et al., 2022]	CVA6 (ARIANE) (RV64IMAC)	Coprocessor	v1.0 (Not fully)	✓	✓	128-4096/64	General Purpose	SystemVerilog	AXI
ARA2 [Perotti et al., 2024]	CVA6 (ARIANE) (RV64IMAC)	Coprocessor	v1.0	✓	✓	256-8192/64	General Purpose	SystemVerilog	AXI
Pircher et al. [2021]	Custom RV32I	Coupled	v0.9	✓	X	1024/64	Cryptography	Domain Specific Language (DSL)	N/S
Jacobs et al. [2024]	Custom RV32E	N/S	v1.0	X	N/S	64/32	Embedded systems	N/S	N/S
Johns and Kazmierski [2020]	Custom RV32E	Coupled	v0.9 subset	X	X	32/32	Embedded systems, Microcontroller	N/S	N/S
Ali et al. [2021]	cv32e40p (RV32IM[F|Zfinx]C)	Coprocessor	v0.1	✓	✓	scalable/32	Embedded systems	SystemVerilog	APU
ZeroVex [Zhao and Ye, 2024]	ZeroVex (RV32IMV)	Coupled	v1.0	✓	✓	256/32	Embedded systems	SystemVerilog	N/S
SPEED [Wang et al., 2024]	RV64GCV1.0	Coupled	v1.0	✓	✓	scalable/64	Multi-Precision DNN Inference	N/S	N/S
This Dissertation	RS5 Nunes et al. [2024]	Coupled	v1.0	✓	✓	scalable/32	Embedded systems	SystemVerilog	AXI

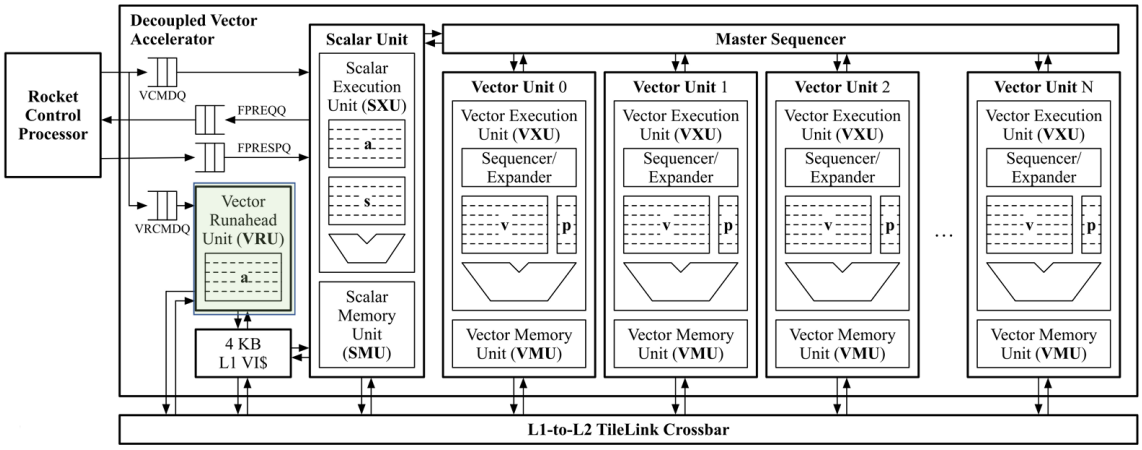


Figure 3.9: Hwacha [Schmidt et al., 2018] block diagram.

instructions are **VMAC** and **VLOAD**, which are SIMD-based and are used to accelerate the dot product computation in the convolution operation. The data width adopted is 16 bits, and each vector register holds nine elements, considering that the convolution kernel of a typical CNN is mostly 3 * 3. The VMAC instruction is a data computational instruction. It has two source registers from the vector registers file, each with nine elements. The result is written back to a general-purpose register, *rd*. The VLOAD instruction is a data transfer instruction that loads a set of vectors from memory into vector registers. It implements a Vector Registers File (VRF) with eight registers of 144 bits (9 x 16-bit elements) and a Dot

Product Unit (DPU or DOTP), which accepts two vectors for calculation at the same time and executes a vector dot product with nine elements.

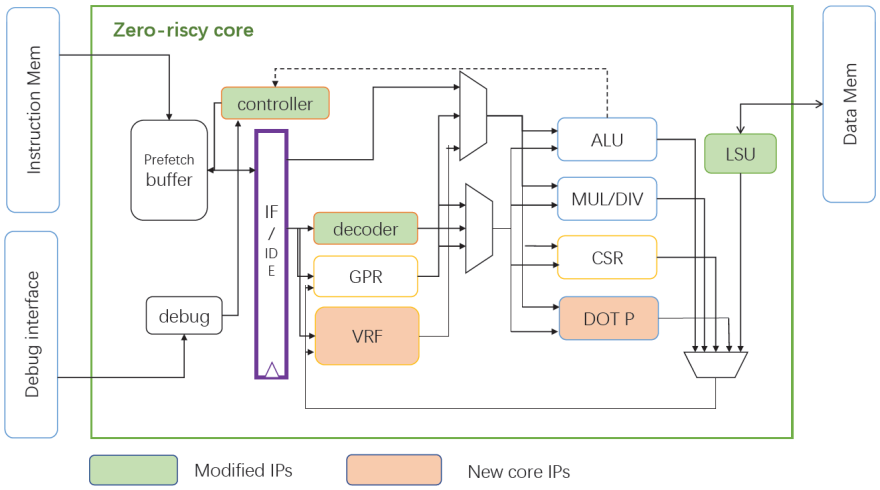


Figure 3.10: Extended Zero-riscy pipeline with VRF and DOTP units [Yu et al., 2022].

RVVe [Schmidt et al., 2024] is a Minimal RISC-V Vector Processor for Embedded AI Acceleration. Their main target is an accelerator targeting the low area with a small-scale vector processor. The accelerator is built on a NeoRV32 RISC-V core, extended with a lightweight vector extension called RVVe. The designers opted only to support an 8-bit quantized network. The RVVe includes VMAC, VADD, and VMUL instructions. The extension uses the integer register file instead of introducing a dedicated VRF. The extension also includes VLOAD and VSTORE instructions supporting unit-strided and strided access patterns of 8 and 32-bit values. The design was synthesized into an AMD Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA.

The **XpulpNN** [Garofalo et al., 2020] is a custom RISC-V extension to accelerate Quantified Neural Networks (QNNs) on RISC-V processors. It uses sub-byte fixed-point data types to avoid requiring numerous instructions for packing and unpacking data when running low-bit-width (i.e., 2- and 4-bit) QNN kernels. It is integrated into the cv32e40p (RI5CY) processor, as depicted in Figure 3.11. The proposed XpulpNN instructions extend the RV32IMCXpulpV2 ISA with SIMD operations for 4-bit and 2-bit operands, namely nibble and crumb. The main functionality consists of dot product operations and an instruction that handles the quantization process in hardware. In total, 15 new instructions are added. The XpulpV2 ISE provides 8-bit and 16-bit SIMD instructions. The work extends the Dot-Product Unit of XpulpV2 to support 2- and 4-bit vector operands. The proposed multipliers compute the dot product between two vectors, each containing either eight 4-bit or sixteen 2-bit elements, and accumulate the result in a 32-bit register through an adder tree in one clock cycle.

RK et al. [2021] introduces a variable bit-precision vector extension for RISC-V processors, designed to support lower bit-widths or variable precision (1 to 16 bits) in MAC

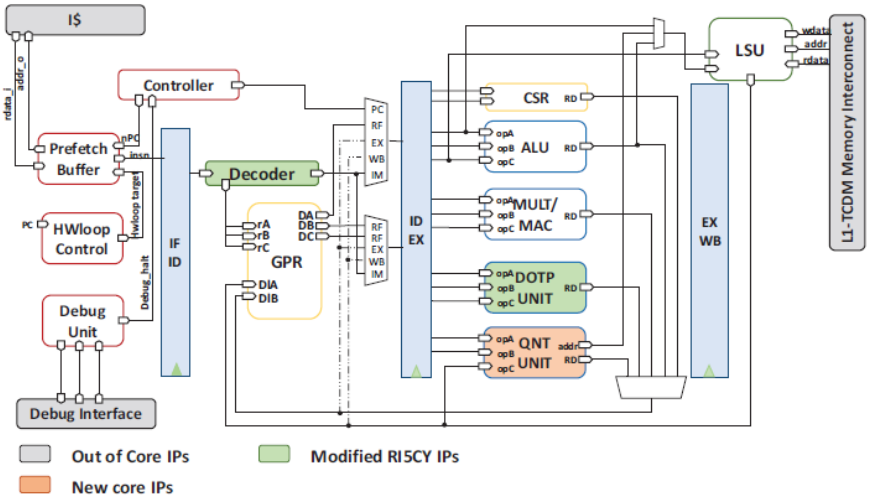


Figure 3.11: Baseline and extended CV32e40p(RI5CY) pipeline for XpulpNN [Garofalo et al., 2020].

operations – Figure 3.12. This extension is demonstrated by integrating it with a PicoRV32 core, using a Xilinx Zynq-based FPGA Zedboard as the implementation platform. This development addresses the bit-width gaps identified in works like XpulpNN [Garofalo et al., 2020]. They opted for a bit-serial implementation for the multiplier, facilitating the integration of variable bit precision. Validation was performed using matrix multiplication with fully variable bit-width precision. Three new states were added to the non-pipelined PicoRV32's Finite State Machine (FSM). The architecture includes a dual-ported vector register file with 32 registers, each 512 bits wide. The vector processing element comprises 16 lanes of parallel elements capable of processing two 32-bit input operands. The Bit Serial Multiplier takes two operands, A (16-bit wide) and B (N-bit wide), and performs a shift-and-add process N times, where N is the width of Operand B. This results in a computation time of N cycles. The architecture was validated using a 3x3 matrix multiplication test program, with Matrix A having 16-bit width elements and Matrix B containing variable bit-width elements. Three versions were tested: the baseline PicoRV32 processor and the PicoRV32 processor with the Vector processor with and without variable bit-precision support. It achieved an average speedup of 1.14x in the version with variable precision compared to the version without variable precision on a matrix multiplication test program. It also reduces the memory footprint by up to 1.88x.

FlexBex [Dao et al., 2020] is a RISC-V with a reconfigurable instruction extension using an open-source framework for adding embedded FPGAs (eFPGA) into RISC-V CPUs. An eFPGA is directly coupled with the CPU, supporting partial reconfiguration, and the instructions can be swapped at runtime. FlexBex integrates an Ibex through an additional execute block called eFU (eFPGA Unit), taped out in 180μm TSMC. The eFPGA allows up to three custom instructions using two source registers. Individual instructions include a vari-

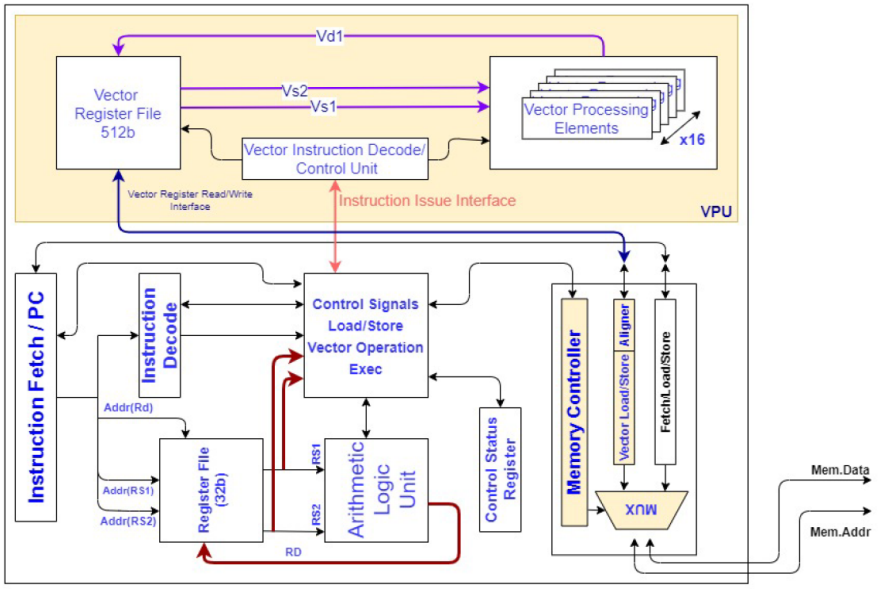


Figure 3.12: Microarchitecture of variable width precision of RK et al. [2021].

able delay to stall the RISC-V core as needed. Compiled custom instruction configuration bitstreams can be loaded directly into the eFPGA using a serial external programming port.

Simodense [Papaphilippou et al., 2021] adopts a similar approach to Flexbex by integrating small FPGAs to implement custom instructions in an open-source RISC-V softcore optimized for exploring custom SIMD instructions. The framework facilitates easy integration of custom vector instructions and allows for their evaluation in simulation and hardware. Simodense implements the RV32IM instruction set with a single pipeline stage and does not rely on 1-cycle-latency memories for instructions or data. It communicates with DRAM in bursts via an interconnect, such as AXI. The Last Level Cache (LLC) is implemented using BRAMs, with the Level-1 block size matching the vector register width. Advanced instructions, including pipelined vector instructions, are processed through a dedicated pipeline. Simodense was implemented on a Xilinx UltraScale+ ZU3EG device, demonstrating a reduction in the number of instructions and cycles required for the evaluated tasks. Wang et al. [2021] details the implementation of a custom instruction, “CONV23”, extended from the RISC-V ISA, specifically designed for Winograd-based convolution acceleration. The Winograd minimal filtering algorithm efficiently minimizes multiplication operations during convolution calculations. Its computation requires just 16 multiplications, compared to the 36 required by the traditional direct dot product method. The custom “CONV23” instruction enables the convolution of a 3×3 kernel with a 4×4 input matrix, producing a 2×2 output matrix. The CONV23 module operates as a 3-state machine (idle, data acquiring, and calculation). The instruction can be executed within 19 clock cycles, significantly improving over the 140 cycles required if implemented using standard RISC-V ISA instructions. A Cv32e40p (RI5CY) RV32IMC core was synthesized onto the NEXYS A7 FPGA platform.

Before the convolution operation, the input matrix is stored in memory, while the elements of the convolution kernel are housed in the convolution acceleration module.

Table 3.2 summarizes the RVV accelerators presented in this section.

Table 3.2: Summary of RISC-V custom accelerators.

	Processor	Coupling	Technique	Target Application
Hwacha [Schmidt et al., 2018]	Rocket core (RV64GC)	Coprocessor	Custom vector extension	General Purpose
Yu et al. [2022]	IBEX (RV32IMCB)	Coupled	Custom vector extension	CNNs on Edge devices
RVVe [Schmidt et al., 2024]	NeoRV32 (RV32IMCB)	Coupled	Custom lightweight vector extension	Embedded AI
XpulpNN [Garofalo et al., 2020]	cv32e40p (RV32IM[F|Zfinx]C)	Coupled	Custom SIMD extension	QNNs
RK et al. [2021]	PicoRV32 (RV32IMC)	Coprocessor	Custom vector extension	Variable bit-precision (QNNs)
FlexBex [Dao et al., 2020]	IBEX (RV32IMCB)	Coupled	Embedded FPGA w/ custom SIMD extension	General Purpose
Simodense [Papaphilippou et al., 2021]	Custom (RV32IM)	Coupled	Embedded FPGA w/ custom SIMD extension	General Purpose
Wang et al. [2021]	cv32e40p (RV32IM[F|Zfinx]C)	Coupled	Custom SIMD extension	Winograd Convolution

3.2.3 Accelerators Controlled by RISC-V

Zhang et al. [2021] presents a CNN accelerator featuring two RISC-V PicoRV32 controllers designed to enhance control flexibility and hardware utilization. The CNN computations are divided into two groups: full-precision and low-precision, and the controllers are employed to manage their execution with fine-grained control. Such architecture applies to different CNNs to support simultaneous computations using multi-blocks. A logic is designed to control the queued operations for the full precision processing element (FPPE) with high flexibility. The architecture is illustrated in Figure 3.13. MACs and storage of PE arrays have 8-bit precision. The 16-bit weight calculations are done by FPPEs that contain 16-bit MAC and storage. The PicoRV32[IMC] can run the driver code for the entire accelerator. With this embedded processor, the workload of the external processor can be reduced. The simplified PicoRV32[I] is a minimized PicoRV which only has the I instruction subset that only does the configuration and control for FPPEs.

AskariHemmat et al. [2021] present a RISC-V Barrel Processor designed to control a DNN Acceleration. It is a 5-stage pipeline datapath with eight hardware threads (hearts). It implements the RV32I and custom CSRs for controlling the PEs. Each thread is executed under a strict round-robin scheduler, as shown in Figure 3.14, and is responsible for providing data and control signals to a neural network PE. Each PE can perform arbitrary-precision GEneral Matrix Vector (GEMV) operations. Each thread executes independently of other threads, and the software sends any communication between threads through shared memory. No data hazard detection or branch prediction logic is needed, and each hart has

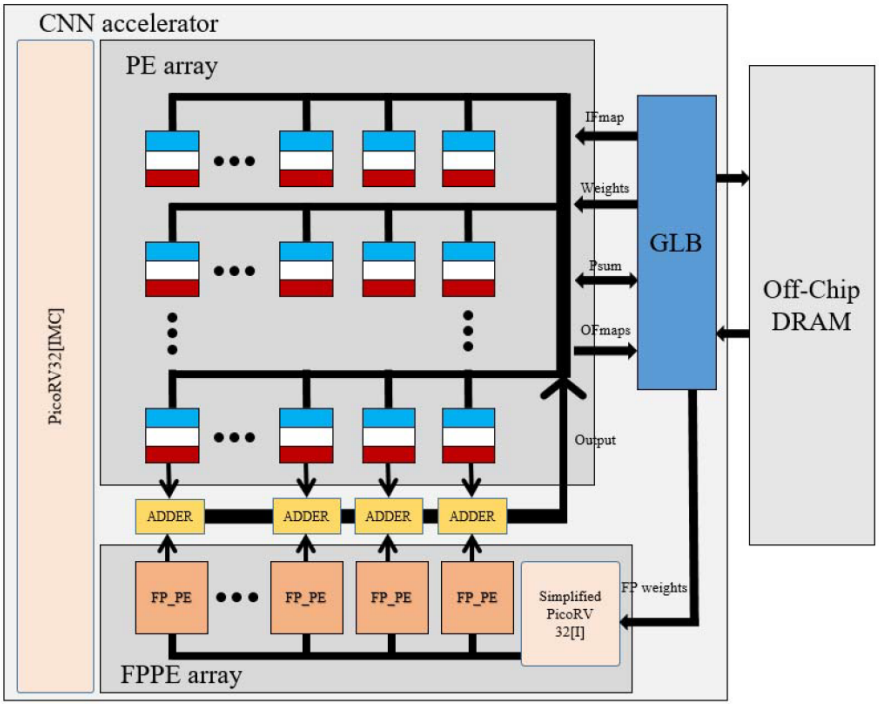


Figure 3.13: Architecture of CNN accelerator with dual RISC-V controller [Zhang et al., 2021].

its own register file. A PE uses mixed fixed-point precision operands, computes 64 output vector elements in parallel, and performs 4096 1-bit multiply-and-accumulate operations on every clock cycle. The PEs have local memory for storing elements of the weight tensors and a local memory for storing data tensors. Every hart has its own instance of CSRs, which are directly linked to its associated PE. Memory is 32KB in size and is implemented in the FPGA using BRAMs. It was prototyped using Xilinx Vivado, targeting the Kintex Ultrascale 040 (KU040) FPGA. Each matrix-vector product took 16 clock cycles to finish. Since all these matrix-vector products were performed simultaneously on eight different harts, the result of the entire matrix-matrix product took 16 clock cycles to finish.

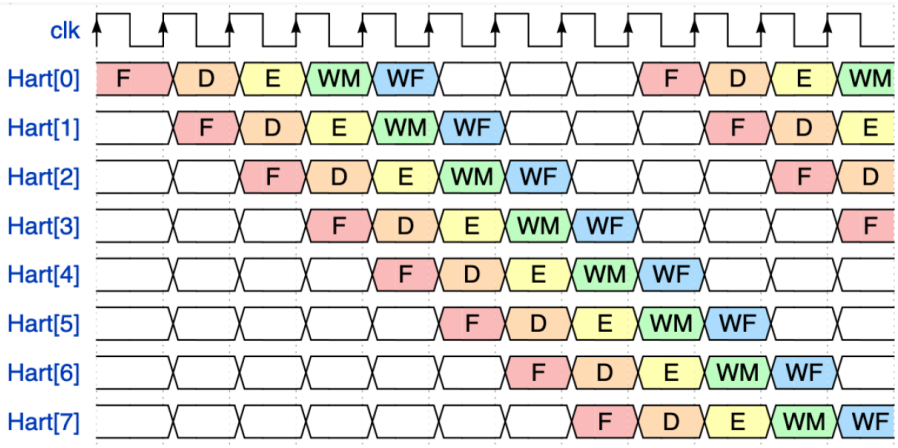


Figure 3.14: Scheduling and pipeline progression of 8 harts running in the barrel processor [AskariHemmat et al., 2021].

3.3 Manycore accelerators

This Section presents works that use manycores for domain-specific acceleration. It is divided into two Subsections: Section 3.3.1 presents works that use multicore clusters for acceleration, and Section 3.3.2 presents works related to NoC-based accelerators.

3.3.1 Multicore Accelerators

This Section reviews related works that leverage multicore architectures for acceleration. Some works use vector accelerators, while others employ custom accelerators. These designs are predominantly organized into core clusters. The SiFive Performance P270 [SiFive, Inc, 2022] and Intelligence X280 [SiFive, Inc, 2023] can also be configured as multicore clusters.

Beldianu and Ziavras [2014] overviews the design of a shared vector accelerator for multicore processors. The main motivation of the Authors is the underutilization of vector processors due to the lack of sustained data-level parallelism or the presence of vector-length variations in application code. Figure 3.15 shows the vector processor (VP) sharing two cores. Multiple cores can dynamically share any vector lane, and each lane contains a subset of the elements from a vector register distributed along the lanes, an FPU, and a memory load/store (LDST) unit. A vector controller (VC) is attached to each core and receives its vector instructions. These instructions can be instructions to move and process vector data, which are forwarded to vector lanes, or control instructions, which are forwarded to the attached scheduler. Vector lanes do not compete for resources except to access a memory module. VP sharing techniques are explored, and fine-grain temporal (FTS) doubles the speedup and reduces the energy by about 50% as compared to cores with VP exclusive access. VP sharing for a dual-core yields speedups of 1.2-2 and halves the energy needs compared to a system with a single core with an attached VP. ASIC was developed targeting a 40nm TSMC technology and can operate at 1.1 GHz.

A RISC-V SoC for AI at the Edge of the Internet of Things (IoT) called **GAP-8** was developed by Flamand et al. [2018]. GAP8 is a multi-core processor derived from the PULP platform fabricated in TSMC 55nm LP CMOS technology. It leverages a microcontroller, a set of peripherals, secure execution, and a programmable parallel processing engine which includes a dedicated convolutional engine for DNN inference. The Fabric Controller is an MCU extended for energy-efficient digital signal processing. The memory hierarchy is organized as a single namespace. Every core in the chip can access all memory locations unless they are protected by the Memory Protection Unit. The cluster, which resides in a dedicated voltage and frequency domain, is turned on when applications running on the fabric con-

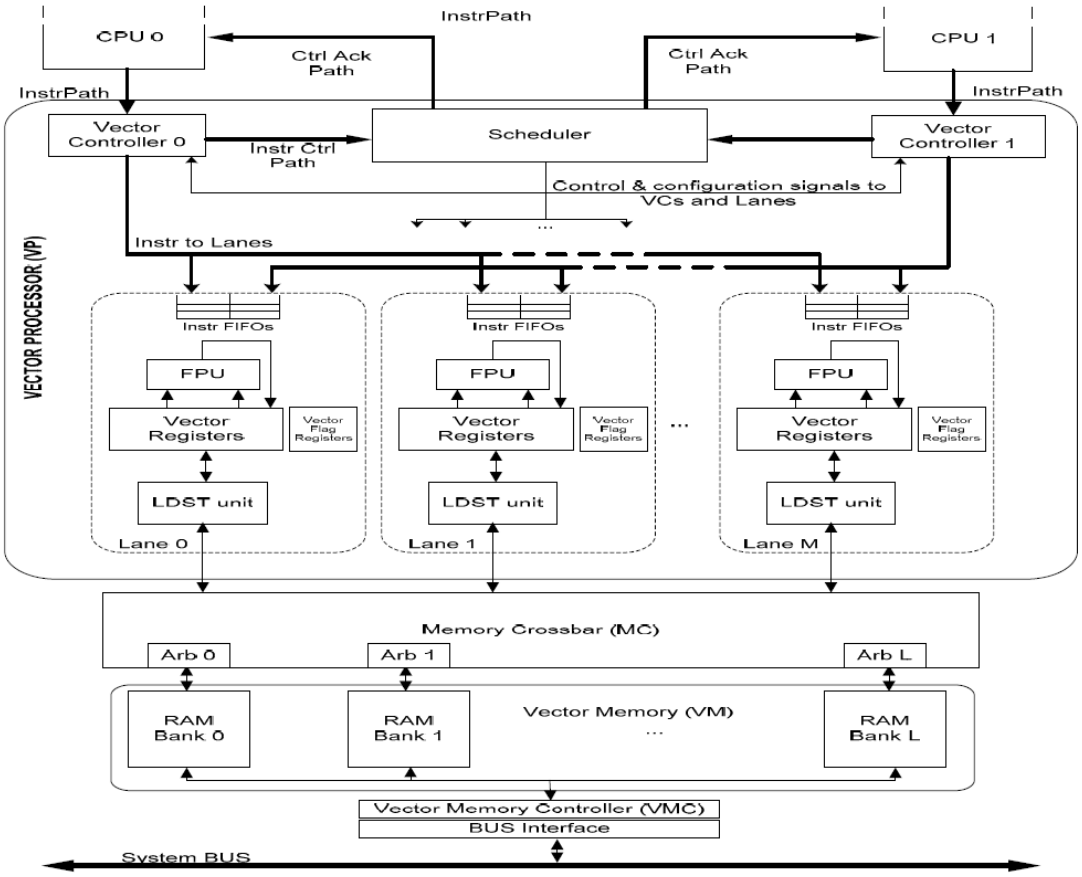


Figure 3.15: Block diagram of a Vector Accelerator shared by two cores [Beldianu and Ziavras, 2014].

troller offload highly computation-intensive kernels. It contains 8 RISC-V cores identical to the one used in the FC, allowing the SoC to run the same binary code on either the fabric controller or the cluster. The cores in GAP-8 feature an in-order, 4-stage pipeline, compliant with the RVIMC extended and optimized with single-cycle MACs, single-cycle complex multiplication, and dedicated instructions for efficient rounding, normalization, and clipping. Also, SIMD instructions enable vectors of 4-byte elements or two short elements and bit-manipulation instructions. When the cluster runs CNN-based applications, it can offload the computation of convolutional layers to a dedicated accelerator, the Hardware Convolution Engine (HWCE), which can evaluate a 5×5 convolution or three 3×3 convolutions on 4-, 8—or 16-bit operands in a single cycle. The combination of parallelism and a hardware accelerator improves performance 10 times compared to a single-core model while improving energy efficiency.

ARA2 [Perotti et al., 2024] provides a multicore analysis, examining the behavior of the vector core architecture in a multicore environment and investigating its performance and efficiency trade-offs. Multiple clusters of the previously described system and the SRAM main memory were multi-banked to support multiple ARA2 instances. For comparisons, they explore the number of cores per cluster, the number of lanes of each vector core, and the

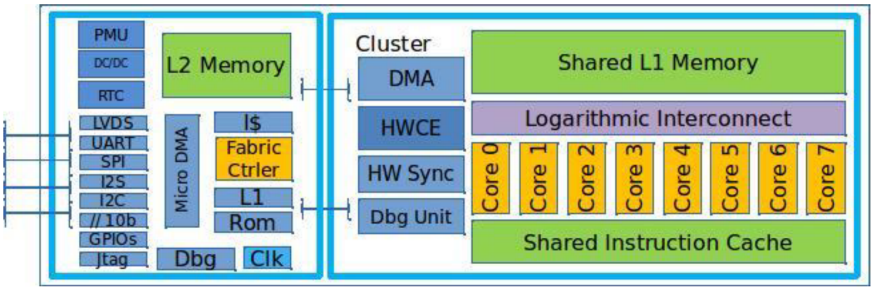


Figure 3.16: Architecture block diagram of GAP-8 [Flamand et al., 2018].

application vector length. Figure 3.17 compares a single-core Ara2 system with a two-core Ara2 system, with both configurations having the same amount of PEs. They conclude that vector processors exploit one dimension only, while multicore systems can parallelize on multiple dimensions.

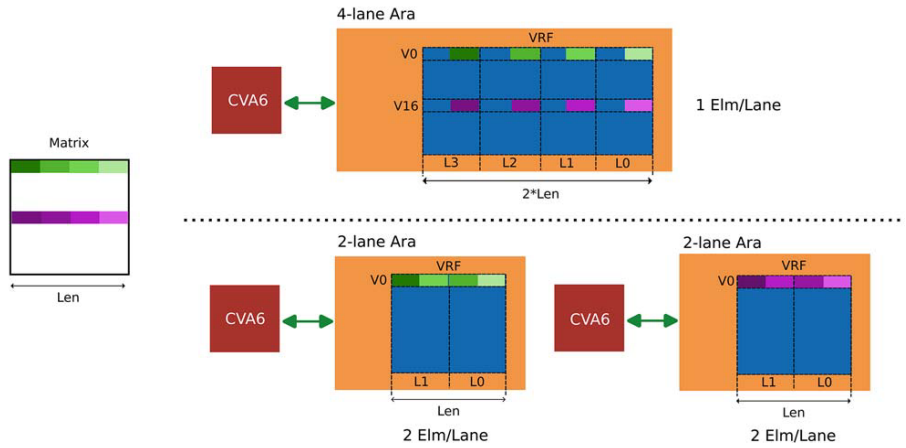


Figure 3.17: Comparison of a single-core 4-lane Ara2 system architecture and a two-core architecture made of 2-lane Ara2 processors. Both configurations have 4 PEs in total [Perotti et al., 2024].

The experiment includes four systems with the same number of FPUs (16), ranging from a single-core 16-lane Ara2 system to a multicore system consisting of eight 2-lane Ara2 systems coupled with eight CVA6 instances. Figure 3.18 compares the throughput results in the four systems, exploring the vector length. For medium/short vectors (8, 16, 32, 64 bit elements), multicore systems composed of smaller instances outperform those with fewer but larger vector cores. As the vector size increases, the PEs are fully utilized even with the maximum number of lanes. The dual-core 8-lane and single-core 16-lane systems surpass the others at vector lengths of 128 and 256 elements, respectively, as the synchronization overhead and pressure on the memory system decrease with fewer cores when evaluating the system's performance abstracted from the physical implementation. The 16-lane system experiences a non-negligible frequency drop that penalizes the whole computation, which becomes slower than all the other designs for all the different vector lengths. The Authors conclude that: (1) for longer vectors, more lanes in single instances are more beneficial; (2)

for smaller vectors: many smaller vector processors are better than a large vector processor that cannot effectively exploit all the dimensions of parallelism in the application.

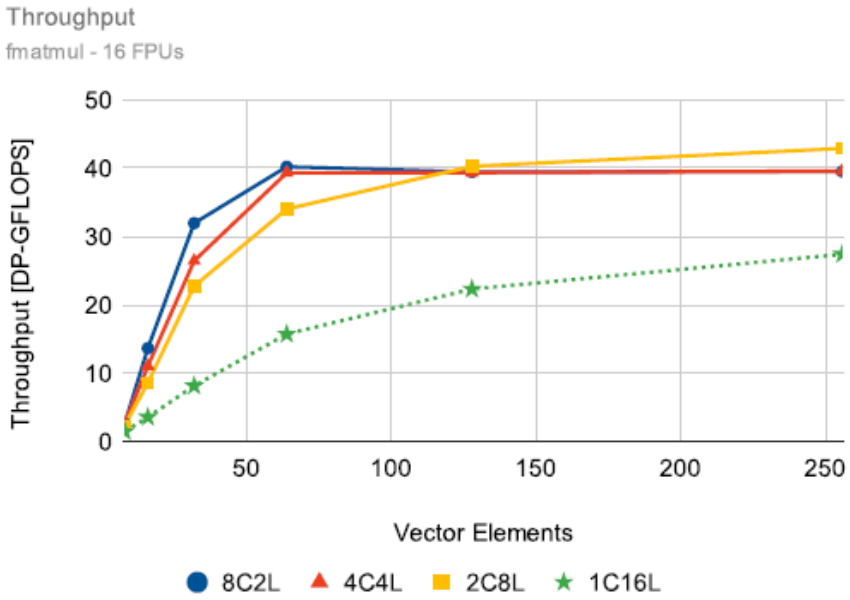


Figure 3.18: Multi- and single-core throughput for different 16-PEs configurations of the Ara2 system in typical conditions (fmatmul) [Perotti et al., 2024].

Table 3.3 summarizes multicore accelerators presented in this section.

Table 3.3: Summary of multicore accelerators.

	Architecture	Target Application
Beldianu and Ziavras [2014]	Dual-Core With Shared Vector Accelerator	General Purpose
GAP-8 [Flamand et al., 2018]	RISC-V Controller + RISC-V 8-Core Cluster W/ Dedicated Accelerator	AI at the Edge
ARA2 [Perotti et al., 2024]	Multicore Cluster of CVA-6 W/ RVV Co-Processor	AI and ML
SiFive Performance P270 [SiFive, Inc, 2022]	Multicore and multi-cluster up to 8 cores of RISC-V RV64GBCV	High-Performance Computing
SiFive Intelligence X280 [SiFive, Inc, 2023]	Multicore and multi-cluster up to 8 cores of RISC-V RV64GCV + Custom	High-Performance AI and ML

3.3.2 Network-on-Chip Accelerators

This section describes works related to the acceleration of AI and ML in the context of Networks-on-Chip. NoC allows more flexible data transmission between PEs, removing the need for complex interconnections. As a result, the NoC-based accelerator has emerged as a promising design paradigm.

Ultra-NoC [Chen et al., 2024] employs a mesh-based NoC interconnection to propose a unified low-transmission routing mechanism. It supports flexible data transmission

and accommodates diverse communication patterns commonly employed in DNN operations. Figure 3.19 shows concepts like Input and Weight reuse, which, in consequence, introduce techniques like Output, Weight, and Row stationary. Data transmission modes supported by the Ultra-NoC are unicast, multicast, and broadcast. The head flit is used to record the multiple destinations of the packet. The proposed Ultra-NoC reduces 26% to 49% and 30% to 56% memory access under AlexNet and ResNet-50, respectively. The design features a controller, on-chip memory, and a 13×13 mesh NoC comprising 169 tiles. Each tile comprises a router, a network interface (NI), and a processing element (PE). Each PE contains a multiplier and an adder, while the PE controller oversees the processing state.

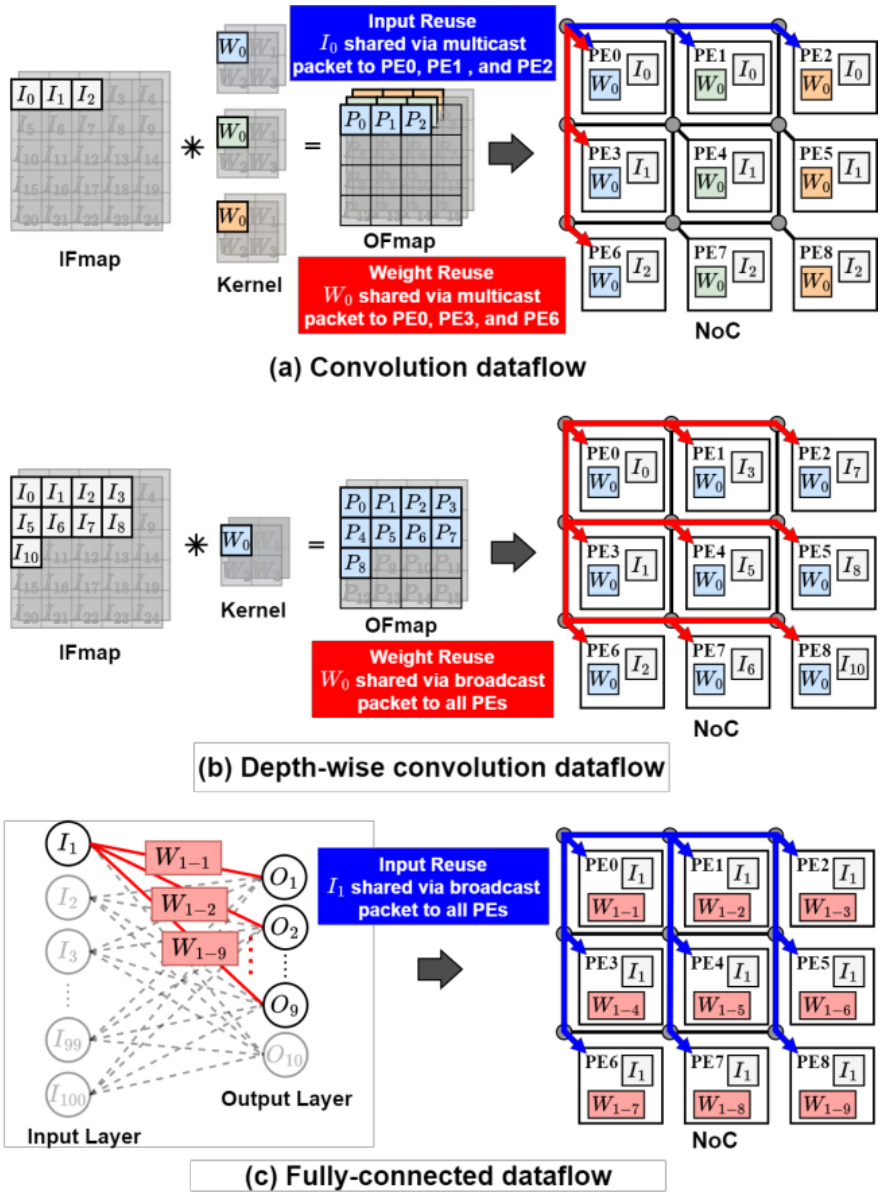


Figure 3.19: The computation dataflow of different DNN operation types [Chen et al., 2024].

Eyeriss v2 [Chen et al., 2019] presents a NoC architecture designed for DNN processing, capable of accommodating various filter shapes and sizes utilized in compact DNNs

like MobileNet. This is achieved by designing a flexible NoC, which architecture is called hierarchical mesh. When data reuse is low, the NoC can provide high bandwidth (via unicast) from the memory hierarchy to keep the PEs busy. When data reuse is high, the NoC exploits spatial data reuse (via multicast or broadcast) to achieve high energy efficiency. Each PE contains MAC datapaths that process 8-bit fixed-point input activations and weights. Eyeriss v2 comprises an array of PEs, a local scratchpad (SPad) memory to exploit data reuse, and global buffers (GLB), which serve as an additional level of memory hierarchy between the PEs and the off-chip DRAM. Thus, it has a two-level memory hierarchy. PEs and GLBs are grouped into clusters to support a flexible NoC. It uses separate NoCs to transfer the three data types, i.e., input activation (iact), weight, and partial sums (psums), between the GLBs and PEs. Each NoC is tailored for the corresponding dataflow of that data type. It comprises 16 PE clusters and 16 GLB clusters arranged in an 8×2 array. Each PE cluster contains 12 PEs arranged in a 3 × 4 array. Figure 3.20 depicts Eyeriss v2 top-level architecture. The chip processes one layer at a time. Once it finishes processing a layer, it is reconfigured to handle the next one.

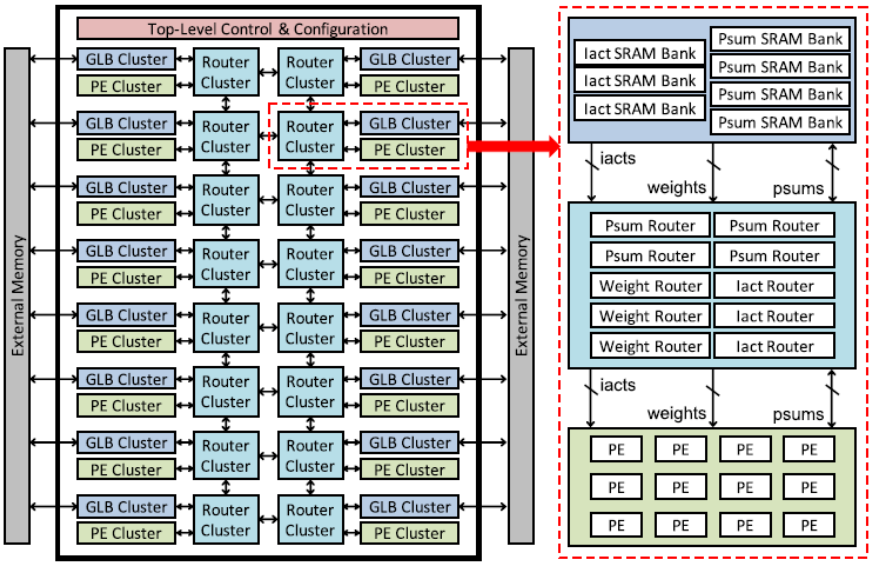


Figure 3.20: Eyeriss v2 top-level architecture [Chen et al., 2019].

The following works seek to add NN capabilities in the NoC components, such as routers, referred to in this work as intra-Noc acceleration. They add extra hardware to perform specific acceleration tasks tailored to each network.

NOVA [Upadhyay et al., 2024] presents a NoC-based vector unit that approximates the non-linear operations within the NoC, relying on the NoC to calculate and broadcast values to different processing elements on-chip. The accelerator can be used on top of existing hardware accelerators using NOVA as a NoC overlay. The NOVA router has two input and output ports (Neighbor and Local). The outputs from each PE are processed by the comparators to generate lookup addresses, which are then sent to the corresponding NOVA router.

Tiwari et al. [2022] propose modifications in the NOC routers with the In-Network Accumulation (INA) method, focusing on reducing the network load and potentially memory transactions that will help alleviate the communication bottleneck. To support the partial sum accumulation, an INA block enclosing an adder is added to each router. The simulation was performed on different DNN workloads and showed a 1.22x improvement in runtime latency and a 2.16x improvement in power consumption.

Gao et al. [2023] propose a dual-layer NoC-based accelerator for lightweight CNN networks. The design includes four parts: (1) NoC array; (2) PE clusters; (3) Loop controller; and (4)Instruction decoder. The NoC array delivers data in different patterns to support calculation. The Instruction decoder adds support to various data for CNN algorithms using a dedicated Instruction Set Architecture (ISA) as the control core of the accelerator. The loop controller controls the memory access interface, input data packagers, and weight packagers. Each PE cluster includes eight processing elements, each containing 1 MAC operation.

Implementing NN accelerators based on NoC architectures needs application mapping to achieve optimal performance. Various NoC mapping techniques have been suggested to reduce power consumption and latency in network communication. Some of the works that explore this area are described below.

Zhao et al. [2020] presents a mapping method for CNNs on NoCs. The proposed approach initially groups convolution operations and allocates them to processing elements (PEs) while considering the characteristics of pooling operations. Subsequently, it calculates the traffic between PEs to generate the Processing Element Communication Graph (PCG). Then, PEs are mapped to the nodes on the NoC by a Genetic Algorithm to reduce communication power consumption and latency. When the number of channels of the image is one, each PE completes one convolution operation per row, and all PEs can complete one convolution operation per clock cycle. Allocating individual PEs for pooling wastes resources and increases communication power, so pooling and convolution operations share the same PE. One PE per channel is allocated in the input layer when the image has multiple channels. For the remaining layers, the number of convolution operations and the balance of computations of different layers are considered. According to their research, more than 70% of the computation operations in CNN are occupied by convolution operations. Experimental results have shown that the proposed method effectively saves power consumption by about 50%.

Liang et al. [2024] employs the Whale Optimization Algorithm, characterized by simplicity, a small number of parameters, and strong global search capabilities. However, it cannot be directly applied to discrete problems. According to them: “the complexity of convolutional neural network applications requires consideration of optimization for multiple objectives, such as energy consumption, latency, and load balancing”. The results show that

their approach significantly improves power consumption and latency compared to similar mapping algorithms.

Ye et al. [2023] propose a task allocation method on a resource-limited NoC Platform. The CNN's computing tasks are first grouped and assigned to PEs to obtain the PCG. The computational load of each task group cannot exceed the limitation of PE's computing capability. Then the PEs are mapped to the NoC through a mechanism called dynamic dense reverse mapping. Compared with the related work, the proposed method reduces 31% power consumption and 38% latency.

Table 3.4 summarizes NN acceleration methods and techniques on NoCs presented in this section.

Table 3.4: Summary NN acceleration methods and techniques on NoCs.

	Method	Technique	Target Application
Ultra-NoC [Chen et al., 2024]	NoC Modification	Hybrid data transmission to leverage different data reuse methods	DNN
Eyeriss v2 [Chen et al., 2019]	Custom NoC Architecture	Separate NoC for different data types and clusters of PEs and GLBs	DNN on Mobile Devices
NOVA [Upadhyay et al., 2024]	Intra-NoC Acceleration	Special router to support non-linear operations	CNN Attention Layers
Tiwari et al. [2022]	Intra-NoC Acceleration	Routers support partial sum accumulation (In-Network Accumulation (INA))	DNN
Gao et al. [2023]	Custom NoC architecture	Dual-layer NoC-based accelerator with dedicated custom ISA	CNN for Edge Computing
Zhao et al. [2020]	Mapping method	PEs communication graph optimization algorithm (Genetic Algorithm)	CNN
Liang et al. [2024]	Mapping method	PEs communication graph optimization algorithm (Whale Optimization Algorithm)	CNN
Ye et al. [2023]	Mapping method	PEs communication graph optimization algorithm considering resource limitation (dynamic dense reverse mapping)	CNNs on resource-limited NoC

3.4 Final Remarks

This Chapter reviewed works on Domain-Specific Accelerators (DSAs). While their target applications vary, all aim to achieve high efficiency in their specialized tasks. In most cases, these applications are related to AI and ML, often designed for edge devices.

The works discussed in Section 3.1 highlight that many companies have incorporated vector and SIMD processing features into their products, often as instruction set extensions (ISEs). Examples include ARM SVE, Intel AVX, and NVIDIA CUDA.

Section 3.2 introduced RISC-V-based accelerators. Leveraging the open-source nature of the RISC-V ISA, many works explore its use across a wide range of domains, from high-performance commercial processors to compact embedded cores. RVV-based

accelerators target diverse applications and are deployed in various configurations, such as coprocessors, tightly coupled designs, pipelined architectures, and multi-lane setups. Some approaches are even more domain-specific, proposing custom extensions to RISC-V cores, as discussed in Section 3.2.2. These works define minimal instruction sets that provide only the necessary operations to accelerate specific target applications, avoiding compliance with broader specifications or incurring unnecessary overhead. However, this strategy sacrifices programmability, requiring compiler adaptation to support the new instructions. Finally, as shown in Section 3.2.3, the simplicity of the RISC-V ISA and processor cores is utilized for controlling accelerators. These works employ RISC-V processors as embedded controllers to optimize processing element (PE) utilization and manage complex dataflows in specialized accelerator clusters.

Section 3.3 reviewed works involving manycore architectures, categorized into multicore accelerators and NoC-based accelerators. The multicore accelerators discussed in Section 3.3.1 primarily organize processing cores into clusters. Most of these accelerators leverage vector-based technology, either shared or individual, except for GAP-8. Many use clusters of processors with the RVV extension, targeting applications in AI and ML.

Section 3.3.2 explores works that employ NoCs for domain-specific acceleration. These works use various approaches, including custom or modified NoCs, intra-NoC acceleration (offloading computation from processing elements), and mapping methods or algorithms. The reviewed works rely on NoCs to interconnect processing elements (PEs) specifically tailored for their target applications, often optimized for operations such as multiply-accumulate (MAC). Consequently, most NoC-based works focus on NoC architectures designed explicitly for acceleration.

In RISC-V cores, there is a gap in the availability of an embedded processor that implements the RVV extension while maintaining compatibility with its target environments and providing effective acceleration. Most existing approaches either modify the memory interface or overly simplify the vector unit, thereby reducing parallelism. Our approach addresses this by implementing an architecture that combines single- and multi-lane components, integrating it as a tightly coupled accelerator using a shared memory bus. Additionally, most processors focus on high-end implementations or support more comprehensive subsets of the vector extension. In contrast, our work aims to implement a reduced subset specifically optimized for CNN applications.

Although ARA2 [Perotti et al., 2024] uses a multicore environment with the RVV extension, it does not use an NoC to interconnect PEs, characterizing a gap in the literature for NoCs with PEs composed of RVV processors. Regarding manycores, the optimization grain is often too small, presenting modifications in the NoC routers that could be performed in other ways with more capable PEs. This also is derived from their approach using NoCs built only for that purpose, with the PEs being only MAC components in most cases, justifying their NoC modifications to achieve better performances. Our approach seeks to use general-

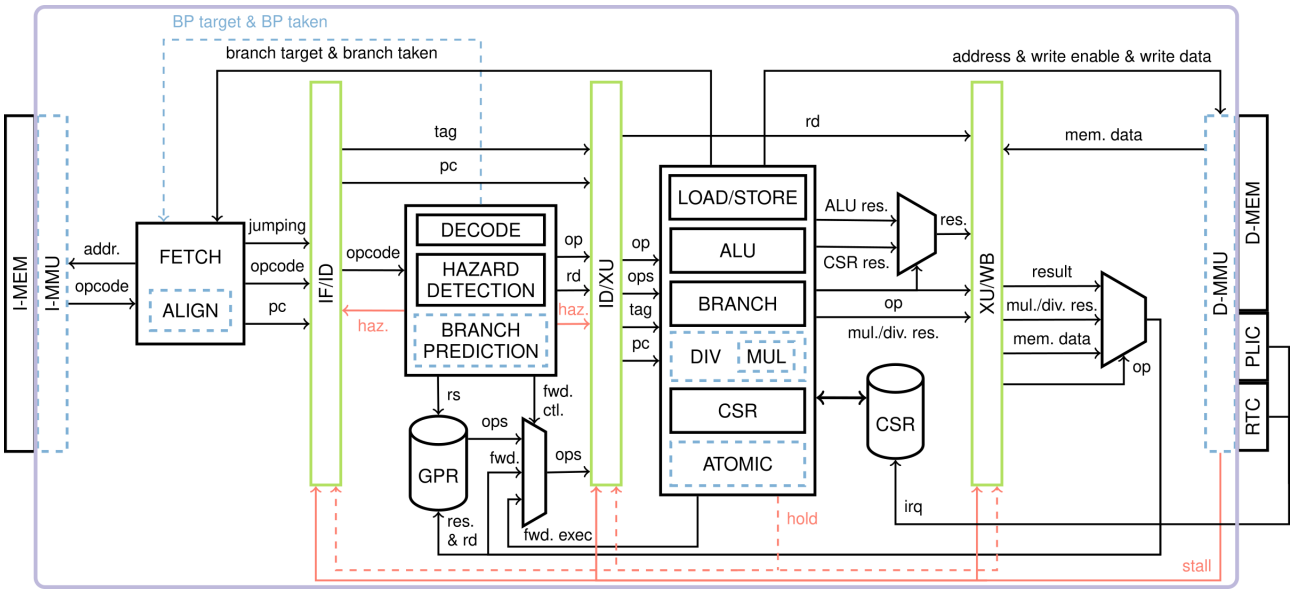
purpose NoCs, which were not tailored for domain-specific applications. This highlights the gap that our work tries to fulfill: enhancing the CNN inference in a NoC environment with accelerated PEs that can handle multiple workloads.

This work leverages vector/SIMD acceleration techniques with multicore and NoC opportunities. It employs a manycore platform comprising several RISC-V cores, including those with the RVV extension, acting as vector accelerators. This design enables the exploration of SIMD/vector parallelization within the RISC-V cores, combined with MIMD parallelization across the manycore dimension while using NoC interconnections for efficient communication. Additionally, the NoC platform and RISC-V processors can concurrently perform other tasks without performance degradation. In summary, this work addresses the gaps in CNN acceleration using RISC-V vector processors within a NoC-based manycore.

4. RS5 PROCESSOR

This Chapter presents the first **contribution** of this Dissertation: an improved version of the RS5 processor. This version includes new RISC-V extensions and mechanisms like branch prediction, compressed instructions, atomic memory accesses, and cryptography acceleration.

RS5, developed initially by the Author of this work – Nunes et al. [2024], is a RISC-V processor that implements the RISC-V RV32I ISA alongside multiple optional extensions and the Machine and User privilege levels. Figure 4.1 presents the RS5 organization, where green rectangles separate the four pipeline stages: instruction fetch (**IF**), instruction decode (**ID**), execution unit (**XU**), and retire unit (**WB**). Control signals are represented in red and optional units are dashed.



The purple line delimits the core. Components in dashed blue lines are optional. Green components are temporal barriers. Signals that control the pipeline stall are marked in red, and the hold signal is dashed due to only occurring on multicicle instructions that are only present in optional extensions.

Figure 4.1: RS5 Organization.

The RS5 offers design parameters to define the inclusion of optional features (extensions and features). Table 4.1 shows all the RS5 top-level parameters.

The parameters correspond to:

- Environment – allows to quickly switch between ASIC and FPGA constructions.
- MULEXT – parameter to select whether or not to include hardware acceleration for multiplication and division (M Extension) or multiplication only (Zmmul Extension).
- AMOEXT – atomic extension that adds instructions that atomically read-modify-write memory contents.

Table 4.1: RS5 Design Parameters.

Parameter	Description	Options
Environment	Environment type	ASIC, FPGA
MULEXT	Include Hardware Multiplication/Division extension	MUL_OFF, MUL_ZMMUL, MUL_M
AMOEXT	Include Atomic operation extension	AMO_OFF, AMO_ZALRSC, AMO_ZAAMO, AMO_A
COMPRESSED	Include Compressed extension	TRUE, FALSE
XOSVMEnable	Include XOSVM extension (MMU)	TRUE, FALSE
ZIHPMEnable	Include ZIHPM extension (Performance Monitors)	TRUE, FALSE
ZKNEEnable	Include ZKNE extension (AES Hardware acceleration)	TRUE, FALSE
BRANCHPRED	Include Branch prediction	TRUE, FALSE
VEnable	Include Vector extension	TRUE, FALSE
VLEN	Vector length in bits	64, 128, 256, ...

- COMPRESSED – compressed extension that adds support for 16-bit instructions, reducing memory footprint.
- XOSVMEnable – custom Memory Management Unit (MMU) in the form of a custom extension called XOSVM.
- ZIHPMnable – optional configuration that includes the performance monitors used for instruction profiling.
- ZKNEEnable – optional extension that accelerates the AES encryption algorithm.
- BRANCHPRED – branch predictor is an optional feature that improves performance but can have area and timing drawbacks.
- VENABLE – optional extension that adds vector processing capabilities.
- VLEN – define the size of the registers in the vector unit when vector extension is enabled. Otherwise, it is a don't care.

4.1 Pipeline Stages

The **IF** stage controls the reception of instructions from the instruction memory (I-MEM). It selects the Program Counter (*pc*) according to the core reset, trap occurrence, trap return, jumps, and sequential operation. This stage works with virtual memory addresses and can optionally be connected to an I-MMU to translate them to physical memory addresses. If the compressed extension is present, this stage includes an address aligner and an instruction decompressor, which transform 16-bit instructions into 32-bit instructions to be decoded.

The **ID** stage extracts the operation (*op*) requested by the fetched instruction and the destination register (*rd*). The source registers (*rs*) have already been decoded at this stage. It then fetches the operands (*ops*) from the register bank (GPRs) and performs hazard detection. This stage can identify if an *rs* of the current instruction is the same as the *rd* being written by the retirement stage or being generated by the execute unit to perform a **1- or 2-stage forwarding** (*fwd*. *ctl*.), avoiding waiting for the register bank write, thus bypassing the data directly to *ops*. The data hazard detection emits no-operations (*haz*.), inserting *bubbles* into the pipeline for the number of cycles necessary to resolve a conflict. This stage can count with the optional branch predictor, which generates signals that connect directly with the first stage (*BP target* & *BP taken*).

The **XU** stage is responsible for the computation based on *op* and *ops*. This stage can have different execution units: (*i*) ALU. Its operation result (*ALU res*.) can be used to compose the data memory (D-MEM) access *address* and the *branch target*; (*ii*) BRANCH: manages conditional branching (*branch target* & *branch taken*); (*iii*) LOAD/STORE: manages D-MEM access (*write enable* and *write data*); (*iv*) CSR: executes CSR atomic operations, enforces privileges, and handles traps; and (*v*) Multiplication (MUL) and Division (DIV): the MUL unit can be included independently (Zmmul extension), while the inclusion of the Division unit implies the inclusion of the Multiplication unit, resulting in the M extension.; (*vi*) ATOMIC: state machine that controls read-modify-write operations on memory.

The **WB** stage executes the write-back of *op* results. It is responsible for writing data obtained from D-MEM after a data read was queued from the LOAD/STORE unit. Note that an optional D-MMU can virtualize D-MEM addresses.

Note that memory writing is performed during the XU stage. The XU also manages the *hold* signal for multi-cycle operations, such as multiplication, division, and atomic operations. Multiplication operations typically require 4 to 5 cycles to complete, whereas division operations may take up to 32 cycles.

4.2 Stall Signals

The signals highlighted in red in Figure 4.1 control the pipeline *stalls* and act as an *enable* for the temporal barriers that delimit the stages. Three signals stall the pipeline:

- **hazard** – raised when the *hazard detection* mechanism detects a memory read-after-write conflict (given a 1-cycle latency memory). The hazard signal stalls only the first two stages, allowing the later stages to proceed with their respective computations to resolve the detected conflicts.
- **stall** – external signal that indicates to the core that a requested operation is being processed. It causes all the stages to hold their current state while the signal is in high-level.

Memory reads that caused cache-miss or memory writes to peripherals that are not ready to receive the data (e.g., FIFO full) are examples of situations that can cause the stall signal to be raised.

- **hold** – control signal that stalls all the pipeline temporal barriers. This signal is generated in the execution of multi-cycle instructions and remain active during the cycles needed for computation of the given instruction. It is present in the core only when optional extensions are present (i.e., MUL/DIV/ATOMIC operations).

These three signals work together, enabling the processor to operate in diverse environments and applications. They allow the processor to support multi-cycle instructions and enable memory architectures that do not assume zero-latency access.

4.3 CSRs and Interrupt Control

Including CSRs is essential for enabling the core to run an operating system. The Zicsr extension defines the instructions required to support these registers. The Zicsr instructions operate on a predefined register map consisting of 4,096 registers (12-bit addresses) and perform atomic operations, which include (*i*) read and write; (*ii*) read and set; and (*iii*) read and clear. A dedicated execution unit controls these operations for CSRs and sends control signals to a CSR Bank, which contains several inputs and outputs for communicating the CSRs with the several units of the processor. The RISC-V privilege approach allows the processor to have different privilege levels: Machine, Supervisor, and User, with the Machine at the highest and the User at the lowest. This allows the separation of user applications from system tasks such as trap handling.

Trap handling is an essential feature for processor cores that interact with the external world, as they must be capable of receiving interrupts and raising exceptions. The processor employs a local interrupt controller with fixed priority. Interrupts can be globally disabled or masked using the CSRs. When a trap occurs, several CSRs are simultaneously modified. The current privilege level is set to Machine mode, and the previous privilege level is saved. The program counter associated with the exception or active during the accepted interrupt is stored, and the program counter is updated with the address of the trap-handling routine. Interrupts are disabled, the trap cause is written to a CSR, and other minor trap-specific adjustments may also occur.

RS5 adopts an interrupt treatment approach based on a proposal by SiFive [SiFive, Inc, 2020]. Two types of interrupt controllers are discussed: (*i*) Core Local Interrupt Controller (CLIC); (*ii*) Platform Level Interrupt Controller (PLIC). CLIC is primarily designed to manage interrupts for multiple hardware threads within the same core, multiplexing timer interrupts, and managing software interrupts. Given that RS5 is a single-thread core, CLIC does not apply to this work. PLIC manages global or external interrupts generated by pe-

ripheral devices. Within PLIC, each interrupt can be assigned a configurable priority level ranging from 1 to 7 (highest priority), and 0 disables the interrupt.

RS5 supports timer interrupts through a Real-Time Clock (RTC) and external interrupts managed by PLIC (*irq*). PLIC is configurable and provides an interface to Memory-Mapped Registers (MMR).

4.4 Real-Time Clock

RISC-V standardizes RTC access through MMRs. RS5 implements a 64-bit cycle counter that is accessible through MMR and also routed to the **time** counter in Zicntr. The OS uses the RTC to generate timer interrupts to control the scheduling time slice.

4.5 Memory Interface

RS5 assumes a True Dual-Port memory architecture, where one port is used for instructions and the other for data access. The memory interfaces, illustrated in Figure 4.2, assume that each port includes a single address bus responsible for addressing both read and write operations. The distinction between operations is determined by a byte-wide *"write"* control signal. If this signal is all zeros, a read operation is performed; otherwise, a byte-wide write operation occurs, with each bit of the signal serving as an enable for the corresponding byte position in memory. Additionally, the port features an *enable* signal that disables all memory operations when set to zero. The instruction memory port has its enable and write signals hardwired to zero, ensuring that only read operations are performed.

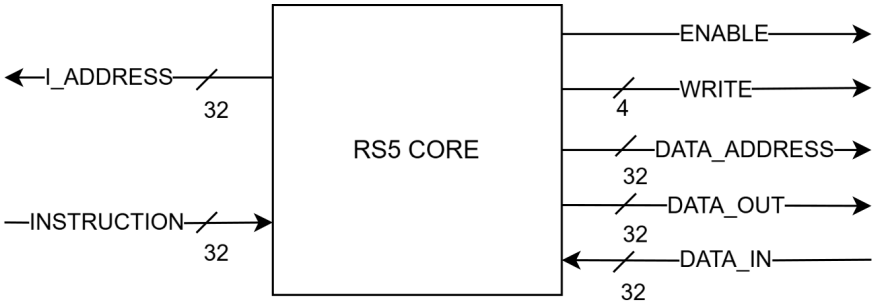


Figure 4.2: RS5 Memory Interface.

This interface allows the RS5 processor core to be used in an AXI bus interface without any modifications within the processor.

4.6 Memory Management Unit (MMU)

RS5 includes an optional compact MMU through the **Xosvm** extension ("X": custom extensions, "osvm": *offset and size virtual memory*). This extension enables paged memory organization by dividing the memory into contiguous, statically-sized pages for each process. It was introduced to address two significant issues: firstly, the standard RISC-V Sv32 extension for 32-bit virtual memory systems requires the Supervisor mode (S-Mode) alongside M- and U-Mode, leading to increased hardware overhead with the inclusion of extra CSRs and a Translation Look-aside Buffer (TLB); secondly, while RISC-V may offer a Physical Memory Protection (PMP) extension for security-centric embedded applications, it lacks virtual memory support needed by dynamic application loading.

The Xosvm extension requires three additional CSRs (mask, offset, and size) for each memory (instruction and data) and an additional register for MMU enablement. The `offset` registers control the page offset of each memory, restricted to the power of two values. Consider, for example, all page sizes equal to 4096 bytes (`0x1000`). The offset for the first four user pages are `0x1000`, `0x2000`, `0x3000`, `0x4000`. The `size` mask registers contain the page size of each memory minus one. In the above example, for a page size equal to `0x1000`, `size` is equal to `0x0FFF`.

Equation (4.1) describes how to obtain the Physical Memory Address (*pma*) from the Virtual Memory Address (*vma*), i.e., the address generated by a code running in the processor. The *or* operation is equivalent to adding the page offset to the *vma*, but with a smaller hardware cost.

$$pma = vma \vee offset \tag{4.1}$$

In the context of the Xosvm extension, memory isolation between pages is ensured, thus safeguarding the contents stored on each page. This is achieved through Equation (4.2). In this equation, the 32-bit *size* mask is negated and then bitwise AND-masked with *vma* (also 32-bits). If any bit of the operation result differs from zero, an exception is raised to the processor core.

$$exception = (vma \wedge \neg size) \neq 0 \tag{4.2}$$

In the above example `size` is equal to `0x0FFF`, if an operation tries to access any element above the *vma* address `0x0FFF` an exception would be generated. For example, when trying to access *vma* of `0x1000`, the negated mask would be `0xF000`, which would result in a non-zero value when it goes through the bitwise AND operation ($0x1000 \wedge 0xF000 = 0x1000$), generating an exception.

4.7 Performance counters (Zicntr and Zihpm)

RISC-V standardizes performance counters to monitor and measure program execution and system behavior. By analyzing the data collected from these counters, developers can make decisions to improve code efficiency, identify bottlenecks, and enhance overall system performance.

RS5 implements the three counters of the Zicntr extension: (*i*) **cycle**: a counter that is increased at every clock cycle and keeps track of how many cycles have passed since the core boot; (*ii*) **time**: a counter that reports an arbitrary precision time counter that is not affected by clock gating, which is connected to the RTC; and (*iii*) **instret**: a counter that tracks how many instructions were retired by the core. Additionally, RS5 offers **optional** debugging counters accessible through the standardized Zihpm extension. These counters track the number of bubbles issued, context switches, and instructions executed by type, such as branches, arithmetic, or logic.

4.8 RS5 Validation Setup

RS5 validation was conducted in three environments:

- RTL simulation, which employs a testbench to emulate peripherals and memory, enabling debugging and rapid validation;
- post-synthesis simulation, aimed at ASIC validation under synthesis constraints;
- FPGA prototyping, which evaluates the core in real-world scenarios.

For FPGA prototyping, the setup uses a Digilent NEXYS A7 board with a Xilinx xc7a100tcsg324-1 FPGA. The main memory is implemented using Block RAM (BRAM), a high-density memory module in Xilinx FPGAs. The GPR is implemented with Lookup Table RAM (LUTRAM), a Xilinx FPGA primitive that is both area- and power-efficient. However, due to LUTRAM's limitations in granularity and atomicity, the CSR bank is implemented using flip-flops.

The core employs a byte-addressed True Dual-Port memory that supports synchronous reads and writes. This memory is initialized by the software using a binary file. The setup includes a UART module for communication, and the board's built-in buttons are used to emulate external interrupts.

Compliance with the RISC-V ISA is evaluated using the rv32ui and rv32um unit tests [RISC-V Foundation, 2015], while performance is assessed using the EEMBC CoreMark benchmark [Consortium, 2024].

4.9 Final Remarks

This Chapter provided a comprehensive overview of the RS5 processor, an implementation of a RISC-V processor that integrates several optional extensions and received multiple architectural improvements during the development of this work, characterizing the first contribution of this Dissertation. The processor has various performance and security features, including branch prediction, compressed instructions, atomic memory operations, cryptographic acceleration, and an optional compact MMU for virtual memory management. Designed with flexibility, RS5 can be configured for various applications through design parameters that enable or disable specific extensions. Its four-stage pipeline, stall management mechanisms, and performance monitoring support contribute to its efficiency and adaptability. The RS5 core was validated through RTL and post-synthesis simulations and FPGA prototyping. These tests demonstrated its compliance with the RISC-V ISA and performance in real-world scenarios. The results confirm RS5 is a robust and scalable solution for embedded and general-purpose computing applications.

5. RVV IMPLEMENTATION

This Chapter presents the efforts to implement the RVV extension in the RS5 processor. It was developed as a tightly coupled accelerator consisting of a new execution unit of the RS5 pipeline. This is the second and main **contribution** of this work.

This Chapter is organized as follows.

- Section 5.1 presents the subset chosen for implementation and the reasoning for the design choices regarding the instructions and functionalities implemented or not on the RS5 vector extension implementation.
- Section 5.2 details the vector extension implementation as a tightly coupled accelerator in the RS5 processors. It uses a top-down approach to detail how the accelerator is fitted inside the RS5 pipeline structure. This section also details the main implementation design choices. The vector execution unit (VXU) organization is presented with its components and functionalities.

  Section 5.2.1 presents the Vector Arithmetic and Logic Unit, a VXU component responsible for executing the vector arithmetic and logic operations. Section 5.2.2 presents the Vector Load and Store Unit (VLSU), detailing how it is organized and controlled and how it communicates with the VXU and the RS5 memory.
- Section 5.3 presents the validation process of the RS5 VXU implementation.
- Section 5.4 concludes this Chapter.

5.1 RVV Subset

The RISC-V Vector extension was previously introduced in Section 2.3. The RVV extension can be implemented as a whole in an application processor, where, in this case, it is named a single letter “V”. The specification also presents predefined instruction subsets intended for embedded uses named with a *Zve* prefix. These subsets were presented in Table 2.5.

The *Zve*32*x* is the smaller instruction subset. It requires a minimum VLEN of 32 bits and supports SEWs of 8, 16, and 32 bits. It supports only integer operations. The *Zve*32*f* adds 32-bit floating-point support to the *Zve*32*x*, requiring the scalar core to implement the F extension. The *Zve*64$*$ extensions increase the RVV extension’s capabilities to 64-bit processors, expanding the minimum VLEN and supported SEWs to 64 bits. The *Zve*64*d* subset supports double-precision floating-point operations while requiring the scalar core to implement the D extension.

Chapter 4 presented the RS5 processor. Given that the RS5 processor targets embedded systems, it became evident that the RS5 RVV extension implementation should target one of the *Zve* subsets designed for that purpose. Since it is a 32-bit processor, it limits the options to *Zve*32*x* or *Zve*32*f* subsets. As the RS5 processor does not implement the floating-point extension (F), the only suitable subset is the *Zve*32*x*.

The requirements for the *Zve*32*x* subset are presented in Table 5.1 accompanied by the RS5 decision to implement it or not and the reasoning to do so. Some requirements were not fulfilled due to the large number of instructions or their complexity. In both cases, the target applications did not use the instructions, leading to a considerable area overhead of never-used instructions. A further explanation for each follows:

- Precise traps - since the RS5 does not trap during multi-cycle instructions, there is no need to have precise traps, and thus, the *vstart* logic is not needed, simplifying the design. Error during vector instructions execution are not treated;
- Vector integer instructions - instructions like add-with-carry and subtract-with-carry are used only in very specific scenarios (not the case of the target applications), and others like widening add/subtract are easier to implement in case they are needed but offer significant area overhead;
- Fixed-point arithmetic - this class of instructions is not used by the target applications and offers a significant amount of instructions, which would increase the area overhead and complexity by a very significant amount;
- Vector mask instructions - this class of instructions varies in complexity, it goes from logic instructions between mask registers (ignoring VLMUL) to more complex instructions like count population and find-first-set mask bit, which would rarely be used;
- Vector permutation instructions - this class contains very complex instructions like slide instructions, register-gather, and compress/decompress instructions, which would add significant area overhead and are not used by the target applications.

Thus, we implemented a subset of the *Zve*32*x* instructions. Since the compiler cannot be told which instructions are present in the implemented subset, the *Zve*32*x* subset is still used to compile the target applications. If an instruction not implemented in the RS5 subset is emitted, the RS5 will decode it as invalid and raise an exception. To avoid that, the applications can be rewritten with different constructs to avoid using unimplemented instructions, or the implementation of those instructions must be considered.

Table 5.2 details the instructions selected for implementation in the RS5 processor and justifies their selection based on their possible use/application. In total, 60 instructions were selected, almost doubling the 36 instructions of the RV32I base ISA implemented by the RS5 processor. This number highlights the need to implement only a portion of the *Zve*32*x* subset, which, with all the instructions, would surpass 100 instructions.

Table 5.1: Requirements of the Zve32x subset and RS5 compliance with each of them.

Requirement	**Imple-mented**	**Reasoning**
Precise traps	No	The vector instructions do not trap
SEW of 8, 16, and 32 bits	Yes	-
Vector configuration instructions	Yes	-
All vector load and store instructions	Yes	-
All vector integer instructions	Partially	Widening Add/Subtract, Integer Extension, Add-with-Carry/Subtract-with-Borrow and Narrowing Right Shift instructions were not implemented as the target applications did not use them
All vector fixed-point arithmetic instructions	No	The target applications did not use fixed-point arithmetic
All vector integer single-width and widening reduction operations	Partially	Only single-width reductions were implemented
All vector mask instructions	No	These instructions were not implemented due to complexity and the target applications not using them
All vector permutation instructions	Partially	Slide, Register Gather, and Compress instructions were not implemented due to complexity, and the target applications not using them

Table 5.2: Instructions implemented in the RS5 Vector Unit and their possible application

Instruction Class	**Instruction Names**	**Application**
Configuration	VSETVL, VSETVLI, VSETIVLI	Configuration instructions
Arithmetic	VADD, VSUB, VRSUB	General-purpose arithmetic
Logic	VAND, VOR, VXOR	General-purpose logic
Shifts	VSLL, VSRL, VSRA	General-purpose shift
Mask Compares	VMSEQ, VMSNE, VMSLTU, VMSLT, VMSLEU, VMSLE, VMSGTU, VMSGT	Mask generation (Used in ReLU)
Min/Max	VMIN, VMINU, VMAX, VMAXU	Get min/max value between two elements
Multiplication	VMUL, VMULH, VMULHU, VMULHSU	General-purpose multiplication
Widening Multiplication	VWMUL, VWMULU, VWMULSU	General-purpose widening multiplication
Multiply and Accumulate	VMACC, VNMSAC, VMADD, VNMSUB	MACs (Convolution operations, dot product)
Division	VDIV, VDIVU, VREM, VREMU	General-purpose division
Sum Reduction	VREDSUM	Aggregation of partial sums
Min/Max Reduction	VREDMIN, VREDMINU, VREDMAX, VREDMAXU	Get min/max value of vector (Max-Pooling)
Logic Reduction	VREDAND, VREDOR, VREDXOR	Get logic reduction of vector
Register Moves	VMV, VMVR, VMVSX, VMVXS	Moving data between vector registers
Merge	VMERGE	Merge data between vector registers based on mask
Unit-Strided Load/Store	VLE, VSE	Access contiguous data from memory (e.g. kernel weights)
Strided Load/Store	VSLE,VSSE	Access pattern-distributed data from memory (e.g. feature maps)
Index load/Store	VLUXEI, VLOXEI, VSUXEI, VSOXEI	Access distributed data from memory (e.g. fully connected layers)

Another important detail is that the instructions presented in Table 5.2 may have variants. The “VADD" instruction, for example, has three variants:

- vadd.vv - sums two vector operands;
- vadd.vx - sums a vector operand to a replicated scalar operand (from a scalar register);
- vadd.vi - sums a vector operand to a replicated immediate operand (from the instruction encoding).

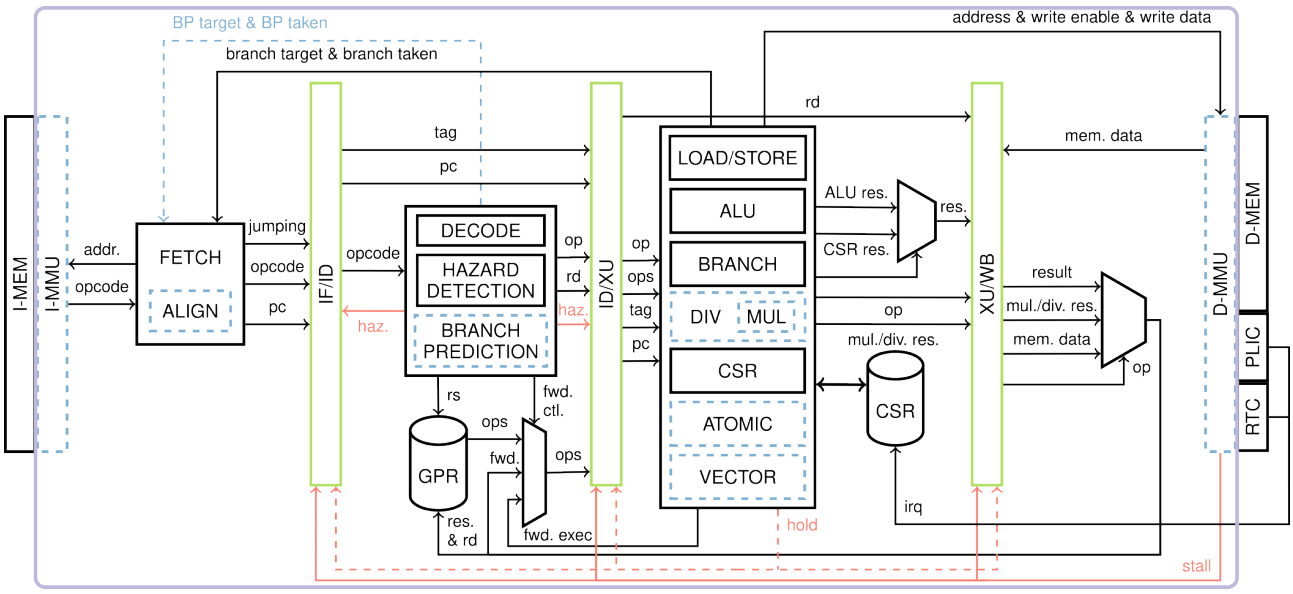
The variants do not create a new instruction, but need operand replication (for scalar and immediate operands) and multiplexing before execution.

5.2 RS5 RVV implementation

The RS5 processor includes multiple execution units, specialized modules for handling specific operations like load-store and multiplication. The execute unit of the RS5 processor, which represents the third stage of the pipeline, instantiates these modules. Some execution units require more than one cycle to complete an instruction. For example, the multiplication unit takes 4 to 5 cycles, and the division unit can take up to 33 cycles. During the execution of a multi-cycle instruction, the pipeline stalls through a *hold* signal assertion until the computation is completed.

The vector extension is implemented in the RS5 [Nunes et al., 2024] processor as a **tightly coupled accelerator**. It is implemented as another multi-cycle execution unit and integrated into the execute module. The vector instructions are decoded in the RS5 decoding unit and executed by an execution unit dedicated to vector processing, present in the third stage (execution) of the pipeline. From now on, this unit will be called the “Vector Execution Unit” (VXU). Figure 5.1 presents the modified version of the RS5 processor that supports the RISC-V Vector extension. The only difference between the old version of RS5 presented in Figure 4.1 and the new version presented in Figure 5.1 is the addition of the **VXU** in the third stage of the processor.

The second stage of the RS5 processor’s pipeline (instruction decode) was modified to support vector instruction decoding. The RS5 pipeline instruction control received only three new instructions: 1) VLOAD, 2) VSTORE, and 3) VECTOR. This approach helps minimize the impact of including the vector extension on the scalar core. If the vector extension is disabled, any vector instruction will be decoded as invalid, and an exception will be raised since the optional components will not be present to execute the instruction, keeping the area overhead minimal. The VLOAD and VSTORE instructions are classified as dedicated instructions. They facilitate the control of the multiplexers for the memory control signals and do not require consideration of more than one signal to produce multiplexing logic. All the vector arithmetic and configuration instructions are decoded as VECTOR instructions, with an auxiliary signal that identifies each. This auxiliary signal is only considered inside the Vector Unit, leaving the entire pipeline agnostic to its existence. Thus, the impact of the vector unit’s inclusion on the other pipeline units is minimal.



The purple line delimits the core. Components in dashed blue lines are optional. Green components are temporal barriers. Signals that control the pipeline stall are marked in red, and the hold signal is dashed because it only occurs on multicycle instructions that are only present in optional extensions. **Note the inclusion of the Vector Unit**.

Figure 5.1: RS5 Organization with VXU.

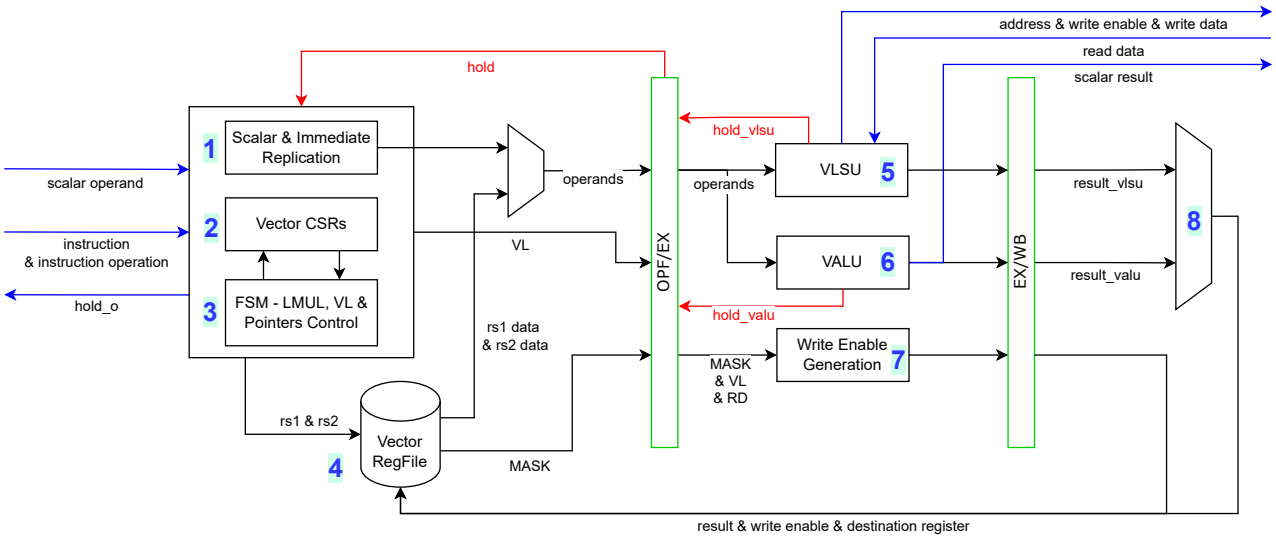
The third stage of the RS5 processor's pipeline (instruction execution) was modified by including the VXU. The VXU is a child unit that receives the execution unit signals (instructions and operands) and performs the required operation. It is a multi-cycle unit with a dedicated "hold_vector" signal that is OR-ed with the hold signal from other execution units (division, multiplication, and atomic, if any are present) to generate the main "hold" signal responsible for freezing the first two pipeline stages until the operation is complete. Since the VXU can access the memory, the choice was to use the RS5 processor memory interface without any modifications. This implies that memory control signals generated by the VXU's Vector Load and Store Unit (VLSU) must be multiplexed with the memory control signals from the other sources (Load/Store and atomic units), which is done based on the current instruction. This design choice helps ensure that the RS5 processor is compliant in every environment without modifying or adding any memory interface. It also helps reduce area overhead and memory complexity.

The CSR bank also suffered modifications as the vector CSRs were added. Unlike regular CSRs, vector CSRs are not present in the CSR bank. They are treated as external registers that can only be written by the vector configuration instructions. Thus, only read logic is implemented, reading the value held in the registers inside the VXU. The CSRs are: 1) Vstart - hardwired to zero; 2) VlenBytes - Constant value (VLEN/8); 3) Vtype; and 4) VL.

The VLEN is a design parameter and can be changed at the module instantiation. When VLEN is changed, the entire vector unit adapts to handle VLEN-length data, by adding more lanes. The VXU unit features VLEN as a design parameter, allowing it to be modified

upon instantiation. VLEN is flexible in powers of 2, and the entire VXU is designed to adapt to process VLEN-length data without designer interactions aside from selecting the desired VLEN.

The VXU is implemented as a three-stage pipeline as illustrated in Figure 5.2. The first stage comprises the control logic associated with the vector CSRs and the operands fetch and replication. The second stage is the execution, where the vectors are processed using two additional units: the Vector Load and Store Unit (VLSU), which is single-lane, and the Vector Arithmetic and Logic Unit (VALU), which is multi-lane. Based on the mask and VL, it also generates the Vector Register File's write enable signal. The results from the VLSU, VALU, write enable, and destination register are propagated to the write-back stage, the last stage of the pipeline. The write-back stage multiplexes the VALU and VLSU results according to the instruction operation. Then, it performs the writeback on the VRF using the write enable and destination register signals.



Green components are temporal barriers. Signals that control the unit pipeline stall are marked in red. Blue signals represent the communication with the scalar core.

Figure 5.2: VXU Organization.

The VXU employs a Finite State Machine (FSM) to control its pipeline, operating in two states: *V_IDLE* and *V_EXEC*. The *V_IDLE* state is the initial state, where the unit remains when it is not operational (i.e., not executing a vector instruction). Upon detecting a vector instruction, it transitions to the *V_EXEC* state, raising the *hold_o* signal and propagating the operands to the second pipeline stage. In the *V_EXEC* state, computations are carried out. This state persists until all registers in the current group (LMUL of the instruction, up to 8 registers) are processed. The *hold_o* signal is de-asserted during the final execution cycle, allowing the scalar pipeline to resume operation. A data hazard detection mechanism is implemented to prevent pipeline stalls due to write-back conflicts. This mechanism is triggered when a vector instruction enters the first pipeline stage with a source register

that matches the destination register in the last pipeline stage. In such cases, the pipeline is stalled for one cycle to ensure that results are properly written to the VRF.

The numbers in Figure 5.2 correspond to the elements that implement the RVV extension:

1) Scalar and Immediate Replication. Controls the replication of the scalar and immediate operands and the operands selection via multiplexers.

2) Vector CSRs. Implements the vector CSRs and associated logic. Checks for invalid configurations and communicates with other components providing SEW, VL, and LMUL.

3) Control FSM. Controls the vector unit with the previously mentioned states. It contains the following modules:

- **3.1) LMUL Control.** It manages register groups by tracking the number of registers processed within a group. It is used to determine when an instruction's execution is complete.
- **3.2) Register Pointers Control.** Register groups are addressed by their smallest register number, requiring adjustments to the operand read address for the current register. The base register address is combined with cycle counters to obtain the correct register address. Vector instructions may use source and destination registers differently for normal, narrowing, or widening operations, necessitating specific pointer control. Narrowing instructions halve the destination register group size, while widening operations double it. Normal operations keep the source and destination register groups the same size.
- **3.3) Vector Length Control.** Manages the number of elements a register can hold and tracks how many have been processed during instruction execution. Instructions may be completed early if the processed elements exceed the current VL.

4) Vector Register File (VRF). Implements the vector registers. The VRF has one write port and three read ports: two for source operands and one for mask (V0 register). It consists of 32 registers of VLEN bits, with a write enable signal for each byte.

5) Vector Load-Store Unit (VLSU). It manages the vector memory access. Each vector access is multi-cycle, and the latency depends on the VLEN, address mode, SEW, and address alignment. It operates on a single-lane, meaning only one 32-bit PE is used. Increasing VLEN barely affects this unit as only the data registers and flow control elements grow with VLEN.

6) Vector Arithmetic and Logic Unit (VALU). Manages the vector arithmetic and logic operations, capable of processing an entire VLEN register in parallel for any element width up to ELEN(32) by using a multi-lane approach. The number of lanes is linked

to VLEN: if VLEN doubles, the number of lanes also doubles. The operations can take different cycles to be completed (depending on SEW for wider operations). For example, additions take one cycle per register, and multiplications can take up to 5 cycles.

7) Write Enable Generation. Generation of the register file's write enable signal. It is a byte-enable signal that considers SEW, VL, and mask.

8) Result Control and Demultiplexing. Demultiplex the result of the VALU and VLSU and send to write back in the VRF or to the scalar register file

The VXU is designed as a **multi-lane** vector unit, meaning it can process multiple elements in parallel. The number of lanes in the RS5 VXU is tied to the VLEN and ELEN parameters. The ELEN is tied to the processor's data width, 32. Thus, the number of lanes in the VXU unit is VLEN/32. This indicates that each execution of vector instructions can process an entire VLEN-wide register in a singular iteration. The vector Load and Store instructions use the RS5 memory interface without modification, so they are limited to a single-lane (32 bits). This implies that the RS5's VXU is multi-lane for arithmetic and logic instructions and single-lane for Loads and Stores.

The number of lanes is a design choice with advantages and disadvantages. Fewer lanes reduce the processing elements required (which often use a significant area footprint) but add multiplexing logic (punishing the circuit timing) and process fewer elements per cycle (smaller throughput). More lanes imply a larger area footprint (more PEs), often with better circuit timing and the advantage of higher throughput.

One of the main design choices was to leave the RS5 memory interface untouched to keep it suitable for its working environments. Therefore, we opted to execute the arithmetic and logic instructions in a multi-lane fashion to compensate the reduced throughput of the vector memory accesses. To help reduce the area overhead of the multi-lane design, the PEs were developed to explore opportunities for hardware reuse.

The VRF supports three simultaneous reads: two for *source* operands and one for the mask (V0 register). MAC instructions require three *source* operands, leading to the need to switch the VRF source addressing. This is possible because MAC operations are multicycle, and the accumulation step is only performed during the last cycle. Thus, the two multipliers are initially fetched from VRF, and the multiplication starts. When the multiplication is in the last cycle, the third operand is fetched from the VRF, becoming available at the next cycle to be accumulated with the multiplication result.

The instructions implemented by the VXU are a subset of the RVV extension selected for ML applications as previously presented by Section 5.1. They are categorized into classes, as shown in Table 5.3, indicating the cycles needed to process a register for each SEW and class. The last column, **optional**, lists instruction classes that can be disabled via design-time code parameters, further reducing the implemented subset.

Table 5.3: Instructions implemented in the RS5 VXU and the respective cycles per register

Instruction Class	Instruction Names	SEW=8 - Cycles per register	SEW=16 - Cycles per register	SEW=32 - Cycles per register	Optional
Arithmetic	VADD, VSUB, VRSUB	1	1	1	N
Logic	VAND, VOR, VXOR	1	1	1	Y
Shifts	VSLL, VSRL, VSRA	1	1	1	N
Mask Compares	VMSEQ, VMSNE, VMSLTU, VMSLT, VMSLEU, VMSLE, VMSGTU, VMSGT	1	1	1	N
Min/Max	VMIN, VMINU, VMAX, VMAXU	1	1	1	Y
Multiplication	VMUL, VMULH, VMULHU, VMULHSU	2	2	4-5	N
Widening Multiplication	VWMUL, VWMULU, VWMULSU	3	3	5-6	N
Multiply and Accumulate	VMACC, VNMSAC, VMADD, VNMSUB	3	3	5-6	N
Division	VDIV, VDIVU, VREM, VREMU	1-8	1-16	1-32	Y
Sum Reduction	VREDSUM	1	1	1	Y
Min/Max Reduction	VREDMIN, VREDMINU, VREDMAX, VREDMAXU	1	1	1	Y
Logic Reduction	VREDAND, VREDOR, VREDXOR	1	1	1	Y
Register Moves	VMV, VMVR, VMVSX, VMVXS	1	1	1	N
Merge	VMERGE	1	1	1	Y
Unit-Strided Load/Store	VLE, VSE	∼VLEN/32	∼VLEN/32	∼VLEN/32	N
Strided Load/Store	VSLE,VSSE	VLEN/8	VLEN/16	VLEN/32	N
Index load/Store	VLUXEI, VLOXEI, VSUXEI, VSOXEI	VLEN/8	VLEN/16	VLEN/32	N

There is always one additional cycle for every instruction listed in Table 5.3. The additional cycle corresponds to the state V_IDLE in which the operands are fetched from the VRF and multiplexed, becoming available when the unit is in the V_EXEC state. Equation (5.1) shows how to compute the number of cycles, N, that a given vector instruction takes to execute in the VXU. The constant 1 represents the additional cycle. X represents the number of “cycles per register" presented in Table 5.3 for the given instruction and the current SEW. The value of X is multiplied by LMUL as it represents the amount of registers the instruction must process. Executing back-to-back vector instructions with the same operands may lead to another additional cycle caused by data hazard conflicts.

$$N = 1 + (X_{CLASS_{SEW}} * LMUL) \tag{5.1}$$

Configuration instructions take only one cycle to execute. Vector Load/Store instructions differ according to the addressing mode. The VLSU has internal control over the number of elements processed per cycle. Due to its nature, unit-strided operations can access more than one element per cycle (SEW-dependent, 32-bit limited). Strided and Vectorized accesses process a single element per cycle. The Vector Load/Store instructions will be further detailed in Section 5.2.2.

5.2.1 Vector Arithmetic and Logic Unit

The vector arithmetic and logic unit (VALU) is the main component of the VXU. It executes all instructions except the loads, stores, and configuration instructions. It receives the operands, mask, CSR configurations (LMUL, VL, and SEW), and control signals (operation, current cycle, and current state). It outputs the VALU hold signal and the operation results.

The hold signal is raised when a multicycle instruction is executed and remains high until the operation is complete. Instructions from the same class can vary in cycle duration depending on the SEW (e.g., 8- and 16-bit multiplications are performed in two cycles while 32-bit requires 4 to 5 cycles) or specific conditions (e.g., dividing by zero takes a single cycle while dividing by valid values takes more).

Logical instructions (AND, OR, and XOR) are SEW-agnostic because they are bitwise operations. All the other instructions consider SEW in their execution. Since ELEN is 32, the supported SEWs are 8, 16 and 32.

Some operations offer clear opportunities for hardware reuse among SEWs, such as the addition. The 32-bit VALU adder comprises 4x 8-bit adders with carry-in and carry-out capabilities. Figure 5.3 details the architecture of the 32-bit adder. Adder 0 is responsible for the least significant 8-bit part of the data, while Adder 3 processes the most significant 8-bit part.

- When SEW is 8, carry-in connections on all adders are zero, and carry-out is ignored.
- When SEW is 16, adders 1 and 3 receive the carry-out of adders 0 and 2 as carry-in.
- When SEW is 32, the entire chain of carry-in and carry-out is connected, producing a 32-bit result.

This design avoids the need for specialized hardware for each supported SEW, minimizing the area overhead.

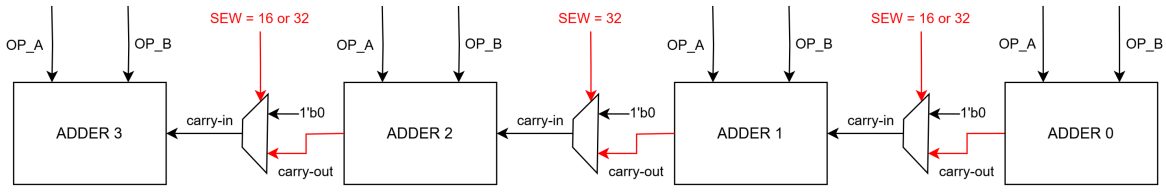


Figure 5.3: Hardware reuse in VXU's adders. The red color in the muxes control indicates which path is propagated when the condition evaluates truly.

The operands that feed the adder are multiplexed as they can come from different sources. The possible operands are the following: 1) First and second operands for regular addition (VADD); 2) First operand and the two's complement of a second operand for subtraction (VSUB and VRSUB); 3) Multiplication result and third operand (either regular or

two's complement) for the accumulation in MAC operations (VMACC, VNMSAC, VMADD, and VNMSUB).

The multipliers also offer hardware reuse opportunities. Multipliers can occupy a significant amount of the circuit's area footprint. Thus, in the RS5 execution unit and the VXU unit, the 32-bit multiples are composed of 17-bit multipliers controlled by an FSM, making it a multicycle operation. This way, the 32-bit multipliers can perform 8- or 16-bit multiplications using sign or zero extensions on the missing bits. A generic multiplier module with parameterized width is used for the smaller multipliers (8 and 16). A VLEN of 64 bits needs 2x 32-bit, 4x 16-bit, and 8x 8-bit multipliers. By applying hardware reuse, we can fulfill this requirement by using 2x 32-bit, 2x 16-bit, and 4x 8-bit multipliers.

- When SEW is 32, only the two 32-bit multipliers are used.
- When SEW is 16, the two 16-bit multipliers handle two multiplications, and the two 32-bit multipliers handle the other two.
- When SEW is 8, all the multipliers are used.

The MAC instructions use the same hardware as the multiplications and additions. The multiplication results are stored in a temporal barrier and used as a source operand for the adder in the next cycle. The accumulation step uses the adder to sum the stored multiplication results and the third operand, maximizing hardware reuse and requiring just an extra cycle compared to regular multiplications.

Due to its implementation, the divider module does not allow hardware reuse techniques similar to those used in multiplications. Division requires at least N cycles to perform an N-bit division. The divider module counts with a dedicated FSM with exclusive cycles to start and calculate the signal, making it very difficult to use the same hardware for smaller divisions.

Other instruction classes were implemented considering their simplicity or possible timing problems, given their complexity. The shift instructions were designed for simplicity, considering only the necessary shift control bits for each SEW. For instance, 8-bit operands require three shift control bits, whereas 32-bit operands need five control bits. The comparisons (equal and greater) operations focus on better timing, as the comparison results are used for other instructions, such as minimum (VMIN, VMINU), maximum (VMAX, VMAXU), and mask compare instructions (VMSEQ, VMSNE, VMSLTU, VMSLT, VMSLEU, VMSLE, VMSGTU, VMSGT).

Register Moves (VMV, VMVR, VMVSX, VMVXS) bypass the input operands to the outputs without any modifications. The VMERGE instruction combines two registers using a mask. If the mask is active, data is sourced from one register; otherwise, it comes from the alternate register.

In widening operations, each register processed by an instruction generates two output registers. A widening operation with a SEW of 32 bits would generate 64-bit elements,

which is invalid as it would surpass the ELEN=32, which defines the maximum element size an instruction can produce as 32 bits. Only widening multiplications were implemented in RS5 VXU. Widening operations require an extra cycle to execute, given the need to write two registers in the VRF.

The reduction instructions (VREDSUM, VREDMIN, VREDMINU, VREDMAX, VREDMAXU, VREDAND, VREDOR, and VREDXOR) condense a group of registers into a single scalar result, effectively merging all elements through the defined operation. For instance, in sum reductions, all elements in the register are added together, producing one output value rather than an output vector.

The reduction instructions also receive a scalar operand (the first position of the second vector operand) to be reduced with the input vector register. This extra scalar operand allows the reduction of a group of multiple registers. In the first cycle, the scalar operand will be the value coming from the first position of the second operand register. For subsequent registers in the group, the scalar operand will be the reduction result from the previous register.

In the RS5 VXU unit, the reduction instructions are implemented using a reduction tree exemplified in Figure 5.4. There are dedicated reduction trees for each supported SEW. The three have $\log_2 N$ levels, where N is the number of elements in the vector registers. For larger VLENs, the design will feature deep reduction trees, especially for 8-bit elements, which may create a critical path for timing constraints. The VXU unit offers the option to implement the reduction trees in using combinational or sequential circuits. The combinational version can reduce the entire vector register in a single cycle, offering good performance in exchange for a possible critical path for timing closure. The sequential version operates one tree level per cycle, resulting in slower operations but significantly better timing. However, this creates additional area overhead due to the need for extra flip-flops at each tree level.

Figure 5.4 represents the multiplexers for filtering the inputs in green. They are responsible for inserting neutral operands for the given operation (the neutral operand for addition is 0, and for logical AND operations is a binary 1). Neutral operands are inserted if the given element is masked off or the element index is bigger than the current VL. Figure 5.4 displays the reduction operation highlighted in orange boxes. In this figure, the depicted operation is a sum; however, for different instructions, the operation varies according to the instruction type (for instance, in the case of VREDAND, it is a bitwise AND).

The first level of the reduction tree is composed of the filtered input operands, with the least significant position being the filtered input operand index zero reduced with the scalar operand. The vector size is halved in each subsequent tree level until only one element remains at the final level, representing the reduction outcome.

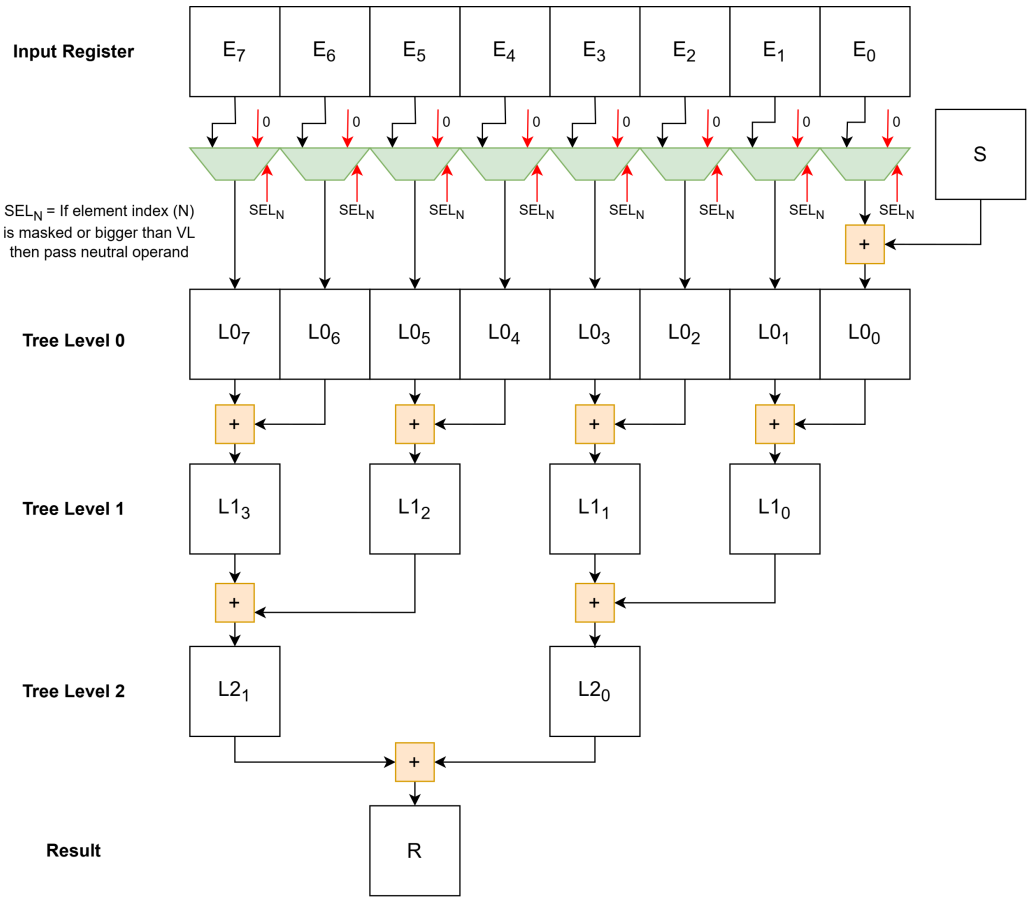


Figure 5.4: Sum reduction tree hardware for an eight-element register. The input register is filtered (considering mask and VL), and the first element accumulates the scalar value. The following tree levels reduce the elements until a single scalar value is achieved in the last level.

5.2.2 Vector Load and Store Unit

The Vector Load-Store Unit (VLSU) is the component responsible for memory accesses in RS5 VXU. It comprises a two-stage pipeline, with the first stage performing memory addressing and the second stage receiving data from memory reads. The second stage is needed because the memory model used in RS5 only supplies the requested data in the subsequent cycle. In the first stage of the pipeline, read signals are dispatched to memory, making the read data accessible in the subsequent stage.

RS5 memory interface always accesses the memory word-aligned (4 bytes), which means that the two least significant bits of the address are discarded. However, the VLSU supports “word-misaligned” access, where the memory access addresses are naturally aligned to SEW. This means that:

1) Elements with a SEW of 8 bits must be in byte-aligned addresses (e.g., 1, 2 and 3);

2) Elements with a SEW of 16 bits must be in half-word-aligned (2 bytes) addresses (e.g., 2, 4 and 6);

3) Elements with a SEW of 32 bits must be word-aligned (4 bytes) addresses (e.g., 0 and 4).

The support of word-misaligned access implies that different numbers of elements can be accessed between memory operations depending on the address alignment. Figure 5.5 shows the memory accesses for a vector of eight 8-bit elements. for different memory addressing modes. Figure 5.5 (a) and (b) show unit-strided accesses, respectively, with word-aligned and word-misaligned. In the word-aligned accesses (Figure 5.5 (a)), only two memory accesses are needed to load/store the entire vector. This is because the address is word-aligned, allowing four elements to be accessed per cycle. However, the first access can load only one element when the base address is word-misaligned, and the start address is 3 (Figure 5.5 (b)). The second memory access successfully reads four elements, so the number of elements accessed at the end of the second memory access is 5. Thus, an additional memory access cycle is required for just three elements, ignoring the fourth one. When the addressing mode is Unit-Strided, and SEW is 8, up to 4 elements per cycle can be accessed. When SEW is 16, up to 2 elements can be accessed per cycle. Logically, when SEW is 32 bits, only one element per cycle is accessed.

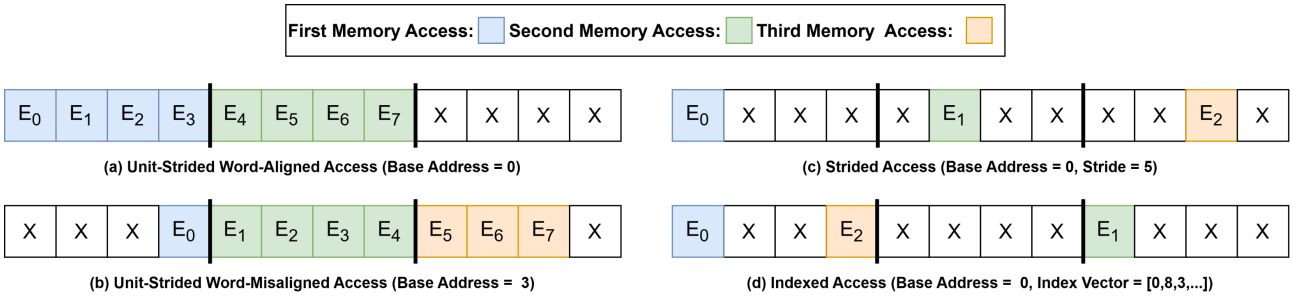


Figure 5.5: Memory access for different addressing modes of RVV extension in RS5 for eight elements of 8 bits. Figures (a) and (b) illustrate word-aligned and word-misaligned unit-strided access, whereas figures (c) and (d) demonstrate strided and indexed modes.

For strided and indexed memory addressing modes, represented in Figure 5.5 (c) and (d) respectively, the amount of elements accessed per cycle is always 1, independent of SEW configurations. There is no guarantee that the next element is within the window of the current memory access for the indexed addressing mode since the indexes might not correlate, leading to irregular memory accesses as the example in Figure 5.5 (d). In the strided addressing mode, although memory access is correlated (the next access address is offset from the current access address), subsequent elements are not guaranteed to be in the same access window (32 bits). For example, for memory accesses with a SEW of 8 bits, a stride bigger than 5 is enough to restrict the number of elements per access to just one, as Figure 5.5 (d) presents. The benefits of implementing extra hardware to support access to

multiple elements in strided and indexed addressing modes do not outweigh the complexity and area overhead.

Table 5.3 (page 83) presents the vector memory access instruction grouped by class (addressing mode) and their respective number of cycles needed to process an entire vector register.

To control all the logic in the VLSU, it counts with an internal FSM of four states:

VLSU_IDLE indicates that the unit is idle, waiting for a memory access instruction;

VLSU_FIRST_CYCLE indicates that the processing of a new vector register by a memory access instruction started, resetting all counters/accumulators related to vector register processing;

VLSU_EXEC performs the execution of the memory access instruction until the vector register is fully processed;

VLSU_LAST_CYCLE indicates that a vector register was fully processed by the memory access instruction. The resulting vector can be written in the VRF if it is a read operation. The FSM can either return to the VLSU_FIRST_CYCLE state until all the vector group (LMUL) is processed or to the VLSU_IDLE state when the group is fully processed.

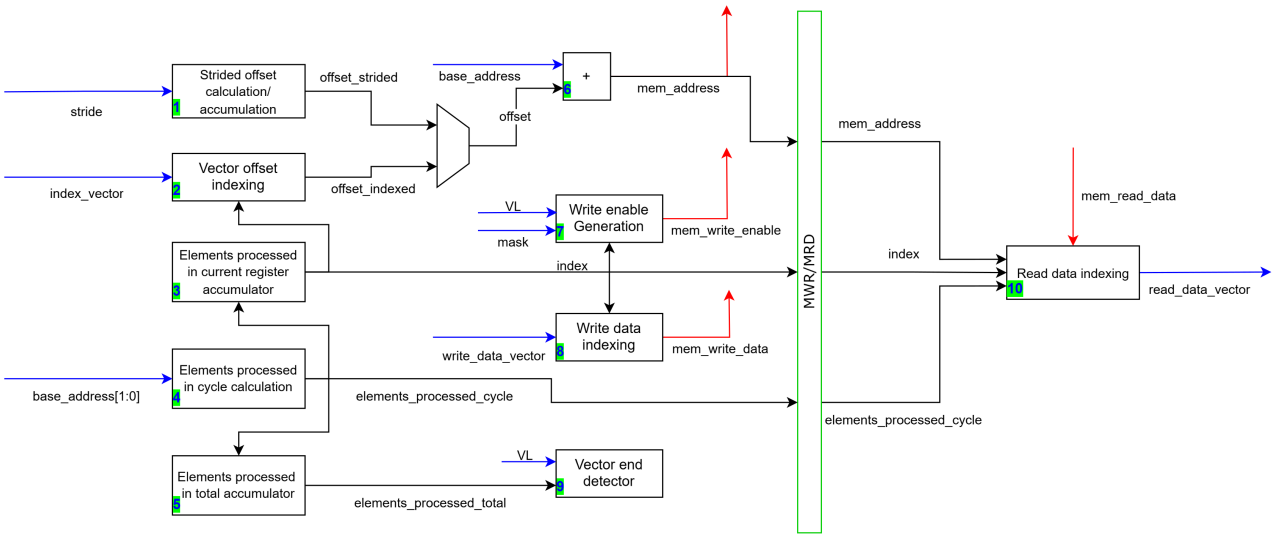


Figure 5.6: VLSU organization. Module signals to/from VXU are drawn as blue arrows. Red arrows represent signals to/from memory. The temporal barrier is drawn as a green rectangle

Figure 5.6 shows the organization of the VLSU. It shows the module input and output signals from VXU as blue arrows and the memory signals as red arrows. The inputs from VXU comprise the stride (for strided operations), the index vector (for indexed memory access), the base address, the vector to be written, the mask, and the Vector Length (VL).

The outputs to the VXU module are the hold signal (generation abstracted in Figure 5.6) and the data read vector. The memory signals are the memory address, the memory write enable, the data to be written to memory, and the data read from memory. The numbers in Figure 5.6 correspond to the components that implement the VLSU module:

1) Strided offset calculation/accumulation. Generates the strided offset. A fixed value increases an accumulator in every execution cycle. It resets to zero when the memory access instruction is completed or when a memory instruction is not being executed. The value added to the accumulator is either 4 when the memory addressing mode is unit-strided or the stride value when the memory addressing mode is strided.

2) Vector offset Indexing. Generates the indexed offset. It receives the index vector (one of the operands) and gets the current element offset from the vector considering SEW and the current element index.

3) Elements processed in current register accumulator Calculates the current element index. This accumulator resets when a register completes processing or when a memory instruction isn't executed. During each memory access cycle, the number of processed elements is added to the accumulator.

4) Elements processed in cycle calculation Generates the number of elements processed in each memory access cycle. It considers SEW, addressing mode, and memory alignment (the two least significant base address bits).

5) Elements processed in total accumulator Generates the number of elements processed during the current instruction execution. It is an accumulator similar to the one that holds elements processed in the current register, except that it does not reset when a register is fully processed but only at the end of the instruction execution. This allows for comparing the total number of processed elements with VL to check if the vector was not completely processed.

6) Mem address calculation Generates the effective memory address. It is an adder that sums the base address with the offset. The offset is multiplex between the strided and indexed based on the instruction addressing mode. The result of the sum is propagated to the memory.

7) Write enable Generation Generates the memory write enable. It is a mechanism that uses the current index to address the mask and check if the element is active (if not, the write will not be executed). It also checks if the current index is not bigger than the VL, indicating that it should also not be written.

8) Write data indexing Generates the data to be written in memory. It uses the current element index to index the write data vector (one of the operands) and get the data that should be written in the given memory position.

9) Vector end detector Generates a signal that indicates vector ending. It compares the VL and the total value of elements processed. If they match, then the vector processing is over, and the instruction can end execution.

10) Read data indexing Generates the data to be written in VRF. It receives the data read from memory and the other values stored in the temporal barriers. The index addresses the correct position in which the read data should be allocated on the output vector. Memory address and the number of elements processed in that cycle are used to calculate which part of the read data should be considered.

5.2.3 Related Work

This section aims to compare the RS5 implementation of the RVV extension with comparable cores presented in Section 3.2.1.

Arrow [Assir et al., 2021] implements a subset of the version 0.9 of the RVV extension. It also presents design time parameters for VLEN, and offer the option to change the number of Lanes and the ELEN. In the paper they analyzed a dual-lane architecture with VLEN of 256 bits and ELEN of 64 bits. Arrow is a coprocessor and have a pipelined datapath, similar to a processor pipeline, with decoding, operand fetch, execute or memory access, and write-back. The memory access is performed through a dedicated AXI interface and all memory accesses are 64 bits wide. Among the instructions implemented in the subset there are: unit-stride and strided memory access; single-width integer addition, subtraction, multiplication, and division; bitwise logic and shift; and integer compare, min/max, merge, and move operations.

ARA [Cavalcante et al., 2019; Perotti et al., 2022, 2024] can be configured with a variable number of identical lanes, with each having three execution units, an integer ALU, an integer MUL, and an FPU, all of them operating on a 64-bit datapath. RS5 VXU does not implement Floating point support, thus it differes from Ara in this point. Ara has a single memory port with a configurable width through a dedicated AXI interface.

Johns and Kazmierski [2020] uses a non-pipelined scalar core, thus the vector accelerator follows the same architecture as a multi-cycle component tighten to the processor's pipeline. It uses a 32-bit VLEN thus containing only one lane. For instructions using vector register groups, one register is operated on per cycle similar to what is done in RS5 VXU, except that each register comprises only one ELEN-element, and RS5 support registers with multiple ELEN elements. The VRF only implements 16 instead of 32 vector registers. The memory port is also limited to 32 bits, sharing the memory bus with regular load-store instructions.

Ali et al. [2021] uses a 32-bit ELEN to match the processor instruction width. The architecture is a two-stage pipeline with an instruction fetch and decode stage (fd-stage) and an instruction execution and write-back stage (ew-stage). The second stage also comprehends VLSU and VALU components. The VALU is composed of multiple lanes whose number can be configured. The VLSU has an interface with the CPU's data memory, and the OBI protocol is used for communication. The data and the address bit length are set to 32 bits.

ZeroVex [Zhao and Ye, 2024] uses a similar approach to RS5 VXU, tying the number of lanes with the VLEN. However, it uses a distributed VRF where each lane is furnished with 32 32-bit vector registers. The memory access has dedicated ports through an OBI interface, with each lane having a memory access port.

The related work demonstrates the variability in RVV extension implementations, with some designs featuring multiple lanes and dedicated memory ports, while others prioritize minimal hardware usage by simplifying the architecture as much as possible. The RS5 VXU adopts a balanced approach by combining multi-lane processing for data operations with single-lane memory access while maintaining simplicity through a reduced instruction subset. This design enables the integration of the RVV extension into RS5 without requiring modifications to the memory interface, while still achieving high throughput, reinforcing its role as a general-purpose processor.

5.3 VXU validation

Early validation of the RS5 VXU implementation was performed using manually generated assembly codes. It helped validate the initial instructions and the main components of the VXU, such as VRF accesses, memory accesses, and instruction executions. However, this approach was not the best since it had the designer's bias.

In the RV32I ISA, there are multiple validation suites. One of them is the unit tests [RISC-V Foundation, 2015], which test each instruction functionality individually for multiple operands. However, this suite lacks tests for vector instructions.

By searching for RVV extension validation suites, the Author crossed with the ARA [Cavalcante et al., 2019; Perotti et al., 2022, 2024] repository, which contains tests for the ARA vector accelerator. It includes tests for all the rv64uv instructions. Since the RS5 RVV implementation does not include all the instructions from the V extension, only the implemented instructions were selected. As the Ara accelerator includes 64-bit support, the tests needed to be modified to exclude the tests with SEW of 64 bits.

The majority of the unit tests use vectors of 16 elements. To hold an entire vector of 16 elements in a single vector register, as the tests assume, it is needed: 1) a VLEN of

128 bits for a SEW of 8 bits; 2) a VLEN of 256 bits for a SEW of 16 bits; and 3) a VLEN of 512 bits for a SEW of 32 bits. Thus, the VXU was tested with a VLEN of 512 to comply with the ARA's tests.

```
#include "vector_macros.h"

void TEST_CASE1(void) {
  VSET(16, e8, m1);
  VLOAD_8(v1, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8);
  VLOAD_8(v2, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8);
  asm volatile("vadd.vv v3, v1, v2");
  VCMP_U8(1, v3, 2, 4, 6, 8, 10, 12, 14, 16, 2, 4, 6, 8, 10, 12, 14, 16);

  VSET(16, e16, m1);
  VLOAD_16(v1, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8);
  VLOAD_16(v2, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8);
  asm volatile("vadd.vv v3, v1, v2");
  VCMP_U16(2, v3, 2, 4, 6, 8, 10, 12, 14, 16, 2, 4, 6, 8, 10, 12, 14, 16);

  VSET(16, e32, m1);
  VLOAD_32(v1, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8);
  VLOAD_32(v2, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8);
  asm volatile("vadd.vv v3, v1, v2");
  VCMP_U32(3, v3, 2, 4, 6, 8, 10, 12, 14, 16, 2, 4, 6, 8, 10, 12, 14, 16);
}
```

Code 5.1: C-code of the first test case for the VADD instruction unit-test.

Listing 5.1 presents the first test case of the VADD instruction. It tests the VADD instruction for different SEW configurations and checks the results. It initially sets the configurations using the "VSET" macro, which calls for a configuration instruction with the given parameters. It sets the VL to 16 elements, the LMUL to 1, and the SEW to 8, 16, and 32. After the configuration, the *v1* and *v2* registers are loaded with the vectors to be processed using VLOAD macros. The VADD instruction is then called using inline assembly. Lastly, the instruction results are compared to the expected results using a comparison macro.

Each instruction unit test contains multiple test cases involving different scenarios, such as masked and unmasked executions, different VLs, and different types of operands (vector, scalar, and immediate). A script was developed to help run all the unit tests. It compiles and executes each unit test individually and then classifies the result as passed or failed. It is a sort of regression that can be run after changing the processor to help identify any errors that may have been introduced by the modifications.

5.4 Conclusion and Final Remarks

This Chapter presented the VXU, a hardware component that implements the RVV extension in the RS5 processor as a tightly coupled accelerator. The modifications made to RS5 were discussed in detail, along with an in-depth analysis of the chosen design. Additionally, the design choices were explained, focusing on the VALU and VLSU components.

The RS5 VXU features a distinctive design that integrates both single-lane and multi-lane capabilities for memory access and computations, making it suitable for a general-purpose processor without needing modifications in the memory interface while providing high throughput.

Chapter 6 will present the applications used as benchmarks for testing the VXU accelerator performance, while Chapter 7 will show the results achieved with the accelerator.

6. CNN BENCHMARKS AND MANYCORE MAPPING

This Chapter presents the third and fourth **contributions** of this Dissertation: new CNN benchmarks for the RS5 processor and the manycore mapping technique. These were used to validate the RVV extension implementation of the RS5 processor and to generate the results that will be presented in Chapter 7.

The applications presented in this Chapter were successfully compiled using the vector extension in GCC 14.2. This compiler version added automatic vectorization, and its previous version, GCC 13, introduced support for the RVV extension instructions via inline assembly.

The ISA string in the Makefile was updated with the RVV extension, using the implemented subset (*zve32x*) to compile the application to the RS5 processor. The Minimum Vector Length specification (*zvl*) used for applications was 64 bits in general. However, tests were made using a supplementary compiling flag *-mrvv-vector-bits* that tells the compiler to use the value specified in the *zvl* ISA as target VLEN. This option often produces better performance results in exchange for compatibility. Another compiler flag explored was the *-mrvv-max-lmul* that gives a hint to the compiler of what size of vector register groups it should use, varying from 1 (*m1*) to 8 (*m8*) registers. Thus, an example of an ISA string in the makefile is: *-march=rv32im_zve32x_zvl256b -mrvv-vector-bits=zvl -mrvv-max-lmul=m8*.

This chapter is organized as follows.

- Section 6.1 presents the first benchmark application, a one-dimensional CNN. It details how it can be vectorized using the RVV extension instructions. It also evaluates the GCC compiler by analyzing the generated assembly codes. It also presents a manually-vectorized version that shows that the compiler auto-vectorization can still be improved.
- Section 6.2 introduces the second benchmark application, a two-dimensional CNN. It describes the characteristics of the selected network and explains how the initial code version was ported to RS5. Additionally, it details the process of dividing the original functions into smaller ones to enhance opportunities for manycore parallelism.
- Section 6.3 details how the 2D CNN was mapped to a manycore environment. It describes the selected technique for dividing the CNN application into smaller processes to improve parallelism. For that, it goes through a mathematical basis and shows how the smaller processes communicate and how they can be organized in multiple PE layers.
- Section 6.4 concludes this Chapter.

6.1 1-D CNN

This section presents the ML application for testing the RS5 VXU, a one-dimensional CNN (1D-CNN). The 1D-CNN application has a relatively simple architecture by today's standards [Reusch et al., 2023]. The primary objective is not to achieve state-of-the-art accuracy but rather to illustrate the effectiveness of simplistic CNN models in the context of Human Activity Recognition tasks. Figure 6.1 presents the 1D-CNN baseline model. It contains three convolutional layers and two fully connected (FC) layers. It adopts the rectified linear activation function (ReLU) after each convolution. The work presented by Reusch et al. [2022] extended this CNN, adding temporal awareness to the reference model, keeping its reduced complexity.

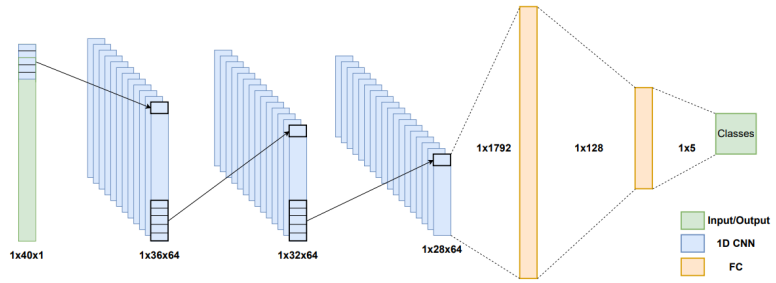


Figure 6.1: 1D-CNN reference model [Reusch et al., 2023].

In Reusch et al. [2023], the 1D-CNN was converted from the Pythorch model to a C model using integer representation. The model described in C language (C model) executes only the inference phase. The C model was built without standard libraries or function calls, making it a perfect fit for the test application or the RS5 VXU.

The model receives a unidimensional vector corresponding to the raw data extracted from the dataset as input. This raw data input has a shape of 1 × 120 (Figure 6.1 presents the reference model, which uses a 1 x 40 input). Considering that the input vector has 120 entries, *KERNEL_SIZE=5*, and the number of filters equals 64, this first layer executes 37,120 multiply-accumulate (MAC) operations (64 × 116 × 5). This model uses a bidimensional input shape of 64×116 and 64×112 in the second and thirdlayers. The first dimension, 64, corresponds to the number of filters. The second parameter corresponds to the input features size minus one (*kernel size* –1). The number of MAC operations is equal to 2,293,760 (64 × 112 × 64 × 5) and 2,211,840 (64 × 108 × 64 × 5) for the second and third layers, respectively. The FC 1 layer has an input size of 6,192,128 filters, resulting in 128 outputs, and executes 884,736 MACs. The FC 2 layer has an input size of 128 with 5 filters, resulting in 5 outputs, performing 640 MAC operations. For a single inference, it is necessary to execute 5,428,096 MAC operations.

The application was compiled to RISC-V using GCC 14.2. It was then executed on the RS5 processor to generate the baseline results. During the compilation process, the

assembly instructions generated by the compiler were analyzed, leading to several conclusions. One notable observation concerns the generation of division operations. GCC 14.1 tries to avoid hardware divisions, which are often slow even in hardware. This optimization is achieved whenever the divider or divisor is constant, and division instructions are inserted only when both operands are variables. This behavior was observed only in GCC 14.1; all older GCC versions did not implement this optimization and issued division instructions regardless.

By running the application in the baseline RS5, the validity of RS5 was once more proven, since it could issue the correct result for a data-intensive application. The memory for holding the input vector had to be expanded to more than 5 Megabytes. Using the performance monitor counters for instruction profiling allowed tracking the number of multiplications performed on execution. The resulting number was very similar to the theoretical value (5,428,096).

6.1.1 C-Language Model

Listing 6.1 presents a simplified version of the first convolution layer for the 1D-CNN in C-Language. The code implements a simplified version of the Equation (2.4), where one of the summations does not exist because the CNN only has one dimension (width). Initially, the application defines the network parameters (number of filters, kernel size, and input size) using the C-defined macros. In the *main* function, after the initialization of the variables (like input_vector), the auxiliary variables are declared:

- *bias* - an integer to hold the **bias** for the current filter;
- *kernel* - An integer vector to hold the five elements-wide **kernel** for each filter;
- *featureMap* - An integer matrix to hold the output feature map. It is a matrix because each filter generates an output feature map. The size of each feature map is the size of the input minus the kernel size (plus one).

After declaring the variables, the convolution operation is performed. The convolution is composed of a main loop that processes each filter. Inside the filters loop, the *kernel* and *bias* variables are loaded respectively with the weights and bias of the current filter.

From lines 23 to 28, the dot product operation is executed, consisting of the core convolution operation. The process includes a loop that produces each feature map output, comprising 116 iterations for every filter. Initially, it initializes the accumulator variable *totalSum* with zero. Then, it walks through the kernel window (5 elements), performing MAC operations. The MAC operations consist of multiplications of the kernel weight with the given input vector position and accumulate the multiplication result to the *totalSum* variable. After

the kernel window is done, the accumulation result is added with the bias and stored in the output feature map, concluding the convolution.

Lines 32 to 35 present the ReLU operation executed after each convolutional layer. It consists of two loops that iterate over the output feature map. Each position is checked to see if it is smaller than zero, if so, it is replaced with the zero value.

```
1  #define NUM_FILTERS 64
2  #define KERNEL_SIZE 5
3  #define CONV0_INPUT_SIZE 120
4
5  int main(){
6      ...
7      int bias = 0;
8      static int kernel[KERNEL_SIZE];
9      static int featureMap[NUM_FILTERS][CONV0_INPUT_SIZE-4];
10
11     for(i = 0 ; i < INPUT_SIZE ; i++)
12         input_vector[i] = dataset120[i];
13
14     // First Convolution Layer
15     for (int k = 0; k < NUM_FILTERS; k++)
16     {
17         for (i = 0; i < KERNEL_SIZE; i++) // Load Current Weights
18             kernel[i] = (int) (conv0_weights[i + (k * KERNEL_SIZE)]);
19
20         bias = (int) (conv0_bias[k]);
21
22         // Perform dot product (convolution)
23         for (i = 0; i <= (input_vector - KERNEL_SIZE); i++)
24             int totalSum = 0;
25             for (int j = 0; j < KERNEL_SIZE; j++)
26                 totalSum += (input_vector[i+j]) * kernel[j];
27             featureMap[k][i] = totalSum + bias ;
28         }
29     }
30
31     // ReLU
32     for (int k = 0; k < NUM_FILTERS; i++)
33         for (int i = 0; i < CONV0_INPUT_SIZE-4; i++)
34             if (featureMap[k][i] <= 0)
35                 featureMap[k][i] = 0;
36     ...
37 }
```

Code 6.1: Simplified C-code of the first convolution layer for the 1D-CNN.

6.1.2 Assembly Code - Scalar

This Subsection presents the assembly code generated from the C-code presented in Section 6.1.1 by the GCC 14.1 compiler using the only the base ISA with the M extension.

The input vector loading performed by Lines 11 and 12 in Listing 6.1 is translated to the vector assembly code presented in Listing 6.2.

Line 1 and 2 load *a5* and *a4* with the pointers to the memory areas that were allocated for the *dataset120* and the *input_vector,* respectively. Line 3 loads register *a3* with the address of the last element to be moved.

From Lines 6 to 9, four elements from the source vector are loaded to the scalar registers. The addressing uses immediate offset to access the four consecutive elements. Lines 11 to 14 store the four previously loaded values to the output register memory address using immediate offsets.

Lines 16 and 17 perform the memory pointers bumping by 16 bytes (4x elements of 4 bytes = 16 bytes). Line 18 is the loop control variable, which checks if the entire vector was moved. If not, it re-executes the loop until the operation is complete. The loop takes 11 clock cycles and processes four elements per iteration. To move the entire vector, it will take 333 clock cycles in the scalar assembly code (3 for loading pointers and $30 * 11$ (amount of iterations ($\frac{120}{4}$ = 30) multiplied by instructions per iteration)).

```
1  la a5, <dataset120>
2  la a4, <input_vector>
3  la a3, vector_end
4
5  move_vector:
6      lw  a6,0(a5)      # Load first  element to scalar reg
7      lw  a0,4(a5)      # Load second element to scalar reg
8      lw  a1,8(a5)      # Load third  element to scalar reg
9      lw  a2,12(a5)     # Load fourth element to scalar reg
10
11     sw  a6,0(a4)      # Store first  element in new position
12     sw  a0,4(a4)      # Store second element in new position
13     sw  a1,8(a4)      # Store third  element in new position
14     sw  a2,12(a4)     # Store fourth element in new position
15
16     addi  a5,a5,16 # Bump input  pointer
17     addi  a4,a4,16 # Bump output pointer
18     bne a5,a3, move_vector # Check if all elements were moved
```

Code 6.2: Commented scalar assembly instructions generated for reading raw data to input vector.

Listing 6.3 presents the vector assembly code for the convolution operation loop, presented by Listing 6.1 in Lines 23 to 28.

Lines 1 to 5 perform the memory pointers loading. Lines 7 and 8 initialize the loop control registers (the elements processed counter and the number of elements to be processed). Line 10 initializes the total sum variable to zero in register *a6*. Line 11 loads the bias to the register *t4*.

Lines 14 to 31 perform the convolution operation on each output element (output loop). Initially, Lines 14 and 15 copy the pointers for the current input vector address and the kernel weights vector. The copy instructions are needed because the pointers will be modified inside the loop to perform the kernel(dot product) operation (five input elements are convoluted with the five kernel weights). Lines 17 to 23 consist of the *kernel loop* and will be further detailed in the next paragraph. Line 25 sums the bias to the total sum calculated by the *kernel loop*. Line 26 stores the final result in the output memory position. Line 28 increases the elements processed counter. Lines 29 and 30 bump the memory pointers for the input and output vectors. Finally, Line 31 controls the loop, comparing the elements processed counter to the number of elements to be processed and jumping back to Line 14 until the loop is finished.

```
1  la  t1, <input_vector>
2  la  a5, <weights>
3  la  a0, <weights_end>     # Weights End Position (weights address +5 positions)
4  la  t2, <bias>            # bias memory address pointer
5  la  a7, <featureMap>      # Feature Map memory address pointer
6
7  li  a6, 0                 # Loop control (elements processed)
8  li  t3, 116               # Loop control (elements to be processed)
9
10 li  a2,0                  # The totalSum variable
11 lw  t4,0(t2)              # Load bias to t4
12
13 output_loop:
14     mv a3, t1             # Copy the current input element address
15     mv t5, a5             # Copy the first weight address
16     kernel_loop:
17         lw    a4,0(a3)    # Load element from input_vector
18         lw    a1,0(t5)    # Load kernel weight
19         addi  a3,a3,4     # Bump input pointer
20         addi  t5,t5,4     # Bump weights pointer
21         mul   a4,a4,a1    # Multiply input and weight
22         add   a2,a2,a4    # Accumulate with totalSum variable
23         bne   t5,a0, kernel_loop
24
25     add a2,t4,a2          # Adds the bias to the totalSum
26     sw  a2,0(a7)          # Stores result in the feature map vector
27
28     addi a6,a6,1
29     addi t1,t1,4
30     addi a7,a7,4
31     bne  a6,t3,188 output_loop
```

Code 6.3: Simplified and commented scalar assembly instructions generated for Convolution operation.

The kernel loop consists of the loop executed by the Listing 6.1 in Lines 25 and 26. It is executed 5 times (kernel size). Initially, in Lines 17 and 18, the input vector element and the kernel weight are loaded. Lines 19 and 20 bump the memory pointers. Line 21 performs the multiplication of the input vector element and the kernel weight. Line 22 accumulates

the multiplication result with the total sum variable. Finally, Line 23 controls the kernel loop iterations.

During the convolution operation, the output loop runs 116 times, with the kernel loop executing 5 times for each iteration. The kernel loop comprises seven instructions, including a multiplication instruction that requires four cycles to execute on the RS5 processor. Thus, each kernel loop iteration takes 10 clock cycles to be executed in the RS5 processor. The output loop consists of 8 instructions plus the kernel loop instructions. Thus, each output element needs 8 clock cycles from the output loop plus 50 clock cycles from the five iterations of the kernel loop. Thus, each output element takes 58 clock cycles to be calculated. The entire output loop would need 6728 ($116 * 58$) clock cycles. By adding the pointers loadings, the total number of instructions needed by the presented code is 6737 clock cycles.

Listing 6.4 presents the scalar assembly code for the ReLU operation, presented by Listing 6.1 in Lines 32 to 35.

```
1  la  a5,  <featureMap>        # First position of feature map
2  la  a3,  <featureMap_end>    # Last  position of feature map
3
4  relu:
5      lw   a4,0(a5)            # Load element
6      bgtz a4, incr_pointer    # If bigger than 0, then skip store instruction
7      sw   zero,0(a5)          # Stores 0 in negative elements memory position
8  incr_pointer:
9      addi a5,a5,4            # Bumps memory pointer
10     bne  a3,a5, relu        # Loop Control (116 elements)
```

Code 6.4: Commented scalar assembly instructions generated for ReLU (omitted redundant configuration instructions and filter loop).

Initially, it sets the register *a5* with the pointer to the output feature map vector address and the register *a3* with the vector end position (116 elements ahead). Line 5 loads the current element to the register *a4*. Line 6 is a branch instruction that checks if the loaded element is bigger than zero, if so, it skips the next instruction. The instruction in Line 7 is a memory store that stores the value zero in the element's memory position if the element is negative. Line 9 bumps the elements pointer. Line 10 manages the loop's termination after processing all 116 elements; if not, it returns to Line 5.

It takes five instructions to perform the ReLU on each vector element (considering the sometimes skipped load instruction). Thus, for the 116 elements, 580 clock cycles are needed, plus the initial two instructions that load pointers, for a total of 582 clock cycles.

6.1.3 Assembly Code - Vector Auto-Vectorized

This Subsection presents the assembly code generated from the C-code presented in Section 6.1.1 by the GCC 14.1 compiler using the vector extension and targeting a VLEN of 256 bits.

The input vector loading performed by Lines 11 and 12 in Listing 6.1 is translated to the vector assembly code presented in Listing 6.5

```
1  la a2, <dataset120>
2  la a3, <input_vector>
3  li  a4,480                  # Number of bytes to load (480 bytes = 120 words)
4  move_loop:
5      vsetvli a5,a4,e8,m8,ta,ma # Configure the vector unit (VLEN=480, SEW=8, LMUL=8)
6      vle8.v v8,(a2)          # Load bytes to vector register group
7      sub  a4,a4,a5           # Subtract the amount of processed bytes from the total (480)
8      add  a2,a2,a5           # Bump input pointer
9      vse8.v v8,(a3)          # Store bytes from vector register group
10     add  a3,a3,a5           # Bump output pointer
11     bnez a4, move_loop      # Loop until all 480 bytes are moved
```

Code 6.5: Commented vector assembly instructions generated for reading raw data to input vector.

Line 1 and 2 loads *a2* and *a3* with the pointers to the memory areas that were allocated for the *dataset120* and the *input_vector,* respectively. Line 3 loads register *a4* with the number of bytes to be loaded; as the number of elements is 120 (words), there are 480 bytes. Line 5 is the vector configuration instruction, setting VLEN to 480 elements, SEW to 8 bits, and LMUL to 8 registers.

In line 5, the *a5* register receives the number of elements of the vector that the vector unit will process in one iteration. Considering the parameters of *vsetvli* instruction, the number returned by the configuration instruction is 256 ((VLEN/SEW) * LMUL)). Lines 6 to 11 comprise the move loop that moves the entire vector. Line 6 presents the unit-strided memory read (vle8 = vector load elements of 8 bits). It accesses the memory at the base address in the *a2* (dataset120 address) register and stores the data in the vector register group starting at *v8*.

Line 7 performs the subtraction of bytes processed by the instruction iteration (held in *a5*) from the number of bytes to be loaded (*a4* initialized with 480). Line 8 bumps the memory pointer *a2* with the number of bytes already loaded. Line 9 stores data in memory using unit-strided access (vse8 = vector store elements of 8 bits). It stores the vector register group that starts at *v8* (data previously loaded by Line 6 instruction) at the memory address held by *a3* register (input_vector address). Line 10 then bumps the *a3* pointer with the number of bytes stored in this iteration. Finally, line 11 checks if the value held by *a4* (number

of bytes to be processed) has reached zero, indicating the vector was processed entirely. If so, it proceeds with the application's next section. Otherwise, it jumps to the instruction in Line 5 until *a4* reaches 0.

Each loop iteration processes eight elements and consists of 7 instructions, whereas the vector instructions are multi-cycle.

- vsetvli - Configuration instruction. It takes only one cycle to execute.
- vle8 - Vector load instruction. Loading one register takes $\frac{VLEN}{32}$ (from Table 5.3) cycles as the access is word-aligned. Thus, it takes $\frac{256}{32}$ = 8 cycles in the memory access plus the additional cycle needed in the VXU unit, resulting in 9 cycles to load eight 32-bit elements.
- vse8 - Vector store instruction. It takes the same amount of cycles as the vector load instruction.

Thus, each loop iteration takes five single-cycle instructions and two vector multi-cycle instructions, each taking nine cycles, resulting in 23 cycles to move eight elements. To move the entire vector of 120 elements, it needs 15 loop iterations. This results in 345 clock cycles plus the three needed to initialize the pointers and counters, for a total of 348 clock cycles to move the entire vector.

Listing 6.6 presents the vector assembly code for the convolution operation loop, presented by Listing 6.1 in Lines 23 to 28.

Lines 1 to 4 load the addresses of the input vector, kernel weights, bias, and output vector. Line 6 loads the total amount of elements to be processed (116). Lines 7 and 8 load the kernel weight initial index(0) and size(5).

Lines 11 to 32 perform the convolution operation on each output element (output loop). Line 11 is the vector configuration instruction; it sets the VL to 8, SEW to 32 bits, and a group of 1 register (LMUL=1). Line 12 is a move instruction that moves the immediate value zero to the vector register *v3*, It consists of the total sum variable initialized with value zero. The instruction in Line 12 uses scalar replication to spread a scalar through the entire vector register. Line 13 loads the bias to the vector register *v4* using a strided load instruction (it replicates the bias in all positions).

Lines 16 to 23 comprise the kernel loop, executed by the Listing 6.1 in Lines 25 and 26. It translated the C-code to a different structure, processing eight elements simultaneously, all operating with the same kernel index. Line 16 loads eight consecutive positions of the input vector using a unit-strided load instruction. Line 17 loads the weight using a strided load instruction with stride zero, resulting in all elements being loaded with the same weight. Lines 19 and 20 perform the kernel counter and address pointer bumping. Line 22 performs the multiplication of the input vector elements and kernel weights accumulating with the total sum register. Line 23 performs the kernel loop ending check. It checks if the current input vector slice has been accumulated with the entire kernel vector, jumping to line 16 if not finished.

```
1  la  a1, <input_vector>        # Input vector address
2  la  a4, <kernel_weights>      # Kernel weights address
3  li  a5, <bias>                # Bias pointer
4  la  a6, <featureMap>          # Output feature map address
5
6  li  t1, 116                   # Elements to be processed
7  li  a2, 0                     # Kernel counter
8  li  a3, 5                     # Kernel size
9
10 output_loop:
11     vsetivli zero,8,e32,m1,ta,ma # Configure the vector unit (VLEN=8, SEW=32, LMUL=1)
12     vmv.v.i  v1,0                # Move zeros to v1 (totalSum = 0)
13     vlse32.v v4,a5               # Load bias to v4  (stride zero)
14
15     kernel_loop:
16         vle32.v  v3,(a1)         # Load 8 elements of the input vector
17         vlse32.v v2,(a4),zero    # Load unique weight (replicated w/ stride zero)
18
19         addi a2,a2,1             # Increase kernel counter
20         addi a4,a4,4             # bump weights pointer
21
22         vmacc.vv v1,v3,v2 # totalSum[i+:8] += input_vector[i+:8] * weight[j]
23         bne a5,a3, calculate_total_sum      # Kernel Loop Control
24
25     vadd.vv v1,v1,v4 # Sum the kernel operation result with the bias
26     vse32.v v1,(a6)  # Store the results in memory
27
28     addi t1, t1, -8  # Update elements to be processed counter
29     addi a1, 32      # Bump input  vector pointer by 8 positions
30     addi a6, 32      # Bump output vector pointer by 8 positions
31     li   a2, 0       # Resets kernel counter
32     bne  zero, t1, <output_loop>
```

Code 6.6: Simplified and commented Assembly instructions generated for Convolution operation (omitted redundant configuration instructions and external loops).

After the kernel loop, line 25 adds the accumulation result (total sum variable) with the bias. Line 26 then stores the kernel results in the output vector memory position.

Line 28 decreases the amount of processed elements from the total amount. Lines 29 and 30 bump the input and output vector pointers by eight positions. Line 28 resets the kernel counter to 0. Finally, Line 32 performs the loop-ending check, jumping to Line 11 until all 116 elements are processed.

Each kernel loop iteration processes eight elements and consists of 6 instructions, whereas the vector instructions are multi-cycle.

- vle32 - Vector load instruction (unit-strided). Loading one register takes $\frac{VLEN}{32}$ (from Table 5.3) cycles as the access is word-aligned. Thus, it takes $\frac{256}{32} = 8$ cycles in the memory access plus the additional cycle needed in the VXU unit, resulting in 9 cycles to load eight 32-bit elements.
- vlse32 - Vector load instruction (strided). It takes the same number of cycles as the unit-strided load instruction because both are 32 bits.

- vmac - Vector multiply and accumulate instruction. It takes five cycles to complete (4 for multiplication and 1 for accumulation), plus the additional cycle needed in the VXU unit, resulting in 6 cycles to multiply and accumulate eight 32-bit elements.

Thus, the kernel loop iteration takes three clock cycles for single-cycle instructions and 24 for multi-cycle instructions (18 from loads and 6 to MAC), totalizing 24 clock cycles to process eight elements. The five kernel loop iterations take 120 clock cycles to complete.

The output loop has nine instructions (apart from the kernel loop), where four are regular and 5 are vector instructions.

- vsetivli - Configuration instruction. It takes only one cycle to execute.
- vmv - Register move instruction. It takes one cycle per register to execute, plus an additional cycle needed in the VXU unit, resulting in 2 cycles.
- vlse32 - Vector load instruction (strided). Loading one register takes $\frac{VLEN}{32}$ (from Table 5.3) cycles as the access is word-aligned. Thus, it takes $\frac{256}{32}$ = 8 cycles in the memory access plus the additional cycle needed in the VXU unit, resulting in 9 cycles to load eight 32-bit elements.
- vadd - Vector addition instruction. Additions in RS5 VXU take 1 cycle to complete, plus the additional cycle needed in the VXU unit, resulting in 2 cycles to add eight 32-bit elements.
- vse32 - Vector store instruction (unit-strided). It takes the same number of cycles as the load strided load instruction because both are 32 bits.

Thus, executing one iteration of the kernel loop takes four clock cycles from regular instructions and 23 from vector instructions (1 for configuration, 2 for move, 18 for load/stores, and 2 for addition). Considering the kernel loop, each iteration of the output loop takes 143 clock cycles to calculate eight output elements. As the number of iterations required to process the 116 elements is 15, the total number of clock cycles spent in the output loop will be 2145. With the seven initial instructions (pointers loading), the code snippet would take 2152 clock cycles.

Listing 6.7 presents the vector assembly code for the ReLU operation, presented by Listing 6.1 in Lines 32 to 35. Initially, it sets the register *a4* with the pointer to the output feature map vector address and the register a5 with the vector size (116 elements). Line 4 (the configuration instruction) sets the register group as a single register (LMUL=1), SEW = 32, and VL = 8 elements. Line 5 consists of loading vector register *v1* with zeros (using immediate replication).

The ReLU loop goes from Line 7 to 12. Line 7 loads a partition of the vector into the *v0* vector register. It loads 8x (VL) 32-bit elements into the *v0* register. Line 8 subtracts the number of processed elements from the total amount previously loaded into register *a5*. Line 9 is responsible for seeking elements with negative values. For that, it uses a mask instruction called *vmsle*, which is the abbreviation for "*vector mask set if less or equal*". Mask instructions create a mask based on a condition; each element generates a single mask bit,

setting if the condition evaluates truly. The instruction in line 9 compares if the elements are negative by comparing them with the immediate 0. The generated mask is stored in the vector register *v0*, used as the mask for masked instructions.

```
1  la   a4, <featureMap>          # Vector address
2  li   a5,116                    # Vector size (elements to be processed)
3
4  vsetivli zero,8,e32,m1,ta,ma # VL=8, SEW=32, LMUL=1
5  vmv.v.i v1,0                   # Fill register v1 with zeros
6  relu_loop:
7      vle32.v  v0,(a4)           # Load 8 elements to v0
8      addi   a5,a5,-8          # Decreases the processed elements from total
9      vmsle.vi v0,v0,0           # Set mask on negative elements positions
10     vse32.v  v1,(a4),v0.t      # Stores zero on elements with mask set
11     addi   a4,a4,32          # Bumps memory pointer in 8 positions
12     bne      zero,a5,<relu_loop> # Loop-ending chek
```

Code 6.7: Commented Assembly instructions generated for ReLU.

Line 10 performs the last part of the ReLU: replacing negative elements. For that, it uses a masked store instruction (argument *v0.t* indicates it is a masked instruction). The masked store will only store the data in the *v1* vector register (zeros) if the mask bit corresponding to that element is set. It uses the previously generated mask. Thus, it will store zeros only in the memory positions that contain negative elements. Line 11 performs the memory pointer bumping for the output vector address. Finally, Line 12 controls the loop-ending, returning to loading elements until the entire vector has passed through ReLU.

The ReLu loop processes eight elements per iteration, consisting of three regular and three vector instructions. In the RS5 VXU implementation, the given load and store instructions take nine cycles to execute, while the mask instruction takes only two cycles. Thus, it takes 23 clock cycles to execute the ReLU loop. The ReLU loop will be executed 15 times, resulting in 345 clock cycles needed. With the preparation instructions, this number totalizes 350 clock cycles.

This section highlights the capabilities of GCC 14.2 auto-vectorization, being capable of translating many loops into vector instructions. Table 6.1 compares the number of clock cycles needed for each operation for the scalar and vector assembly versions and the respective ratio between both. It becomes clear that pure memory instructions do not have any advantage, as expected, since the VXU does not alter the RS5 memory capabilities. Instead, the vector version of the vector move operation required additional cycles due to the VXU startup cycles and additional control/configuration instructions.

The picture changes when the operations involve arithmetic operations since it is where the main VXU parallelism occurs. In the convolution operation, the vector version takes less than a third of the clock cycles required for the scalar version, highlighting the

Table 6.1: Comparison between the number of clock cycles needed for each operation for the scalar and vector assembly versions. The last column presents the ratio between both ($\frac{vector}{scalar}$).

Operation	Scalar	Vector (256 bits)	Ratio
Vector move	333	348	1.04
Convolution	6737	2152	0.32
ReLU	582	350	0.60

acceleration capabilities. The GCC choice to have multiple elements operating with the same kernel weight was very interesting in this operation, as it made a mix of two loops.

In the ReLU operation, the vectorized version took about only sixty percent of the cycles that the scalar version takes. The compiler's choice to use a masked load store is also an interesting use of the available instructions in the vector extension.

Although the compiler proved very capable of auto-vectorizing the 1-D CNN code, it did not fully explore the RVV extension capabilities. Instead, it limited itself to using register groups of just one register in arithmetic operations. It also uses several redundant configuration instructions omitted from the presented codes. Addressing these issues may lead to even better results.

6.1.4 Assembly Code - Vector Manually Vectorized

The Subsection aims to address some of the issues pointed out in the Section 6.1.3 by making a manually vectorized assembly. It is based on the auto-vectorized version but has significant improvements.

Listing 6.8 presents the vector assembly code for the entire convolution layer, presented by Listing 6.1.

This code aims to delve deeper into parallelism by utilizing groups of 8 registers (LMUL=8). An optimization was that instead of storing the input vector in the dataset memory position, we directly used the values in the registers to start the computation. This avoided storing the values in a new memory position and loading them from there. Another optimization regarding memory access is performing the ReLU operation on the data immediately after the convolution and before storing its results in memory. It also uses the variants of the arithmetic instructions using replicated scalar register values as operands, thus avoiding vector move instructions.

Lines 1 to 4 perform the pointers loading. Lines 6 to 8 initialize the control variables needed (kernel index, kernel size, and output size). Line 9 loads the bias to a scalar register.

Lines 12 to 34 correspond to the output loop. Line 12 presents the configuration instruction, it uses a SEW of 32 bits and LMUL of 8. The AVL value is 116, but the VXU,

with a VLEN of 256 bits, can not handle that many elements in just 8 registers. Thus, the instruction result, placed in the *a1* register, is the number of elements it will process in the current iteration. With a VLEN of 256 bits and the selected configurations, the maximum number of elements a register group can hold is 64 (see Equation (2.3)). Line 13 shifts the value held in *a1*, effectively multiplying it by four. The shift result is the number of bytes that the memory pointers must be bumped. Line 14 moves the immediate value zero to the register group starting with register *v24*, effectively filling all the registers from *v24* to *v31*. This register group is the total sum variable vector.

```
1  la          s5, <dataset120>
2  la          a3, <featureMap>
3  la          a4, <weights>
4  la          a5, <bias>
5
6  li          t1,  0                          # Kernel counter
7  li          t5,  5                          # Kernel size
8  li          a0, 116                         # Output size
9  lw          t2, 0(a5)                       # Load bias to t2
10
11 output_loop:
12     vsetvli     a1, a0, e32, m8, ta, ma     # VL=116, SEW=32, LMUL=8
13     slli        t3, a1, 2                   # processed elements * 4 (for memory bump)
14     vmv.v.i     v24, 0                      # Fill regs (v24-v31) with zeros (totalSum)
15
16     kernel_loop:
17         vle32.v     v8, (s5)                # Load elements to (v8-v15) regs
18         lw          t4, (a4)                # Load bias to scalar t4 reg
19         addi        a4, a4, 4               # Bumps kernel pointer
20         addi        t1, t1, 1               # Increases kernel counter
21         vmacc.vx    v24, t4, v8             # Mult (v8-15) by scalar in t4 + (v24-v31)
22         bne         t1, t5, kernel_loop
23
24     vadd.vx     v24, v24, t2                # Add bias to totalSum variable
25
26     ReLU:
27         vmsgt.vi    v0, v24, 0              # Set mask on positive elements
28         vse32.v     v24, (a3), v0.t         # Stores only positive conv results
29
30     sub         a0, a0, a1                  # Subtract processed elements from total
31     add         s5, s5, t3                  # Bumps input pointer by processed positions
32     add         a3, a3, t3                  # Bumps output pointer by processed positions
33     li          a4, 0                       # Resets kernel index
34     bgt         zero, a0, <output_loop>     # Loop control
```

Code 6.8: Commented manually generated assembly instructions generated for first conv layer of the CNN.

The kernel loop is implemented by Lines 17 to 22. It uses the same structure as the kernel loop of the auto-vectorized assembly. The only difference is that here it operates on a larger register group.

Line 24 adds the kernel operation results to the bias. It uses the scalar variant of the *vadd* instruction, getting the second operand directly from a scalar register. The content of register *t2* (bias) is replicated into an entire vector register to before the addition operation.

Lines 27 and 28 perform the ReLU operation, similar to the auto-vectorized assembly. However, instead of storing all the values in memory and later replacing negative values with zeros, the manually vectorized version only stores the non-negative values. For that, it uses the *vmsgt* (vector mask set if greater than) instruction comparing with the immediate value zero. Finally, Line 28 performs the memory store only on the positive values using the generated mask.

Lines 30 to 33 are the pointers bumping and index/counters control. Line 34 performs the loop-ending control by checking if the entire vector was processed. As one iteration processes 64 elements, two iterations would be needed.

In this case, as the LMUL is eight, the vector instructions would take more cycles to be executed, directly related to the number of elements it will process.

- vsetivli - Configuration instruction. It takes only one cycle to execute.
- vmv - Register move instruction. One cycle per register, plus an additional cycle needed in the VXU unit, resulting in 9 cycles.
- vle32 - Vector load instruction (strided). Loading one register takes $\frac{VLEN}{32}$ (from Table 5.3) cycles as the access is word-aligned. Thus, it takes $\frac{256}{32} = 8$ in the memory access plus the additional cycle needed in the VXU unit, resulting in 65 cycles to load 64x 32-bit elements.
- vmacc - Vector Multiply and accumulate. It takes five cycles per register and the additional cycle needed in the VXU unit, resulting in 41 cycles multiplying and accumulating 64x 32-bit elements.
- vadd - Vector addition instruction. One cycle per register and the additional cycle needed in the VXU unit, resulting in 9 cycles to add 64x 32-bit elements.
- vmsgt - Vector mask instruction. One cycle per register and the additional cycle needed in the VXU unit, resulting in 9 cycles to generate mask for 64x 32-bit elements.
- vse32 - Vector store instruction (unit-strided). It takes the same number of cycles as the load instruction because both are 32 bits.

The kernel loop takes 109 clock cycles per iteration. Thus, executing an iteration of the output loop would take 10 clock cycles for regular instructions and 638 ($93 + 5 * 109$) for vector instructions. The total number of clock cycles to execute the code snippet is 1284 (initial control instructions (8 clock cycles) and the two iterations of the control loop).

The number of cycles needed to process the first convolution layer of the 1-D CNN in the manually vectorized assembly and the auto-vectorized assembly can be compared. The auto-vectorized version's execution would take 2502 clock cycles (ignoring the vector move operation to make a fair comparison), while the other would take only 1284. This is a ratio of *0,51* of the clock cycles needed, an improvement of almost 2x. When comparing

this result to the scalar version, the improvement reaches around 5,71x (a ratio of 0,175). These theoretical results highlight the acceleration potential in the RS5 VXU unit for CNN applications.

6.2 2-D CNN

The two-dimensional CNN chosen as a benchmark is the Alexnet [Krizhevsky et al., 2017]. In 2012, AlexNet won the ImageNet competition, marking a significant milestone in neural network development. The AlexNet model comprises five convolutional layers that utilize filters with dimensions 11x11, 5x5, and 3x3. It encompasses 60 million parameters. AlexNet's success demonstrated the power of deep learning in solving complex vision problems. Although old, AlexNet is one of the most relevant networks due to its historical importance, as it paved the way for more advanced/modern CNN architectures.

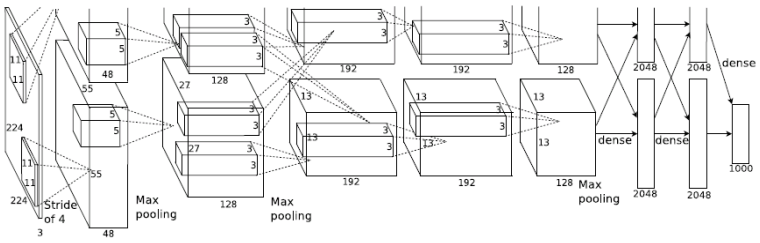


Figure 6.2: 2D-CNN reference model - AlexNet [Krizhevsky et al., 2017].

The network architecture, as illustrated in Figure 6.2, features five convolutional layers followed by three fully-connected (FC) layers. From left to right, the structure includes the input layer, followed by four convolutional hidden layers and two fully-connected layers, culminating in the final fully-connected output layer.

Table 6.2: AlexNet layers information (Source [Korol, 2019]).

Layer	Operation	Input Size (padded)	Weight/ Filter Size	Output Size
1	CONV	227x227x3	11x11x3 (x96)	55x55x96
1	MAX-POOL	55x55x96	3x3	27x27x96
2	CONV	31x31x96	5x5x96 (x256)	27x27x256
2	MAX-POOL	27x27x256	3x3	13x13x256
3	CONV	15x15x256	3x3x256 (x384)	13x13x384
4	CONV	15x15x384	3x3x384 (x384)	13x13x384
5	CONV	15x15x384	3x3x384 (x256)	13x13x256
5	MAX-POOL	13x13x256	3x3	6x6x256
6	FC	6x6x256	6x6x256x4096	4096
7	FC	4096	4096x4096	4096
8	FC	4096	4096x1000	1000

The parameters of each layer are detailed in Table 6.2. This table shows that the input layer consists of three channels, representing the RGB color channels. The first layer kernel has a *11x11x3* filter, which generates a *55x55* output. For the first layer, there are

96 Filters, resulting in 96 output channels. The pooling layer helps to reduce the first layer dimensions using a max-pool filter, it reduces the output size from *55x55* to *27x27* by using a filter size of 3x3. To resolve the incompatible sizes of the first layer output feature map (*27x27x96*) and the second layer input (*31x31x96*), a pad operation is applied. The pad operation adds zeros on the borders of the incompatible size until it achieves the desired shape. The other layers follow a similar structure and will not be further detailed here. It is worth citing that convolutional layers 3 and 4 are not connected to pooling layers.

Table 6.3 presents the number of multiplication operations per layer in the AlexNet CNN, highlighting its computationally intensive nature. This further demonstrates how such applications can benefit from specialized hardware. Hardware acceleration enhances the performance of data-intensive applications, making them more suitable for various environments.

Table 6.3: Multiplication operations per layer in Alexnet (Source [Korol, 2019]).

Layer	**Multiply Ops**
1 - CONV	105,415,200
2 - CONV	447,897,600
3 - CONV	149,520,384
4 - CONV	224,280,576
5 - CONV	149,520,384
6 - FC	37,748,736
7 - FC	8,388,608
8 - FC	2,048,000
Total =	**1,124,819,488**

The AlexNet CNN source code used by the Author was written in C-language and used floating-point parameters. This network implementation had to be ported to use integer values because neither the RS5 nor the VXU supports floating-point operations. Normalization of the parameters was performed using Python scripts. These scripts read the source parameters and multiply them by a fixed factor 1024 before casting them into integers. The 1024 value was chosen because it simplifies some hardware operations. Since 1024 is a power of two, multiplying or dividing any number by 1024 can be done using bit shift instructions, which are less costly than multiplying/dividing instructions. After normalizing the parameters, the next step was to adjust the C-language code to implement integer parameters/operations instead of floating-point. The normalization process was not concerned with the resulting network precision. The primary goal was to execute the network on the RS5 processor.

The C-language code uses macros to define the dimensions of each network layer accordingly to Table 6.2. It also uses generic functions that receive (through function parameters) the layer parameters and dimensions, as Listing 6.9 shows. This is a good practice because it allows the same C function to be used for different layers. It also allows the same

code to be used for different networks by simply providing the new parameters and adjusting the macros to reflect the new layer shapes.

```
1 void relu(float data[], const unsigned input_height, const unsigned input_width, const
      unsigned input_depth)
```

Code 6.9: Header of the C-Language ReLU function.

The convolution C-function was initially very complex. It performed all the computations required by the convolution layers, considering multiple input and output channels, strides, and layer shapes. This function was composed of a total of 6 nesting loops (in order: output depth, input depth, output height, output width, kernel height, and kernel width). The convolution function was divided into multiple functions for simplicity while targeting parallelism opportunities. This modification resulted in four C functions.

conv_kernel(...) - This function performs the kernel operation in the current kernel window. It receives the input data, the given kernel, and the kernel dimensions. It has two nested loops that iterate over the kernel dimensions (kernel height and width). The function returns the accumulation result.

conv_channel(...) - This function operates on an entire input channel. It receives the input data (and dimensions), the channel kernel/filter (and dimensions), and output dimensions. It has two nested loops that iterate over the output channel dimensions (output height and width). On each inner loop iteration, it calls the conv_kernel function. This way, the kernel slides over the entire input channel (applying stride) and generates the resulting output channel. The function returns the calculated output channel.

conv_sum_bias(...) - This function sums the given bias to the layer's given output channel. It receives the data, data dimensions, the biases, and the output channel identification (to get the bias for the given output channel). It has two nested loops that iterate over the output channel dimensions (output height and width). The function has no return as it operates directly on the output memory area received via parameter.

conv(...) - This function performs the entire convolution operation by calling the other functions. It receives all the layer data and parameters. It has two nested loops that iterate over the output channels (output depth) and input channels (input depth). The inner loop, responsible for the input channels, calls the conv_channel(...) function to perform the channel computation. The function return (resulting channel) is then accumulated with the output channel. After the end of the inner loop, the outer loop, responsible for the output channels, calls the conv_sum_bias(...) function to finish the computation of the output channel. The function has no return as it operates directly on the output memory area received via parameter.

The two versions of the application (integer and integer with divided functions) were tested on the host computer to compare the output's equivalence. After validation, both applications were executed on the scalar version of the RS5 processor. The compilation with the RVV extension was successful. The compiler can auto-vectorize the 2-D convolution operations, similar to the 1-D CNN. Finally, the auto-vectorized code was executed in the RS5 processor with the vector extension, generating results that will be presented in the next Chapter.

6.3 Manycore Mapping

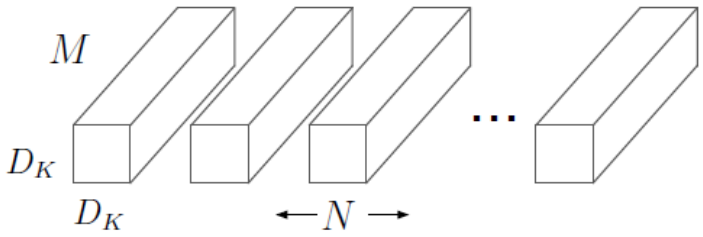
Chapter 3 presented the state-of-the-art regarding NoCs and CNNs. Table 3.4 summarized the selected works. The acceleration methods involve NoC modifications, dedicated NoCs, and mapping methods, all seeking to maximize the parallelism. This work seeks to use specialized Processing Elements (PEs) for acceleration in the context of a manycore. The Author does not intend to modify the NoC, as it already has high throughput. This Dissertation explores the impact of a specialized PE on CNN applications in manycores.

The initial approach involved mapping different CNN layers to different PEs. While this enables some parallelism, direct dependencies and varying computational costs across layers can result in suboptimal performance. Additionally, this method does not fully exploit parallelism opportunities within each layer. The selected manycore mapping technique maximizes parallelism by leveraging the independence of input channels. A literature review revealed works that employ this method, referred to as **depthwise convolutions**. Howard et al. [2017] introduces the concept of **depthwise separable convolutions**, which consist of two layers: depthwise convolutions and pointwise convolutions.

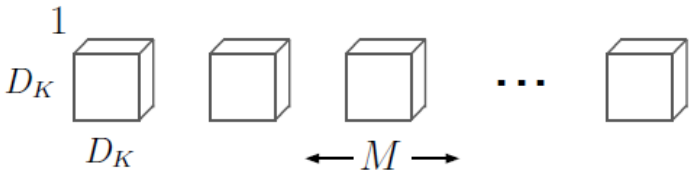
- Depthwise Convolutions: each input channel is convolved independently using its filter. This approach decreases computational load compared to standard convolutions, which aggregate information from all channels simultaneously.
- Pointwise Convolutions: a 1×1 convolution merges the outputs from depthwise convolutions. This process integrates data across channels, functioning as a linear combination of the input channels.

Figure 6.3 presents the filter structure for standard convolutions in (a), depthwise convolution in (b), and pointwise convolution in (c). Where D_k is the spatial dimension of the kernel, M is the number of input channels, and N is the number of output channels.

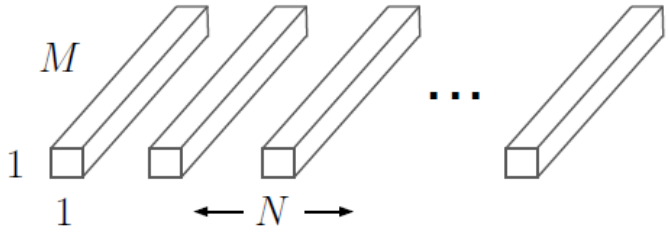
Using depthwise convolutions allows mapping different input channels to different PEs, improving the parallelism inside each layer. Since each group of input channels and their corresponding filters creates a single output channel, this suggests that the processing



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 6.3: Standard convolutional filters in (a). Depthwise convolution in (b) and pointwise convolution in (c) (Source [Howard et al., 2017]).

of various output channels can be viewed as independent. This leads to another parallelism opportunity: a first PE layer processes the input channels using depthwise convolution and then sends the resulting feature map (of a single output channel) to a second PE layer, which gathers all the resultant feature maps from the depthwise convolution. This allows the first PE layer to proceed to the following output channel while the second PE layer processes the merge of results of all the first layer PEs.

Formally, Equation (6.1) describes the convolution operation performed by the first PE layer. It differs from Equation (2.4) by not having the outer summation (which sums the input channels).

$$\mathbf{O_p}[c_o][x][y] = \sum_{i=0}^{\text{Width}-1} \sum_{j=0}^{\text{Height}-1} (\mathbf{I}[K][Sx+i][Sy+j] * \mathbf{W}[c_o][i][j]) \tag{6.1}$$

where: c_o is the current output channel; x and y are the horizontal and the vertical positions; Width and Height correspond to the filter size; S is the stride; O_p is the partial output; I is

the input; W is the filter tensor; K is the input channel (fixed parameter in PEs that performs depthwise convolution).

To fulfill the absence of the outer summation, there needs to be c_i (number of output channels) PEs in the first PE layer performing the computation presented by Equation (6.1). Each PE has a fixed K parameter indicating which input channel it is responsible for processing. The operation output O_p is now partial and needs to be summed with the partial outputs from the other input channels to generate the output feature map. The operation described by Equation (6.1) is executed for every output channel in the first PE layer.

Formally, Equation (6.2) describes the convolution operation performed by the second PE layer. It fulfills the Equation (6.1) to complete the convolution operation implemented by Equation (2.4). It summates all the partial outputs from the first PE layer for each output channel. Then, sum the Bias (**B**) to generate the complete output feature map. The second PE layer also performs the ReLU and Max-Pool operations if needed to balance computations. After that, it sends each output feature map to a third PE layer, which starts the second convolution layer using depthwise convolution.

$$\mathbf{O}[c_o] = \mathbf{B}[c_o] + \sum_{k=0}^{C_i-1} O_p[c_o][C_i] \tag{6.2}$$

The amount of PEs can vary according to the layer depth and manycore dimensions. The first convolutional layer of the chosen application (AlexNet) has only three input channels corresponding to RGB layers. Thus, in this case, the first PE layer contains 3 PEs, each responsible for a unique input channel. Each PE will handle 96 (first layer output depth) iterations of the depthwise convolution. After the end of each iteration, the results are sent to the second PE layer.

The second PE layer comprises one PE and is responsible for receiving the data sent by the first PE layer after the processing of each output channel. It then sums the partial feature maps, adds the bias, and performs the ReLU and Max-Pool operations. The output feature map is then sent to the third PE layer.

The third PE layer has an input depth of 96 and an output depth of 256. Because the input layer is more profound (96), having dedicated PEs for each input channel becomes unfeasible, as this would require a significant number of PEs. Thus, each PE in the third PE layer processes multiple input channels. The number of channels that each PE will process depends on how many PEs will be allocated for the third PE layer($\frac{96}{PEs_3}$). The PE in the third PE layer already accumulates the partial outputs of the processed input channels before being sent to the fourth PE layer.

The fourth PE layer is similar to the second one. It is responsible for gathering the partial outputs, summing them, performing the ReLU and Pooling operations if required, and then sending the result to the fifth PE layer, which has a structure similar to the third layer.

The 2-D CNN application (Alexnet) was modified to divide the CNN structure into depthwise convolution. For that, four different C-Language programs were created. They are named with a leading P (which stands for program) followed by the number identifying the PE layer to which it belongs. Only four PE layers are implemented, so the programs handle only the first two convolutional layers of AlexNet.

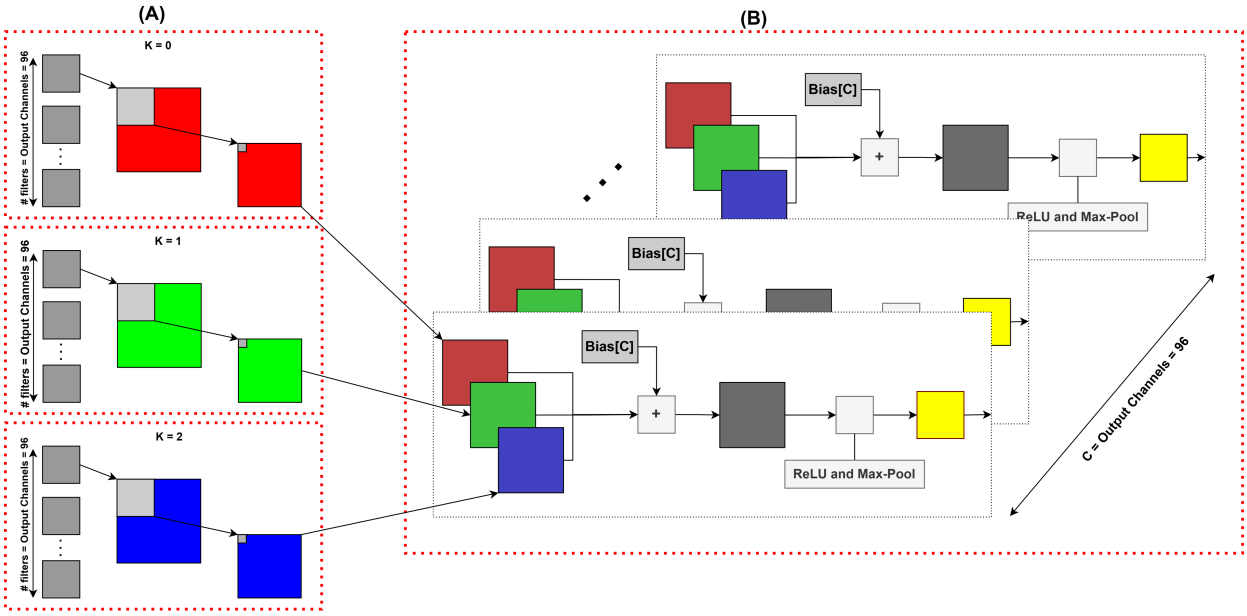
- *P1* - Entry point program, implementing the operations performed by the first PE layer. The program handles a single input channel and then sends it to the second program. It requires three instances of this program (3 processes running the same program with a different K parameter (input channel)).
- *P2* - implements the operations performed by the second PE layer. Receives the partial outputs from the P1 processes and sums them to generate the output feature map. The ReLu and Max-Pool operations are applied, and each output channel is sent to the next PE layer.
- *P3* - implements the operations performed by the third PE layer. It handles multiple input channels (depending on the number of PEs in the layer and convolution layer depth). Receive the input channels from the P2 process, perform the depthwise convolution, and then send it to the next PE layer.
- *P4* - implements the operations performed by the fourth PE layer. Receives the partial outputs from the P3 processes and sums them to generate the output feature map. The ReLu and Max-Pool operations are applied, and each output channel is sent to the next PE layer.

Figure 6.4 presents the P1 (with three instances/processes) and P2 (with a single instance/process) programs handling the first convolutional layer of AlexNet. The red dashed lines delimit the computations performed by each process.

Each P1 process represented in Figure 6.4 (A) is depicted with the channel color it processes, having the K parameter described at the top. The 96 filters for the 96 output channels are represented in gray. The partial outputs are sent to the P2 process. There are 96 partial outputs sent to the P2 process (one for each filter). Process P2 represented in Figure 6.4 (B) starts receiving the partial outputs and then performs their accumulation and the Bias sum. It also represents the ReLU and pooling operation. P2 operates 96 times (one for each output channel), generating 96 output feature maps (in yellow).

Figure 6.5 presents the P3 (with eight processes) and P4 programs handling the second convolutional layer of AlexNet. The red dashed lines delimit the computations performed by each process.

The 96 outputs from the P2 process are represented in Figure 6.5 (A) (same yellow color scheme as Figure 6.4). The P3 processes are represented in Figure 6.4 (B), in total there are 8 P3 instances, each responsible for processing 12 channels. The K parameters in P3 represent the indexes of the input channel that the process instance is responsible



Red dashed lines delimit the computations performed by each process. (A) Represents the computation performed by the P1 processes. Each process handles an input channel. (B) Represents the computation performed by the P2 process. It receives the partially convoluted input channels from all the P1 processes. Then, it accumulates them, generating the convolution output feature map, in which the ReLU and Max-Pool operations are applied to generate the final output feature map (in yellow).

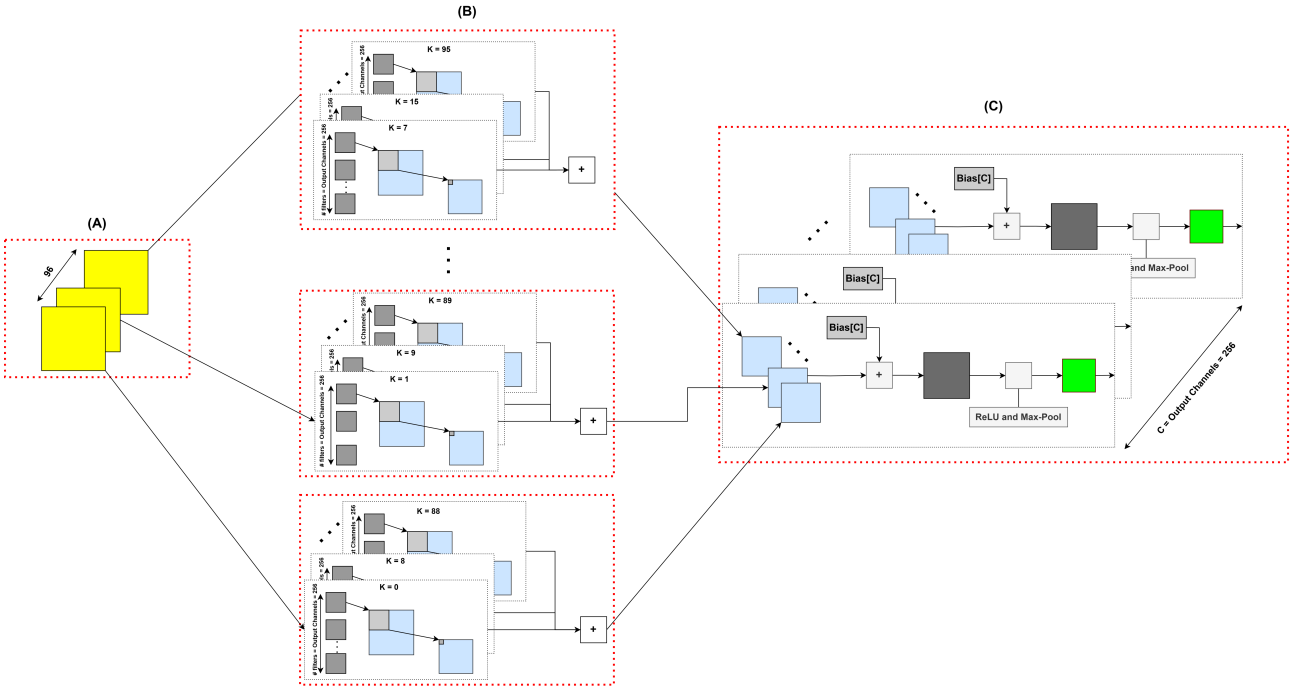
Figure 6.4: Depthwise convolution structure.

for processing. The P2 program sends the (P2) input channels in a round-robin manner, distributing one channel per P3 process until all P3 processes have received an input channel. For example, the first P3 process will receive channels 0, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80 and 88. All the partial outputs generated within a P3 process (12) are accumulated and then sent to the P4 process. There are 256 partial outputs sent to the P4 process (one for each output filter). Process P4 represented in Figure 6.5 (C) starts receiving the partial outputs and then performs their accumulation and the Bias sum. It also performs the ReLU and pooling operation. P4 operates 256 times (one for each output channel), generating 256 output feature maps (in green).

The Memphis-V platform was used with the RS5 processor as the PE. The mapping was performed considering the XY routing. The NoC dimension chosen was a 4x4 NoC. The NoC PEs 0x0 and 3x3 are used as injectors, and the PEs 3x0 and 0x3 are not used.

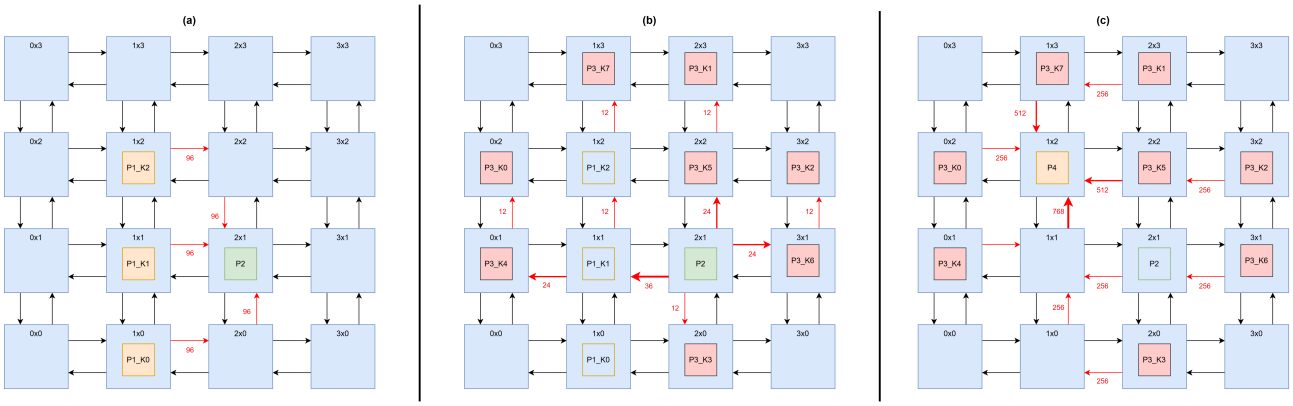
Figure 6.6 presents the mapping of the presented processes to the NoC, considering the amount of PEs chosen for each PE layer (3 for first, 1 for second, 8 for third, and 1 for fourth). The NoC PEs can run multiple tasks, but only the PE at position 1x2 is used in more than one task/process.

Figure 6.6 (a) presents the three P1 instances and the P2 process. The P2 process is centralized to receive data from the maximum ports possible, avoiding packets using the same router. With the P1 and P2 mapping positions, the P2 process receives data from



Red dashed lines delimit the computations performed by each process. (A) Represent the outputs of the P2 process. (B) Represents the computation performed by the P3 processes. Each process instance handles multiple input channels ($\frac{N_{input\ channels}}{P3\ instances}$). (C) Represents the computation performed by the P4 process. It receives the partially convoluted data from the P3 processes. Then, it accumulates them, generating the convolution output feature map, in which the ReLU and Max-Pool operations are applied to generate the final output feature map (in green).

Figure 6.5: Second Layer of the depthwise convolution structure.



Red arrows represent communications, and the number associated represents the number of channels/packets sent. (a) presents the three P1 instances communicating with the P2 process. (b) presents the P2 process sending the channels to be processed by the eight P3 processes. (c) presents all the eight P3 processes sending data to the P4 process.

Figure 6.6: Mapping in a 4x4 NoC.

the South (S), West (W), and North (N) ports, while the East (E) is unused. Figure 6.6 (b) presents the P2 process sending the first convolution layer output channels to the P3 processes (8x P3 instances, which see the received channels as input channels). Each

P3 process receives 12 channels to process. They are mapped to PEs surrounding the centralized PE to divide the channels between all the ports. The routing is not optimal given that the routing algorithm used by the NoC is the XY, which first routes the channel in the X-axis. The channels are sent through P2 ports in the following directions (in order): W, N, E, S, W, N, E, and W. Finally, Figure 6.6 (c) presents the P3 processes sending data to the P4 process. The P4 process is centralized and receives the packets in the following order: W, N, E, S, S, E, S, N. Although some ports are used sequentially, the NoC is not a bottleneck and can handle the data transmission without reducing the application's performance.

6.4 Conclusion and Final Remarks

This Chapter presented the benchmark applications used to validate and evaluate the RS5 VXU unit. The 1-D and 2-D CNNs were ported to the RS5 processor and run in the scalar core to validate its functionality. It also presented the depthwise convolution technique chosen to improve the parallelism in the manycore context further.

Section 6.1 detailed the 1-D CNN application and extensively analyzed the fit of the RVV extension with this applications class. It used assembly codes to explain how the operations can be vectorized and presented a theoretical analysis of the acceleration for each convolution part. This analysis proved that the RVV extension can offer significant acceleration for the application with very few instructions.

Section 6.2 presented the 2-D CNN application with the details on how it was ported and optimized to simplify further operations in manycores. The application layers and the number of multiplications were presented, highlighting the need for specialized hardware to run inference efficiently.

Finally, Section 6.3 demonstrated the method for enhancing parallelism possibilities within a manycore environment. It provided a mathematical foundation for how CNNs can be separated through depthwise convolutions. The arrangement of smaller processes within a manycore environment illustrated how PE layers are organized. Finally, it showed the mapping implemented for the smaller applications, considering communication.

The benchmarks will be further analyzed in the next Chapter, Chapter 7, with (hardware and software) parameter analysis and comparisons of the theoretical and achieved accelerations.

7. RESULTS

This Chapter presents the fifth **contribution** of this Dissertation: an exploration of the CNN benchmarks execution in the different hardware environments, exploring parameters in the single-core environment, and testing the mapping technique in the manycore environment.

This Chapter is organized as follows.

- Section 7.1 presents the results for a single RS5 processor, with the area, performance, and power evaluations. It explores the different instruction subsets and all the benchmark applications presented in Chapter 6. It also evaluates different compilation options for the RVV extension and compares results with the related works.
- Section 7.2 presents the results for a manycore NoC platform with the RS5 processor as the processing element. It explores the parallelization technique presented in Section 6.3 while exploring the impact of having an accelerated PE with the RVV extension.
- Section 7.3 concludes this Chapter.

7.1 Single-core Results

The RS5 core [Nunes et al., 2024], introduced in Chapter 4, enables the selection of optional extensions for use in various software applications. Since its early development, it has been designed for two primary environments: ASIC and FPGA. Section 4.8 presented the three validation environments:

- RTL simulation, which employs a testbench to emulate peripherals and memory, enabling debugging and rapid validation;
- post-synthesis simulation, aimed at ASIC validation under synthesis constraints;
- FPGA prototyping, which evaluates the core in real-world scenarios.

7.1.1 Area Evaluation

Table 7.1 presents the area and timing results of the RS5 processor synthesis in a 28-nanometer library at a frequency of 500 MHz. The synthesis tool used was the Cadence Genus 22.15. The first two lines of the table present the *RV32I* ISA, the most simple processor version. These two lines illustrate the effect of adding a branch prediction mechanism to the processor, which has a minimal area increase of only 250 μm^2 and contributes to enhanced performance. Therefore, the mechanism is present in all other syntheses. The

Table 7.1: ASIC area results at 500MHz for RS5 core (TSMC 28 nm).

Instruction Set	**Total # Gates**	**Cell Area (μm^2)**	**Total Area (μm^2)**	**Slack (ps) - Slowest**
RV32I (no branch prediction)	7,041	8,074	10,902	9
RV32I	7,237	8,343	11,151	5
RV32I_ZIHPM	9,374	11,066	14,409	1
RV32I_ZKNE	7,535	8,612	11,638	9
RV32I_XOSVM	8,080	9,124	12,315	2
RV32IC	7,825	8,800	11,938	1
RV32I_ZMMUL	8,280	9,352	12,546	3
RV32IM	9,105	10,298	13,938	4
RV32IMC *-baseline implementation*	9,546	10,674	14,196	16
RV32IAC	8,778	9,660	13,207	6
RV32IMAC	10,491	11,558	15,443	6
RV32IMAC_ZIHPM_XOSVM	13,952	15,641	20,628	1
RV32IMAC_ZIHPM_XOSVM_ZKNE	14,323	15,900	21,164	0

effect of including each extension individually is assessed by the subsequent lines. All the combinations achieved timing closure for the target frequency.

The *RV32IMC* and *RV32IMAC* ISAs are the most common combinations in embedded RISC-V processors. Hardware accelerators for multiplication and division (M extension) are crucial for performance. Nunes et al. [2024] demonstrated that the *RV32IM* ISA achieved a CoreMark score of 212.3 points [Consortium, 2024], whereas the *RV32I* ISA reached only 86.3. Including compressed instructions (C extension) reduces code size, while atomic instructions (A extension) enable read-modify-write operations in memory, which is essential for implementing OS semaphores.

Since the benchmark applications presented in Chapter 6 do not use an OS, the A extension is not required. Thus, due to its flexibility and performance, the chosen baseline was the *RV32IMC* instruction set. In the 28 nm ASIC synthesis, this instruction set occupied approximately 14K μm^2. Figure 7.1 shows the resource distribution for the baseline core, where the register bank and execution unit are the largest modules, each occupying around 30% of the area, followed by the CSR bank at approximately 20%.

Table 7.2 presents the FPGA synthesis results for a Digilent NEXYS A7 board with a Xilinx xc7a100tcsg324-1 FPGA at 100MHz. It shows the results in terms of look-up tables (LUTs), Flip-Flops (FFs), and Digital Signal Processors (DSPs). The timing result is presented in picoseconds. The Xilinx Vivado 2023.2 default synthesis strategies were used. The most complete version of the RS5 processor (*RV32IMAC_ZIHPM_XOSVM*) with and without the *ZKNE* extension (AES cryptography acceleration) failed to meet timing closure using the default and performance synthesis strategies. Thus, its operating frequency was reduced to 50 MHz. The chosen baseline (*RV32IMC*) uses ≈3300 LUTs, ≈1300 FFs, and 1 DSP block (used in multiplication hardware)

.

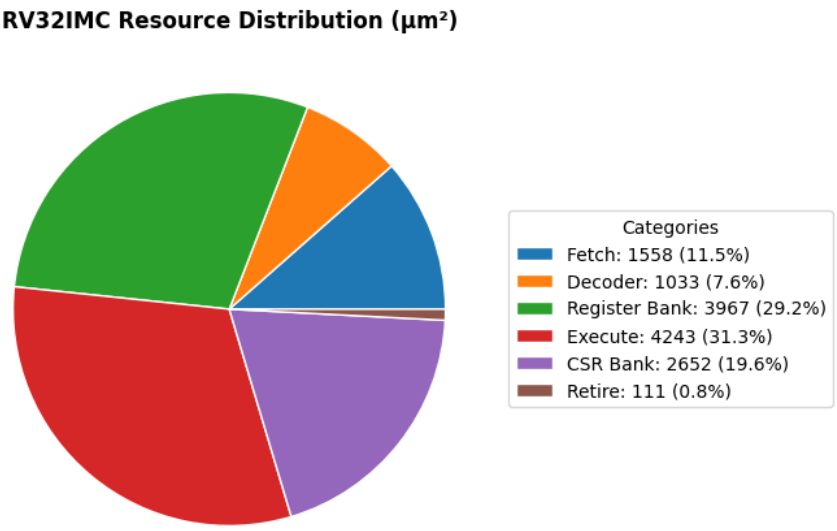


Figure 7.1: Resource distribution for the RS5 with the RV32IMC instruction set.

Table 7.2: RS5 core FPGA area results at 100MHz with default synthesis strategies.

Instruction Set (* = No branch prediction, ** = Running on 50MHz)	**LUTs**	**FFs**	**DSPs**	**Slack (ps)**
RV32I*	2,301	897	0	392
RV32I	2,374	948	0	409
RV32I_ZIHPM	2,664	1,497	0	410
RV32I_ZKNE	2,745	951	0	41
RV32I_XOSVM	2,428	1,171	0	297
RV32IC	2,522	1,014	0	380
RV32I_ZMMUL	2,726	1,043	1	201
RV32IM	3,047	1,226	1	169
RV32IMC	3,296	1,292	1	325
RV32IAC	2,893	1,199	0	120
RV32IMAC	3,482	1,475	1	173
RV32IMAC_ZIHPM_XOSVM**	3,921	2,375	1	4,862
RV32IMAC_ZIHPM_XOSVM_ZKNE**	4,164	2,381	1	3.892

The benchmarks compiled with GCC 14.2 and auto-vectorization offered insight into the necessary vector instructions for running the applications. The vector instructions used by the benchmark applications are:

- Configuration: used to set the CSRs of the RS5 VXU.
- Load: Unit-Strided addressing mode is used for loading vectors from memory. The strided addressing mode is often used with a stride zero to load the same value to an entire register group.
- Store: Unit-Strided addressing mode stores vectors in memory.
- Multiply and accumulate: Used in dot product operations of convolution operations.
- Sum reduction: used in the dot product operations of convolution layers.
- Addition: used for adding the bias to the dot product operation results in convolutional layers.

- Mask compares: used in ReLU operation to define which values should be replaced by zero.
- Max: used in Max-Pool operations in the pooling layers.
- Max Reduction: used in Max-Pool operations in the pooling layers.

The RS5 VXU has non-optional instructions that the target applications did not use. An example is the Load/Store instructions with indexed addressing modes and widening multiplications. Although not used, these instructions can not be disabled in the synthesis steps. Therefore, the instruction subset that comprises the mandatory instructions plus the optional instructions used by the applications is called the "*Minimal*" subset. The subset called "*Complete*" comprises all instructions implemented. Another subset called "*No_Div*" explores the exclusion of vector division operations to analyze its impact in the area.

Table 7.3 presents the area results for synthesizing RS5 with the vector extension across three subsets (Minimal, No_Div, and Complete). The Table includes results for VLENs of 64, 128, and 256 bits. Larger VLENs do not justify the area-performance trade-off, as the area increases excessively. All evaluated VLENs and subsets achieved timing closure.

Table 7.3: ASIC area results at 500MHz for RS5 core with vector extension.

Instruction Set	VLEN	Subset	Total # Gates	Cell Area (μm^2)	Total Area (μm^2)	Slack (ps) - Slowest
RV32IMC	-	-	9,546	10,674	14,196	16
RV32IMC_ZVE32X	64	Minimal	31,573	30,517	41,080	0
	64	No_Div	31,800	31,244	42,575	0
	64	Complete	37,604	36,910	49,546	0
RV32IMC_ZVE32X	128	Minimal	50,448	48,168	65,036	0
	128	No_Div	51,792	49,014	66,177	0
	128	Complete	63,639	60,464	80,528	0
RV32IMC_ZVE32X	256	Minimal	89,515	83,353	114,117	0
	256	No_Div	93,118	85,990	117,078	0
	256	Complete	115,751	107,734	145,832	0

A 64-bit VLEN offers an area overhead of 2.89x (minimal subset) to 3.49x (complete subset) compared to the RV32IMC baseline. This means that the VXU inclusion, on average, triples the processor area. Including other optional instructions, except the division, represents an area footprint increase of just 3.64%. However, the division operations offer a significant area footprint overhead of 16.37%. A VLEN of 128 bits expands the area footprint by 4.58x for the minimal subset and 5.67x for the complete subset. Comparing the 128-bit VLEN area with the 64-bit VLEN shows an increase ranging from 1.58x to 1.62x for their respective minimal and complete subsets. With a VLEN of 256 bits, the area expansion for minimal and complete subsets is 8.04x to 10.27x compared to the scalar core, and 1.75x and 1.81x compared to the 128-bit version.

Figure 7.2 presents the resource distribution for the RS5 core with the VXU unit using the minimal subset. Figure 7.2 (a) and (b) show the VXU with a VLEN of 64 bits, and (c) and (d) represent a 256-bit VLEN. The (a) and (c) subplots show the RS5 resource

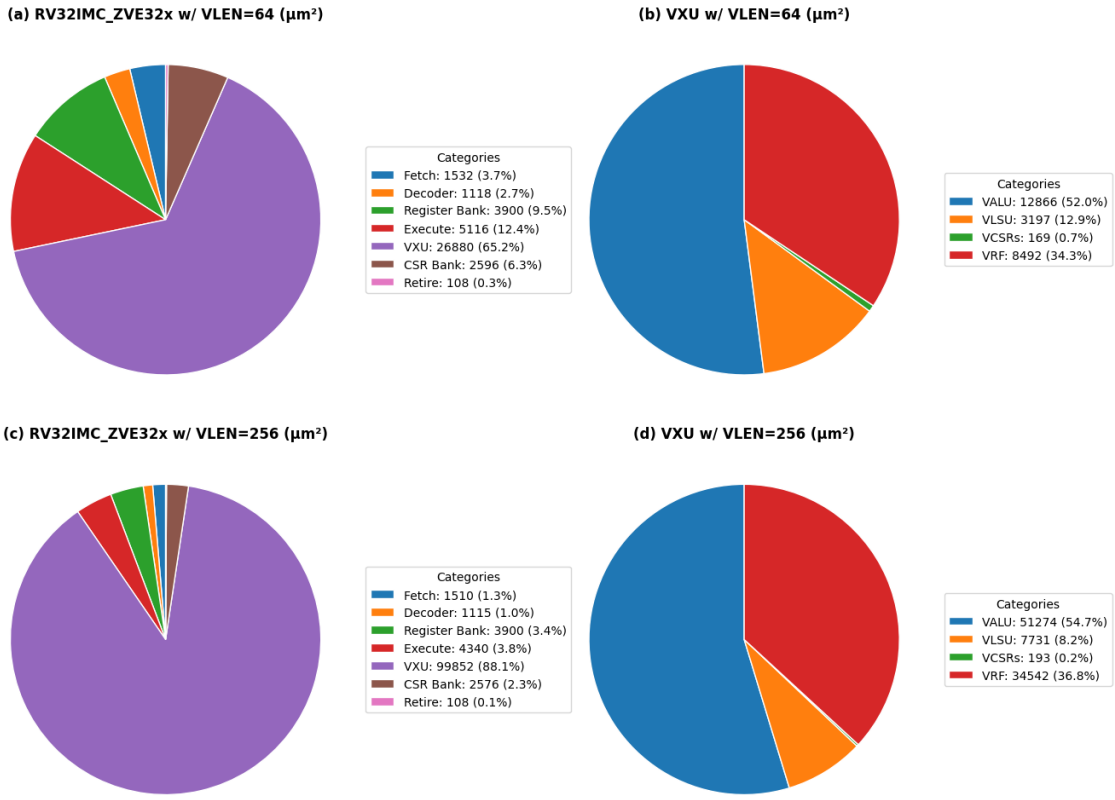


Figure 7.2: Resource distribution for the RS5 with the RV32IMC_ZVE32x instruction set with a VLENs of 64 and 256 bits.

distribution. Even though the VXU is part of the execute module, it is depicted as a separate module to illustrate how it contributes to the RS5 processor area. With this, it is possible to observe that the execute module maintains a similar area to the scalar core execute unit (4243 μm^2 – Figure 7.1), highlighting that including the VXU does not significantly impact other components. However, the VXU takes 65% (VLEN=64) and 88% (VLEN=256) of the processor area.

Figure 7.2 (b) and (d) show the VXU's internal resource distribution. The VALU is the component that uses more resources, followed by the VRF. The 64-bit VALU occupies 12,866 μm^2 (52% of VXU area) for a dual-lane (2x32) design, and the 256 bits VALU occupies 51,274 μm^2 (54.7% of the VXU area) for an octa-lane (8x32) design. The multipliers represent around a third of the VALU area, highlighting the importance of resource sharing for multiplying operations. The growth ratio from the dual-lane VALU to the octa-lane VALU is 3.98x, showing that the VALU area grows linearly with the number of lanes. The VRF, which is directly impacted by the size of the registers, has a similar growth ratio of 4.06x and occupies from 34% to 36.8% of the VXU area. The VLSU occupies 3,197 μm^2 (12.9% of the VXU area) for a 64-bit VLEN and 7,731 μm^2 for a 256-bit VLEN. The growth ratio is lower, at just 2.42x, because this unit operates in a single lane. Vector CSR has a minimal effect, accounting for less than 1% of resource usage.

Table 7.4: RS5 core with vector extension FPGA area results.

Instruction Set (* = Using multicycle reductions)	VLEN	Subset	Frequency (MHz)	Strategy	LUTs	FFs	DSPs	Slack (ps)
RV32I	-	-	100	Default	2,374	948	0	409
RV32IMC	-	-	100	Default	3,296	1,292	1	325
RV32IMC	-	-	50	Default	3,202	1,292	1	7,881
RV32IMC_ZVE32XB	64	Minimal	50	Default	10,800	2,344	5	109
		No_Div	50	Default	11,266	2,347	5	642
		Complete	50	Default	12,673	3,429	5	245
RV32IMC_ZVE32XB	128	Minimal	50	Performance	17,156	3,095	9	8
		No_Div	50	Performance	17,646	3,105	9	5
		Complete	50	Performance	23,328	5,345	9	20
RV32IMC_ZVE32XB*	128	Minimal	50	Default	15,467	5,652	9	847
		No_Div	50	Default	16,729	6,810	9	1,563
		Complete	50	Default	21,984	9,114	9	418
RV32IMC_ZVE32XB*	256	Minimal	50	Default	27,500	10,142	17	494
		No_Div	50	Default	29,830	12,565	17	727
		Complete	50	Default	39,344	16,871	17	408
RV32IMC_ZVE32XB*	256	Minimal	50	Performance	29,602	11,953	17	968
		No_Div	50	Performance	31,777	16,407	17	716
		Complete	50	Performance	43,636	20,843	17	2,139

Table 7.4 presents the FPGA synthesis results for the RS5 processor with the VXU unit. As Table 7.2 showed, the most complete versions of the RS5 processor failed timing closure even with performance strategy. The same applied to the vector extension, which implied that the frequency had to be reduced to 50MHz. The baseline core was synthesized using the same frequency for fair comparisons, using 3200/1300 LUTs/FFs. The RS5 with the vector extension and a VLEN of 64 bits achieved timing closure using the default strategy for the three subsets. They presented growth in LUTs/FFs usage of 3.37x/1.81x (minimal subset) to 3.96x/2.64x (complete subset), representing similar growth ratios observed in ASIC syntheses. Using a VLEN of 128 bits with default synthesis strategies did not achieve timing closure. Thus, the performance synthesis strategy was used. The growth in the 128-bit version with performance synthesis is 5.36x/2.40x in LUTs/FFs for the minimal subset. The VXU with a VLEN of 256 bits did not achieve positive slacks even using the performance strategy.

The sections of Table 7.4 with a "*" use multicycle reduction trees, improving timing in exchange for performance. This can be seen in the 128-bit VLEN, which failed timing closure using the default synthesis strategy but had positive slacks when this option was active. When comparing the two variants of 128-bit VLEN with the minimal subset, the version employing multicycle reductions has approximately 2,000 fewer LUTs. However, this optimization increases the number of FFs by about 2,500, nearly doubling the usage.

The impact of using the performance synthesis strategy can be seen in the versions with a 64-bit VLEN. The performance synthesis strategy can increase the number of LUTs and FFs used, ranging from 2,000 to 4,000. The 256-bit VLEN with the default strategy indicates growth factors in LUTs/FFs ranging from 8.59x/7.85x (minimal subset) to 12.29x/13.06x (complete subset). This growth is similar to that presented in ASIC synthesis, which achieved 8.04x to 10.27x in the respective subsets.

7.1.2 Performance Evaluation

1-D CNN Performance Evaluation

Table 7.5 displays the performance results, measured in clock cycles, for executing the first convolutional layer of the 1-D CNN (with ReLU) presented in Section 6.1. It first presents the results for a scalar core using the base ISA and the versions with multiplication. It emphasizes once more that division operations are seldom necessary because compilers typically avoid them due to their high cost. The chosen baseline ISA (RV32IMC) is highlighted and took 636,531 clock cycles to execute. The following table rows explore the performance results for the vector extension with VLENs ranging from 64 to 1024 bits.

Table 7.5: Performance results running the simplified first layer of the 1-D CNN application in RS5.

Instruction Set	Scalable	VLEN	Clock Cycles	Speed-up	Cycle Reduction (%)
RV32I	-	-	7,220,542	-	-
RV32I_ZMMUL	-	-	595,735	-	-
RV32IM	-	-	595,735	-	-
RV32IMC	-	-	**636,531**	-	-
RV32IMC_ZVE32X_ZVL64B	X	64	508,199	1.25	20.16
	✓	64	608,163	1.05	4.46
	✓	128	394,479	1.61	38.03
	✓	256	288,597	2.21	54.66
RV32IMC_ZVE32X_ZVL128B	X	128	300,726	2.12	52.76
	✓	128	347,705	1.83	45.38
	✓	256	232,415	2.74	63.49
RV32IMC_ZVE32X_ZVL256B	X	256	228,127	2.79	64.16
	✓	256	232,415	2.74	63.49
RV32IMC_ZVE32X_Manual	**X**	**256**	**82,895**	**7.68**	**86.98**
RV32IMC_ZVE32X_ZVL512B	X	512	170,893	3.72	73.15
	✓	512	168,910	3.77	73.46
RV32IMC_ZVE32X_ZVL1024B	X	1024	138,334	4.60	78.27
	✓	1024	138,454	4.60	78.25

Table 7.5 results explore the compiler option *-mrvv-vector-bits=zvl*. Usually, the compiler use the VLEN specified by the *ZVL* instruction set to generate scalable code. A scalable code can be run in VLENs bigger or equal to the specified in *ZVL*, thus assuring portability among cores. The inclusion of the *-mrvv-vector-bits=zvl* flag indicates to the compiler that it should generate code targeting only the VLEN specified in the *ZVL* instruction set. This is displayed in the table using the "scalable" column, which, when has an "X", indicates that the optional flag was used.

The RV32IMC_ZVE32X_ZVL64B instruction set presents four results: 1) Non-Scalable code running on a 64-bit VLEN; 2-4) scalable code running on a 64, 128 and 256-bit VLEN. Comparing the 64-bit VLEN versions, the non-scalable version shows a sig-

nificant advantage by reducing the application runtime cycles by over 20% (1.25x speed-up). In contrast, the scalable version manages a reduction of just 4.46% (1.05x speed-up). For a 128-bit VLEN, the non-scalable version has an advantage of ≈7% in cycles reduction. For larger VLENs, the difference between scalable and non-scalable reduces to less than 1%, with the non-scalable having advantages in most cases.

Running a code compiled for a scalable ZVL64B on a 128-bit VXU unit achieved a speed-up of 1.61x while running a code compiled with ZVL128B on the same hardware yielded a speed-up of 1.83x. Based on this information, we can conclude that compiling the code with the ZVL instruction set with the same VLEN as the hardware that will run the code is the better option. In addition, using non-scalable code is the better option as it yields better speed-ups/cycle reductions. *Thus, only the non-scalable version will be considered for other benchmarks.* The maximum speed-up achieved was 1.25x, 2.12x, 2.79x, 3.77x, and 4.60x for VLENs of 64, 128, 256, 512, and 1024 bits, respectively.

The Table also shows a version of the code that was manually vectorized in assembly, targeting a VLEN of 256 bits, as Section 6.1 presented. The manual version achieved a speed-up of 7.68x, beating all the speed-ups from codes generated by the compiler, even those with larger VLENs. This represents a reduction of almost 87% in the cycles required to run the application, meaning that the application was executed in just 13% of the time that a scalar version would take. This result closely resembles the theoretical expectations discussed in Section 6.1.4, indicating an approximate speed-up of 5.71x. The improved result (7.68x) arises from the more straightforward memory access in the assembly version.

Table 7.6 presents the results of running the **complete** 1-D CNN on the RS5 core. This application was presented in Section 6.1 and, for a single inference, takes 5,428,096 MAC operations. Running on the base ISA would take around 955 million clock cycles to execute, while in the baseline ISA it takes around 78 million cycles. Table 7.6 brings the column *Max-LMUL* which refers to another compilation option/flag for the RVV extension explored in this table. The option/flag is the *-mrvv-max-lmul* that gives a hint to the compiler of what size of vector register groups it should use, varying from 1 (*m1*) to 8 (*m8*) registers.

The complete 1-D CNN application achieved a maximum speed-up of 1.98x for the vector extension with a VLEN of 1024 bits. A 256-bit VLEN offered a speed-up of 1.91x, very close to the 1.98x of the 1024-bit version, thus becoming the best trade-off of performance and area for this application. It reduced the required clock cycles by 47.6%, representing more than 37 million cycles. Smaller VLENs also showed good performance. The 64-bit version achieves a maximum speed-up of 1.43x, which reduces the cycles needed by approximately 30% (about 23 million cycles). The 128-bit version achieved a maximum speed-up of 1.66x, reducing the clock cycles by almost 40% (around 31 million cycles).

A 64-bit VLEN VXU yielded a speed-up of only 1.04x without adding the hint flag. Including a register group hint for bigger register groups allowed the application to achieve more significant speed-ups. The hint of using registers of 1 group limited the compiler and

Table 7.6: Performance results running the 1-D CNN application in RS5.

Instruction Set	Max-LMUL	VLEN	Clock Cycles	Speed-up	Cycle Reduction (%)
RV32I	-	-	955,198,109	-	-
RV32IMC	-	-	**78,083,969**	-	-
RV32IMC_ZVE32X_ZVL64B	-	64	74,787,999	1.04	4.22
	1	64	77,975,577	1.00	0.14
	2	64	61,326,366	1.27	21.46
	4	64	54,475,600	1.43	30.23
	8	64	71,505,872	1.09	8.42
RV32IMC_ZVE32X_ZVL128B	-	128	56,267,471	1.39	27.94
	1	128	56,509,994	1.38	27.63
	2	128	46,955,281	1.66	39.87
	4	128	51,790,385	1.51	33.67
	8	128	69,925,007	1.12	10.45
RV32IMC_ZVE32X_ZVL256B	-	256	40,931,358	1.91	47.58
	1	256	40,935,557	1.91	47.57
	2	256	43,071,013	1.81	44.84
	4	256	48,571,769	1.61	37.80
	8	256	67,311,347	1.16	13.80
RV32IMC_ZVE32X_ZVL512B	-	512	40,000,534	1.95	48.77
	1	512	40,058,847	1.95	48.70
	2	512	42,858,139	1.82	45.11
	4	512	48,069,445	1.62	38.44
	8	512	66,505,523	1.17	14.83
RV32IMC_ZVE32X_ZVL1024B	-	1024	39,527,786	1.98	49.38
	1	1024	39,987,473	1.95	48.79
	2	1024	42,539,441	1.84	45.52
	4	1024	47,403,823	1.65	39.29
	8	1024	66,392,487	1.18	14.97

made the performance even worse. Groups of 2 and 4 registers improved performance in 1.27x and 1.43x, respectively. Using groups of 8 registers did not improve performance compared to groups of 4, but it still improved compared with the version without the hint. The same behavior was observed for a 128-bit VLEN, but the maximum speed-up was reached with groups of two registers, demonstrating that the improvement is related to the size of the processed vector.

Dot-product Performance Evaluation

A **dot-product benchmark** was ported from ARA [Perotti et al., 2022] to explore the relationship between vector size and acceleration. The application runs the same operation (dot-product) in different vector lengths/sizes. It also explores different data sizes, from 8 to 32 bits. The application was modified to allow the cycles per operation calculation.

Figure 7.3 presents the results for the dot-product benchmark running on RS5 processor, exploring the LMUL for vector operations. Figure 7.3 (a) shows the results for a VLEN of 64 bits and (b) for a VLEN of 256 bits. For smaller vector sizes, all versions take similar cycles to execute, even compared with the scalar version.

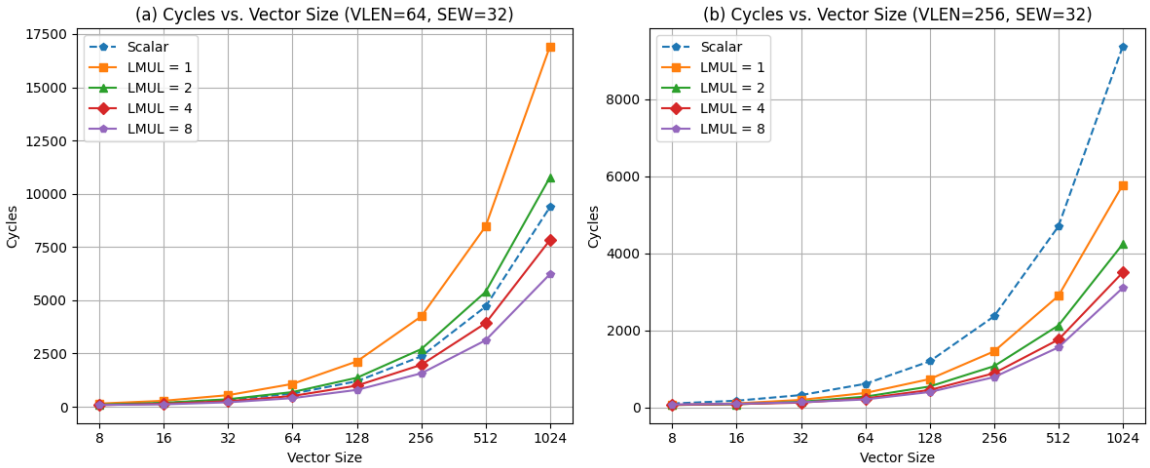


Figure 7.3: Performance behavior for different LMULs in the dot-product benchmark. In (a) VLEN=64 and in (b) VLEN=256.

For a VLEN of 64 bits (Figure 7.3 (a)), the scalar version (dashed line) outperforms versions using register groups (LMUL) of 1 and 2 registers due to the additional cycles required for configuring and controlling the VXU accelerator, which occur more frequently with smaller register groups. However, larger register groups perform better than the scalar version for a 64-bit VLEN.

For a VLEN of 256 bits (Figure 7.3 (b)), all vectorized versions outperform the scalar version, as larger vector registers accommodate more elements, enabling greater parallelism. However, increasing the VLEN does not necessarily yield significant gains if the vector size remains small. For large vectors, groups of eight registers proved to be the most effective option.

Figure 7.4 shows the results for the dot-product benchmark running on RS5 processor, exploring the VLEN. Figure 7.4 (a) shows the results for operations using 32-bit data (SEW=32), and (b) shows the result for operations using 8-bit data (SEW=8). In (a), it can be observed that the VLEN size only starts to make a significant difference from around 64 to 128 elements in the vector. It can also be observed that doubling the VLEN does not double the performance. For a VL of 1024 elements, a 256-bit VLEN requires about 3,000 cycles, whereas the 128-bit version takes around 4,000 cycles, and the scalar version takes about 9,000 cycles. This results in a speed-up of about 3x and 2.25x, respectively.

Figure 7.4 (b) shows the impact of using smaller data sizes (SEW) in the vector unit. In RS5 VXU, 8-bit data have the maximum number of elements processed in parallel, 4 times more than in a 32-bit SEW. In the (b) chart with SEW=8, it is evident that for the maximum vector size, all VLENs show a speed-up exceeding 6x compared to the scalar core, reducing the cycles from around 12,000 to below 2,000. For smaller vector sizes, the differences are negligible, only starting to be visible after 512 elements. This happens because the vector register can hold and process more data in parallel.

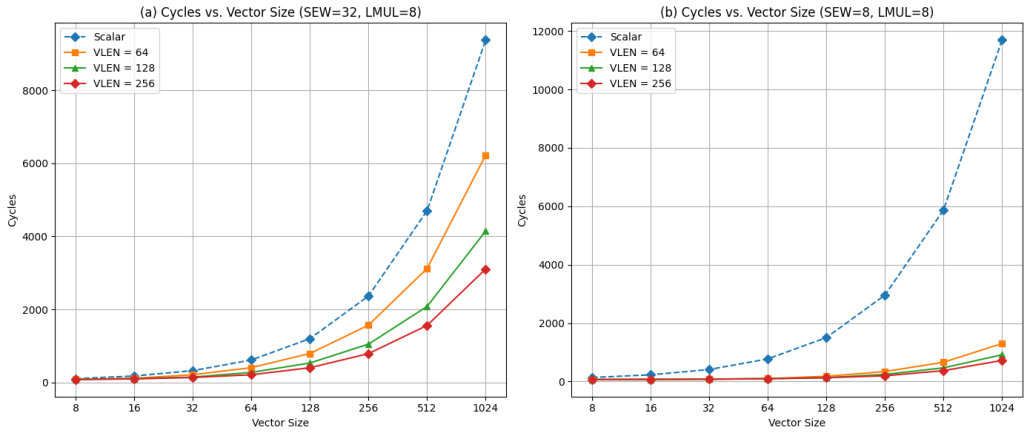


Figure 7.4: Performance behavior for different VLENs in the dot-product benchmark. In (a) SEW=32 and in (b) SEW=8.

Figure 7.5 shows the results for the dot-product exploring the SEW variation. Figure 7.5 (a) and (b) have a 64-bit VLEN, and (c) and (d) have a 256-bit VLEN, with the LMUL being 1 in (a) and (c) and 8 in (b) and (d). Scalar results are shown as dashed lines.

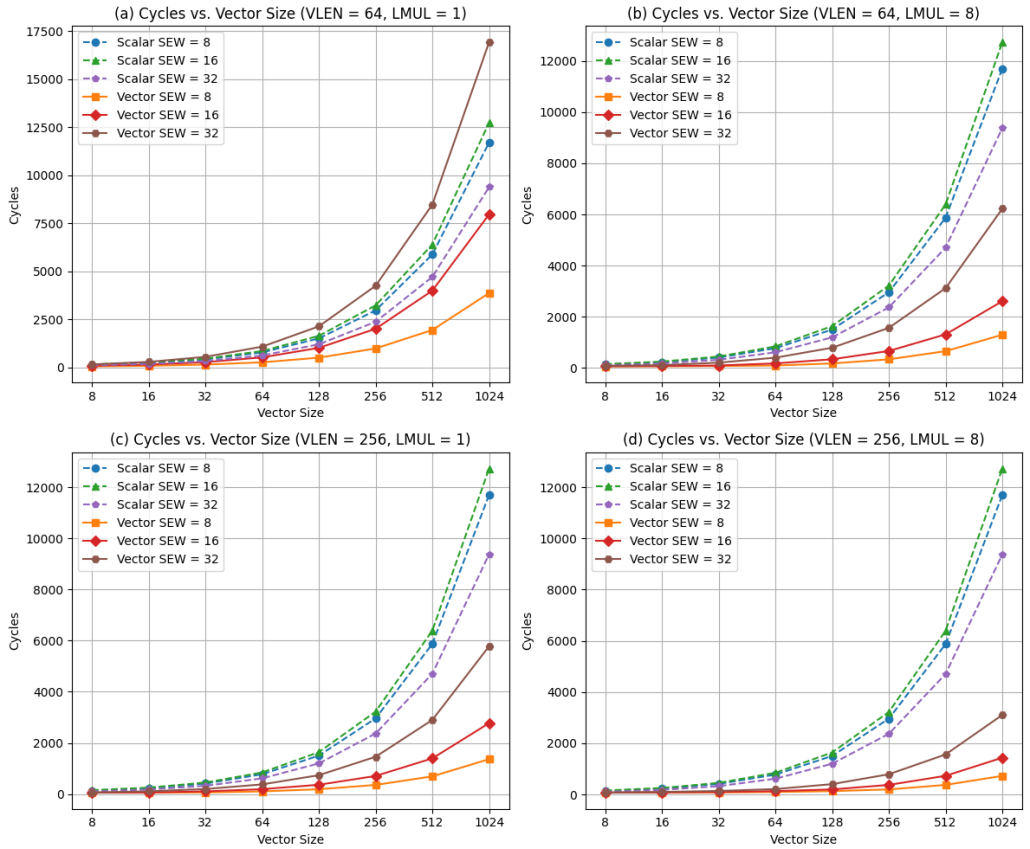


Figure 7.5: Performance behavior for different SEWs in the dot-product benchmark. Horizontal charts share the same VLEN, while vertical charts maintain the same LMUL.

Figure 7.5 (a) shows that the vectorized version with small VLEN (64) and LMUL (1) can yield worse results compared to even the scalar. For an LMUL of 8 registers, the 64-bit VLEN yields more regular results, with the worst being the scalar versions and the

vectorized with better results. Charts (b), (c), and (d) show that smaller SEWs provide better results than bigger SEWs. The main difference observed when the VLEN increases ((c) and (d)) is that the cycle vs. vector size growth ratio is smaller.

Figure 7.6 shows the logarithmic results for the dot-product benchmark, exploring the SEW, LMUL, and VLEN variations. Individual charts show LMUL variation, and the horizontal charts explore the different SEWs. Charts from (a) to (c) show results for a 64-bit VLEN and (d) to (f) for a 256-bit VLEN.

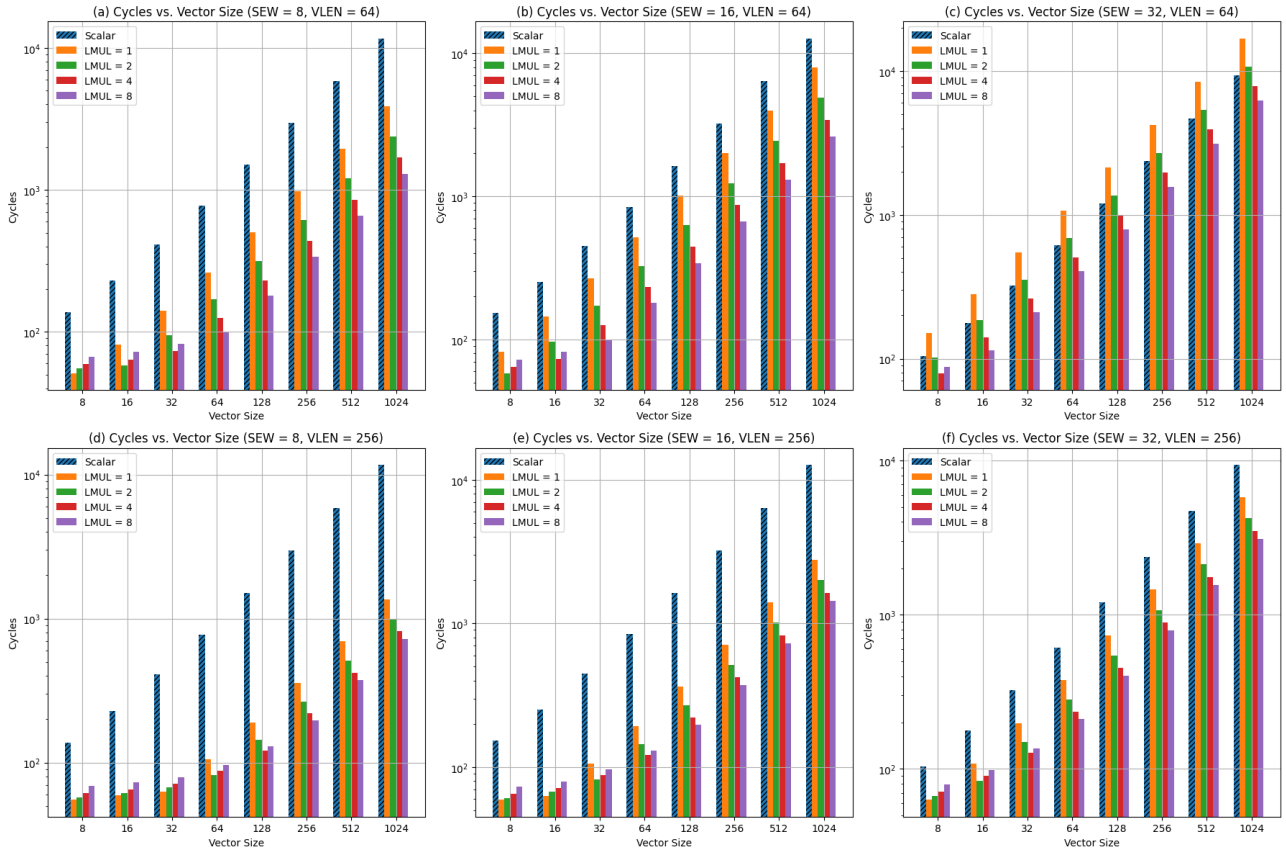


Figure 7.6: Logarithmic performance behavior for different LMULs in the dot-product benchmark. Horizontal charts share the same VLEN, while vertical charts maintain the same SEW.

In Figure 7.6, it can be observed that as the SEW grows (horizontal charts), the difference between the scalar and the vectorized versions diminishes. It can also be observed that as the VL grows, the register groups with eight registers become superior to others. By applying a logarithmic scale, the smaller VLs can be observed better. The best speed-up achieved for the dot-product application was using with a 256-bit VLEN, with LMUL=8 and SEW=8, achieving 16.15x of speed-up.

AlexNet Performance Evaluation

Table 7.7 presents the results for running the first two convolutional layers of the AlexNet [Krizhevsky et al., 2017]. The scalar core with the base ISA takes almost 85 billion clock cycles, while the baseline takes around 8.5 billion cycles. The configuration that

yielded better speed-up was those with 512 and 1024 VLENs, both achieving 1.37x. The VLEN of 256 bits (chosen as the best trade-off in CNN 1-D) achieved a maximum speed-up of 1.33x, a reduction of almost 25%, representing more than 2 billion clock cycles. A 128-bit VLEN presented a top reduction of 15.6% (1.18x speed-up). The 64-bit VLEN did not provide good acceleration, with most cases being worse than the scalar and having a maximum reduction of 3.6% (speed-up of 1.04x).

Table 7.7: Performance results running the first two convolutional layers of AlexNet in RS5.

Instruction Set	VLEN	Max-LMUL	Clock Cycles	Speed-up	Cycle Reduction (%)
RV32I	-	-	84,792,153,908		
RV32IMC	-	-	**8,513,618,557**		
RV32IMC_ZVE32X_ZVL64B	64	-	8,513,618,583	1.00	0.00
	64	1	8,513,892,321	1.00	0.00
	64	2	9,148,991,562	0.93	-7.46
	64	4	8,206,769,205	1.04	3.6
	64	8	9,576,737,921	0.89	-12.49
RV32IMC_ZVE32X_ZVL128B	128	-	8,259,518,056	1.03	2.98
	128	1	8,259,518,056	1.03	2.98
	128	2	7,187,565,401	1.18	15.58
	128	4	7,810,109,889	1.09	8.26
	128	8	9,401,203,341	0.91	-10.43
RV32IMC_ZVE32X_ZVL256B	256	-	6,394,263,639	1.33	24.89
	256	1	6,394,263,639	1.33	24.89
	256	2	6,736,728,017	1.26	20.87
	256	4	7,532,481,241	1.13	11.52
	256	8	9,123,126,403	0.93	-7.16
RV32IMC_ZVE32X_ZVL512B	512	-	6,211,825,071	1.37	27.04
RV32IMC_ZVE32X_ZVL1024B	1024	-	6,211,793,917	1.37	27.04

With the first and second layers of AlexNet containing 105,415,200 and 447,897,600 multiplication operations, accumulating over 500 million multiplications the vector acceleration outcomes fell short of expectations. By observing the assembly code generated by the compiler, it was possible to see that it could not auto-vectorize every part of the convolution operations. For example, the function that sums the bias to the output feature did not use any vector instruction, although the VADD instruction could have been used. This might be due to how the C-language code is written, which uses multiple C memory pointers for the array/matrix structures.

7.1.3 Power and Energy Evaluation

The synthesized circuits were used in simulation with annotated delays to evaluate the power. These simulations used the synthesized netlist allied with the delays of each gate to validate the circuit functionality after the synthesis. This simulation allows the export of a switching activity file. The switching activity file can then be loaded into the synthesis tool again to calculate the power more precisely for that application.

The Cadence Xcelium 23.03 simulator was used to perform the annotated simulations. The testbench includes the circuit netlist, the “.sdf” file with the delays to be annotated,

and RAM with the application loaded. The testbench is also responsible for dumping the switching activity file. A commonly used flow uses value change dumping (VCD), but this approach is slower than the Cadence SHM flow. The SHM files dumped by the testbench are loaded into the synthesis tool (Cadence Denus) alongside the circuit netlist and the libraries. The synthesis tool then performs a more precise power analysis based on actual switching activities.

As the applications vary in characteristics, they can have different switching activities. For this analysis, two applications were selected: the dot-product of vectors with 4096 elements (with 8, 16, and 32 bits) and the simplified first layer of the 1-D CNN.

Table 7.8 presents the results for the target applications. Smaller circuits, such as the scalar core and the variations with smaller VLENs, consume less power. The scalar core consumes around 4.4 to 5 mW, while the accelerated versions can consume up to 16.74 mW.

Table 7.8: Power analisys of diferent applications on RS5 w/vector unit.

Application (* = Manually-Vectorized)	**VLEN**	**Power (*mW*)**	**Time (*ns*)**	**Energy (μJ)**	**Energy comparison (%)**
dot-product 4096x8b	-	4.40	112,610	0.50	-
dot-product 4096x8b	64	9.45	28,750	0.27	-46.00
dot-product 4096x8b	128	10.34	25,486	0.26	-48.00
dot-product 4096x8b	256	16.74	23,854	0.40	-20.00
dot-product 4096x16b	-	4.83	120,926	0.58	-
dot-product 4096x16b	64	9.50	39,990	0.38	-34.48
dot-product 4096x16b	128	10.11	33,084	0.33	-43.10
dot-product 4096x16b	256	16.36	29,756	0.49	-15.52
dot-product 4096x32b	-	5.00	93,234	0.47	-
dot-product 4096x32b	64	7.72	69,682	0.54	+14.89
dot-product 4096x32b	128	9.44	52,274	0.49	+4.25
dot-product 4096x32b	256	15.45	43,570	0.67	+42.55
conv_simplified	-	4.92	1,273,062	6.26	-
conv_simplified	64	7.47	1,016,398	7.59	+21.25
conv_simplified	128	9.70	601,452	5.83	-6.87
conv_simplified	256	16.06	456,254	7.33	+17.09
conv_simplified*	256	14.73	165,790	2.44	-61.02

As seen previously, the versions with the RVV extension can significantly reduce the time it takes to execute the applications. The third table column presents the time taken, in nanoseconds, to execute each application for the given application and core version. The power and time information can be used to calculate the energy required to run the application (energy = power x time).

The energy required to run the dot-product application in its 8-bit version on a scalar core is 0.50μJ. In the versions that use the RVV extension, the energy used was reduced by up to 48% (0.2 μJ), proving that even with a significant area overhead, the speed-up of the applications can be beneficial. A similar behavior was observed in the 16-bit version of the

dot-product application. For the 32-bit dot-product, no accelerated version obtained gains in energy consumption. This occurs because 32-bit data is the worst-case in parallelism in the RS5 VXU, obtaining the smaller speed-ups.

The simplified convolution (the first layer of a 1-D CNN) consumed energy similar to the scalar version, with the 64- and 256-bit VLEN versions increasing and the 128-bit version decreasing by around 7 percent. This occurs again because the application uses only 32-bit data, limiting the speed-up that can be achieved. The manually vectorized version, which has a more significant speed-up, obtained an energy reduction of 61%, showcasing how effective the RVV extension can be.

7.1.4 Related work comparison

This Section presented the RS5 area and performance evaluation, including the RISC-V vector extension. Table 7.9 summarizes the area of RS5 with VXU for different VLENs and their relative speed-up for the benchmark applications.

Table 7.9: RS5 W/ Vector extension summary of area and speed-up results.

ISA/subset	VLEN	Lanes	ASIC Area (μm^2)	LUTs/FFs (* = Using multicycle reductions)	Simplified convolution (** = manually vectorized)	Dot-Product	1-D Conv	2-D Conv
IMC	-	-	14,196	3,296/1,292	-	-	-	-
IMC_ZVE32X/Minimal	64	2	41,080	10,800/2,344	1.25x	9.00x	1.43x	1.04x
IMC_ZVE32X/Minimal	128	4	65,036	17,156/3,095	2.12x	12.77x	1.66x	1.18x
IMC_ZVE32X/Minimal	256	8	114,117	27,500/10,142*	2.79x/7.68x**	16.15x	1.91x	1.33x

Chapter 3 discussed related works that employ various techniques to accelerate CNNs, with RVV-based approaches differing in multiple aspects. Table 3.1 summarized these works regarding RVV accelerators, with the final row positioning our work relative to the state-of-the-art.

Wang et al. [Wang et al., 2024] synthesized Ara [Perotti et al., 2022] and SPEED using Synopsys Design Compiler 2014.09 on the TSMC 28 nm process with four lanes and a vector length of 4,096 bits. At a frequency of 500 MHz and an RV64GCV1.0 ISA, they reported areas of 0.44 mm^2 for Ara and 1.10 mm^2 for SPEED. The RS5 results consider the same frequency and technology used in Wang et al. work. With the same number of lanes (4) and a VLEN of 128 bits, RS5 requires no more than 0.080 mm^2, 5.5 times smaller than ARA and 13.75x smaller than SPEED. The largest RS5 RVV version (with eight lanes and 256-bit VLEN) requires 0.145 mm^2, corresponding to one-third the area of Ara and $\approx$7.5 times smaller than SPEED. These results also show that despite having bigger VLEN (4096), the area results are contained in ARA and SPEED, showcasing that fewer processing lanes (only four) can help the contention of the overheads.

Regarding equivalent gates, Table 7.10 compares the complete RS5 subset with other works that provide this information. With the same number of lanes, the RS5 im-

Table 7.10: Equivalent gates results in KGE for RS5 and other RVV implementations.

RVV implementation	2-Lanes	4-Lanes	8-Lanes
ARA [Cavalcante et al., 2019]	2228	3434	5902
ARA2 [Perotti et al., 2024]	2291	3688	6768
This Work	131	213	386

plementation in this work occupies significantly less area, ranging from 15.29× to 17.53× smaller. It is important to note that ARA focuses on high-end acceleration and implements the full RVV extension, whereas this work targets only a subset. Nevertheless, this result shows how large implementations can become.

ZeroVex [Zhao and Ye, 2024] synthesized their design to the SkyWater open source PDK, achieving an area for the scalar core of 0.15 mm^2. For a 2-lane vector accelerator, the area increases to 0.40 mm^2, almost tripling the size, similar to what happens in the RS5 core with a 2-lane vector accelerator, which achieves an area of 0.041 mm^2 (note that each design employs different technology nodes).

Ali et al. [2021] synthesized their Vector accelerator (VPU) for Xilinx FPGA Zynq xc7z020clg484-1 using Xilinx Vivado v2018.3. Each lane utilizes three DSP blocks. When synthesized to a similar Xilinx FPGA board, the RS5 with a VLEN of 64 bits required around 7,000 additional LUTs, while in Ali et al. [2021] for the same VLEN, it required around 3,800. Even though RS5 requires almost twice the amount of LUTS used by Ali et al. [2021] with a VLEN of 64 bits, RS5 implements an additional lane justifying this overhead. In the 256-bit versions, both accelerators have eight lanes, with RS5 requiring approximately 24,000 LUTs and their work requiring over 20000 LUTs.

The authors of ARROW [Assir et al., 2021] found that, in the ARROW accelerator, a 2D convolution runs only 1.4 – 1.9× faster. They concluded that this relatively low performance of 2D convolution is mainly due to their highly repetitive use of scalar arithmetic operations to manage data pointers. Our work achieved similar performance for a 2D convolution in AlexNet, achieving a maximum of 1.33x speed-up.

Johns and Kazmierski [2020] presented a vector processor coupled with an embedded RISC-V. They use a VLEN of only 32 bits with a single lane. The authors used three benchmarks: 1)RGB to grayscale using shifts; 2) Grayscale filter; and 3) Integer matrix multiplication of two 120x120 matrices with 8-bit data. All the applications were written in assembly. Compared to the scalar core, the performance results ranged from 2.7x to 5.8x speed-ups, increasing FPGA resource usage by 2.6x.

These results indicate that the RS5 VSU achieves an area comparable to or smaller than similar works. However, direct comparisons between implementations are challenging and often unfair due to differences in subsets, technologies, protocols, and memory arrangements. Unlike many related works, the RS5 VXU lacks a dedicated memory port, which is a known performance bottleneck. Additionally, it does not allow independent scaling of the

number of lanes, as this is tied to the VLEN parameter, potentially leading to significant area overheads as VLEN increases.

7.2 Manycore Results

This Section presents the results for running the AlexNet [Krizhevsky et al., 2017] 2-D CNN in a manycore environment. The manycore environment is the Memphis-V platform, which uses the RS5 processor as the processing element (PE). This allows the exploration of the RVV extension in a broader range of application characteristics.

Section 6.3 presented the 2-D CNN application and the chosen technique for partitioning the application into different programs. The programs are:

- *P1* - Handles only one out of three input channels of the first layer, performing the depthwise convolution and sending the results to the P2 process;
- *P2* - Joins the results from all the P1 instances, performing ReLU and Max-Pool operations on each output channel. Sends data to all P3 instances;
- *P3* - handles multiple input channels of the second layer, performing the depthwise convolution and sending the results to the P4 process;
- *P4* - Joins the results from all the P3 instances, performing ReLU and Max-Pool operations on each output channel.

The chosen NoC dimension was a 4x4 NoC, which mapping can be seen at Figure 6.6. Simulations were performed using a scalar PE and three accelerated versions. The three accelerated versions include the RVV extension with VLENs of 64, 128, and 256 bits.

Table 7.11 presents the results of the P1 application/processes on the Memphis-V platform. Each process only computes one of the three RGB input channels and generates the 96 output channels of the first convolutional layer for that given input channel.

Table 7.11: Results on running RGB channels computation (P1) on MEMPHIS-V with vector extension.

VLEN	Avg cycles per channel in computation	Speed-up in computation	Total cycles in computation (x96)	Avg cycles per channel in communication	Total cycles in communication (x96)
-	4.74e+6	-	4.55e+8	1.19e+5	1.14e+7
64	4.75e+6	1.00x	4.56e+8	1.19e+5	1.14e+7
128	2.65e+6	1.79x	2.54e+8	1.34e+5	1.28e+7
256	2.37e+6	2.00x	2.27e+8	1.39e+5	1.34e+7

The average number of cycles taken to compute each output channel can be seen in the second column of Table 7.11. The scalar and the 64-bit VLEN versions take a similar cycle count. In contrast, the versions with larger VLENs obtain significant speed-ups in computation, reducing up to 2x the number of cycles required for the calculation of each output channel. The VLEN growth negatively impacted the time spent communicating (sending

each output channel to the P2 process), and no plausible explanation was found for this. The time taken for communication is smaller by one order of magnitude, as the application computation load is much bigger than the amount of data required to be sent.

Table 7.12 presents the cycle results of the P2 process, responsible for joining the input channels of the first convolutional layer and performing ReLU and Max-Pool operations. This process's first operation is waiting for the outputs of each P1 process. The average cycle count spent waiting for P1 results ranges from 4.54e+6 (scalar) to 2.33e+6 (256-bit VLEN), which is very similar to the time taken per channel in the P1 process (4.74e+6 to 2.37e+6). This indicates that the P2 process relies more on the results from P1 than it spends time computing ReLU and max-pooling or sending data to P3.

Table 7.12: Results on running RGB channels join (P2) plus ReLU and Max-Pool on MEMPHIS-V with vector extension.

VLEN	Avg cycles per channel receive	Total cycles receiving (x96)	Avg cycles per channel computation	Total cycles computation (x96)	Avg cycles per channel sending	Total cycles sending (x96)
-	4.54e+6	4.36e+8	2.92e+5	2.80e+7	3.65e+4	3.50e+6
64	4.62e+6	4.44e+8	2.14e+5	2.06e+7	3.65e+4	3.51e+6
128	2.57e+6	2.46e+8	1.85e+5	1.77e+7	3.65e+4	3.50e+6
256	2.33e+6	2.23e+8	1.48e+5	1.42e+7	3.65e+4	3.50e+6

The average number of cycles per computation performed in the P2 process is shown in the fourth column of Table 7.12. It is one order of magnitude smaller than the time it waits for data from the P1 process, indicating that it quickly processes it and sends it forward once it has data. The average number of cycles required to send the data to the P3 process is another order of magnitude smaller than the computation time. The time taken in the communication in P2 is around 3 times smaller than in P1. This happens as the packages transmitted by P2 are reduced in size due to the data passing through a max-pooling layer.

Table 7.13 compares the single and manycore versions of the first layer of the 2-D CNN benchmark, both exploring the scalar and vectorized versions for multiple VLENs.

Table 7.13: Peformance on the AlexNet first layer on single x manycore.

Single/Manycore	VLEN	Cycles	Speed-up	Cycle reduction (%)
Single	-	1,418,604,031	-	-
Single	64	1,418,604,052	1.00x	0.00
Single	128	1,012,741,796	1.40x	28.61
Single	256	858,535,157	1.65x	39.48
Many	-	474,913,995	2.99x	66.52
Many	64	475,713,468	2.98x	66.47
Many	128	275,530,467	5.15x	80.58
Many	256	248,906,823	5.70x	82.45

While in the single-core version, the speed-up of using the vector extension reached 1.65x, by applying the manycore mapping technique, the speed-up reached almost 3x in using scalar PEs. The 3x speed-up in the scalar core is significant. Still, the theoretical gain of

using 4 PEs would be 4x, which cannot be achieved in real applications for multiple reasons, like communication and non-optimal computation division. As previously demonstrated, the P2 process performs significantly less computation during operation. It spends the majority of its time awaiting data from the P1 processes. If we exclude the P2 process, the 3x speed-up gets closer to the 3x theoretical gain.

The best trade-off was the many-core version with a 128-bit VLEN, offering a speed-up of 5.15x, meaning a reduction of around 80% of the cycles compared to a scalar single-core and 1.72x of the scalar manycore.

The results of the manycore simulations show that the applied technique significantly improved the time per inference, achieving a maximum of 5.70x of speed-up compared to a single scalar core.

The number of PEs used more than doubles when running the first two layers of the 2-D CNN application. The first layer uses 3 PEs for the P1 instances and 1 PE for the P2 instance, while the second layer uses 8 PEs for the P3 processes and another one for the P4 process. However, extracting results for the application with all four programs was impossible because the P3 processes presented data access fault exceptions, which halted the simulation. This exception occurs when accessing data outside the process data page, meaning the parameters can not fit inside the allocated memory page. Despite using the maximum page size, this error continued to occur. The data memory in the MEMPHIS-V platform is limited to 24 bits, and platform modifications would be required to allow expansions beyond 24 bits.

In the first layer of AlexNet, there are 189,531 integer parameters (154,587 inputs, 34,848 weights, and 96 biases), and in the second, 614,656 integer parameters(614,400 weights and 256 biases). In a manycore environment, these parameters are distributed through the program instances. The P1 program counts with three instances, distributing a third of the input and weights to each instance. Thus, each process has only 63,145 parameters compared to a single instance's 189,435 (not including bias) parameters, representing a reduction of 66.7%. The P3 program counts with eight instances, where each will count with 76,800 parameters, a significant reduction compared to the 614,400 required when only one instance is used, representing a reduction of 87.5%. The P2 and P4 programs will receive only the biases, representing 96 and 256 parameters, respectively. Considering a single processor running the first two layers of AlexNet, it would use 804.187 parameters. In a manycore environment, the process instance that uses the most parameters (P3 program) uses 76,800 parameters, representing a reduction of 90.4% in code size.

To avoid dealing with limited space in the process memory pages, a possible solution is to have an external memory to store the application parameters. This approach helps keep the local storage small, with the weights and biases stored on an external memory being loaded dynamically to the local memory when needed.

7.2.1 Related work comparison

As this work explored mainly the impact of an accelerated PE, the works that better relate to it are those presented by Table 3.3. Although we use a NoC, comparing results with NoC-related works (Table 3.4) is difficult as they focus on having dedicated NoC, modifying NoC components, or implementing mapping algorithms, charactheristics that our work does not present. Our work does not implement dedicated NoCs because it focuses on having a general-purpose PE, in which one of the applications executed can be a CNN. It is not built to target only CNNs. The NoC used presents a significant throughput, as demonstrated by the developers in previous works, and is not a bottleneck for the target applications. This work does not seek to implement the most optimal mapping in the NoC platform. Our main goal was to evaluate the impact that an accelerated PE would have, so we did not use any mapping algorithm.

SiFive Performance P270 [SiFive, Inc, 2022] and SiFive Intelligence X280 [SiFive, Inc, 2023] use multicore and multi-cluster configurations of up to 8 processing cores. This configuration would resemble a NoC with a 3x3 dimension, resulting in 9 processing cores. However, they target high-performance computing, and due to these processors being proprietary, results for comparison could not be found.

GAP-8 [Flamand et al., 2018] uses a RISC-V controller allied with an 8-Core cluster with an additional dedicated accelerator for convolutions, the Hardware Convolution Engine (HWCE). For applications like convolutions, Max-Pooling, and matmul, they achieved speed-ups ranging from 5.3x to 8.4x when using eight cores in the cluster. These results are comparable to what this work presented in acceleration regarding the first layer of the CNN using a distributed approach, where with only four PEs we achieved a speed-up of up to 5.7x. For full CNNs, the performance improvement of GAP-8 ranged from 9.3x to 11.7x using 8-PEs.

ARA2 [Perotti et al., 2024] concluded that a multi-core design with smaller Ara2 instances behaves better than a single-core larger Ara2. This work achieved a similar result as having multiple scalar cores achieved a better speed-up (almost double) than the biggest single core with VLEN of 256 bits.

7.3 Conclusion and Final Remarks

This Chapter presented Power, Performance, and Area evaluations for the accelerated RS5 processor with the vector extension.

It first explored the impact of including the RVV extension in the RS5 core, presenting the area overheads for different VLEN configurations in ASIC and FPGA environments.

The smaller VLEN has an area overhead ranging from 2.89 to 3.49x, while the bigger VLEN evaluated presented an area overhead ranging from 8.04x to 10.27x. Then, it presented the RS5 area utilization for each component, highlighting that including the VXU unit does not negatively impact other components. It also showed that the VXU components that consume the most area and are directly connected to the VLEN growth are the VRF and the VALU. The VALU is primarily responsible for the growth of the expressive area, which accounts for up to 54.7% of the VXU area. Comparing our work with related work positioned it with comparable results. The comparison showcased that the RS5 VXU is small compared to more complex RVV implementations. A configurable number of processing Lanes might be the best approach for keeping the area resources low.

The performance evaluation explored the different RS5 extensions and the impact of including the VXU. It also examined the compiler flags, concluding that the best performances often occur for non-scalar codes. The LMUL parameter was explored, showing that acceleration is directly connected to vector lengths. The RS5 equipped with the VXU attained speed increases of up to 4.6x in a basic 1-D CNN and up to 1.98x in the full version. An impressive speed-up of 7.68x was achieved with a manually vectorized code targeting a 256-bit VLEN in the simplified CNN. In a 2-D CNN, it achieved speed-ups of up to 1.37x. The performance evaluation also explored a dot-product application evaluation, presenting an in-depth analysis of the impact of vector size, VLEN, LMUL, and SEW on the number of cycles required to run the computations. Due to its increased parallelism, the 8-bit SEW showed the best performance for larger vectors.

Power and Energy evaluation highlighted that the accelerated core can significantly reduce energy consumption despite the increased area overhead. With a large vector (4096) of 8-bit elements, it reduced energy consumption by up to 48% in the dot-product application. Energy gains were not achieved for wider data like 32 bits due to the limited parallelism, which implies smaller speed-ups. For the simplified 1-D CNN benchmark, the VLEN of 128 bits showed a reduction of approximately 7%, despite using 32-bit data. The manually vectorized version obtained around 64% of energy consumption reduction. These results show that vector extension can significantly reduce energy consumption, even with the area overheads when used under the right conditions to explore the maximum parallelism.

Finally, using a manycore to accelerate the 2-D CNN showed impressive results. The parallelization technique significantly improved the time taken to execute a single layer of a large CNN. The speed-up ranges from 2.98x to 5.70x, reducing the cycles required by more than 80% compared to a single scalar core, while the code size was reduced by up to 90.4%.

8. CONCLUSIONS AND FUTURE WORK

This work presented a detailed study on accelerating machine learning workloads, particularly Convolutional Neural Networks (CNNs), using the RISC-V Vector Extension (RVV) in both single-core and manycore architectures. The research expanded the RS5 processor, a RISC-V-based core implementing a subset of RVV instructions, and evaluated its performance in different hardware environments.

The area evaluation proved the need to select the correct subset to implement in the processor, as the overheads can become quite large as the VLEN parameter grows. Including vector division operations proved very costly in terms of area, adding an overhead of up to 24.6% for operations not often used by the target application. Therefore, vector division operations should only be included if strictly necessary.

Although the area overheads of including the vector extension in the RS5 were significant, other factors should be considered. The first fact is that the RS5 processor is a small processor with a compact area footprint. Consequently, any addition can lead to significant area overhead. The second factor is that it is up to 17 times smaller than similar implementations. The third factor is that in RS5 VXU, the lanes are the most area-intensive resource, and the number of lanes is linked to VLEN, which grows exponentially.

Overall, the VXU with a VLEN of 256 bits presented the best results in terms of performance trade-off, as larger VLENs often do not significantly increase the speed-ups to justify their area overhead. A 256-bit VLEN design uses 8x 32-bit lanes to process data. A configurable number of lanes would allow us to explore the impact of the VLEN and the number of lanes separately. Regarding the area trade-off, the best option was the four-lane design (128-bit VLEN), which uses no more than 5.67x the area of a scalar core.

The energy evaluation proved that even with a significant area overhead, which increases the power consumption, the circuits can reduce energy consumption thanks to decreased time taken per operation.

The RISC-V vector extension proved very effective for accelerating CNN applications, achieving significant speed-ups for different benchmarks. However, by doing a manual vectorization, this work proved that the vector extension can achieve even better results, highlighting the compiler's auto-vectorization limitations.

By exploring the vector extension compiler options, this work concluded that using non-scalable code can yield better performance in most cases and that using larger register groups is the better option for larger vectors. A dot-product benchmark helped to identify multiple relationships between the parameters of the VXU and the application with the yielded speed-ups. Smaller vectors do not present significant speed-ups that justify the inclusion of the vector extension, while larger vectors can have better results for larger VLENs and register groups.

Overall, increasing the VLEN size increases the data parallelism, thus improving performance. The best-case scenario for achieving maximum parallelism uses a SEW of 8 bits, while the worst-case scenario uses a SEW of 32 bits (SEW=ELEN). Applications using smaller SEWs can yield better speed-ups with smaller VLENs, while bigger SEWs require bigger VLENs to yield similar speed-ups. Another finding to note is that using smaller SEWs allows the processing of larger vectors more effectively.

The results for complete CNNs fell short of expectations. The achieved speed-ups of 1.91x and 1.3x in the 1-D and 2-D CNN do not entirely justify the area overhead of up to 10x for the associated hardware with a 256-bit VLEN. This can be due to the C-language code's use of multiple memory pointers, which can limit the compiler's auto-vectorization. Another factor limiting the speed-up is that the benchmarks use 32-bit SEWs, the worst-case scenario regarding data parallelism.

Implementing the vector load-store unit using a single lane and sharing the processor memory bus proved a good choice. Despite being a known bottleneck, this approach allowed the processor to be used without requiring any modifications in the environments it was used, especially in the manycore environment. Also, the enhanced data parallelism in computation can compensate for the slower memory access.

In the manycore setup, the parallelization strategy improved execution by up to 5.7x compared to a scalar single-core implementation. These results indicate that combining vectorized processing and manycore mapping strategies significantly enhances the efficiency of CNN inference. Distributing a CNN layer to multiple PEs also helps keep memory usage low by reducing the number of parameters each PE must hold.

Although distributing the parameters through multiple PEs reduces memory usage, using memory pages in the manycore environment limits the number of parameters the applications can have. To avoid this limitation, it is recommended to store weights and bias in an external shared memory and read these parameters when necessary.

The results highlight the potential of RISC-V vector extensions as a viable solution for accelerating CNN tasks in single-core and manycore platforms. By integrating an open-source, customizable architecture with efficient vectorized execution, this work contributes to the ongoing development of scalable and energy-efficient CNN accelerators.

8.1 Future Work

As future work, the following tasks can be explored:

- Implementing a configurable number of lanes in the RS5 VXU to allow the area footprint to be contained for bigger VLEN configurations;

- Explore other CNN benchmarks to evaluate more recent CNN architectures and to explore the impact of the CNN window sizes in the acceleration and their relationship with the VLEN;
- Rework the benchmark codes to improve auto-vectorization, reducing pointer handling, which can be malefic to the compiler's vectorization capabilities;
- Explore other machine learning structures beyond CNN applications, testing the suitability of the vector extension for other application classes;
- Modify the MEMPHIS-V platform to support larger page sizes or external shared memories with the application parameters to allow having multiple CNN layers in a manycore application;
- Explore other mapping techniques in the manycore environment to evaluate the effectiveness of different approaches and test the better NoC configurations to improve the parallelism.

8.2 Publications During de MsC Period

Accelerating Machine Learning with RISC-V Vector Extension and Auto-Vectorization Techniques
NUNES, Willian Analdo; SANTOS, Antônio Vinicius Corrêa; MORAES, Fernando Gehm; Fernando Gehm Moraes
In: ISCAS, 2025
to be published

Deploying human activity recognition in embedded RISC-V processors
NUNES, Willian Analdo; REUSCH, Rafael Schild; LUZA, Lucas; BERNARDON, Eduardo; DAL ZOTTO, Angelo Elias; JURACY, Leonardo Rezende; MORAES, Fernando Gehm.
Design Automation For Embedded Systems, vol. 28, n. 3, pp. 187-217, December 2024.
http://dx.doi.org/10.1007/s10617-024-09288-w

RS5: An Integrated Hardware and Software Ecosystem for RISC-V Embedded Systems
NUNES, Willian Analdo; ZOTTO, Angelo Elias Dal; BORGES, Caroline da Silva; MORAES, Fernando Gehm
In: LASCAS, 2024
http://dx.doi.org/10.1109/LASCAS60203.2024.10506171

Validating an Automated Asynchronous Synthesis Environment with a Challenging Design: RISC-V
NUNES, Willian; SARTORI, Marcos; MOREIRA, Matheus; MORAES, Fernando Gehm; CALAZANS, Ney
In: SBCCI, 2023
http://dx.doi.org/10.1109/SBCCI60457.2023.10261656

REFERENCES

Abiodun, O. I., Jantan, A., Omolara, A. E., Dada, K. V., Mohamed, N. A., and Arshad, H. (2018). State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11). https://doi.org/10.1016/j.heliyon.2018.e00938.

Ali, M., von Ameln, M., and Goehringer, D. (2021). Vector Processing Unit: A risc-v based simd co-processor for embedded processing. In *Euromicro Conference on Digital System Design (DSD)*, pages 30–34. https://doi.org/10.1109/DSD53832.2021.00014.

ARM (2016). The DSP capabilities of ARM® Cortex®-M4 and Cortex-M7 Processors. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjF5tu2sPaCAxU4ppUCHazLCmgQFnoECA8QAQ&url=https%3A%2F%2Fcommunity.arm.com%2Fcfs-file%2F__key%2Fcommunityserver-blogs-components-weblogfiles%2F00-00-00-21-42%2F7563.ARM-white-paper-_2D00_-DSP-capabilities-of-Cortex_2D00_M4-and-Cortex_2D00_M7.pdf&usg=AOvVaw1b5YLuHoGCWt6fADtBPQci&opi=89978449.

ARM (2025). ARM Official Website. https://www.arm.com/.

AskariHemmat, M., Bilaniuk, O., Wagner, S., Savaria, Y., and David, J.-P. (2021). RISC-V barrel processor for deep neural network acceleration. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. https://doi.org/10.1109/ISCAS51556.2021.9401617.

Assir, I. A., Iskandarani, M. E., Sandid, H. R. A., and Saghir, M. A. (2021). Arrow: A RISC-V Vector Accelerator for Machine Learning Inference. *CoRR*, abs/2107.07169:1–6. https://arxiv.org/abs/2107.07169.

Barnes, G., Brown, R., Kato, M., Kuck, D., Slotnick, D., and Stokes, R. (1968). The ILLIAC IV Computer. *IEEE Transactions on Computers*, C-17(8):746–757. https://doi.org/10.1109/TC.1968.229158.

Beldianu, S. F. and Ziavras, S. G. (2014). ASIC design of shared vector accelerators for multicore processors. In *IEEE International Symposium on Computer Architecture and High Performance Computing*, pages 182–189. https://doi.org/10.1109/SBAC-PAD.2014.13.

Blythe, D. (2008). Rise of the Graphics Processor. *Proceedings of the IEEE*, 96(5):761–778. https://doi.org/10.1109/JPROC.2008.917718.

Carara, E. A., de Oliveira, R. P., Calazans, N. L. V., and Moraes, F. G. (2009). HeMPS - a Framework for NoC-based MPSoC Generation. In *IEEE International Symposium on*

Circuits and Systems (ISCAS), pages 1345–1348. https://doi.org/10.1109/ISCAS.2009.5118013.

Cavalcante, M., Schuiki, F., Zaruba, F., Schaffner, M., and Benini, L. (2019). Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multiprecision Floating-Point Support in 22-nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(2):530–543. https://doi.org/10.1109/TVLSI.2019.2950087.

Chang, A. X. M. and Culurciello, E. (2017). Hardware Accelerators for Recurrent Neural Networks on FPGA. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. https://doi.org/10.1109/ISCAS.2017.8050816.

Chen, C., Xiang, X., Liu, C., Shang, Y., Guo, R., Liu, D., Lu, Y., Hao, Z., Luo, J., Chen, Z., et al. (2020). Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 52–64. https://doi.org/10.1109/ISCA45697.2020.00016.

Chen, K.-C. J., Peng, H.-H., and Shen, P.-C. (2024). Ultra-NoC: Unified Low-Transmission Routing Assisted NoC for High-flexible DNN Accelerator. In *IEEE International System-on-Chip Conference (SOCC)*, pages 1–5. https://doi.org/10.1109/SOCC62300.2024.10737754.

Chen, Y.-H., Yang, T.-J., Emer, J., and Sze, V. (2019). Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308. https://doi.org/10.1109/JETCAS.2019.2910232.

Consortium, E. E. M. B. (2024). CoreMark - An EEMBC Benchmark. https://www.eembc.org/coremark/index.php.

Dalzotto, A. E., Ruaro, M., Erthal, L. V., and Moraes, F. G. (2021). Management Application - a New Approach to Control Many-Core Systems. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6. https://doi.org/10.1109/SBCCI53441.2021.9529989.

Dao, N., Attwood, A., Healy, B., and Koch, D. (2020). FlexBex: A RISC-V with a Reconfigurable Instruction Extension. . In *International conference on field-programmable technology (ICFPT)*, pages 190–195. https://doi.org/10.1109/ICFPT51103.2020.00034.

Flamand, E., Rossi, D., Conti, F., Loi, I., Pullini, A., Rotenberg, F., and Benini, L. (2018). GAP-8: A RISC-V SoC for AI at the Edge of the IoT. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 1–4. https://doi.org/10.1109/ASAP.2018.8445101.

Gao, J., Shao, Q., Deng, F., Wang, Q., Jing, N., and Jiang, J. (2023). An NoC-based CNN Accelerator for Edge Computing. In *International Conference on ASIC (ASICON)*, pages 1–4. https://doi.org/10.1109/ASICON58565.2023.10396346.

Garofalo, A., Tagliavini, G., Conti, F., Rossi, D., and Benini, L. (2020). XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions. In *ACM/IEEE Design, Automation Test in Europe Conference (DATE)*, pages 186–191. https://doi.org/10.23919/DATE48585.2020.9116529.

Goel, A., Goel, A. K., and Kumar, A. (2023). The role of artificial neural network and machine learning in utilizing spatial information. *Spatial Information Research*, 31(3):275–285. https://doi.org/10.1007/s41324-022-00494-x.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. https://doi.org/10.1109/CVPR.2016.90.

Hennessy, J. L. and Patterson, D. A. (2018). John Hennessy and David Patterson 2017 ACM A.M. Turing Award Lecture. https://www.youtube.com/watch?v=3LVeEjsn8Ts&t=1h15m.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). MobileNets: efficient convolutional neural networks for mobile vision applications (2017). *CoRR*, abs/1704.04861:1–9. http://arxiv.org/abs/1704.04861.

Hudomalj, U., Mandla, C., and Plattner, M. (2020). FPGA Implementations for Real-Time Processing of High-Frame-Rate and High-Resolution Image Streams. In *International Conference on Computing, Electronics & Communications Engineering (iCCECE)*, pages 211–216. https://doi.org/10.1109/iCCECE49321.2020.9231119.

Hwu, W.-m. and Patel, S. (2018). Accelerator Architectures — A Ten-Year Retrospective. *IEEE Micro*, 38(6):56–62. https://doi.org/10.1109/MM.2018.2877839.

Intel Corporation (2025). Intel Official Website - Brasil. https://www.intel.com.br/content/www/br/pt/homepage.html.

Jacobs, E., Utyansky, D., Hassan, M., and Roecker, T. (2024). RISC-V V Vector Extension (RVV) with reduced number of vector registers. *CoRR*, abs/2410.08396:1–9. https://doi.org/10.48550/arXiv.2410.08396.

Johns, M. and Kazmierski, T. J. (2020). A minimal RISC-V vector processor for embedded systems. In *Forum for Specification and Design Languages (FDL)*, pages 1–4. https://doi.org/10.1109/FDL50818.2020.9232940.

Juracy, L. R., Moreira, M. T., Amory, A. M., and Moraes, F. G. (2021). A TensorFlow and System Simulator Integration Approach to Estimate Hardware Metrics of Convolution Accelerators. In *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–4. https://doi.org/10.1109/LASCAS51355.2021.9459183.

Kim, M. and Shao, Y. S. (2018). Hardware Acceleration. *IEEE Micro*, 38(6):6–7. https://doi.org/10.1109/MM.2018.2881546.

Korol, G. (2019). An FPGA implementation for convolutional neural network. Master's thesis, Pontifical Catholic University of Rio Grande do Sul.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90. https://doi.org/10.1145/3065386.

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., and Jackel, L. (1989). Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems*, 2:396–404.

Liang, Z., Hu, F., Xu, B., and Wei, C. (2024). Multi objective non dominated sorting whale optimization genetic algorithm for convolutional neural network-based on-chip networks. In *International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, pages 653–656. https://doi.org/10.1109/AINIT61980.2024.10581699.

Lomont, C. (2011). Introduction to Intel Advanced Vector Extensions. *Intel white paper*, 23:1–21. https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf.

Mahesh, B. (2020). Machine learning algorithms-a review. *International Journal of Science and Research (IJSR)*, 9(1):381–386. https://www.ijsr.net/archive/v9i1/ART20203995.pdf.

Microsoft (2024). Copilot+ PCs. https://www.microsoft.com/en-us/windows/copilot-plus-pcs?r=1.

Moraes, F. G., Calazans, N. L. V., Mello, A. V., Möller, L. H., and Ost, L. C. (2004). HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip. *Integration, the VLSI journal*, 38(1):69–93. https://doi.org/10.1016/j.vlsi.2004.03.003.

Nunes, W. A., Dal Zotto, A. E., Borges, C. d. S., and Moraes, F. G. (2024). RS5: An Integrated Hardware and Software Ecosystem for RISC-V Embedded Systems. In *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–5. https://doi.org/10.1109/LASCAS60203.2024.10506171.

NVIDIA Corporation (2025). NVIDIA Official Website. https://www.nvidia.com/en-us/.

Papaphilippou, P., HJ, K. P., and Luk, W. (2021). Simodense: a RISC-V softcore optimised for exploring custom SIMD instructions. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 391–397. https://doi.org/10.1109/FPL53798.2021.00082.

Patterson, J. and Gibson, A. (2017). *Deep learning: A practitioner's approach*. O'Reilly Media, Inc., 1st edition. 507 pages.

Peccerillo, B., Mannino, M., Mondelli, A., and Bartolini, S. (2022). A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture*, 129:1–51. https://doi.org/10.1016/j.sysarc.2022.102561.

Peleg, A. and Weiser, U. (1996). MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50. https://doi.org/10.1109/40.526924.

Perotti, M., Cavalcante, M., Andri, R., Cavigelli, L., and Benini, L. (2024). Ara2: Exploring Single- and Multi-Core Vector Processing With an Efficient RVV 1.0 Compliant Open-Source Processor. *IEEE Transactions on Computers*, 73(7):1822–1836. https://doi.org/10.1109/TC.2024.3388896.

Perotti, M., Cavalcante, M., Wistoff, N., Andri, R., Cavigelli, L., and Benini, L. (2022). A New Ara for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 43–51. https://doi.org/10.1109/ASAP54787.2022.00017.

Phung, V. H. and Rhee, E. J. (2018). A Deep Learning Approach for Classification of Cloud Image Patches on Small Datasets. *Journal of information and communication convergence engineering*, 16(3):173–178. https://doi.org/10.6109/jicce.2018.16.3.173.

Pircher, S., Geier, J., Zeh, A., and Mueller-Gritschneder, D. (2021). Exploring the RISC-V Vector Extension for the Classic McEliece post-quantum cryptosystem. In *International Symposium on Quality Electronic Design (ISQED)*, pages 401–407. https://doi.org/10.1109/ISQED51717.2021.9424273.

PyTorch (2024). PyTorch. https://pytorch.org/.

Rahman, A., Khan, A. A., and Shoumik, T. M. (2022). *A novel lightweight CNN approach for Bangladeshi sign language gesture recognition*. PhD thesis, Brac University. https://dspace.bracu.ac.bd/xmlui/handle/10361/17893.

Rauber, T. and Rünger, G. (2013). *Parallel Programming for Multicore and Cluster Systems*. Springer, 2nd edition. 516 pages.

Red Hat (2024). newlib 4.3.0. https://sourceware.org/newlib/.

Reusch, R. S., Juracy, L. R., and Moraes, F. G. (2022). Assessment and Optimization of 1D CNN Model for Human Activity Recognition. In *Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–7. https://doi.org/10.1109/SBESC56799.2022.9964520.

Reusch, R. S., Juracy, L. R., and Moraes, F. G. (2023). Deploying Machine Learning in Resource-Constrained Devices for Human Activity Recognition. In *Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–6. https://doi.org/10.1109/SBESC60926.2023.10324073.

RISC-V Foundation (2015). Unit Tests for RISC-V Processors. https://github.com/riscv/riscv-tests.

RISC-V Foundation (2021). RISC-V "V" Vector Extension. https://github.com/riscv/riscv-v-spec/releases/tag/v1.0.

RISC-V Foundation (2024a). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20240411. https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link, January 2025.

RISC-V Foundation (2024b). The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20240411. https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view.

RK, R., Sinha, S., and Rao, N. (2021). Variable Bit-Precision Vector Extension for RISC-V Based Processors. In *IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 114–121. https://doi.org/10.1109/MCSoC51149.2021.00024.

Ruaro, M., Caimi, L. L., Fochi, V., and Moraes, F. G. (2019). Memphis: a framework for heterogeneous many-core SoCs generation and validation. *Springer Design Automation for Embedded Systems*, 23(3-4):103–122. https://doi.org/10.1007/s10617-019-09223-4.

Ruaro, M., Lazzarotto, F. B., Marcon, C. A., and Moraes, F. G. (2016). DMNI: A Specialized Network Interface for NoC-based MPSoCs. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1202–1205. https://doi.org/10.1109/ISCAS.2016.7527462.

Russell, R. M. (1978). The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72. https://doi.org/10.1145/359327.359336.

Sanders, J. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional.

Schmidt, C., Ou, A., and Asanović, K. (2018). Hwacha V4: Decoupled Data Parallel Custom Extension. In *Proc. Inaugural RISC-V Summit*, pages 1–40.

Schmidt, P., Pfau, J., Hotfilter, T., Stammler, M., Harbaum, T., and Becker, J. (2024). RVVe: A Minimal RISC-V Vector Processor for Embedded AI Acceleration. In *IEEE International System-on-Chip Conference (SOCC)*, pages 1–6. https://doi.org/10.1109/SOCC62300.2024.10737723.

SiFive, Inc (2017a). SiFive E31 Core Complex Manual v1p2. https://static.dev.sifive.com/E31-RISCVCoreIP.pdf.

SiFive, Inc (2017b). SiFive E51 Core Complex Manual v1p2. https://static.dev.sifive.com/E31-RISCVCoreIP.pdf.

SiFive, Inc (2020). *SiFive Interrupt Cookbook, Version 1.2*. https://www.starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf.

SiFive, Inc (2022). SiFive Performance P270 Data Sheet. https://sifive.cdn.prismic.io/sifive/859c28c0-8bd5-4fc4-9113-a25a2a89bf9c_P270+Data+Sheet.pdf.

SiFive, Inc (2023). SiFive Intelligence X280 Data Sheet. https://sifive.cdn.prismic.io/sifive/9405d3d0-35a1-4680-a259-7a5598d1ecb2_sifive-intelligence-x200-datasheet.pdf.

Silvano, C. et al. (2023). A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms. *CoRR*, abs/2306.15552:1–58. https://doi.org/10.48550/arXiv.2306.15552.

Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., Reid, A., Rico, A., and Walker, P. (2017). The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39. https://doi.org/10.1109/MM.2017.35.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going Deeper With Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9. https://doi.org/10.1109/CVPR.2015.7298594.

TensorFlow (2024). TensorFlow. https://www.tensorflow.org/.

Tiwari, B., Yang, M., Wang, X., and Jiang, Y. (2022). In-Network Accumulation: Extending the Role of NoC for DNN Acceleration. In *IEEE International System-on-Chip Conference (SOCC)*, pages 1–6. https://doi.org/10.1109/SOCC56010.2022.9908106.

Upadhyay, M., Juneja, R., Wong, W.-F., and Peh, L.-S. (2024). NOVA: NoC-based Vector Unit for Mapping Attention Layers on a CNN Accelerator. In *ACM/IEEE Design, Automation Test in Europe Conference (DATE)*, pages 1–6. https://doi.org/10.23919/DATE58400.2024.10546727.

Wachter, E., Caimi, L. L., Fochi, V., Munhoz, D., and Moraes, F. G. (2017). BrNoC: A broadcast NoC for control messages in many-core systems. *Microelectronics Journal*, 68:69–77. https://doi.org/10.1016/j.mejo.2017.08.010.

Wang, C., Fang, C., Wu, X., Wang, Z., and Lin, J. (2024). A Scalable RISC-V Vector Processor Enabling Efficient Multi-Precision DNN Inference. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. https://doi.org/10.1109/ISCAS58744.2024.10558028.

Wang, S., Zhu, J., Wang, Q., He, C., and Ye, T. T. (2021). Customized Instruction on RISC-V for Winograd-Based Convolution Acceleration. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 65–68. https://doi.org/10.1109/ASAP52443.2021.00018.

Ye, J., Ge, F., and Zhou, F. (2023). A Method of Mapping Convolutional Neural Networks on Resource-limited NoC Platform. In *International Conference on ASIC (ASICON)*, pages 1–4. https://doi.org/10.1109/ASICON58565.2023.10396382.

Yu, X., Yang, Z., Peng, L., Lin, B., Yang, W., and Wang, L. (2022). CNN Specific ISA Extensions Based on RISC-V Processors. In *International Conference on Circuits, Systems and Simulation (ICCSS)*, pages 116–120. https://doi.org/10.1109/ICCSS55260.2022.9802445.

Zhang, K., Ding, L., Cai, Y., Yin, W., Yang, F., Tao, J., and Wang, L. (2017). A high performance real-time edge detection system with NEON. In *International Conference on ASIC (ASICON)*, pages 847–850. https://doi.org/10.1109/ASICON.2017.8252609.

Zhang, L., Zhou, X., and Guo, C. (2021). A CNN Accelerator with Embedded RISC-V Controllers. In *China Semiconductor Technology International Conference (CSTIC)*, pages 1–3. https://doi.org/10.1109/CSTIC52283.2021.9461576.

Zhao, T. and Ye, Z. (2024). ZeroVex: A Scalable and High-performance RISC-V Vector Processor Core for Embedded Systems. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 32–33. https://doi.org/10.1109/ASAP61560.2024.00018.

Zhao, Y., Ge, F., Cui, C., Zhou, F., and Wu, N. (2020). A Mapping Method for Convolutional Neural Networks on Network-on-Chip. In *International Conference on Communication Technology (ICCT)*, pages 916–920. https://doi.org/10.1109/ICCT50939.2020.9295883.

Zhu, F., Xu, P., and Zong, J. (2023). Moore's Law: The potential, limits, and breakthroughs. *Applied and Computational Engineering*, 10:307–315. https://doi.org/10.54254/2755-2721/10/20230038.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br