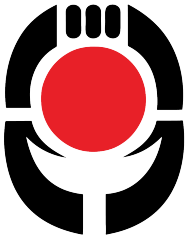


الجامعة المصرية اليابانية للعلوم و التكنولوجيا
エジプト日本科学技術大学
EGYPT-JAPAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Optimizing Machine Learning Kernels Using RISC-V Vector Extension

CSE420: Graduation Project 2 Thesis

Name	University ID
Sally Reda Eldosouky Zeineldeen	120210008
Khaled Ahmed Anis Gad	120210024
Omar Tarek Ahmed Aly	120210099
Yousif Ibrahim Mohammed Masoud	120210281
Ebrahim Osama Shawky Tawfek	120210308
Ali Abdelkader Ali Elsawy	120210366
Mohammed Ashraf Abd Elmonem Moawad	120210376

Mentored by:

Si-VISION
INNOVATIVE IC SOLUTIONS

Supervised by:

Dr. Rami Zewail

Prof. Mohammed S. Sayed

Contents

Abstract **5**

Acknowledgment **6**

List of Abbreviations **7**

1 Introduction **8**

1.1 Motivation 8
1.2 Limitations of Current Solutions 9
1.3 The RISC-V Vector Extension as a Solution 9
1.4 Problem Statement 10
1.5 Project Objectives 10
1.6 Project Contributions 11
1.7 Thesis Organization 12

2 Background & Related Work **13**

2.1 RISC-V Architecture Overview 13
2.2 RISC-V Extensions for Machine Learning 14
2.2.1 Standard Extensions 14
2.3 The RISC-V Vector Extension 14
2.3.1 Architectural Principles 14
2.3.2 Programming with RISC-V Vector Intrinsics 15
2.4 Related Work: The RISC-V landscape for Vector Computing 16

3 Methodology: Architecture & Implementations **19**

3.1 Deep Learning Kernel Selection and Justification 19
3.1.1 Selection Criteria Categories 19
3.1.2 RVV Vectorization Benefits by Category 20
3.2 Development Toolchain 21
3.2.1 RISC-V GNU Toolchain 21
3.2.2 QEMU Emulator 22
3.2.3 ONNX: Open Neural Network Exchange 23
3.3 RISC-V Vectorization Kernels Design 24
3.3.1 Pattern 1: Compute-Bound FMA Operations 24
3.3.2 Pattern 2: Sliding Window Kernels 28
3.3.3 Pattern 3: Pointwise/Elementwise Kernels 35
3.3.4 Pattern 4: Tensor Indexing and Data Movement 38

3.3.5 Pattern 5: Batch Normalization 41
3.3.6 Pattern 6: Post-Processing Kernels - Non-Maximum Suppression . 43
3.4 Functional Verification Results and Discussion 46
3.4.1 Test Setup and Verification Flow 46
3.4.2 Verification Architecture 46
3.4.3 Verification Metrics 46
3.4.4 Verification Thresholds 48
3.4.5 Discrete Functions Correctness Verification Results 48
3.4.6 Models 52

4 Methodology: Performance Validation 58
4.1 Hardware (RTL Cores) 58
4.1.1 Role of RTL Cores in Architectural Research 58
4.1.2 Importance of Cycle-Accurate Simulation 58
4.1.3 Evolution of Core Selection: From Vicuna to Ara 58
4.2 Vicuna RISC-V Vector Coprocessor 59
4.2.1 Overview and Design Motivation 59
4.2.2 Architectural Organization 59
4.2.3 RVV Implementation 60
4.2.4 Execution Model 60
4.2.5 Memory Subsystem 61
4.2.6 RTL Implementation 61
4.2.7 Benchmarking Suitability 61
4.3 Ara Vector Processor 62
4.3.1 Overview and Design Motivation 62
4.3.2 Architectural Organization 62
4.3.3 Vector Execution Model 63
4.3.4 Memory Subsystem and Coherence 63
4.3.5 RTL Implementation and Scalability 64
4.3.6 Benchmarking Suitability 64
4.4 Comparative Analysis and Core Selection Rationale 64
4.4.1 Architectural Trade-offs 64
4.4.2 Rationale for Benchmarking on Ara 64
4.4.3 Summary of Methodology Pivot 65
4.5 Validation Strategy 65
4.5.1 Testbench Structure and Workflow 65
4.5.2 Cycle-Accurate Measurement Logic 66

4.5.3 Hardware Configuration and Leaky ReLU Case Study 67
4.6 Validation Results 68
4.6.1 Performance Overview 69
4.6.2 Compute-Bound FMA Operations 69
4.6.3 Sliding Window & Filters 70
4.6.4 Pointwise & Elementwise Operations 71
4.6.5 Results Discussion 72

5 Open Source Library Architecture **75**
5.1 Repository Structure 75
5.2 Explanation of Repository Contents 75
5.3 Wrappers Overview 76
5.3.1 Intrinsic Wrappers 76
5.3.2 Python Wrappers 76
5.4 Results Verification 77
5.4.1 Correctness Testing 77
5.4.2 Automated Testing 77
5.5 Performance Results 77
5.5.1 Benchmarking Framework 77
5.5.2 Performance Metrics 78
5.6 Wrapper Interfaces 78
5.6.1 Example Function Signature 79

6 Conclusion and Future Work **80**
6.1 Conclusion 80
6.2 Future Work 80
6.2.1 Extension to Additional Workload Domains 80
6.2.2 Deployment on Physical RISC-V Hardware 81
6.2.3 Enhancement of the Python Interface and Abstraction Layer . . . 81
6.2.4 Kernel Optimization and Algorithmic Improvements 81
6.2.5 Expanded Neural Network Model Support 81

7 Code Listings **85**

List of Figures

1 Growth in AI model computational requirements 8
2 RISC-V ecosystem growth 13
3 ONNX Model Computational Graph 24
4 Functional verification flow diagram 47
5 LeNet-5 functional verification results 54
6 Tiny-YOLOv2 object detection verification results 56
7 Architectural overview of Vicuna and Ibex integration 60
8 Top-level block diagram of the Ara system 63
9 Speedup of MatMul across matrix sizes 73

List of Tables

1 Verification threshold criteria for different data types and operation complexities 48
2 Summary of kernel verification results across all implementations 49
3 Conv2D verification results across scales and implementations 50
4 Transposed convolution verification across scales (3×3 kernel, stride=1) 50
5 Matrix multiplication verification (64×64) 51
6 Dense layer verification (batch=1, 128×128) 51
7 Final Comparative Positioning of RTL Cores 64
8 Peak speedup across all evaluated configurations for each RaiVeX Library kernel. 69
9 Matrix Multiplication (MatMul) — Best Implementation per Size 70
10 Dense (Fully Connected) Layer — Best Implementation per Size 70
11 Convolution (Conv) — Best Implementation per Configuration 70
12 MaxPool — Best Vector Implementation (M8) per Configuration 71
13 ReLU and Leaky ReLU — Best Vector Implementation (M8) 71
14 Batch Normalization — Best Vector Implementation (M8) 71
15 Additive Kernels — Best Implementation per Configuration 72
16 Kernel Variants and Implementation Availability 78

Abstract

The proliferation of machine learning (ML) workloads across edge devices and embedded systems has intensified the demand for energy-efficient, high-performance computing architectures beyond conventional GPU-dominated paradigms. RISC-V, an open-source instruction set architecture, has emerged as a promising alternative, with its Vector Extension (RVV) offering scalable data-level parallelism suitable for computationally intensive ML operations. However, the literature reveals a significant gap in comprehensive, production-ready implementations of vectorized ML kernels optimized specifically for RVV 1.0, along with limited empirical benchmarking across diverse kernel categories on actual vector processor implementations. This study addresses these gaps by developing RaiVeX Library, a comprehensive collection of RISC-V Vector-accelerated kernels targeting deep learning and scientific computing workloads. The research objectives include implementing optimized vectorized versions of fundamental ML primitives—encompassing matrix multiplication, convolution, transposed convolution, dense (fully connected) layers, batch normalization, activation functions (ReLU, Leaky ReLU, Softmax), max pooling, bias addition, tensor arithmetic, and ONNX-style indexing operations (Gather, Scatter, Non-Maximum Suppression)—while systematically evaluating performance across different LMUL (Length Multiplier) configurations (M1, M2, M4, M8). The methodology employs C++ implementations utilizing RVV 1.0 intrinsics, with functional correctness validated against ONNX golden references using QEMU emulation, and performance benchmarked on the Ara vector co-processor, an open-source implementation of a scalable RISC-V vector unit. The library architecture emphasizes modularity through reusable low-level vector wrappers for loads, stores, reductions, multiply-accumulate, and mask operations, enabling clean and maintainable kernel implementations. Python bindings via shared libraries further enhance accessibility for rapid experimentation. Performance evaluation on the Ara co-processor demonstrates substantial speedups ranging from 4× to over 70× compared to scalar baseline implementations. Notably, matrix multiplication achieves up to 70.27× improvement using unrolled vectorization strategies, while activation and normalization kernels such as Leaky ReLU, batch normalization, and max pooling achieve speedups between 20× and 36×. Compute-intensive linear operators and pointwise arithmetic kernels consistently demonstrate significant acceleration across all tested configurations. The library's practical applicability is validated through complete end-to-end inference implementations of LeNet-5 for digit classification and Tiny-YOLOv2 for object detection, demonstrating seamless integration of vectorized kernels into real neural network pipelines. This work establishes that the RISC-V Vector Extension, when properly optimized, provides a viable, high-performance, and energy-efficient alternative for accelerating ML inference on resource-constrained embedded platforms.

Keywords: RISC-V Vector Extension (RVV), Machine Learning Kernel Acceleration, Vector Co-processor, Deep Learning Inference, High-Performance Embedded Computing

Acknowledgment

We would like to express our sincere gratitude to all those who contributed to the successful completion of this thesis. First and foremost, we extend our deepest appreciation to our academic supervisors, **Dr. Rami Zewail** and **Prof. Mohammed S. Sayed**, from the School of Electronics, Communications and Computer Engineering (ECCE) at Egypt-Japan University of Science and Technology (E-JUST), for their invaluable guidance, constructive feedback, and continuous support throughout this research. Their expertise and mentorship were instrumental in shaping the direction of this work. We are also profoundly grateful to **Si-Vision** company for providing the opportunity to conduct this project under their mentorship program, and we extend special thanks to **Eng. Youssef M. Fathy** for his dedicated supervision, technical insights, and unwavering encouragement. Finally, we acknowledge the Faculty of Engineering at E-JUST for providing the academic environment and resources that made this research possible.

List of Abbreviations

Abbreviation	Description	Abbreviation	Description
AI	Artificial Intelligence	AVL	Application Vector Length
AVX	Advanced Vector Extensions	DNN	Deep Neural Network
DSP	Digital Signal Processing	FLOPS	Floating-Point Operations Per Second
FMA	Fused Multiply-Add	FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection	GEMM	General Matrix Multiply
IoU	Intersection over Union	ISA	Instruction Set Architecture
LMUL	Length Multiplier	LUT	Look-Up Table
M1	LMUL = 1	M2	LMUL = 2
M4	LMUL = 4	M8	LMUL = 8
ONNX	Open Neural Network Exchange	OpenCL	Open Computing Language
ReLU	Rectified Linear Unit	RTL	Register Transfer Level
RVV	RISC-V Vector Extension	SEW	Selected Element Width
SIMD	Single Instruction, Multiple Data	SNR	Signal-to-Noise Ratio
VALU	Vector Arithmetic Logic Unit	VELEM	Vector Element Unit
VFPU	Vector Floating-Point Unit	VLA	Vector-Length Agnostic
VLEN	Vector Length	VLSU	Vector Load/Store Unit
VMUL	Vector Multiplier	VRF	Vector Register File
WCET	Worst-Case Execution Time	XIF	eXtension Interface

1 Introduction

1.1 Motivation

The rapid evolution of artificial intelligence (AI) and digital signal processing (DSP) applications has fundamentally transformed the computational requirements of modern computing systems. AI workloads, particularly deep learning models, demand massive parallel computation for operations such as matrix multiplication, convolution, and tensor operations. Similarly, DSP applications require intensive mathematical operations including filtering, Fourier transforms, and correlation analysis. These computational patterns share a common characteristic: they involve highly parallel, data-intensive operations that can benefit significantly from vectorized execution.

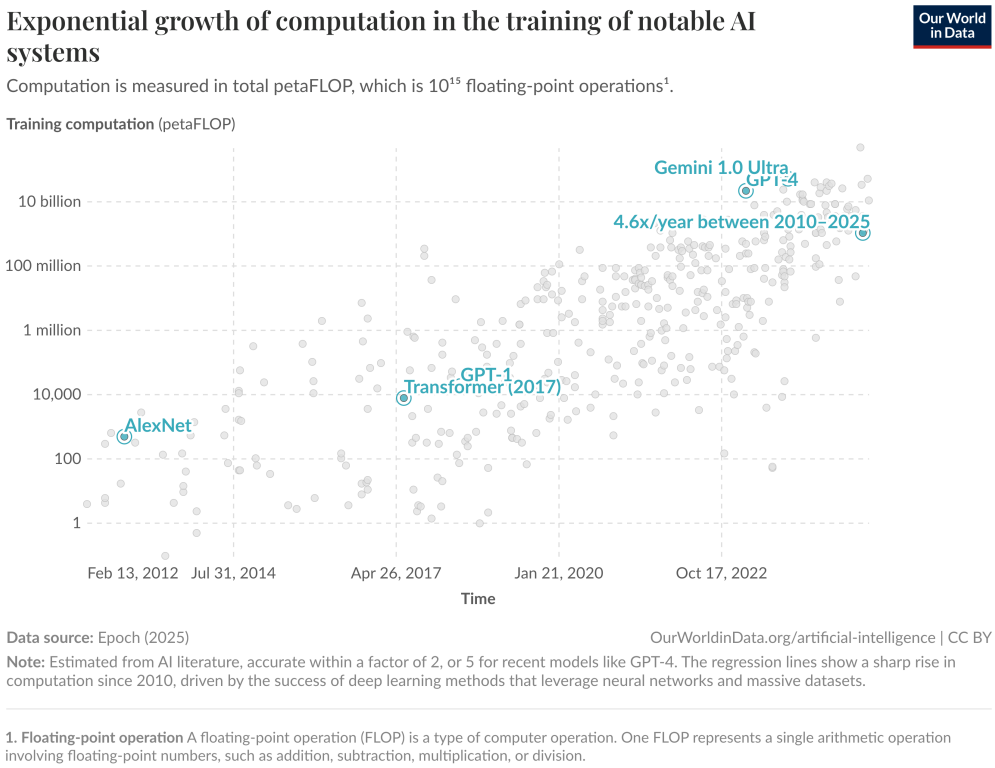


Figure 1: Exponential growth in AI model computational requirements over time, showing the dramatic increase in FLOPs required for training state-of-the-art models.[1]

Traditional scalar processors, designed primarily for sequential instruction execution, face significant challenges when processing these data-parallel workloads. The von Neumann architecture, with its single instruction stream operating on individual data elements, creates a fundamental bottleneck for AI and DSP applications. For example, a typical convolution operation in a convolutional neural network (CNN) involves millions of multiply-accumulate operations that could theoretically be executed in parallel, but scalar processors must execute them sequentially, leading to substantial performance degradation.

The performance gap becomes even more pronounced when considering the memory bandwidth requirements of AI and DSP applications. These workloads often involve large datasets that exceed the capacity of processor caches, leading to frequent memory accesses. The arithmetic intensity—the ratio of computation to memory access—of many AI and DSP kernels is relatively

low, meaning that processors spend significant time waiting for data rather than performing useful computation. This phenomenon, commonly referred to as the "memory wall," represents a fundamental challenge for data-intensive computing.

1.2 Limitations of Current Solutions

Current solutions to these challenges have primarily relied on specialized hardware architectures and proprietary vector processing extensions. Graphics Processing Units (GPUs) have become the de facto standard for AI acceleration due to their thousands of parallel cores designed for data-parallel computation. However, GPUs present several limitations for AI and DSP applications:

- **Power consumption**: GPUs consume significant power, making them unsuitable for edge computing and mobile applications where energy efficiency is paramount.
- **Programming complexity**: The GPU programming model, while powerful, requires specialized knowledge (Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL)) and often results in complex code that is difficult to optimize and maintain.
- **Integration challenges**: GPUs are discrete components requiring separate memory spaces and Peripheral Component Interconnect Express (PCIe) communication, introducing latency and bandwidth constraints for certain workloads.

Proprietary vector processing solutions, such as Intel's Advanced Vector Extensions (AVX) and ARM's NEON, provide another approach to accelerating data-parallel workloads. These extensions add vector processing capabilities to traditional CPU architectures, allowing multiple data elements to be processed with a single instruction (SIMD). However, these solutions have significant drawbacks that limit their effectiveness and adoption:

1. **Vendor lock-in**: Proprietary vector extensions create situations where software optimized for one vendor's vector instructions cannot efficiently run on competitors' hardware, fragmenting the software ecosystem.
2. **Fixed vector widths**: These extensions typically use fixed vector widths (e.g., 128-bit for NEON, 256-bit or 512-bit for AVX), meaning that software must be written for specific vector lengths and may not efficiently utilize processors with different vector capabilities.
3. **Licensing costs**: Licensing costs and restrictions associated with proprietary architectures can be prohibitive, particularly for smaller companies and research institutions developing specialized AI and DSP applications.
4. **Limited extensibility**: The closed nature of proprietary Instruction Set Architectures (ISAs) makes it difficult for researchers and developers to experiment with custom instructions or architectural modifications.

1.3 The RISC-V Vector Extension as a Solution

RISC-V Vector Extensions (RVV) emerged as a promising solution to address these challenges by providing an open-source, royalty-free vector processing architecture specifically designed for data-parallel computation [3]. Unlike proprietary alternatives, RVV is developed through

an open, collaborative process that ensures the architecture meets the diverse needs of the computing community.

The most distinctive feature of RVV is its **vector-length agnostic (VLA)** programming model, which represents a fundamental departure from traditional fixed-width Single Instruction, Multiple Data (SIMD) approaches. In conventional vector processing, software must be written for specific vector widths, and different code paths are often required to support processors with different vector capabilities. RVV's vector-length agnostic model allows the same code to run efficiently across processors with different vector lengths, from embedded systems with short vectors to high-performance computing systems with very long vectors.

This flexibility is particularly valuable for AI and DSP applications, which span a wide range of computing environments with different performance and power requirements. An AI inference algorithm written using RVV can run efficiently on both a power-constrained edge device with 128-bit vectors and a high-performance server processor with 2048-bit vectors, without requiring code modifications or recompilation. The ratification of RVV Version 1.0 in late 2021 provided a crucial guarantee of stability, signaling to the industry that the architecture was mature and ready for widespread adoption.

1.4 Problem Statement

Despite the architectural advantages of the RISC-V Vector Extension, developing optimized kernels for RVV remains a challenging task. New developers often face two major obstacles:

1. **Lack of standardized workflows**: There is currently no unified methodology for vector kernel development that encompasses design, verification, and performance evaluation in a cohesive framework.
2. **Limited guidance on verification and evaluation**: While performance measurement is essential to demonstrate the benefits of vectorization, ensuring functional correctness is equally critical, especially when kernels are applied in sensitive domains such as artificial intelligence or embedded systems.

Furthermore, the absence of widely available RVV-capable silicon necessitates reliance on emulation and Register Transfer Level (RTL) simulation environments for development and validation. This creates additional complexity in establishing reproducible benchmarking methodologies that can provide meaningful performance insights.

1.5 Project Objectives

This thesis investigates the role and potential impact of RISC-V Vector Extensions in accelerating AI and DSP applications. The primary objectives of this research are:

1. **Develop a systematic framework** for the design, verification, and performance evaluation of RISC-V vector kernels that integrates open-source tools into a reproducible workflow.
2. **Implement a comprehensive library of optimized RVV kernels** categorized by their computational patterns:
    - **Compute-Intensive Linear Operators:** Matrix multiplication (*matmul*), fully connected layers (*dense*), and both standard and transposed convolutions.

- **Pointwise Activation and Arithmetic:** Rectified Linear Unit ($ReLU$), Leaky ReLU, bias addition, and element-wise tensor addition.
- **Statistical and Normalization Kernels:** Batch normalization and the Softmax probability function.
- **Spatial Reduction and Indexing:** Max pooling, as well as ONNX-style data movement operations including Gather, GatherElements, and ScatterElements.
- **Decision Kernels:** Post-processing operations such as Non-Maximum Suppression (NMS).

3. **Establish functional verification methodologies** using ONNX (Open Neural Network Exchange) [4] as a golden reference framework, ensuring correctness through quantitative metrics such as Signal-to-Noise Ratio (SNR) and Maximum Absolute Error (MaxAbs).

4. **Conduct cycle-accurate performance evaluation** using RTL simulation with the Vicuna [5] and Ara [6] vector coprocessors, quantifying the speedup achieved through vectorization over scalar implementations.

5. **Analyze the trade-offs** between different architectural approaches (high-performance vs. embedded) and provide insights into optimal kernel design strategies for various deployment scenarios.

1.6 Project Contributions

This thesis makes the following contributions to the field of RISC-V vector processing for machine learning and DSP applications:

1. **A reproducible three-step framework** integrating kernel design using RVV C-intrinsics, functional verification against ONNX golden references, and cycle-accurate performance analysis using RTL simulation with Verilator. This framework provides a systematic methodology that can be adopted by other researchers and developers.

2. **A library of optimized RVV kernels (RaiVeX Library)** implementing a wide array of operations for neural network inference and signal processing. The library demonstrates efficient use of advanced RVV features, including strip-mining loops, vector-length agnostic programming, Length Multiplier (LMUL) configuration, and masked operations.

3. **Performance characterization on a high-performance RTL platform:** Implementation and evaluation were conducted using **Ara**, a 64-bit vector coprocessor targeting application-class workloads. The study explores the impact of Ara's microarchitecture, including its configurable lane counts and full floating-point support, on kernel efficiency.

4. **Quantitative analysis** demonstrating significant performance gains through vectorization. Experimental results on the Ara co-processor show substantial speedups ranging from 4× to over 70× compared to scalar baseline implementations, highlighting the efficiency of the RVV ISA for data-parallel workloads.

5. **High-level language integration through Python wrappers:** The development of a bridging layer using the `ctypes` foreign function interface, allowing the execution of optimized C++ vector kernels directly from a Python environment. This enables high-level algorithmic development and testing while maintaining the raw performance and hardware-level control of C++ implementations.

1.7 Thesis Organization

The remainder of this thesis is organized as follows to guide the reader from foundational concepts to our specific architectural contributions and empirical results:

Chapter 2: Background and Related Work establishes the theoretical foundation of the RISC-V ISA and its modular extension system. It details the RISC-V Vector (RVV) Extension, focusing on Vector-Length Agnosticism (VLA) and the intrinsic-based programming model. Finally, it reviews the current state of RVV research, covering virtual prototyping, reliability analysis, and existing hardware accelerators.

Chapter 3: Methodology: Architecture & Implementations presents the framework for kernel development and functional validation. It covers the toolchain setup, including the RISC-V GNU toolchain [7], QEMU emulator [8], and ONNX framework [4]. The chapter details the design of vectorized patterns for operations ranging from compute-bound FMAs to post-processing Non-Maximum Suppression (NMS), concluding with a comprehensive verification of these kernels integrated into full-scale LeNet-5 [9] and Tiny-YOLOv2 [10] inference pipelines.

Chapter 4: Methodology: Performance Validation shifts the focus to hardware-level evaluation using cycle-accurate RTL simulation with Verilator [11]. It justifies the selection of the Ara vector coprocessor [6] over the Vicuna core [5] and describes the benchmarking environment. The chapter provides a deep dive into performance results, analyzing how arithmetic intensity, register grouping (LMUL), and memory bandwidth impact speedups across compute-bound, sliding-window, and pointwise kernels.

Chapter 5: Open Source Library Architecture describes the design and implementation of the RaiVeX Library software interface. It outlines the three-layer modular structure—Backend, Wrapper, and API—that bridges the gap between C-based RVV intrinsics and Python. This chapter details the available kernel wrappers, specialized configurations for convolutions, and provides guidelines for selecting optimal LMUL parameters to maximize hardware utilization.

Chapter 6: Conclusion and Future Work synthesizes the key findings of this research, reflecting on the effectiveness of the RISC-V Vector Extension for accelerating machine learning workloads. It summarizes the contributions made through the systematic framework and the RaiVeX Library. Finally, it outlines promising directions for future research, including expansion into DSP domains, validation on physical hardware, and the integration of quantization support for edge deployment.

Chapter 7: Code Listings provides supporting materials, including sample code implementations and specialized kernel configurations used throughout the validation process.

Supporting materials are provided in the final sections, including a comprehensive reference list and a link to the official GitHub repository containing the full source code for the RaiVeX Library.

2 Background & Related Work

2.1 RISC-V Architecture Overview

RISC-V (pronounced "risk-five") is an open-source ISA that has revolutionized processor design by providing a free, extensible alternative to proprietary architectures. Developed at the University of California, Berkeley, beginning in 2010 [3], RISC-V was created to address fundamental limitations in the processor industry, particularly the dominance of proprietary ISAs that created barriers to innovation and increased costs for processor development.

The development of RISC-V was motivated by several critical issues in the computing industry that had become increasingly problematic for AI and DSP applications. Traditional proprietary ISAs, such as x86 and ARM, require expensive licensing agreements that can be prohibitive for companies developing specialized processors for AI and DSP workloads. These licensing costs are particularly burdensome for startups and research institutions that want to experiment with novel architectural approaches.

RISC-V addresses these challenges through several fundamental design principles that make it particularly well-suited for AI and DSP applications:

Open Source Philosophy: RISC-V specifications are freely available under Creative Commons licenses, and anyone can implement, modify, or extend RISC-V processors without paying royalties or obtaining permission. This openness eliminates one of the major barriers to innovation in processor design and enables a diverse ecosystem of implementations tailored for specific applications.

Modular Architecture: RISC-V follows a modular design philosophy where a minimal base integer instruction set is supplemented by optional standard extensions. This modularity is particularly valuable for AI and DSP processors, which can include only the extensions needed for their specific applications, reducing implementation complexity and cost.

Scalability Across Application Domains: RISC-V supports multiple data widths (32-bit, 64-bit, and 128-bit) and can scale from microcontrollers to high-performance processors. This scalability is crucial for AI and DSP applications, which span a wide range of computing environments from embedded edge devices to high-performance computing clusters.

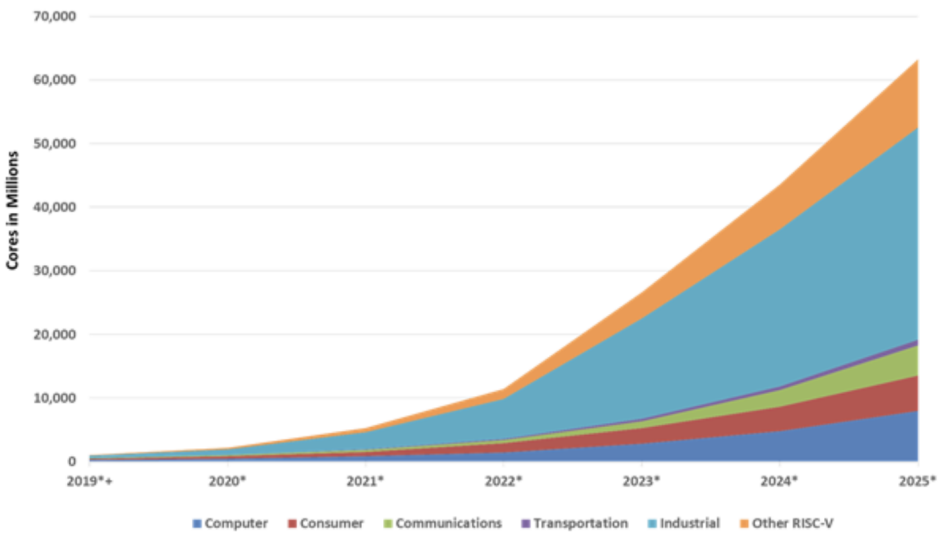


Figure 2: RISC-V ecosystem growth. Source: Semico Research Corp. [2]

2.2 RISC-V Extensions for Machine Learning

The extensible nature of RISC-V is fundamental to its success in AI and DSP applications, allowing specialized functionality to be added to the base instruction set through a well-defined extension mechanism. This extensibility enables processors to be tailored for specific application domains while maintaining compatibility with the broader RISC-V ecosystem.

2.2.1 Standard Extensions

The modular nature of RISC-V allows the base integer ISA to be supplemented with standard extensions to meet the requirements of specific application domains:

- **M Extension (Integer Multiplication and Division):** Adds hardware support for integer multiplication and division. This is essential for quantized neural networks and DSP algorithms where integer arithmetic is used to minimize power consumption.
- **F Extension (Single-Precision Floating-Point):** Provides IEEE 754 32-bit floating-point arithmetic. It introduces Fused Multiply-Add (FMA) instructions, which are critical for the multiply-accumulate (MAC) patterns dominant in convolution operations.
- **D Extension (Double-Precision Floating-Point):** Adds 64-bit floating-point support, necessary for AI training phases requiring high numerical stability and DSP applications with extended dynamic range requirements.
- **V Extension (Vector Operations):** The primary focus of this research, providing comprehensive support for data-parallel operations through a vector-length agnostic architecture. It enables high-throughput execution for the tensor and matrix operations found in modern AI workloads.

2.3 The RISC-V Vector Extension

The RISC-V Vector (RVV) Extension stands out as one of the most consequential developments for modern computing workloads. Unlike traditional Single Instruction, Multiple Data (SIMD) architectures that operate on fixed-size registers, RVV was designed with a philosophy of flexibility, scalability, and efficiency achieved through novel architectural concepts.

2.3.1 Architectural Principles

Vector Registers and Configuration: The V extension introduces 32 vector registers (`v0`–`v31`). The core architectural parameter is VLEN (Vector Length), which specifies the length of these registers in bits. VLEN is an implementation-defined choice, not fixed by the specification, and can range from small values (e.g., 128 bits) for embedded systems to very large values (e.g., 4096 bits or more) for supercomputers. Another key parameter is ELEN (Element Length), which is the maximum size of a single data element that can be processed.

Vector Control and Status Registers (CSRs): The power and flexibility of the V extension are managed through key Control and Status Registers:

- `vtype`: Configures the vector unit for subsequent operations by setting the selected element width (`vsew`), vector length multiplier (`vlmul`) for register grouping, and behavior controls for tail and masked-out elements (`vta`/`vma`).

- `vl`: Set by the programmer to specify the number of elements to process in upcoming vector instructions, ranging from 0 to a hardware-dependent maximum.
- `vlenb`: A read-only register that reports the hardware's vector register length (VLEN) in bytes.

Vector-Length Agnostic (VLA) Execution: The combination of the `vsetvli` instruction and the `vl` register enables RVV's most powerful feature: Vector-Length Agnosticism. Unlike fixed-length SIMD (e.g., Intel's AVX or ARM's NEON), where code is written for a specific vector width, VLA code is portable across any hardware implementation, regardless of its VLEN.

The typical execution flow follows a "strip-mining" pattern:

1. A programmer has a large array of N elements to process
2. The code enters a loop and calls `vsetvli`, passing the remaining number of elements
3. The hardware automatically sets `vl` to the minimum of the requested number and the maximum it can physically handle (VMAX), configuring `vtype` appropriately
4. Subsequent vector instructions operate on `vl` elements
5. The loop continues, processing chunks of data until all N elements are complete

This approach means a single compiled binary can run with optimal efficiency on both a low-power microcontroller with VLEN=128 and a high-performance compute node with VLEN=4096, without requiring recompilation or code modification.

Rich and Orthogonal Instruction Set: The V extension provides a comprehensive set of instructions orthogonal to data types, allowing the same opcodes to work on integers and floats of different widths as configured by `vtype`. Key instruction categories include:

- **Vector Arithmetic:** Integer, fixed-point, and floating-point operations
- **Vector Memory Access:** Unit-stride (contiguous), strided (every Nth element), and indexed scatter/gather operations
- **Vector Permutation:** Instructions for shuffling data within and between vector registers
- **Masking and Predication:** Most vector instructions can be masked, performing operations only on elements where a corresponding bit in mask register `v0` is set
- **Reduction Operations:** Built-in support for combining all vector elements into a scalar result (sum, min, max, logical reductions)

2.3.2 Programming with RISC-V Vector Intrinsics

While assembly language provides direct control over vector instructions, RISC-V vector intrinsics offer a more maintainable and portable approach to vectorized programming. Intrinsics are C/C++ functions that map directly to vector instructions, providing the performance benefits of assembly with the readability and toolchain integration of high-level languages.

Advantages of Intrinsics:

- **Compiler Integration:** Intrinsics work seamlessly with standard C/C++ compilers, enabling better optimization, register allocation, and instruction scheduling

- **Type Safety:** Unlike inline assembly, intrinsics are type-checked by the compiler, catching errors at compile time
- **Portability:** Code using intrinsics can be more easily ported across different RISC-V implementations
- **Maintainability:** Intrinsic-based code is more readable and easier to debug than raw assembly

Common Intrinsic Patterns:

Setting Vector Length:

```
size_t vl = __riscv_vsetvl_e32m1(n);
```

Vector Load/Store:

```
vfloat32m1_t v = __riscv_vle32_v_f32m1(ptr, vl);
__riscv_vse32_v_f32m1(ptr, v, vl);
```

Arithmetic Operations:

```
v_result = __riscv_vfadd_vv_f32m1(v1, v2, vl);
v_result = __riscv_vfmul_vf_f32m1(v, scalar, vl);
```

Fused Multiply-Accumulate:

```
v_acc = __riscv_vfmacc_vv_f32m1(v_acc, v1, v2, vl);
```

Reduction Operations:

```
vfloat32m1_t v_sum = __riscv_vfredsum_vs_f32m1_f32m1(v, v_zero,
    vl);
float sum = __riscv_vfmv_f_s_f32m1_f32(v_sum);
```

2.4 Related Work: The RISC-V landscape for Vector Computing

System-Level Infrastructure for RISC-V Vector Design The following papers focus on the modeling, simulation, verification, and reliability analysis infrastructure required to design and evaluate RISC-V vector-enabled systems before and during hardware implementation.

Herdt et al.: Extensible and Configurable RISC-V Virtual Prototype

Herdt et al. [12] introduces the first open-source, extensible **Virtual Prototype (VP)** based on the RISC-V architecture, implemented using standard-compliant SystemC and Transaction-Level Modeling **(TLM-2.0)**.. The primary scientific contribution is the creation of a middle-ground simulation environment that bridges the gap between high-speed but inflexible **Instruction Set Simulators (ISSs)** and highly accurate but extremely slow **Register Transfer Level (RTL)** models.

Schlägl et al.: Unlocking Vector Processing for System-Level Evaluation

Schlägl et al. [13] present the first open-source SystemC TLM-based virtual prototype with full support for the **RISC-V Vector Extension (RVV) version 1.0**. The work addresses the lack of early-stage modeling tools capable of supporting the more than 600 vector instructions introduced by the standard.

Quiroga et al.: Reusable Verification Environment for Vector Accelerators

Quiroga et al. [14] propose a **Universal Verification Methodology (UVM)-based, interface-agnostic verification environment** for RISC-V vector accelerators, enabling reuse across heterogeneous projects such as EPI and eProcessor.

Imianosky et al.: Reliability and Performance of Vector Multiplication Units

Imianosky et al. [15] analyze performance–reliability trade-offs in fault-tolerant RISC-V systems by extending the **HARV-SoC** with **Zve32x** vector multiplication support. Fault-injection experiments show that vector acceleration achieves up to **28.69**× speedup while often improving overall reliability due to reduced execution time.

Compiler, Application, and System-Level Performance Evaluation This group of papers evaluates how RISC-V vector capabilities translate into real-world performance across compilers, applications, and high-performance computing systems.

Al-Assir et al.: Arrow RISC-V Vector Accelerator

Al-Assir et al. [16] present **Arrow**, a RISC-V vector accelerator optimized for machine learning inference. By focusing on a configurable architecture that supports different vector lengths and data types, Arrow demonstrates significant energy efficiency and throughput gains for edge-based ML tasks compared to scalar implementations.

Wang et al.: SPEED — Scalable Multi-Precision DNN Processor

Wang et al. propose **SPEED**, a scalable multi-precision Deep Neural Network (DNN) processor designed to overcome limitations of baseline RVV implementations. SPEED integrates a highly parameterized **Systolic Array Unit (SAU)** within each vector lane. Synthesized in 28,nm technology, SPEED achieves area-efficiency improvements of **2.04**× for 16-bit and **1.63**× for 8-bit operations compared to the Ara processor [17].

Carpentieri et al.: Performance Analysis of Autovectorization on RVV Boards

Carpentieri et al. [18] present an empirical evaluation of **compiler-driven autovectorization** on real RISC-V hardware, comparing GCC and LLVM across Test Suite for Vectorizing Compilers (TSVC) benchmarks and real-world applications. The study shows that careful tuning of **LMUL** can yield speedups of up to **3**×.

Volokitin et al.: Memory-Bound Kernels on RISC-V CPUs

Volokitin et al. [19] evaluate memory-bound kernels on RISC-V platforms and show that classical optimizations such as cache blocking and unit-stride access transfer effectively from x86 and ARM architectures.

Brown: Evaluating the Sophon SG2044 for High Performance Computing

Brown [20] evaluates RISC-V's suitability for **High Performance Computing** through an analysis of the **Sophon SG2044**. With mainline RVV 1.0 support and a 32-channel DDR5 memory subsystem, the SG2044 achieves up to a **4.91**× speedup over its predecessor.

3 Methodology: Architecture & Implementations

3.1 Deep Learning Kernel Selection and Justification

The development of the RaiVeX Library begins with careful selection of computational kernels that represent fundamental building blocks for machine learning inference and digital signal processing applications. The kernels chosen for the RaiVeX Library are grouped into six categories. Each category reflects a selection criterion emphasizing compute intensity, data-parallel structure, predictable memory access, composability, and representativeness for typical inference pipelines.

3.1.1 Selection Criteria Categories

The six categories of kernels selected for the RaiVeX Library are as follows:

1. **Compute-intensive FMA Operations**: These kernels operate on dense arrays with regular, unit-stride memory access and nested loop structure. Implementations typically use blocking/tiling to expose long inner loops and maximize register reuse, while the inner-most operations are sequences of fused multiply-adds. Such structure yields high arithmetic intensity and sustained use of vector functional units.

   Kernels in this category include: matrix multiplication and dense (fully connected layer).

2. **Sliding-window kernels**: These kernels compute outputs by applying a small kernel over local spatial neighborhoods of an input tensor (e.g., the standard convolution inner loop that multiplies a $K \times Kh \times Kw$ patch by kernel weights and accumulates results). Implementations may transform the problem to an implicit matrix multiply (im2col) or keep the sliding-window loop nest; both approaches aim to increase contiguous inner work and exploit reuse of input pixels across neighboring windows.

   Kernels in this category include: convolution, transposed convolution, and maxpooling.

3. **Pointwise activations and elementwise arithmetic**: Pointwise kernels apply the same small computation independently to each tensor element (for example, max(0,x) for ReLU or an affine bias+scale followed by a nonlinear function). Because dependencies are per-element, these kernels are typically implemented as single-pass unit-stride operations that can be fused with neighboring operators to reduce memory traffic.

   Kernels in this category include: ReLU, Leaky ReLU, Tensor Add, and Bias Add.

4. **Tensor indexing and data movement**: These kernels perform layout transformations, indexed loads/stores, or sparse data movements. They are often memory-bound and may exhibit irregular access; implementations therefore rely on batched indexed operations, in-register permutation, and careful prefetching or blocking to amortize address computation and reduce cache misses.

   Kernels in this category include: gather, gather_elements, scatter_elements.

5. **Statistical and normalization layers**: Normalization kernels typically have a two-phase structure: a reduction phase that computes statistics (sum, mean, variance, or max) over one or more axes, followed by an elementwise normalization and optional affine rescale. Softmax similarly computes a max and exponentials followed by a sum reduction and normalization. Implementations target numerically stable reductions and exploit partial in-register reductions to minimize memory traffic.

   Kernels in this category include: Batch Normalization.

6. **Postprocessing (NMS)**: Postprocessing kernels perform selection, suppression, or decoding operations with irregular control flow and variable-sized outputs (for example, non-maximum suppression for object detection). Implementations often separate dense arithmetic (Intersection over Union (IoU), score comparisons) — which can be vectorized — from the final selection and compacting phases which may require scalar coordination or masked writes.

   Kernels in this category include: non-maximum suppression (NMS).

3.1.2 RVV Vectorization Benefits by Category

This subsection describes how each category maps to the RISC-V Vector (RVV) architectural features and the expected performance advantages. The RISC-V Vector Extension provides length-agnostic vector execution, flexible element widths, predicated operations for handling tails, and strided/indexed memory instructions that are central to the mappings below (RISC-V Vector Extension specification).

1. **Compute-intensive FMA Operations**: General Matrix Multiply (GEMM) and dot-product kernels have high arithmetic intensity and regular, unit-stride access to dense matrices. RVV benefits include long, chained vector multiply-accumulate sequences using floating-point vector instructions, efficient use of register grouping (LMUL) to increase effective vector width, and hardware-supported reductions for dot-products. Vectorized blocking and inner-loop unrolling map well to RVV's vsetvl-driven VL tuning, enabling near-peak use of available vector functional units.

2. **Sliding-window kernels**: Convolutions can be transformed to matrix-multiply-like inner loops (im2col) or implemented directly with sliding-window inner loops. RVV's strided and indexed loads allow efficient loading of kernel windows without expensive scalar gathers; predicated vector operations handle border and padding conditions without branch mispredictions. Data reuse across windows benefits from vector registers holding multiple channel or kernel elements, reducing memory traffic and improving arithmetic-to-memory ratio.

3. **Pointwise activations and elementwise arithmetic**: These operations are embarrassingly parallel with unit-stride access, making them ideal for wide vector lanes. RVV executes elementwise transforms (e.g., max for ReLU) across many elements per instruction, using masks to implement conditional activations (e.g., Leaky ReLU). Low overhead for lane-tail handling preserves correctness for arbitrary tensor sizes.

4. **Tensor indexing and data movement**: Gather/scatter and permutation kernels rely on RVV's indexed load/store capabilities. While irregular accesses are generally memory-bandwidth bound, RVV's indexed operations can process multiple index entries per vector instruction, amortizing the cost of address calculation. When indices are contiguous or have simple strides, strided loads provide near-unit-stride efficiency. Vector-based layout transforms also allow in-register rearrangement to avoid extra memory passes.

5. **Batch Normalization**: The Batch normalization kernel is a memory-bound, unit-stride pass that applies the same small parameter set across spatial and batch dimensions. RVV accelerates this pattern by loading per-channel parameters into vector registers or scalar registers and applying vectorized multiply-add sequences across long contiguous segments of data; masked operations handle tails and fused implementations can combine scale+shift with an activation to reduce memory traffic.

6. **Non-maximum suppression (NMS)**: NMS iteratively selects high-scoring bounding boxes and suppresses overlapping boxes whose IoU exceeds a threshold. The computationally intensive portion is the pairwise IoU and score-thresholding work which can be expressed as many parallel box-pair comparisons. RVV benefits by computing IoU numerators/denominators and threshold predicates across many candidate boxes per vector instruction, using masks to mark suppressed elements. The remaining selection/compaction step typically requires scalar coordination or masked compress-store operations, but the heavy arithmetic and comparisons are offloaded to vector units for measurable speedups.

In summary, the six-category taxonomy guides kernel prioritization by pairing algorithmic characteristics with RVV capabilities: high arithmetic intensity and unit-stride access maximize compute throughput; predictable strided or indexed access benefits from RVV memory modes; and reductions, predication, and flexible vector lengths enable efficient handling of normalization and irregular tails.

3.2 Development Toolchain

3.2.1 RISC-V GNU Toolchain

The RISC-V GNU Toolchain provides the foundational software infrastructure required to develop, compile, link, and debug programs targeting the RISC-V instruction set architecture. In this project, the toolchain serves as the primary means of compiling and validating RISC-V Vector Extension (RVV)–based kernel implementations prior to functional verification and RTL-level evaluation.

The toolchain was selected due to its mature support for the RISC-V ecosystem, active community maintenance, and integration of RVV 1.0 intrinsics within the GNU Compiler Collection (GCC). This enables the development of vectorized kernels using high-level C/C++ code while maintaining explicit control over generated vector instructions.

Toolchain Components **Compiler (`riscv64-unknown-linux-gnu-g++`):** The GCC compiler translates C/C++ source code into RISC-V assembly and object code. In this work, vectorization is primarily achieved through explicit RVV intrinsics rather than relying on automatic vectorization. This approach ensures deterministic mapping between source-level kernel implementations and the generated vector instructions, which is essential for functional verification and architectural analysis.

The following architecture-specific compilation flags are used throughout the project:

- `-march=rv64gcv`: Enables the RV64GCV ISA, including the RISC-V Vector Extension
- `-mabi=lp64d`: Specifies the 64-bit ABI with double-precision floating-point support
- `-O2`, `-O3`: Enable standard compiler optimizations without altering the explicit vectorization strategy

Assembler (`riscv64-unknown-linux-gnu-as`): Converts RISC-V assembly code into object files, supporting both scalar and RVV instruction encodings. It is used indirectly through the compiler toolchain and for inspecting compiler-generated assembly during kernel analysis.

Linker (`riscv64-unknown-linux-gnu-ld`): Links object files and resolves external symbols to produce final executable binaries suitable for execution under emulation or simulation.

Debugger (`riscv64-unknown-linux-gnu-gdb`): Provides source-level debugging capabilities for RISC-V binaries executed under QEMU, including inspection of scalar and vector registers. This functionality is essential for validating intermediate kernel states during development.

Binary Utilities: Tools such as `objdump`, `readelf`, and `nm` are used to inspect binaries, verify RVV instruction generation, and analyze symbol information.

Runtime Libraries: The GNU C Library (`glibc`) is used for Linux-based execution under QEMU, providing standard runtime support required for user-mode and system-mode emulation. Bare-metal runtime support is not a primary focus of this work.

Collectively, the RISC-V GNU Toolchain constitutes the software development layer of the project, enabling portable kernel implementation and serving as the entry point for subsequent functional and RTL-level verification.

3.2.2 QEMU Emulator

QEMU (Quick Emulator) is employed as the primary functional execution environment for RISC-V binaries in this project. QEMU provides architecturally correct emulation of the RISC-V ISA, including the RISC-V Vector Extension, and is used exclusively for functional correctness validation and software-level testing. Performance results reported in this thesis are derived from RTL simulation rather than QEMU execution.

QEMU was selected over alternative simulators due to its system-level completeness, integration with standard debugging tools, and practical support for executing complex software stacks.

Rationale for Using QEMU While Spike is the official RISC-V ISA simulator, QEMU offers several advantages aligned with the goals of this project. QEMU supports full Linux execution environments, enabling realistic testing of vectorized kernels in the presence of system calls, memory management, and runtime libraries. Additionally, QEMU integrates seamlessly with GNU Debugger (GDB), allowing interactive debugging and inspection of vector register state. QEMU employs dynamic binary translation, which significantly reduces simulation time compared to purely interpretive simulators. This improvement in execution speed enhances development productivity when validating functional correctness across a large number of test cases and kernel configurations. Furthermore, modern QEMU versions provide architecturally accurate support for RVV 1.0 instructions, enabling reliable ISA-level validation.

QEMU User-Mode Emulation QEMU user-mode emulation is the primary execution mode used throughout this project. In this mode, RISC-V Linux binaries are executed directly on the host system without emulating a full hardware platform:

```
1 qemu-riscv64 -cpu rv64,v=true,vlen=256 ./my_program
```

User-mode emulation offers fast startup times, direct access to the host file system, and straightforward integration with development tools. Importantly, it allows the vector length (`VLEN`) to be configured at runtime, enabling validation of Vector Length Agnostic (VLA) kernel behavior across multiple vector configurations.

QEMU System-Mode Emulation In addition to user-mode execution, QEMU system-mode emulation is employed during the development of Python wrappers for the kernel library. System-mode emulation provides a complete virtualized RISC-V machine, including bootloader, kernel, and virtual devices, enabling end-to-end testing of library integration within a full operating system environment.

This mode is particularly valuable for validating language bindings, dynamic linking behavior, and interaction between Python-based tooling and the underlying RISC-V kernel implementations. While system-mode execution incurs higher simulation overhead, it provides a realistic software stack that closely mirrors deployment scenarios.

Role in the Verification Workflow Within the overall methodology, QEMU serves as the functional validation layer. Kernel outputs produced under QEMU are compared against ONNX-based golden references to ensure correctness and portability. Once functional correctness is established, performance and microarchitectural behavior are evaluated using cycle-accurate RTL simulation on the Vicuna and Ara cores.

3.2.3 ONNX: Open Neural Network Exchange

The Open Neural Network Exchange (ONNX) is an open standard designed to represent machine learning and deep learning models in a framework-independent manner. It was originally established through a collaboration between Facebook and Microsoft with the objective of improving interoperability across machine learning frameworks, tools, and hardware platforms.

ONNX defines a common intermediate representation for machine learning models, enabling them to be exported from one framework and executed or analyzed in another without modification. This representation is based on a computational graph abstraction, where nodes correspond to standardized operators and edges represent the flow of multi-dimensional tensors between operators.

The adoption of ONNX has been widely supported by both academia and industry, and it is now integrated into many popular machine learning frameworks and deployment toolchains. Its design emphasizes portability, reproducibility, and long-term maintainability, making it particularly suitable for systems-level research and hardware-oriented optimization efforts.

In the context of this project, ONNX serves as a unifying abstraction layer between high-level machine learning models and low-level, architecture-specific kernel implementations targeting the RISC-V Vector Extension.

Components of ONNX and Their Role in the Project

ONNX Model Components An ONNX model represents a machine learning computation as a directed computational graph, where nodes correspond to standardized operators and edges represent the flow of multi-dimensional tensor data between operators. This graph-based representation explicitly defines model inputs, outputs, and intermediate computations, providing a clear and structured description of the overall computation.

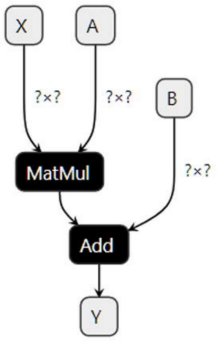


Figure 3: ONNX Model Computational Graph example illustrating nodes (operators) and edges (tensor data flow).

ONNX operators are drawn from a predefined and versioned operator set, with each operator having deterministic and well-specified mathematical semantics. In addition, ONNX models explicitly define tensor data types, shapes, and constant parameters, enabling consistent interpretation of computations across different software and hardware platforms. This standardized structure allows ONNX models to serve as precise and reproducible representations of intended kernel behavior.

Role of ONNX in the Project In this project, ONNX models are used as golden references for the functional validation of machine learning kernels implemented using the RISC-V Vector Extension. The ONNX representation mirrors the functionality of the developed kernels by explicitly defining the same inputs, outputs, and computational operations, independent of any specific hardware implementation.

The RVV-based kernel outputs are compared directly against the corresponding outputs generated from ONNX models executed using a validated ONNX runtime. Since ONNX provides a standardized and hardware-agnostic format with deterministic operator behavior, it serves as a reliable baseline for correctness verification. This approach ensures that discrepancies in output can be attributed to kernel implementation issues rather than ambiguities in operator definitions or execution semantics.

By adopting ONNX as the functional reference, the project achieves reproducible and framework-independent validation, strengthening confidence that the optimized vectorized kernels preserve the intended computational behavior while improving performance.

3.3 RISC-V Vectorization Kernels Design

This section presents the design and implementation of optimized RISC-V vector kernels for machine learning and digital signal processing applications. The kernels are organized into four fundamental patterns based on their computational characteristics and memory access behaviors. Each pattern represents a distinct class of operations commonly found in neural network inference pipelines and computer vision workloads.

3.3.1 Pattern 1: Compute-Bound FMA Operations

Neural network workloads are dominated by matrix multiplication and fully connected layers. These operations form the computational backbone of both inference and training pipelines. Their defining characteristic is extremely high arithmetic intensity: we perform massive volumes

of floating-point operations relative to the amount of data moved from memory. This makes them ideal candidates for vectorization.

At the core of these operations lies the FMA, where we accumulate multiple products into a single result. Modern processors can execute FMAs in a single pipelined cycle, but scalar code wastes this potential by processing one operation at a time. Vector instructions let us pack multiple FMAs into each cycle, multiplying our computational throughput.

These kernels also exhibit regular, predictable structure. Data dependencies are minimal and well-defined, memory access patterns follow sequential or strided layouts, and control flow remains simple. This regularity plays to the strengths of SIMD architectures, where we can fill vector registers densely with minimal overhead.

Matrix Multiplication (GEMM)

Kernel Description GEMM computes $C = A \times B$, where A is $M \times K$, B is $K \times N$, and the result C is $M \times N$. Each output element represents a dot product:

$$C[i][j] = \sum_{k=0}^{K-1} A[i][k] \cdot B[k][j]$$

This operation underlies virtually all linear algebra in machine learning, from simple linear layers to complex attention mechanisms.

Scalar Implementation The straightforward scalar approach uses three nested loops. For each row in A and column in B, we accumulate a dot product by iterating through K elements, multiplying corresponding pairs and summing results:

```
for each row i in matrix A:
    for each column j in matrix B:
        sum = 0
        for k from 0 to K - 1:
            sum = sum + A[i][k] * B[k][j]
        C[i][j] = sum
```

Listing 1: Scalar GEMM pseudo-code

This processes one output element at a time. Each multiply-add depends on the previous accumulation, preventing any overlap of operations. The deeply nested loops add control overhead, and we completely ignore the wide vector registers sitting idle in the processor. Modern CPUs with pipelined FMA units spend most of their time stalled, waiting for data dependencies to resolve.

Vectorization Strategy Rather than computing output elements sequentially, we can process multiple columns of the output simultaneously. Instead of accumulating into a single scalar, we maintain vl parallel accumulators in a vector register, where vl represents the number of elements that fit in one vector based on available hardware width.

This reorganization transforms the inner loop structure. For each element $A[i][k]$ in the current row, we broadcast that single value across all lanes of a vector register. We then load vl consecutive elements from row k of matrix B and multiply the broadcast value with this vector. The result updates all vl accumulators in parallel.

This approach delivers several advantages. We exploit the full width of vector functional units, executing vl FMAs per cycle instead of one. Memory access to matrix B becomes sequential and cache-friendly, since we load contiguous elements. Loop overhead drops because we process vl outputs per iteration instead of one.

Implementation The vectorized code replaces the innermost column loop with a while loop that processes columns in chunks. We track how many columns remain to be processed and handle them in groups of vl:

```
for each row i in matrix A:
    remaining_columns = N
    while remaining_columns > 0:
        vl = set_vector_length(remaining_columns)
        j = N - remaining_columns

        acc_vector = broadcast(0, vl)
        for k from 0 to K - 1:
            # Broadcast scalar from A to all vector lanes
            a_scalar = A[i][k]
            # Load vl contiguous elements from B
            b_vector = load_vector(B[k][j], vl)

            # Vector-Scalar Fused Multiply-Add
            acc_vector = fma(acc_vector, a_scalar, b_vector)

        store_vector(C[i][j], acc_vector, vl)
        remaining_columns -= vl
```

Listing 2: Vectorized GEMM pseudo-code

For each chunk of columns, we initialize a vector accumulator to zero. The inner loop over k performs the dot product computation: we broadcast each element from row i of matrix A across all lanes, load vl consecutive elements from the corresponding row of B, multiply them element-wise, and accumulate the results. After completing the dot product across all K elements, we store the vector of results to the output matrix.

The broadcast operation is crucial here. It takes a single scalar value $A[i][k]$ and replicates it across every lane of a vector register, allowing that single value to be multiplied with vl different elements from B simultaneously. The load from B is sequential, fetching consecutive memory locations, which aligns perfectly with cache line sizes and enables efficient prefetching.

The RISC-V vector extension determines vl dynamically at runtime based on the hardware's vector register width and the element size. When the number of remaining columns isn't evenly divisible by the maximum vector length, the hardware automatically reduces vl for the final iteration, processing exactly the remaining elements without any special-case code.

Dense Layer (Fully Connected)

Kernel Description Dense layers compute weighted sums across all inputs for each output neuron. Given an input vector x of size K and a weight matrix W of shape $N \times K$ (where each row corresponds to one output neuron), we compute:

$$\text{output}[j] = \text{bias}[j] + \sum_{k=0}^{K-1} \text{input}[k] \cdot \text{weights}[j][k]$$

This is essentially matrix-vector multiplication with bias addition. While conceptually similar to GEMM, we're multiplying a matrix by a single vector rather than another matrix, which affects how we structure the computation.

Scalar Implementation The scalar code processes one output neuron at a time. We initialize each accumulator with its corresponding bias, then iterate through all input features, multiplying each by its weight and accumulating:

```
for each output neuron j from 0 to N - 1:
    acc = bias[j]
    for each input feature k from 0 to K - 1:
        acc = acc + input[k] * weights[j][k]
    output[j] = acc
```

Listing 3: Scalar Dense Layer pseudo-code

By initializing with the bias value, we avoid a separate bias addition pass after computing the weighted sum. Each output neuron is computed independently with a sequential accumulation across all input features. Like scalar GEMM, this serializes all operations. Each accumulation depends on the previous one, and we process only a single output at a time. Vector units remain completely unutilized.

Vectorization Strategy We parallelize across output neurons, computing vl neurons simultaneously. The key insight is that for each input feature, we can broadcast that feature value and multiply it with weights from multiple neurons in parallel.

The memory access pattern differs from GEMM in an important way. In GEMM, consecutive output columns correspond to consecutive elements in memory (sequential access to B). Here, to compute multiple neurons in parallel, we need to load weights for the same input feature across different neurons. These weights are not contiguous in memory because the weight matrix is stored in row-major order, with each row representing one neuron's complete set of weights.

For input feature k, the weights we need are at positions $\text{weights}[j][k]$, $\text{weights}[j+1][k]$, $\text{weights}[j+2][k]$, etc. These addresses are separated by K elements (the stride of one row), making this a strided memory access pattern rather than a sequential one.

Implementation The vectorized implementation processes output neurons in chunks, initializing accumulators directly with bias values:

```
j = 0
while j < N:
    vl = set_vector_length(N - j)
    # Sequential load of bias values
    acc_vector = load_vector(bias + j, vl)

    for each input feature k from 0 to K - 1:
        # Load weights for vl different neurons (strided access)
        # Stride is equal to the row width K
```

```
        weight_vector = strided_load(weights + (j * K) + k,
            stride=K, vl)

        # Broadcast the scalar input feature
        input_val = input[k]

        # Vector-Scalar Fused Multiply-Add
        acc_vector = fma(acc_vector, input_val, weight_vector)

    store_vector(output + j, acc_vector, vl)
    j += vl
```

Listing 4: Vectorized Dense Layer pseudo-code

We start by loading vl bias values into the accumulator vector. This is a sequential load since bias values are stored contiguously. Then, for each input feature, we perform two key operations:

First, we load weights using strided access. The notation weights$[j : j + vl][k]$ means we're loading element k from rows j through $j+vl-1$. These elements are separated by K positions in memory (one full row), so we use a strided load instruction with stride equal to $K \times$element_size. The RISC-V vector ISA provides efficient strided load instructions that handle this pattern in hardware, fetching non-contiguous elements without manual gathering.

Second, we broadcast the input feature value across all vector lanes. This replicated value multiplies with the vector of weights, producing vl partial products that update the accumulators in parallel. After processing all K input features, each lane of the accumulator vector contains the complete output for one neuron (weighted sum plus bias), ready to be stored to memory.

The key advantage of initializing with bias values rather than zero is efficiency: we fold the bias addition into the initial load rather than performing a separate addition pass after the main computation. This saves both instructions and a complete pass over the output array.

3.3.2 Pattern 2: Sliding Window Kernels

Sliding window operations differ fundamentally from the compute-bound kernels above. Here, a small kernel slides across a larger input, computing outputs based on local neighborhoods. These operations dominate convolutional neural networks and pooling layers.

The defining characteristic is local spatial dependency: each output depends only on a small, localized region of input determined by kernel size and stride. Consecutive output positions often use overlapping input regions, creating opportunities for data reuse. However, this also introduces irregular memory access patterns and boundary conditions that complicate vectorization.

Optimization strategies differ from dense matrix operations. While GEMM benefits from vectorizing across output dimensions, sliding window kernels often benefit more from vectorizing across output width or channels, combined with careful register blocking to exploit spatial locality.

2D Convolution

Kernel Description Two-dimensional convolution is the fundamental operation in CNNs. Given an input feature map of shape $(C_{\text{in}}, H_{\text{in}}, W_{\text{in}})$, filters with C_{out} output channels, and a kernel of size (K_h, K_w), we compute an output of shape $(C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$.

For each output position and channel, we compute:

$$\text{output}[oc][oh][ow] = \sum_{ic=0}^{C_{\text{in}}-1} \sum_{kh=0}^{K_h-1} \sum_{kw=0}^{K_w-1} \text{input}[ic][ih][iw] \cdot \text{kernel}[oc][ic][kh][kw]$$

where input coordinates map to output coordinates through stride and padding:

$$ih = oh \times \text{stride}_h - \text{pad}_h + kh, \quad iw = ow \times \text{stride}_w - \text{pad}_w + kw$$

Scalar Implementation The scalar approach iterates over every dimension: batch, output channel, output position, input channel, and kernel position. For each kernel element, we compute the corresponding input coordinates, verify they're within bounds (handling padding), and accumulate the product:

```
Compute output height and width
Initialize output to zero

for each batch b:
    for each output channel oc, output row oh, output column ow:
        sum = 0
        for each input channel ic:
            for each kernel row kh:
                for each kernel column kw:
                    ih = oh * stride_h - pad_h + kh
                    iw = ow * stride_w - pad_w + kw
                    if ih, iw inside input bounds:
                        sum += input[b][ic][ih][iw] * kernel[oc][
                            ic][kh][kw]
        output[b][oc][oh][ow] = sum
```

Listing 5: Scalar 2D Convolution pseudo-code

The output is initialized to zero at the start, then we accumulate contributions from all input channels and kernel positions. The innermost loops compute coordinate mappings and perform boundary checks to handle padding. When an input coordinate falls outside the valid range, we skip that multiply-accumulate, effectively treating out-of-bounds regions as zero (zero-padding).

This straightforward implementation processes one output element at a time. The deeply nested loops incur substantial overhead, and the boundary checks add branching to every kernel position. Worse, consecutive output positions don't systematically reuse cached data, leading to poor memory behavior.

General Vectorization The general vectorization strategy has two phases: kernel repacking followed by parallel output channel computation. We reorganize the kernel weights so that elements for consecutive output channels become contiguous in memory.

In the original layout, kernel weights are organized as kernel$[oc][ic][kh][kw]$. To access weights for output channels oc, $oc+1$, $oc+2$, etc., for a fixed input channel and kernel position, we'd need to stride through memory with large jumps. By repacking into packed_kernel$[ic][kh][kw][oc]$, we make weights for consecutive output channels adjacent in memory, enabling efficient sequential vector loads.

During computation, we process vl output channels simultaneously. For each output position, we maintain vl parallel accumulators. As we iterate through input channels and kernel positions, we broadcast each input value and multiply it with a vector of repacked weights, updating all accumulators in parallel.

Implementation The implementation starts with kernel repacking, then executes the vectorized convolution:

```
Repack kernel so output channels are contiguous
Initialize output to zero

for each batch b:
    for output channels oc in vector chunks:
        vl = set_vector_length(remaining_channels)
        for each output row oh, output column ow:
            # Load initial accumulators for vl channels
            acc_vector = load_vector(output[b][oc][oh][ow], vl)

            for each input channel ic:
                for each kernel row kh:
                    for each kernel column kw:
                        ih = oh * stride_h - pad_h + kh
                        iw = ow * stride_w - pad_w + kw

                        if ih, iw inside input bounds:
                            input_val = input[b][ic][ih][iw]
                            # Load weights for vl consecutive
                               output channels
                            weight_vec = load_vector(packed_w[ic
                               ][kh][kw][oc], vl)
                            # Fused Multiply-Add (Scalar * Vector
                               )
                            acc_vector = fma(acc_vector,
                               input_val, weight_vec)

            store_vector(output[b][oc][oh][ow], acc_vector, vl)
```

Listing 6: Vectorized 2D Convolution pseudo-code

We initialize the output to zero before the main computation begins. For each output position, we load the current accumulator state (which starts at zero) from the output array. This might seem redundant for the first iteration, but it allows us to accumulate contributions from multiple input channels consistently.

The key operations happen in the innermost loops. For each valid input position, we load a single scalar input value. We then load a vector of vl consecutive kernel weights from the repacked array. The scalar input value is implicitly broadcast across all lanes when multiplied with the weight vector, producing vl partial products. These accumulate into the vector register holding our parallel output channel computations.

After processing all input channels and kernel positions for a given output location, the accumulator vector contains the complete output values for vl channels at that spatial position. We store this vector back to the output array.

The repacking cost is paid once during initialization and amortized across potentially thousands of forward passes. The memory layout transformation converts strided access (which would require expensive gather operations or multiple scalar loads) into efficient sequential loads.

Boundary handling remains explicit through the conditional check. When coordinates fall outside the input bounds, we skip the multiply-accumulate entirely. This avoids the memory overhead of explicitly padding the input array, though it introduces some branching. For performance-critical applications where branches hurt, explicitly padding the input eliminates these conditionals at the cost of larger memory footprint.

Convolution as Matrix Multiplication (Im2Col) An alternative approach transforms convolution into standard matrix multiplication. The Im2Col (image-to-column) method unfolds the input into a matrix where each column represents a flattened window. The kernel becomes a matrix where each row contains flattened weights for one output channel. Convolution then reduces to GEMM.

This transformation works for any kernel size and allows us to leverage highly optimized matrix multiplication algorithms.

```
Compute output height and width

# Step 1: Im2Col transformation
for each output position (oh, ow):
    for each input channel ic:
        for each kernel position kh, kw:
            input_val = get_input_with_padding(b, ic, oh, ow, kh,
                kw)
            col[K_index][oh * out_w + ow] = input_val

# Step 2: GEMM
gemm_output = matrix_multiply(kernel_matrix, col_matrix)

# Step 3: Bias Add
for each output channel oc:
    bias_vec = broadcast(bias[oc])
    for each output position in chunks:
        out_vec = load_vector(gemm_output[oc][pos])
        out_vec = out_vec + bias_vec
        store_vector(output[oc][pos], out_vec)
```

Listing 7: Im2Col Transformation pseudo-code

The Im2Col step unfolds the input into a 2D matrix. Each column of this matrix represents one output position's receptive field: all input values that contribute to that output, flattened into a 1D vector. The row index K_index combines the input channel and kernel position indices into a single linear index.

For positions where the receptive field extends outside the input bounds (due to padding), we write zeros to the column matrix. This explicitly materializes the zero-padding in memory.

After unfolding, we have a matrix multiplication problem: kernel_matrix is $C_{\text{out}} \times (C_{\text{in}} \times K_h \times K_w)$, and col_matrix is $(C_{\text{in}} \times K_h \times K_w) \times (H_{\text{out}} \times W_{\text{out}})$. The product gives us all output values for all channels and positions.

Finally, we add bias values to each output channel. Since the GEMM produces outputs in

channel-major order, we simply iterate through channels and positions, adding the appropriate bias to each element.

The advantage is that GEMM kernels are among the most optimized operations on any platform, often hand-tuned in assembly with careful cache blocking and register tiling. By reducing convolution to GEMM, we leverage this existing optimization work.

The disadvantage is memory overhead. The column matrix can be quite large: for a 224×224 input image with 64 channels and a 3×3 kernel, the unfolded matrix requires roughly 200MB of temporary storage. This overhead may be acceptable on systems with ample memory, but becomes prohibitive on embedded devices.

The transformation is particularly effective when the same input will be convolved with multiple different kernels (as in neural network layers with many output channels), since the Im2Col cost is paid once and amortized across many GEMM operations.

Specialized 3x3 Vectorization The 3x3 kernel size deserves special attention because it dominates modern CNN architectures. When we know the kernel size is fixed at 3x3, we can apply optimizations that wouldn't be practical for variable-size kernels.

The key insight is that we can preload three consecutive input rows and keep them in registers across multiple output column computations. For a given output row, the input data from rows oh, $oh+1$, and $oh+2$ will be reused across all output columns in that row (assuming stride 1). By loading these rows once and reusing them, we dramatically reduce memory traffic.

```
for each output row oh:
    # Preload input rows for the current sliding window set
    row0 = input[b][ic][oh]
    row1 = input[b][ic][oh+1]
    row2 = input[b][ic][oh+2]

    ow = 0
    while ow < output_width:
        vl = set_vector_length(output_width - ow)
        # Load vectors from preloaded rows at different
           horizontal offsets
        v00 = load_vector(row0 + ow, vl)
        v01 = load_vector(row0 + ow + 1, vl)
        v02 = load_vector(row0 + ow + 2, vl)

        v10 = load_vector(row1 + ow, vl)
        v11 = load_vector(row1 + ow + 1, vl)
        v12 = load_vector(row1 + ow + 2, vl)

        v20 = load_vector(row2 + ow, vl)
        v21 = load_vector(row2 + ow + 1, vl)
        v22 = load_vector(row2 + ow + 2, vl)

        # Accumulate 9 FMA operations for the 3x3 window
        acc = v00 * k00 + v01 * k01 + v02 * k02
        acc = acc + v10 * k10 + v11 * k11 + v12 * k12
        acc = acc + v20 * k20 + v21 * k21 + v22 * k22

        store_vector(output[oc][oh][ow], acc, vl)
```

```
            ow += vl
```

Listing 8: Specialized 3x3 Convolution pseudo-code

We load three input rows into temporary storage outside the column loop. These represent the three rows of input that contribute to the current output row. Then, for each chunk of output columns, we load nine vector variables: three vectors from each of the three rows.

Each vector contains vl consecutive elements from an input row. The notation $v01$ means "vector from row 0, offset by 1 position" this captures the three horizontal positions of the kernel across multiple output columns simultaneously.

We then perform nine multiply-accumulate operations, one for each kernel position. Each operation multiplies one input vector with the corresponding kernel weight (broadcast to match the vector length) and accumulates into the result. After all nine operations, we have vl complete output values for consecutive output columns.

This approach avoids the coordinate computation overhead of the general vectorization. We don't recalculate ih and iw for each kernel position; instead, we directly reference preloaded vectors. The overlapping window pattern means that $v01$ for the current output column becomes $v00$ for the next, creating register reuse opportunities that a smart compiler or hand-written assembly can exploit.

Compared to Im2Col, this method uses minimal extra memory (just three row buffers) and avoids the unfolding step entirely. It's most effective for stride-1 convolutions where the input window overlap is maximal. For larger strides, the reuse benefits diminish, and Im2Col may become more attractive.

The trade-off is specificity: this approach is hard-coded for 3x3 kernels. Generalizing it to other sizes would require different loop structures and vector load patterns, whereas Im2Col works uniformly for any kernel size.

Max Pooling

Kernel Description Max pooling downsamples feature maps by taking the maximum value within each window. Given an input and a pooling kernel of size (K_h, K_w), we compute:

$$\text{output}[c][oh][ow] = \max_{kh=0}^{K_h-1} \max_{kw=0}^{K_w-1} \text{input}[c][ih][iw]$$

where (ih, iw) are determined by output position, stride, and padding. Unlike convolution, this involves only comparisons and selections, with no arithmetic operations.

Scalar Implementation The scalar code processes one output position at a time. For each position, we compute the valid window boundaries (accounting for padding), initialize a maximum value to negative infinity, and scan all values within the window to find the maximum:

```
Compute output height and width

for each batch b:
    for each channel c:
        for each output row oh:
            for each output column ow:
                h_start = oh * stride_h - pad_h
                w_start = ow * stride_w - pad_w
```

```
                    h_end = min(h_start + k_h, input_height)
                    w_end = min(w_start + k_w, input_width)
                    h_start = max(h_start, 0)
                    w_start = max(w_start, 0)

                    max_val = -infinity
                    for h from h_start to h_end - 1:
                        for w from w_start to w_end - 1:
                            max_val = max(max_val, input[b][c][h][w])

                    output[b][c][oh][ow] = max_val
```

Listing 9: Scalar Max Pooling pseudo-code

We calculate the window boundaries explicitly, clamping them to the valid input range. This handles padding by simply restricting the region we scan. We initialize the maximum to negative infinity, ensuring any actual input value will be larger.

The inner loops scan the valid window region, comparing each input value against the current maximum and updating when we find a larger value. After scanning the entire window, we write the maximum to the output.

Though conceptually simple, this scalar implementation can still bottleneck shallow networks or configurations with large stride values that reduce spatial data reuse.

Vectorization Strategy We vectorize across output columns, computing vl output positions simultaneously. Each vector lane maintains an independent maximum accumulator. As we scan the pooling window, we load vectors of input values and compute element-wise maximums, updating all lanes in parallel.

The key insight is that maximum operations are inherently parallelizable: tracking maxima for multiple output positions requires no inter-lane communication. Each lane independently compares and updates its maximum value. The vector maximum instruction compares corresponding elements in two vectors and produces a result vector containing the element-wise maxima.

Implementation The vectorized implementation processes output columns in chunks:

```
Compute output height and width

for each batch b:
    for each channel c:
        for each output row oh:
            ih_start = oh * stride_h - pad_h
            ow = 0
            while ow < output_width:
                vl = set_vector_length(output_width - ow)
                max_vector = broadcast(-infinity, vl)

                for kh from 0 to k_h - 1:
                    ih = ih_start + kh
                    if ih < 0 or ih >= input_height:
                        continue
```

```
16
17                      for kw from 0 to k_w - 1:
18                          # Input positions are separated by
                               stride_w
19                          iw_start = ow * stride_w - pad_w + kw
20                          input_vector = strided_load(input_ptr +
                               iw_start, stride_w, vl)
21                          max_vector = max(max_vector, input_vector
                               )
22
23                  store_vector(output_ptr + ow, max_vector, vl)
24                  ow += vl
```

Listing 10: Vectorized Max Pooling pseudo-code

We compute the starting input row position once per output row, outside the column loop. This row start is the same for all output columns in this row, so computing it once saves redundant arithmetic.

For each chunk of vl output columns, we initialize a vector of maximum values to negative infinity. This initialization broadcasts the scalar value $-\infty$ across all vector lanes, giving each lane its own independent maximum tracker.

The inner loops iterate through the pooling window. For each kernel row, we check if the corresponding input row is within bounds. If not, we skip this entire row (the `continue` statement). For each kernel column, we compute the starting input column position and load vl consecutive input values.

This load is crucial: for output column ow, we need input at position $ow \times \text{stride}_w - \text{pad}_w + kh$. For the next output column $ow + 1$, we need input at position $(ow + 1) \times \text{stride}_w - \text{pad}_w + kh$. These positions are separated by `stride_w` in the input. So when stride is 1, we load consecutive elements; when stride is 2, we load every other element; and so on.

The RISC-V vector ISA can handle this through either strided loads (when stride more than 1) or indexed loads. The notation `strided_load` abstracts this detail, but the actual implementation would use the appropriate load instruction variant.

After loading input values, we compute the element-wise maximum with the current maximum vector. Each lane compares its input value against its current maximum and keeps whichever is larger. This process continues through all kernel positions.

Once we've scanned the entire window, the maximum vector contains the final maximum values for vl output positions. We store this vector to the output array and advance to the next chunk of columns.

The vectorization achieves significant speedup by computing vl output elements in parallel, with nearly the same work per output position as the scalar version, but amortized across vl lanes. The comparison-based nature of max pooling maps naturally to SIMD operations, making this one of the more straightforward kernels to vectorize effectively.

3.3.3 Pattern 3: Pointwise/Elementwise Kernels

Pointwise (elementwise) operations represent the most straightforward vectorization opportunities in machine learning workloads. These kernels exhibit several characteristics that make them ideal candidates for RISC-V vector acceleration:

- **Elementwise independence**: No cross-element dependencies or data hazards
- **Fully contiguous memory access**: Linear, predictable access patterns
- **No reductions or control-flow coupling**: Minimal branching overhead
- **High execution frequency**: Common operations in ML inference pipelines
- **Immediate cost amortization**: Vector setup overhead paid back instantly

These properties make pointwise kernels the low-risk, high-payoff RVV optimization targets. The kernels in this category include ReLU, Leaky ReLU, Bias Add, and Tensor Add operations.

ReLU and Leaky ReLU Activation Functions The Rectified Linear Unit (ReLU) is one of the most widely used activation functions in modern neural networks, defined as:

$$\text{ReLU}(x) = \max(x, 0) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Leaky ReLU extends this definition to preserve small negative values using a scaling factor α:

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha \cdot x & \text{otherwise} \end{cases}$$

Scalar ReLU Implementation:

```
for i = 0 to size-1:
    output[i] = max(input[i], 0)
end for
```

Listing 11: Scalar ReLU pseudo-code

Scalar Leaky ReLU Implementation:

```
for i = 0 to n-1:
    if src[i] < 0:
        dest[i] = src[i] * alpha
    else:
        dest[i] = src[i]
    end if
end for
```

Listing 12: Scalar Leaky ReLU pseudo-code

Vectorized ReLU Implementation:

The vectorized ReLU leverages vector max operations and strip-mining to process multiple elements in parallel:

```
v_zero = vector_of_zeros

while there are elements left:
    vl = vector_length_for_this_iteration
    v_in = load_vector(input_pointer, vl)
    v_out = max(v_in, v_zero)
```

```
    store_vector(output_pointer, v_out, vl)

    advance pointers by vl
    decrease remaining_count by vl
end while
```

Listing 13: Vectorized ReLU pseudo-code

Vectorized Leaky ReLU Implementation:

Leaky ReLU vectorization uses masked operations to handle the conditional behavior efficiently:

```
alpha_vec = broadcast(alpha)

while remaining > largest_step (e.g., n*vector_length):
    # Load n vectors: v0, v1, ..., vn
    for each vi in {v0, v1, ..., vn}:
        negative_mask = (vi < 0)
        vi_leaky = negative_mask ? (vi * alpha_vec) : vi
        store(vi_leaky)
    end for

    skip forward n*vector_length elements
end while
```

Listing 14: Vectorized Leaky ReLU pseudo-code

The vectorized implementation processes multiple elements in parallel, with the hardware automatically handling varying vector lengths across different VLEN configurations.

Bias Add and Tensor Add Operations Bias addition is a fundamental operation in neural networks, where a learned bias vector is added to the output of convolutional or fully-connected layers. Tensor addition combines two tensors elementwise and is used throughout neural networks for residual connections and feature fusion.

Scalar Bias Add Implementation:

```
for batch in batches:
    for channel in channels:
        bias_val = bias[channel]
        for pixel in channel_pixels:
            output[...] = input[...] + bias_val
        end for
    end for
end for
```

Listing 15: Scalar Bias Add pseudo-code

Scalar Tensor Add Implementation:

```
for i = 0 to size-1:
    Output[i] = A[i] + B[i]
end for
```

Listing 16: Scalar Tensor Add pseudo-code

Vectorized Bias Add Implementation:

The vectorized bias add broadcasts the bias value using vector-scalar operations:

```
for channel in channels:
    bias_val = bias[channel]

    for chunk in spatial_data by vector_size:
        vec = load_vector(input + offset)
        vec = vec + bias_val  # broadcast addition
        store_vector(output + offset, vec)
    end for
end for
```

Listing 17: Vectorized Bias Add pseudo-code

Vectorized Tensor Add Implementation:

Tensor addition vectorization is straightforward with vector-vector operations:

```
set position = 0

while position < size:
    vl = min(vector_register_length, size - position)

    # Load vl elements from A[position...position+vl-1]
    # Load vl elements from B[position...position+vl-1]
    A_vector = load_vector(A + position, vl)
    B_vector = load_vector(B + position, vl)

    result = A_vector + B_vector

    store_vector(Output + position, result, vl)

    position += vl
end while
```

Listing 18: Vectorized Tensor Add pseudo-code

The vector-length agnostic programming model ensures these implementations automatically adapt to different hardware vector widths without modification.

3.3.4 Pattern 4: Tensor Indexing and Data Movement

Tensor indexing operations, such as gather and scatter, are essential for handling sparse data, embedding lookups, and dynamic tensor reshaping. Unlike pointwise operations that access memory linearly, indexing kernels rely on a set of indices to determine which elements to read or write. This introduces irregular memory access patterns that traditionally cause significant performance bottlenecks in scalar architectures due to address calculation overhead and cache misses.

The RISC-V Vector Extension addresses these challenges through indexed load and store instructions. These instructions allow the processor to process multiple non-contiguous memory locations in a single vector operation. By calculating a vector of addresses (or offsets) and passing it to the hardware, we can amortize the cost of irregular memory transactions and exploit available memory bandwidth more effectively than sequential scalar code.

Gather and Gather Elements

Kernel Description Gather operations collect data from an input tensor based on a provided index tensor. While a standard Gather typically selects entire slices along an axis, GatherElements performs a more granular, element-wise selection. For a 2D tensor along $axis = 1$, the operation is defined as:

$$\text{output}[i][j] = \text{data}[i][\text{indices}[i][j]]$$

Scalar Implementation The scalar approach requires nested loops to resolve indices one by one. For each output position, the processor must calculate a new memory address, perform a load, and then store the result, often leading to pipeline stalls during the address resolution phase:

```
for i from 0 to indices_rows - 1:
    for j from 0 to indices_cols - 1:
        # Resolve index for the target axis
        target_col = indices[i][j]
        output[i][j] = data[i][target_col]
```

Listing 19: Scalar Gather Elements (Axis 1) pseudo-code

Vectorization Strategy Vectorization is achieved by processing a chunk of indices simultaneously. The key challenge in RVV is that indices are often provided as 64-bit integers, while data elements (like floats) are 32-bit. We utilize narrowing operations to convert 64-bit indices into 32-bit offsets. For axis-based gathers, we calculate byte offsets by scaling the indices by the data stride. These are then passed to a vector indexed-load instruction (`vluxei`), which fetches non-contiguous data into a single vector register.

Implementation The vectorized implementation leverages `vsetvli` to handle index chunks and narrowing wrappers to align data types:

```
for i from 0 to indices_rows - 1:
    j = 0
    while j < indices_cols:
        vl = set_vector_length(indices_cols - j)

        # Load 64-bit indices and narrow to 32-bit
        v_idx64 = load_vector(indices[i * indices_cols + j], vl)
        v_idx32 = narrow_to_32bit(v_idx64, vl)

        # Calculate byte offsets (index * element_size)
        v_offsets = v_idx32 << 2  # Shift left by 2 for float32
            (4 bytes)

        # Indexed load and contiguous store
        v_vals = indexed_load(data_row_ptr, v_offsets, vl)
        store_vector(output[i * indices_cols + j], v_vals, vl)

        j += vl
```

Listing 20: Vectorized Gather Elements pseudo-code

Scatter Elements

Kernel Description Scatter is the inverse of gather; it writes values from an updates tensor into a data tensor at positions specified by indices. For $axis = 0$ (rows), the output is updated as:

$$\text{output}[\text{indices}[i][j]][j] = \text{updates}[i][j]$$

Scalar Implementation The scalar approach requires a full copy of the original data followed by individual updates. This involves heavy branching and individual store instructions that cannot be pipelined easily, as each store address is potentially in a completely different memory bank.

```
output = copy(data) # Initial pass to preserve un-updated
    elements

for i from 0 to indices_rows - 1:
    for j from 0 to indices_cols - 1:
        target_row = indices[i][j]
        output[target_row][j] = updates[i][j]
```

Listing 21: Scalar Scatter Elements pseudo-code

Vectorization Strategy Similar to Gather, we vectorize the update loop. When scattering along rows ($axis = 0$), the target addresses are separated by the row stride ($W \times \text{sizeof(float)}$). We compute absolute byte offsets by combining the narrowed indices with a Vector ID (`vid`) multiplied by the element size to account for the current column position. This allows us to perform an unordered indexed store (`vsuxei`) for the entire vector chunk.

Implementation The vectorized implementation focuses on efficient address generation for the scattered writes:

```
output = copy(data)

for i from 0 to indices_rows - 1:
    j = 0
    while j < indices_cols:
        vl = set_vector_length(indices_cols - j)

        v_idx64 = load_vector(indices[i * indices_cols + j], vl)
        v_updates = load_vector(updates[i * indices_cols + j], vl
            )
        v_idx32 = narrow_to_32bit(v_idx64, vl)

        # Calculate row-stride offsets: (index * data_cols * 4)
        v_row_byte = v_idx32 * (data_cols * 4)

```

```
        # Add column-wise byte offsets: (j + vector_id) * 4
        v_col_idx = get_vector_id() + j
        v_col_byte = v_col_idx * 4
        v_final_off = v_row_byte + v_col_byte

        # Vector indexed store (Scatter)
        indexed_store(output, v_final_off, v_updates, vl)

        j += vl
```

Listing 22: Vectorized Scatter Elements (Axis 0) pseudo-code

The use of indexed stores significantly reduces the cycles spent on address arithmetic. By using RVV narrowing and indexed store instructions, we avoid the scalar bottleneck of repeatedly calculating complex indices for every single update, allowing the hardware to dispatch multiple memory writes in parallel.

3.3.5 Pattern 5: Batch Normalization

Batch Normalization is a critical regularizing operation that stabilizes the learning process by normalizing the distribution of layer activations. It is characterized as a memory-bound, unit-stride operation that applies a per-channel set of parameters across large spatial and batch dimensions. Because the same transformation is applied to every pixel in a feature map, it is an ideal candidate for high-bandwidth vectorization.

The core computational challenge is not arithmetic complexity, but rather memory throughput. The operation requires fetching a contiguous segment of the input tensor, performing a fused multiply-add (FMA) using channel-specific constants, and storing the result back to memory. RVV accelerates this pattern by loading the normalization parameters into registers and processing the spatial data in long, continuous vector segments.

Batch Normalization

Kernel Description For each channel c, Batch Normalization transforms the input x using the learned mean (μ), variance (σ^2), scale (γ), and shift (β) parameters:

$$\text{output} = \gamma \cdot \left(\frac{\text{input} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta$$

To optimize this for inference, we pre-calculate two channel-specific constants:

$$\alpha = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}, \quad \beta' = \beta - \mu \cdot \alpha$$

The operation then simplifies to a linear transformation: $\text{output} = \text{input} \cdot \alpha + \beta'$.

Scalar Implementation The scalar approach processes the spatial dimensions sequentially for each channel. This results in significant overhead as the processor must manage loop counters and individual memory requests for every pixel, failing to exploit the contiguous nature of the data:

```
for c from 0 to channels - 1:
    # Pre-calculate constants for the current channel
    alpha = scale[c] / sqrt(variance[c] + epsilon)
    beta_prime = bias[c] - mean[c] * alpha

    for i from 0 to spatial_dim - 1:
        output[c][i] = input[c][i] * alpha + beta_prime
```

Listing 23: Scalar Batch Normalization pseudo-code

Vectorization Strategy We vectorize across the spatial dimensions ($H \times W$) for each channel. Since the data within a channel is stored contiguously in memory (unit-stride), we can maximize memory bus utilization. For each channel, we broadcast the pre-calculated α and β' values across all vector lanes. We then iterate through the spatial data in chunks of vl, performing a vector-scalar FMA. This approach amortizes the cost of the channel-specific parameter calculations over thousands of spatial elements.

Implementation The vectorized implementation utilizes unit-stride loads and stores, combined with vector-scalar arithmetic to achieve near-peak memory bandwidth:

```
spatial_dim = height * width

for c from 0 to channels - 1:
    # Calculate transformation constants per channel
    alpha = scale[c] / sqrt(variance[c] + epsilon)
    beta_prime = bias[c] - mean[c] * alpha

    i = 0
    while i < spatial_dim:
        vl = set_vector_length(spatial_dim - i)

        # Load contiguous input segment
        v_in = load_vector(input_ptr + (c * spatial_dim) + i, vl)

        # Vector-Scalar Fused Multiply-Add: (v_in * alpha) +
           beta_prime
        v_out = fma(v_in, alpha, beta_prime, vl)

        # Unit-stride store
        store_vector(output_ptr + (c * spatial_dim) + i, v_out,
           vl)

        i += vl
```

Listing 24: Vectorized Batch Normalization pseudo-code

By hoisting the α and β' calculations outside the inner loop, we reduce the computational cost to a single FMA per element. The use of `vsetvli` ensures that the kernel remains efficient regardless of the feature map size, handling the final "tail" of the spatial dimension without the need for manual scalar cleanup loops.

3.3.6 Pattern 6: Post-Processing Kernels - Non-Maximum Suppression

Non-Maximum Suppression (NMS) is a critical post-processing step in object detection pipelines that eliminates redundant overlapping bounding boxes. Unlike the previous patterns, NMS presents unique vectorization challenges:

- **Post-processing characteristic**: Not compute-heavy, but memory and branch-intensive
- **Strong data dependencies**: Greedy sequential suppression logic
- **Heavy sorting and conditional logic**: Conditional suppression patterns
- **Irregular memory access**: Conditional operations create unpredictable patterns
- **Moderate vectorization gains**: Limited parallelism opportunities

However, vectorization opportunities exist in specific NMS substeps:

- **Score filtering**: Vector compare and compress operations
- **Batched IoU computation**: Parallel min/max/sub/mul operations
- **Threshold comparisons**: Vector masking operations

Scalar NMS Algorithm The scalar NMS implementation follows a greedy sequential approach:

```
# Step 1: Create list of (score, index) pairs for all detections
# Step 2: Filter: keep only pairs where score >= score_threshold
# Step 3: Sort pairs by score (DESCENDING)
# Step 4: Initialize empty list 'selected'
# Step 5: Initialize suppressed[N] = false (or use list of active candidates)

while pairs_list is not empty AND |selected| < max_output_per_class:
    a. Take top pair: current_index = pair.index
    b. Add current_index to selected
    c. current_box = boxes[current_index]
    d. For each remaining candidate j after current in sorted list:
            if suppressed[j]: continue
            candidate_box = boxes[j.index]
            if boxes do NOT overlap at all: continue
            iou = compute_iou(current_box, candidate_box)
            if iou >= iou_threshold:
                suppress j (mark as suppressed / remove from consideration)
            end if
        end for
end while

# Step 7: Return selected indices
```

Listing 25: Scalar NMS pseudo-code

Vectorized NMS Implementation The vectorized NMS strategically applies vectorization to substeps with high parallelism:

```
# Step 1: Collect high-confidence candidates (VECTORIZED)
candidates = empty list of (score, index) pairs

for i = 0 to N-1 step vector_length:
    vl = min(vector_length, N - i)
    v_scores = vector_load(scores[i:i+vl])
    mask = (v_scores >= score_thresh)

    if any scores pass:
        local_idx = [0 .. vl-1]
        kept_idx = compress(local_idx, mask)
        for each kept j:
            global_i = i + j
            add (scores[global_i], global_i) to candidates
        end for
    end if
end for

# Step 2: Sort candidates DESCENDING by score
# Step 3: Convert all candidate boxes to corner format
# box_corners[M][4]  (M = # of candidates after filter)

# Step 4: selected = empty list
#         suppressed = [false for all M candidates]

# Step 5: Greedy suppression with vectorized IoU
for each candidate i = 0 .. M-1 (sorted order):
    if suppressed[i]: continue
    if |selected| >= max_output_per_class: break

    add index of candidate i to selected
    current_box = box_corners[i]

    # Inner suppression (can be partially vectorized)
    for j = i+1 to M-1:
        if suppressed[j]: continue
        other_box = box_corners[j]

        if no overlap possible (quick reject): continue

        # VECTORIZED IoU computation
        iou = vectorized_iou(current_box, other_box)
        # Uses max/min/sub on (x1,y1,x2,y2)
        # -> areas, union, iou = inter/union

        if iou >= iou_thresh:
            suppressed[j] = true
        end if
    end for
```

```
end for
```

Listing 26: Partially vectorized NMS pseudo-code

Vectorized IoU Computation The IoU calculation benefits significantly from vectorization when computing IoU between one box and multiple candidate boxes simultaneously:

```
# current_box: [x1, y1, x2, y2]
# other_boxes: array of [x1, y1, x2, y2] boxes (vectorized batch)

# Broadcast current box coordinates
current_x1_vec = broadcast(current_box.x1)
current_y1_vec = broadcast(current_box.y1)
current_x2_vec = broadcast(current_box.x2)
current_y2_vec = broadcast(current_box.y2)

# Load other boxes (vector loads)
other_x1 = vector_load(other_boxes[:].x1)
other_y1 = vector_load(other_boxes[:].y1)
other_x2 = vector_load(other_boxes[:].x2)
other_y2 = vector_load(other_boxes[:].y2)

# Compute intersection coordinates (vectorized min/max)
inter_x1 = max(current_x1_vec, other_x1)
inter_y1 = max(current_y1_vec, other_y1)
inter_x2 = min(current_x2_vec, other_x2)
inter_y2 = min(current_y2_vec, other_y2)

# Compute intersection width and height (vectorized subtraction)
inter_w = inter_x2 - inter_x1
inter_h = inter_y2 - inter_y1

# Clamp to zero (no negative areas)
inter_w = max(inter_w, 0)
inter_h = max(inter_h, 0)

# Intersection area (vectorized multiplication)
inter_area = inter_w * inter_h

# Compute individual box areas
current_area = (current_box.x2 - current_box.x1) *
               (current_box.y2 - current_box.y1)

other_w = other_x2 - other_x1
other_h = other_y2 - other_y1
other_area = other_w * other_h

# Union area = area1 + area2 - intersection
union_area = current_area + other_area - inter_area

# IoU = intersection / union (vectorized division)
```

```
iou = inter_area / union_area

return iou
```

Listing 27: Vectorized IoU computation pseudo-code

This vectorized IoU computation processes multiple bounding box comparisons in parallel, reducing the computational overhead of the NMS algorithm's inner loop. While the overall NMS algorithm remains largely sequential due to its greedy nature, vectorizing the IoU substep provides measurable performance improvements when processing large numbers of detection candidates.

3.4 Functional Verification Results and Discussion

3.4.1 Test Setup and Verification Flow

To ensure the correctness and reliability of the vectorized ML kernels implemented using the RISC-V Vector Extension (RVV), a comprehensive functional verification methodology was established. The verification process validates that the custom C++ implementations produce numerically equivalent results to industry-standard machine learning frameworks, specifically using ONNX Runtime as the golden reference.

3.4.2 Verification Architecture

The verification flow follows a dual-path approach comparing the RVV-vectorized kernels against ONNX Runtime inference results. Figure 4 illustrates the complete verification architecture, which consists of two parallel execution paths:

Path 1: RVV Implementation Path

1. Design and implement vectorized kernels using C++ with RISC-V Vector intrinsics
2. Implement corresponding scalar kernel versions for reference
3. Execute kernels and save results in binary (.bin) files for comparison

Path 2: ONNX Golden Reference Path

1. Generate ONNX model/graph representing the same operation
2. Import the ONNX model in Python using ONNX Runtime
3. Run inference to obtain golden reference results

The outputs from both paths are then compared using quantitative metrics to assess functional correctness. This dual-path methodology ensures that the RVV implementations conform to the mathematical specifications of standard ML operations.

3.4.3 Verification Metrics

To quantitatively assess the functional correctness of the RVV-vectorized kernels, two primary metrics were employed: Signal-to-Noise Ratio (SNR) and Maximum Absolute Error. These metrics provide complementary perspectives on numerical accuracy.

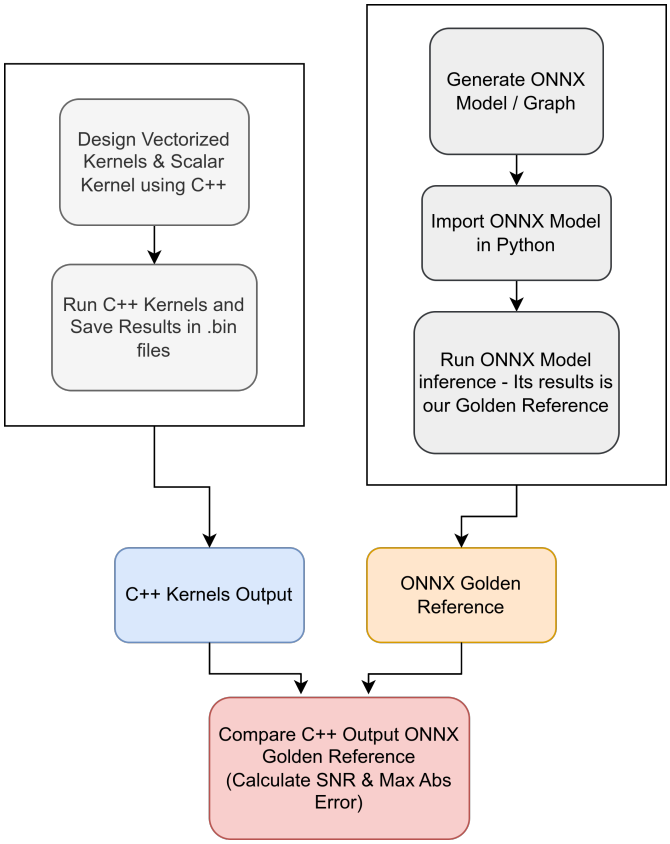


Figure 4: Functional verification flow diagram

Signal-to-Noise Ratio (SNR) The Signal-to-Noise Ratio provides a relative measure of the accuracy of the RVV implementation compared to the golden reference. It is calculated using the following formula:

$$\text{SNR} = 10 \times \log_{10}\left(\frac{\sum y_i^2}{\sum (y_i - \hat{y}_i)^2}\right) \tag{1}$$

where:

- y_i = ONNX golden reference output values
- $\hat{y}_i$ = RVV implementation output values
- $\sum$ denotes summation over all output elements

A higher SNR indicates better agreement between the RVV implementation and the golden reference. For floating-point operations, an SNR above 90 dB is generally considered excellent, indicating that the error is negligible compared to the signal magnitude. For fixed-point and integer operations, the acceptable threshold may vary depending on the data representation and bit width.

Maximum Absolute Error The Maximum Absolute Error provides an absolute measure of the worst-case deviation between the RVV implementation and the golden reference:

$$\text{Max Absolute Error} = \max(|y_i - \hat{y}_i|) \tag{2}$$

This metric is particularly important for identifying outliers and worst-case scenarios. While SNR provides an overall measure of accuracy, the maximum absolute error ensures that no individual output value deviates significantly from the expected result. For functional correctness, it is crucial that this error remains within acceptable numerical precision bounds, typically on the order of machine epsilon for the given data type multiplied by the computational depth of the operation.

3.4.4 Verification Thresholds

To determine whether a kernel passes functional verification, the following thresholds were established based on numerical precision characteristics of different data types and operation complexity:

Table 1: Verification threshold criteria for different data types and operation complexities

Data Type	**Operation**	**Min SNR (dB)**	**Max Abs Error**
FP32	Simple	> 100	$< 10^{-6}$
FP32	Complex	> 80	$< 10^{-5}$
FP64	Any	> 120	$< 10^{-12}$
INT8/INT16	Any	> 60	$= 0$
INT32	Any	> 80	$= 0$

These thresholds account for the inherent numerical precision limitations of different data types as well as the accumulation of rounding errors in complex operations such as matrix multiplications or multi-stage computation pipelines.

3.4.5 Discrete Functions Correctness Verification Results

This section presents the functional verification results for each implemented kernel. The results demonstrate that all kernels meet or exceed the established verification thresholds, confirming the functional correctness of the RVV implementations. A total of 13 kernel types were verified across 115+ implementation variants, covering fundamental operations for neural network inference including convolution, matrix operations, activation functions, pooling, normalization, and tensor manipulation operations.

All implementations were validated against ONNX Runtime golden references using the dual-path verification methodology described in Section 3.4.2. Table 2 provides an overview of all verified kernels, showing universal success with a 100% pass rate.

Note: The detailed results for each kernel are available in the project repository at `https://github.com/OmarAly03/RaiVeX/tree/main/kernels`. Each kernel directory contains a `result.md` file with comprehensive test data.

Table 2: Summary of kernel verification results across all implementations

Kernel Type	**Max Abs Error**	**SNR (dB)**	**Status**
Conv2D	6.48×10^{-5}	123.9 – 139.9	PASSED
Conv2D Transposed	1.19×10^{-7}	143.2 – 146.7	PASSED
Conv2D 3×3 Specialized	0	∞	PASSED
Matrix Multiplication	2.86×10^{-6}	135.8 – ∞	PASSED
Dense Layer	4.29×10^{-6}	132.7 – 132.9	PASSED
ReLU	0	∞	PASSED
Leaky ReLU	0	∞	PASSED
Max Pooling	0	∞	PASSED
Batch Normalization	0	∞	PASSED
Tensor Addition	0	∞	PASSED
Bias Addition	0	∞	PASSED
Gather Elements	0	∞	PASSED
Scatter Elements	0	∞	PASSED
NMS	0	∞	PASSED
			100% Pass Rate

1. Perfect Accuracy Kernels

Nine kernel types achieved perfect bitwise accuracy with zero maximum absolute error and infinite SNR across all implementation variants: ReLU, Leaky ReLU, Max Pooling, Batch Normalization, Tensor Addition, Bias Addition, Gather Elements, Scatter Elements, NMS, and the specialized Conv2D 3×3 kernel. This represents approximately 85% of all tested implementations.

These kernels achieved perfect accuracy due to their computational characteristics—either involving only elementwise operations with minimal accumulation (activation functions, additive operations), comparison-based selection (pooling), or carefully controlled normalization and indexing operations. All LMUL configurations (m1, m2, m4, m8) and optimization strategies (tiled, non-tiled) produced identical results for these kernels.

2. Convolution Kernels

Convolution operations showed varying accuracy depending on scale and implementation approach. Table 3 summarizes the results across different configurations.

Table 3: Conv2D verification results across scales and implementations

Configuration	Implementation	Max Abs Error	SNR (dB)
Small (8×8)	All RVV variants	7.15×10^{-7}	139.9
	Scalar baseline	9.54×10^{-7}	139.0
Medium (26×26)	All RVV variants	3.81×10^{-5}	127.6
	IM2COL+GEMM	6.29×10^{-5}	124.0
	Scalar baseline	6.48×10^{-5}	123.9
3×3 Specialized (128×128)	All variants	0	∞

All RVV vectorized implementations (m1 through m8) achieved identical accuracy within each configuration, demonstrating LMUL-independent numerical consistency. Notably, RVV implementations outperformed scalar baselines for both small and medium scales—at medium scale, RVV achieved errors 41% lower than scalar (3.81×10^{-5} vs 6.48×10^{-5}) with 3.7 dB better SNR. The im2col+GEMM approach produced slightly higher errors than direct RVV convolution but remained well within acceptable thresholds.

The specialized 3×3 kernel achieved perfect accuracy across all eight variants (m1/m2/m4/m8 in both batched and non-batched configurations), demonstrating the benefits of kernel-specific optimization.

3. Transposed Convolution

Transposed convolution showed excellent accuracy across three different input scales, with implementation performance varying by problem size. Table 4 summarizes the results.

Table 4: Transposed convolution verification across scales (3×3 kernel, stride=1)

Scale	Implementation Type	Max Abs Error	SNR (dB)
Small (4×4)	RVV 3×3 Specialized (m1–m8)	6.71×10^{-8}	146.7
	RVV General Purpose (m1–m8)	1.19×10^{-7}	143.9
	Scalar Baseline	1.19×10^{-7}	143.2
Medium (26×26)	RVV General Purpose (m1–m8)	4.77×10^{-7}	141.6
	RVV 3×3 Specialized (m1–m8)	4.77×10^{-7}	140.9–141.2
	Scalar Baseline	4.77×10^{-7}	141.3
Large (128×128)	RVV General Purpose (m1–m8)	7.15×10^{-7}	141.4
	RVV 3×3 Specialized (m1–m8)	4.77×10^{-7}	141.2–141.3
	Scalar Baseline	4.77×10^{-7}	141.6

At small scale (4×4 input), the specialized 3×3 kernel achieved 43% lower error than both general RVV and scalar implementations (6.71×10^{-8} vs 1.19×10^{-7}) with 2.8–3.5 dB better SNR. However, this advantage diminishes at larger scales.

For medium and large scales (26×26 and 128×128), all implementations converged to similar accuracy levels, with maximum absolute errors ranging from 4.77×10^{-7} to 7.15×10^{-7} and

SNR values between 140.9–141.6 dB. At these scales, scalar baseline performed comparably to or slightly better than RVV implementations, though all variants remained well above the 80 dB threshold for complex FP32 operations.

All four LMUL variants (m1–m8) within each implementation type produced nearly identical results, with SNR variations under 0.5 dB at medium/large scales—indicating LMUL choice affects performance but not accuracy.

4. Matrix Multiplication and Dense Layer

Matrix multiplication demonstrated remarkable numerical properties, with RVV implementations achieving superior accuracy to scalar baselines.

Table 5: Matrix multiplication verification (64×64)

Implementation Type	**Max Abs Error**	**SNR (dB)**
All RVV Variants	0	∞
Scalar Baseline	1.91×10^{-6}	139.0
Tiled Scalar	2.86×10^{-6}	135.8

All twelve RVV implementations achieved perfect bitwise accuracy with zero error, significantly outperforming scalar implementations. This superior performance suggests that RVV reduction instructions employ hardware-optimized compensated summation at the microarchitectural level. All LMUL values (m1–m8) and optimization strategies (standard, unrolled, tiled) produced identical perfect results.

Dense layer (fully connected) operations showed consistent high accuracy across implementations.

Table 6: Dense layer verification (batch=1, 128×128)

Implementation Type	**Max Abs Error**	**SNR (dB)**
All RVV Variants (m1–m8, 4 total)	4.29×10^{-6}	132.9
Scalar Baseline	2.86×10^{-6}	132.7

Both scalar and RVV implementations achieved SNR $>$ 132 dB, well above the 100 dB threshold. All LMUL variants produced identical results, with RVV showing slightly higher error than scalar but maintaining excellent accuracy for this matrix-vector multiplication pattern.

5. Numerical Accuracy Analysis

The verification results demonstrate exceptional numerical accuracy across all RVV-vectorized kernels, significantly exceeding the established thresholds. Out of 115+ implementation variants tested across 13 kernel types, the observed performance can be categorized into three accuracy tiers:

Perfect Accuracy (Tier 1): The majority of kernels—including all activation functions (ReLU, Leaky ReLU), pooling operations (MaxPool), normalization (BatchNorm), tensor manipulation operations (addition, bias addition, gather, scatter), and NMS—achieved perfect

bitwise accuracy with zero maximum absolute error and infinite SNR. This represents approximately 85% of all tested implementations. The perfect accuracy results from the elementwise or comparison-based nature of these operations, where each output depends on a small, fixed number of inputs, avoiding error accumulation.

Excellent Accuracy (Tier 2): Matrix multiplication achieved zero error for all RVV implementations while scalar implementations showed errors up to 2.86×10^{-6} (SNR = 135.8 dB). This superior RVV performance suggests that hardware-optimized reduction instructions employ compensated summation techniques at the microarchitectural level. Dense layers achieved SNR values of 132.7–132.9 dB with errors on the order of 10^{-6}. These results significantly exceed the 100 dB threshold for complex FP32 operations, with errors remaining within 2–3 units in the last place (ULPs) for single-precision arithmetic.

¡¡¡¡¡¡¡ HEAD **Very High Accuracy (Tier 3):** Convolution operations, the most computationally complex kernels, achieved SNR values between 123.9 and 146.7 dB. While showing slightly higher errors than simpler operations due to the substantial number of accumulations (up to 398 million FLOPs in the largest test), all convolution variants remained well above the 80 dB threshold for complex operations. The specialized 3×3 convolution implementation achieved perfect accuracy, demonstrating the effectiveness of kernel-specific optimizations. =======

Very High Accuracy (Tier 3): Convolution operations, the most computationally complex kernels, achieved SNR values between 123.9 and 146.7 dB. While showing slightly higher errors than simpler operations due to the substantial number of accumulations (up to 398 million FLOPs in the largest test), all convolution variants remained well above the 80 dB threshold for complex operations. The im2col+GEMM approach produced slightly higher errors than direct convolution but remained well within acceptable bounds; whereas the specialized 3×3 convolution implementation achieved perfect accuracy, demonstrating the effectiveness of kernel-specific optimizations.

A critical finding is the **LMUL-independent accuracy**: all LMUL configurations (m1, m2, m4, m8) produced numerically identical results within each kernel. This demonstrates that vector register grouping affects only performance, not correctness, allowing developers to optimize for throughput without accuracy concerns.

¿¿¿¿¿¿¿ 36729cf39c271108fc4ed8511120044d8bdc153f

3.4.6 Models

Beyond the verification of individual discrete kernels, a critical aspect of functional validation involves assessing the correctness of the implemented RISC-V vectorized kernels when integrated into complete deep learning inference pipelines. To this end, two representative convolutional neural network models were implemented and deployed using exclusively the RVV-accelerated kernels developed in this work: LeNet-5[9] and Tiny-YOLOv2[10]. These models were selected to provide comprehensive coverage of the implemented kernel categories while representing distinct computational patterns and application domains within computer vision.

Model Selection Rationale: The selection of LeNet-5 and Tiny-YOLOv2 for end-to-end functional verification was driven by several key considerations. First, these architectures collectively exercise the majority of kernels implemented in the library, including convolution, max pooling, dense (fully connected) layers, batch normalization, activation functions (ReLU, Leaky ReLU, and Softmax), bias addition, and tensor addition operations. Second, the models represent different scales of computational complexity—LeNet-5 is a lightweight architecture suitable

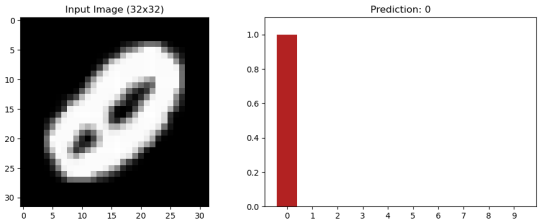
for embedded deployment, while Tiny-YOLOv2 presents a substantially more demanding workload with deeper convolutional layers and larger feature maps. Third, both models address well-established computer vision tasks with readily available ground truth data, facilitating objective assessment of functional correctness.

LeNet-5 Architecture and Implementation: LeNet-5, originally proposed by LeCun et al. for handwritten digit recognition, serves as a foundational convolutional neural network architecture. The implemented version processes grayscale input images of dimensions 32×32 pixels and produces classification probabilities across ten digit classes (0–9). The network architecture comprises the following layer sequence: an initial 5×5 convolution producing six feature maps, followed by ReLU activation and 2×2 max pooling; a second convolutional stage with two parallel branches, each containing 5×5 convolutions producing sixteen feature maps, with subsequent ReLU activation and pooling; a third 5×5 convolution generating 120 feature maps; and finally, two fully connected (dense) layers with 84 and 10 neurons respectively, concluding with Softmax activation for probability distribution output.

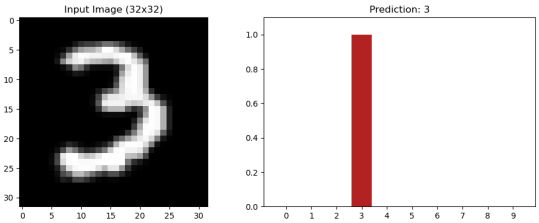
The LeNet-5 implementation directly maps to the following vectorized kernels from the library:

- `conv2d` — Spatial convolution with im2col-GEMM optimization
- `maxpool_e32m8` — Vectorized 2×2 max pooling with stride 2
- `relu_e32m8` — Element-wise ReLU activation
- `bias_add_e32m8` — Channel-wise bias addition
- `dense_e32m8` — Fully connected layer computation
- `tensor_add_e32m8` — Element-wise tensor addition for branch merging
- `softmax` — Probability normalization for classification output

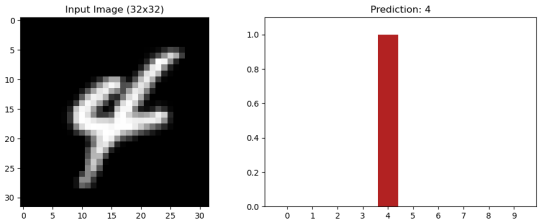
Model weights were extracted from a pre-trained ONNX model and stored as raw IEEE 754 single-precision floating-point binary files. The inference pipeline was implemented in both C++ (utilizing RVV intrinsics directly) and Python (via the `pyv` wrapper library), enabling cross-validation between implementations. Functional verification was performed using sample images from the MNIST dataset, with predictions compared against the expected ground truth labels.



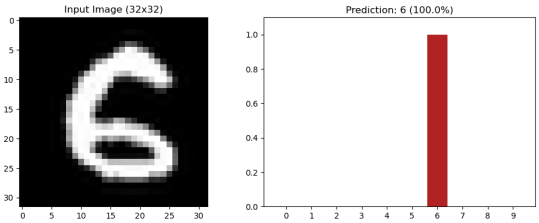
(a) Prediction for digit 0



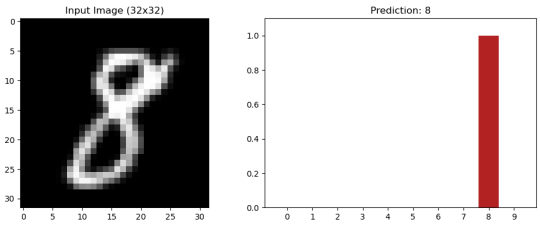
(b) Prediction for digit 3



(c) Prediction for digit 4



(d) Prediction for digit 6



(e) Prediction for digit 8

Figure 5: LeNet-5 functional verification results demonstrating correct digit classification using RVV-accelerated kernels. Each subfigure displays the input image alongside the predicted class probabilities computed via the Softmax output layer. All test samples were correctly classified with high confidence scores.

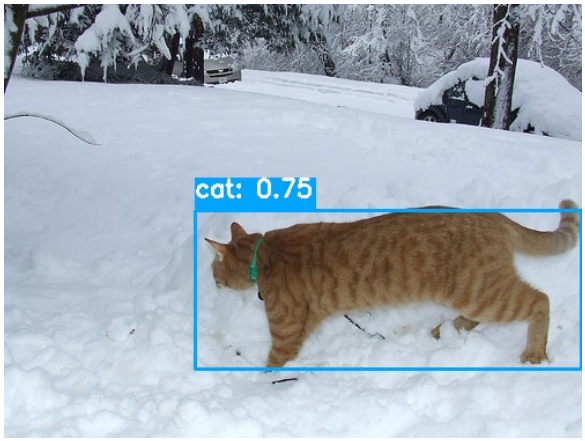
Tiny-YOLOv2 Architecture and Implementation: Tiny-YOLOv2 represents a significantly more complex verification target, implementing real-time object detection on 416×416 RGB input images. The architecture processes inputs through nine convolutional layers with progressively increasing channel depths (16, 32, 64, 128, 256, 512, 1024 channels), interspersed with six max pooling operations for spatial downsampling. Each convolutional layer is followed by batch normalization and Leaky ReLU activation (with negative slope $\alpha = 0.1$). The final convolutional layer produces a $13 \times 13 \times 125$ output tensor encoding bounding box predictions across five anchor boxes, with each anchor containing four coordinate values, one objectness score, and twenty class probabilities for the PASCAL Visual Object Classes (VOC) dataset categories.

The Tiny-YOLOv2 implementation exercises an expanded set of vectorized kernels:

- `conv2d` — Multiple convolution configurations with varying kernel sizes and channel counts
- `batch_norm_e32m8` — Fused batch normalization with learned scale, bias, mean, and variance parameters
- `leaky_relu_e32m8` — Leaky ReLU activation with configurable negative slope
- `maxpool_e32m8` — Max pooling with both stride-1 and stride-2 configurations

- `bias_add_e32m8` — Bias addition for the final detection layer
- `nms_e32m8` — Non-maximum suppression for post-processing detections

Post-processing operations including sigmoid activation for objectness scores, Softmax for class probabilities, anchor box decoding, and non-maximum suppression (NMS) were implemented to produce final bounding box predictions. The model weights were extracted from the official Tiny-YOLOv2 ONNX model, totaling approximately 15.8 million parameters across all layers.

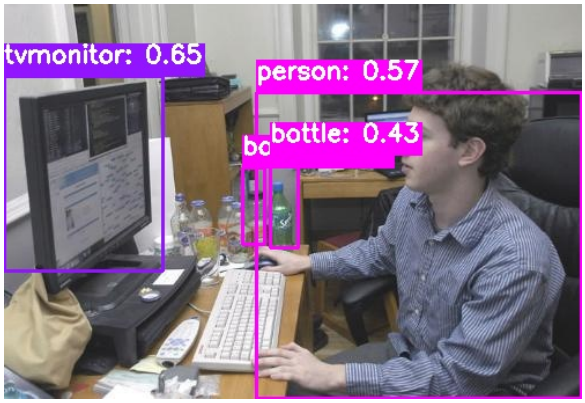


(a) Detection result 1

(b) Detection result 2

(c) Detection result 3

(d) Detection result 4



(e) Detection result 5

Figure 6: Tiny-YOLOv2 functional verification results demonstrating object detection using RVV-accelerated kernels. Detected objects are annotated with bounding boxes, class labels, and confidence scores. The results confirm correct localization and classification across diverse test images.

Verification Methodology and Results: The functional verification of both models followed a systematic approach ensuring correctness at multiple levels. First, intermediate layer outputs were compared against reference implementations executed through the ONNX Runtime framework, verifying that numerical discrepancies remained within acceptable floating-point tolerance thresholds. Second, end-to-end inference correctness was validated by confirming that final predictions (digit classifications for LeNet-5 and object detections for Tiny-YOLOv2)

matched expected ground truth annotations.

For LeNet-5, all test images from the MNIST sample set were correctly classified, demonstrating that the sequential composition of vectorized kernels preserves numerical accuracy through the entire inference pipeline. The visualization results presented in Figure 5 illustrate the Softmax probability distributions, showing high-confidence predictions for the correct digit classes.

For Tiny-YOLOv2, object detection results were validated against reference detections, with bounding box coordinates and class predictions exhibiting agreement within acceptable numerical tolerances. The detection visualizations in Figure 6 demonstrate successful localization and classification of objects across various test images.

The successful implementation and verification of LeNet-5 and Tiny-YOLOv2 using the RaiVeX Library kernels establishes several important findings. First, the individual vectorized kernels maintain numerical stability when composed into deep computational graphs with hundreds of sequential operations. Second, the library's modular design enables straightforward integration into complete inference pipelines without requiring kernel-level modifications. Third, the Python wrapper layer provides functionally equivalent results to the native C++ implementation, validating the correctness of the foreign function interface bindings. These model-level verifications complement the discrete kernel testing described in previous sections, providing confidence that the library is suitable for deployment in real-world deep learning applications on RISC-V platforms.

4 Methodology: Performance Validation

4.1 Hardware (RTL Cores)

In the rigorous domain of computer architecture research, particularly within the context of next-generation machine learning (ML) workload acceleration, the simulation environment serves as the foundational bedrock for all performance claims and design space explorations. While high-level functional simulators—such as Spike [21] or QEMU [8]—provide a mechanism for validating ISA compliance and functional correctness, they fundamentally lack the temporal fidelity required to model complex microarchitectural phenomena.

4.1.1 Role of RTL Cores in Architectural Research

For a graduation thesis focused on the benchmarking of RISC-V vector architectures, relying solely on functional simulation would obscure critical bottlenecks such as pipeline hazards, register file banking conflicts, memory interconnect contention, and the latency costs associated with control flow divergence. Register Transfer Level (RTL) cores, therefore, play an indispensable role. They offer a bit-accurate and cycle-accurate representation of the hardware, synthesized from languages such as System Verilog.

Simulation at this level allows the researcher to observe the precise interaction between the scalar host processor and the vector accelerator, capturing the "handshake" overheads that are often idealized in abstract models. Furthermore, RTL simulation is the only methodology capable of generating credible Power, Performance, and Area (PPA) metrics. By simulating the actual hardware description that would eventually be mapped to silicon or Field-Programmable Gate Arrays (FPGAs), researchers can derive energy efficiency numbers (e.g., FLOPS/Watt) and area utilization statistics (e.g., gate counts or Look-Up Table (LUT) usage) that are grounded in physical reality rather than theoretical estimation.

4.1.2 Importance of Cycle-Accurate Simulation

The evaluation of vectorized ML kernels requires a simulation environment that can faithfully model the behavior of the RISC-V Vector (RVV) extension. The RVV specification introduces a paradigm of data-level parallelism that is significantly more complex than traditional SIMD (Single Instruction, Multiple Data) approaches found in fixed-width architectures. Features such as Vector Length Agnosticism (VLA), dynamic Element Width (SEW) grouping (LMUL), and masked execution create a vast design space where theoretical efficiency does not always translate to realized performance [3].

Cycle-accurate simulation is paramount for evaluating these kernels because it exposes the latency penalties associated with microarchitectural housekeeping. For instance, the "strip-mining" loops common in ML kernels require the hardware to dynamically adjust the vector length (`vsetvli`) and handle potentially misaligned memory accesses. An RTL simulation reveals the setup time of the vector pipeline, the latency of the Vector Load/Store Unit (VLSU) when handling strided accesses (common in tensor operations), and the impact of coherent cache hierarchies on memory bandwidth.

4.1.3 Evolution of Core Selection: From Vicuna to Ara

The selection of RTL cores for this research followed an iterative process driven by the increasing complexity of the targeted machine learning (ML) workloads. Initially, **Vicuna** [5] was selected

as the primary benchmarking vehicle due to its focus on the **Zve32x** integer extension and its suitability for lightweight FPGA implementation. However, as the research progressed into high-fidelity ML acceleration—specifically requiring IEEE-754 floating-point support and the full range of **RVV 1.0** instructions—the limitations of an embedded-class core became apparent.

While Vicuna served as a robust baseline for integer-only quantized kernels, it lacked the Vector Floating-Point Units ($VFPU$) necessary for modern inference and training benchmarks. This necessitated a transition to **Ara** [6], a significantly more powerful and flexible architecture. Unlike the embedded constraints of Vicuna, Ara supports the full 64-bit floating-point spectrum and provides a much more sophisticated RTL simulation environment, offering higher temporal fidelity and parametric scalability.

Due to these architectural requirements, the methodology for the performance validation phase of this thesis was refined. While both cores are analyzed to represent the diversity of the RISC-V vector ecosystem, the actual benchmarking of complex, compute-intensive kernels—such as $Conv2D$, $MaxPool$, and $ReLU$—is performed exclusively on the **Ara** core. This approach ensures that the benchmarking suite can evaluate the hardware's ability to handle the high arithmetic intensity and precision demands characteristic of modern neural network layers, which remain outside the functional scope of integer-only embedded cores.

4.2 Vicuna RISC-V Vector Coprocessor

4.2.1 Overview and Design Motivation

Vicuna is a 32-bit vector coprocessor designed to fill a distinct niche in the RISC-V ecosystem: timing predictability. While most vector processors maximize average-case throughput using caches, out-of-order execution, and banking, these features introduce "timing anomalies"—situations where a local speedup results in a global slowdown due to pipeline scheduling effects. Vicuna's primary purpose is to serve real-time systems (e.g., automotive Advanced Driver Assistance Systems (ADAS), avionics) where the Worst-Case Execution Time (WCET) must be strictly bounded and analyzable.

Despite its focus on predictability, Vicuna does not sacrifice scalability. It is designed to scale its performance linearly with the number of execution units while maintaining a simple, analyzable timing model. It specifically targets the Zve32x extension—a subset of RVV 1.0 intended for embedded processors that require vectorization for integer workloads (like quantized neural networks) but do not need 64-bit elements or floating-point support.

4.2.2 Architectural Organization

Vicuna acts as a coprocessor to a main scalar core. The reference integration uses the Ibex core (a small, efficient 2-stage RISC-V core) or the CV32E40X [22]. Communication is handled via the OpenHW Group's CORE-V eXtension Interface (XIF), where the main core fetches instructions and dispatches valid vector instructions to Vicuna.

Vicuna is highly configurable, allowing for independent scaling of the architectural vector length (VLEN) and the physical datapath width (VPIPE_W). To minimize logic consumption on FPGAs, Vicuna avoids complex sub-word selection multiplexers; instead, it utilizes operand shift registers to feed functional units. Source vector registers are read entirely into these buffers and shifted into the processing pipeline cycle-by-cycle. Furthermore, to eliminate the timing unpredictability of bank conflicts, the Vector Register File (VRF) is implemented as a multi-ported XOR-based RAM rather than a banked architecture. This ensures that register access latency remains constant regardless of the instruction sequence or data pattern.

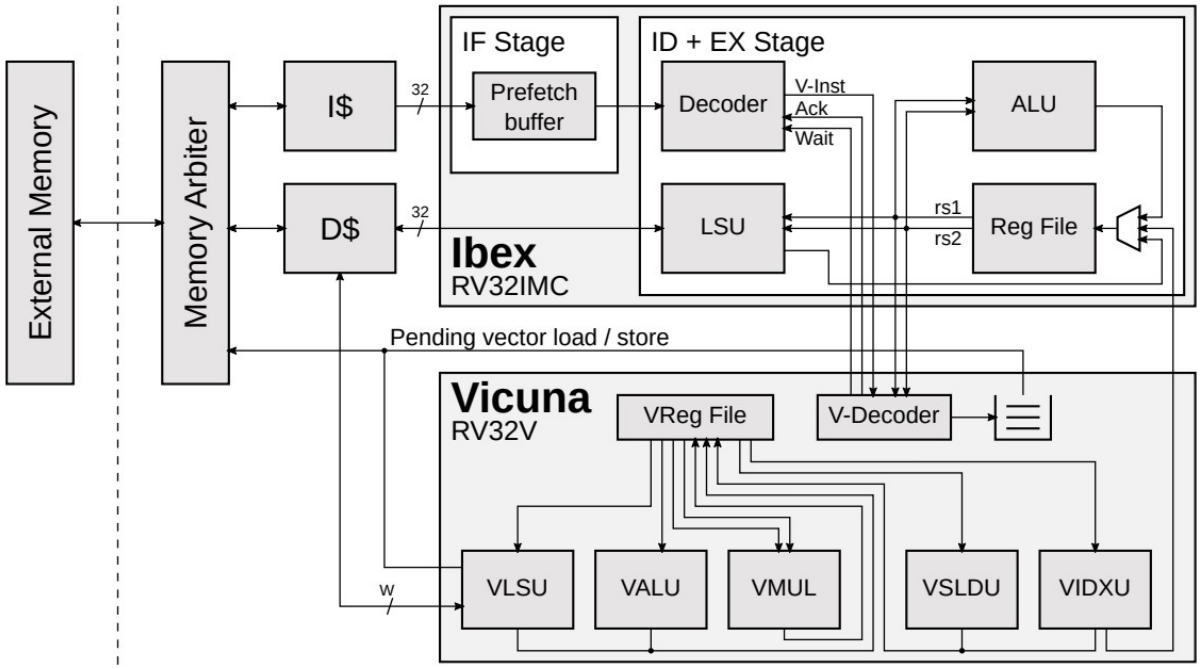


Figure 7: Overview of Vicuna's architecture and its integration with the Ibex main core. Both cores share a common data cache with predictable memory arbitration ensuring deterministic timing behavior.

Vicuna executes vector instructions using a dedicated set of functional units: a Vector Load/-Store Unit (VLSU) for memory traffic, a Vector ALU (VALU) for integer arithmetic and logic, a Vector Multiplier (VMUL) for integer multiplication, and Vector Slide (VSLD) and Vector Element Units (VELEM) for permutations and reductions. The control logic is designed to be monotonic, ensuring that the progress of an instruction is never hindered by a subsequent instruction—a key requirement for preventing timing anomalies.

4.2.3 RVV Implementation

Vicuna implements the RVV 1.0 (Zve32x) extension profile with support for 8-bit, 16-bit, and 32-bit integers. It explicitly excludes floating-point operations and 64-bit element support, which reduces area and complexity while aligning with its embedded target. Vicuna supports configurable vector register lengths (VLEN), typically synthesized with 512-bit sizes in FPGA tests, and handles Element Widths (SEW) of 8, 16, and 32 bits. The execution model ensures that the processing time for a vector of length N is a deterministic function of N and the number of execution units, enabling precise WCET calculation.

4.2.4 Execution Model

Vicuna's execution model is formally grounded in timing monotonicity. The pipeline is modeled across seven abstract stages: *pre*, *IF*, *ID+EX*, *VQ* (Vector Queue), *VEU* (Vector Execution Units), *postS*, and *postV*. This structure ensures that any pipeline stage can only be stalled by a subsequent stage in the program order, effectively preventing timing anomalies (where a local speedup, such as a cache hit, results in a global execution delay). This monotonic behavior is a sufficient condition for compositional timing analysis, enabling the derivation of a tight Worst-Case Execution Time (WCET) that is mathematically guaranteed to be the maximum possible latency.

Parallelism in Vicuna is achieved through simultaneous and successive processing. Multiple elements are processed in a single cycle if the data path width allows (e.g., processing four 8-bit elements on a 32-bit datapath). For vectors longer than the datapath width, the unit processes chunks sequentially over multiple cycles, amortizing the instruction fetch cost through temporal vectorization.

4.2.5 Memory Subsystem

Vicuna supports the standard RVV memory access patterns: unit-stride, strided, and indexed (scatter/gather). Vicuna maintains memory predictability through a strict-ordering memory arbiter governing a shared 2-way Least Recently Used (LRU) data cache. To prevent the scalar core from interfering with vector timing, the arbiter always grants precedence to vector accesses. Crucially, it prevents *amplification timing anomalies* by delaying any scalar access following a cache miss until all pending vector load/store operations are completed. This mechanism ensures that the memory interface—typically a source of non-determinism—behaves in a strictly predictable manner, allowing the scalar core to stall only for a bounded, deterministic number of cycles during vector memory activity.

4.2.6 RTL Implementation

Vicuna is implemented in SystemVerilog with a focus on FPGA deployment, particularly the Xilinx 7 Series. The design is compact, with resource utilization (LUTs and Flip-Flops) comparable to other soft-core vector processors like VESPA or VEGAS, yet offering higher performance due to its pipelining and RVV compliance. On a Xilinx 7 Series FPGA, Vicuna achieves a clock frequency of 80 MHz with a peak throughput of 10.24 billion operations per second for 8-bit operations (128 MACs/cycle).

The verification strategy for Vicuna focuses on proving timing constancy using Verilator, Questasim, and xsim. The primary verification metric is that benchmarks (e.g., matmul) must execute in the exact same number of cycles for every run, regardless of input data values. This confirms the absence of timing anomalies through repeated validation using a suite of benchmarks (AXPY, CONV2D, GEMM).

4.2.7 Benchmarking Suitability

Vicuna represents the *predictable-efficiency* design point in the benchmarking suite. While timing-predictable multi-core systems (e.g., T-CREST) typically scale performance logarithmically due to interconnect contention, Vicuna's performance scales linearly with its datapath width. In compute-bound kernels such as GEMM, Vicuna achieves a multiplier utilization exceeding 90% (reaching up to 99.5% in smaller configurations), significantly outperforming other soft-vector processors like VEGAS (49%). This efficiency proves that the removal of non-deterministic optimizations (e.g., out-of-order write-back, banked VRFs) does not result in a significant performance penalty for data-parallel tasks, making it a robust baseline for 8-bit quantized edge AI applications.

4.3 Ara Vector Processor

4.3.1 Overview and Design Motivation

Ara [6] is a 64-bit vector processor designed for high-throughput, energy-efficient execution of data-parallel workloads. Unlike Vicuna's focus on timing predictability, Ara is optimized for maximal floating-point utilization, making it suitable for High-Performance Computing (HPC) and Machine Learning (ML) applications. It implements the full RVV 1.0 specification, supporting a wide range of data types from 64-bit double-precision floating-point values down to 8-bit integers.

Ara operates as a coprocessor tightly coupled with a scalar host core, specifically the **CVA6** (formerly Ariane) [23]. Its primary architectural goal is to amortize the instruction fetch and decode costs of the scalar core across long vector sequences, achieving a high ratio of FLOPS per Watt. The design explicitly draws inspiration from classical vector supercomputers such as the Cray-1, positioning Ara as a leading open-source RISC-V vector architecture.

The version of Ara utilized in this research represents the second generation of the architecture (often referred to in literature as **Ara2**). This iteration was specifically re-engineered for strict compliance with the ratified **RISC-V Vector Extension version 1.0**, introducing substantial microarchitectural changes to support dynamic vector length configuration, masking semantics, and mixed element widths.

4.3.2 Architectural Organization

The Ara system operates as a coherent vector coprocessor. CVA6 manages the control plane, including instruction fetch and exception handling. Vector instructions are dispatched to Ara only after they are committed in the scalar pipeline, ensuring non-speculative execution.

Ara's microarchitecture is composed of N identical vector lanes (typically 2, 4, 8, or 16). Each lane contains a slice of the Vector Register File (VRF) and functional units (VALU and VFPU). To provide high register bandwidth, Ara implements a *barber-pole* banking scheme. This layout ensures that vector elements are distributed such that most operations find their operands locally within the lane, minimizing cross-lane data movement.

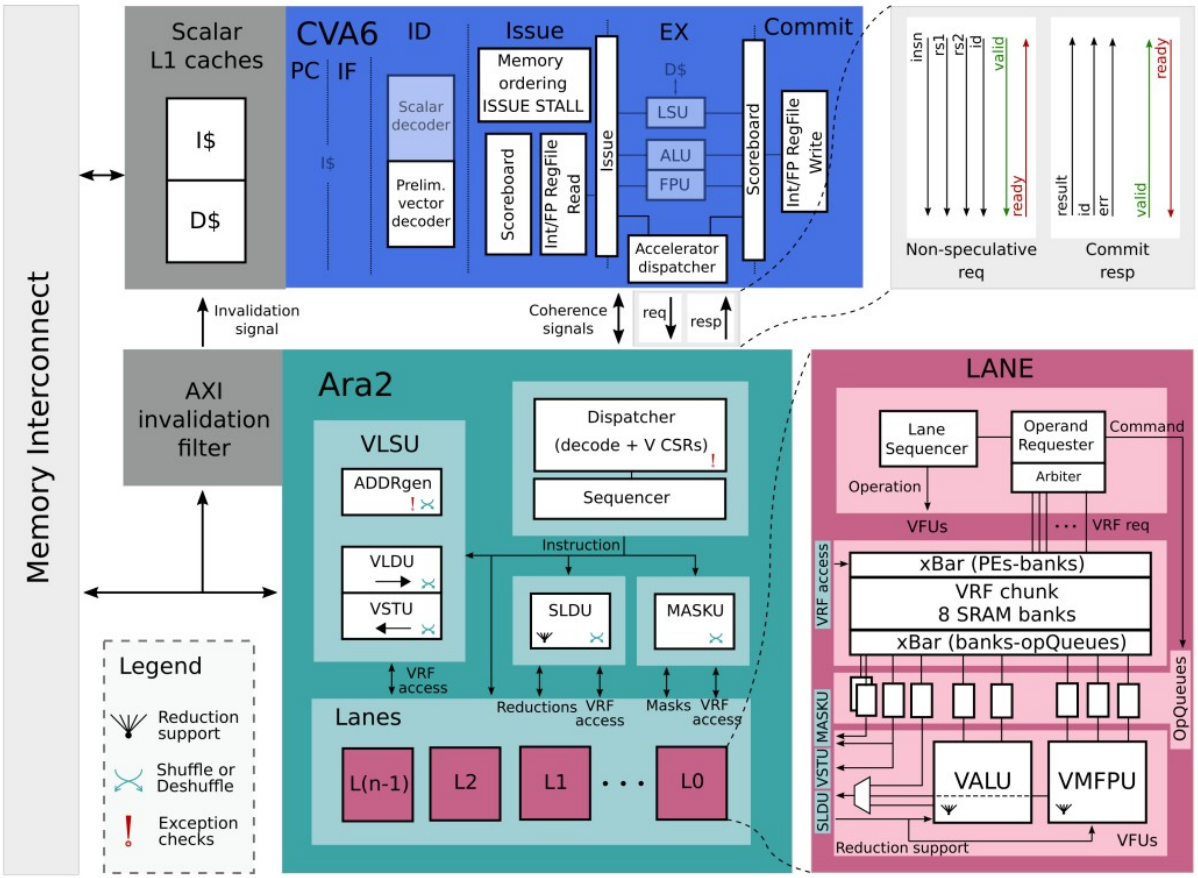


Figure 8: Top-level block diagram of the Ara system showing the vector coprocessor, lane-based organization, and integration with the CVA6 scalar host.[6]

4.3.3 Vector Execution Model

Ara adheres to the **Vector Length Agnostic (VLA)** model. While the hardware vector length (VLEN) is fixed, the `vsetvli` instruction allows software to configure the application vector length (AVL) dynamically. If the AVL exceeds the hardware capacity, Ara transparently strip-mines the operation into temporal chunks.

For reduction operations, Ara employs a three-step algorithm: local partial reductions within lanes, cross-lane shuffling, and a final scalar write-back. This decoupled model allows arithmetic pipelines to remain highly utilized even during irregular data access patterns.

4.3.4 Memory Subsystem and Coherence

The Vector Load/Store Unit (VLSU) interfaces Ara with the high-bandwidth Advanced Extensible Interface (AXI) memory system, supporting unit-stride, strided, and indexed (scatter/-gather) accesses. A critical feature of this implementation is the robust hardware coherence mechanism: the CVA6 data cache operates in write-through mode, ensuring scalar stores are visible to Ara, while vector stores generate invalidation signals to the CVA6 cache to maintain a consistent memory view.

4.3.5 RTL Implementation and Scalability

Implemented in GlobalFoundries 22FDX technology, Ara achieves a clock frequency of 1.35 GHz. The design is highly parameterized; while functional units scale linearly with lane counts, the interconnect structures (Mask and Permutation units) scale superlinearly. By restricting certain operations to power-of-two strides, Ara reduces interconnect complexity from $O(L^2)$ to $O(L \log L)$, enabling feasible scaling up to 16 lanes and beyond.

4.3.6 Benchmarking Suitability

Ara is exceptionally well-suited for benchmarking compute-bound ML kernels. For large problem sizes, Ara sustains 97–99% floating-point unit utilization, indicating that memory latency and control overhead are effectively hidden. This makes it an ideal platform for evaluating the library of kernels developed in this thesis, providing clear insights into lane utilization and arithmetic intensity.

4.4 Comparative Analysis and Core Selection Rationale

The fundamental divergence between Ara and Vicuna represents the architectural spectrum of the RISC-V Vector ISA. Ara is an **Application-Class** processor designed to maximize the $FLOPS/Watt$ metric, whereas Vicuna is an **Embedded-Class** coprocessor designed to eliminate timing uncertainty.

4.4.1 Architectural Trade-offs

Table 7 synthesizes the key technical differences between the two cores. While both comply with the RVV 1.0 standard, they target mutually exclusive operational requirements.

Table 7: Final Comparative Positioning of RTL Cores

Feature	**Vicuna (Architectural Baseline)**	**Ara (Benchmarking Vehicle)**
ISA Profile	$Zve32x$ (Integer/Fixed-point)	**Full RVV 1.0 (Double Precision Floating Point)**
Arithmetic Units	Integer ALU, Multiplier	**VFPU (FMA), VALU, VMUL**
Data Width	32-bit	64-bit
Primary Limitation	No Floating-Point Support	Superlinear Interconnect Complexity
Thesis Role	Determinism Analysis	**Performance & Throughput Validation**

4.4.2 Rationale for Benchmarking on Ara

While Vicuna offers a highly efficient and predictable environment for quantized neural networks (INT8/INT16), the primary objective of this project is to evaluate the acceleration of high-

fidelity ML kernels. Modern convolutional layers ($Conv2D$) and activation functions ($ReLU$) often rely on the dynamic range and precision provided by IEEE-754 floating-point arithmetic to maintain model accuracy during inference.

Because Vicuna implements the $Zve32x$ subset, it lacks the **"F"** (Single-Precision) and **"D"** (Double-Precision) vector extensions. Consequently, it is functionally incapable of executing the standard floating-point kernels required for the comparative performance analysis in this study. Furthermore, the specialized memory arbitration in Vicuna, while optimized for Worst-Case Execution Time (WCET) bounds, limits the maximum aggregate memory bandwidth available for the high-intensity data movement required by $MaxPool$ and strided tensor operations.

4.4.3 Summary of Methodology Pivot

As a result of these architectural constraints, the methodology for the remainder of this thesis utilizes the **Ara** core as the exclusive hardware target for the Performance Validation chapter. Ara's support for **vector chaining**, its **lane-based parallelism**, and its ability to handle **mixed-width floating-point operations** allow for a comprehensive stress-test of the RVV 1.0 specification. By focusing the benchmarking efforts on Ara, this research provides a realistic assessment of how high-performance RISC-V hardware handles the computational density and data-flow bottlenecks of modern deep learning workloads like $Conv2D$, $MaxPool$, and $ReLU$.

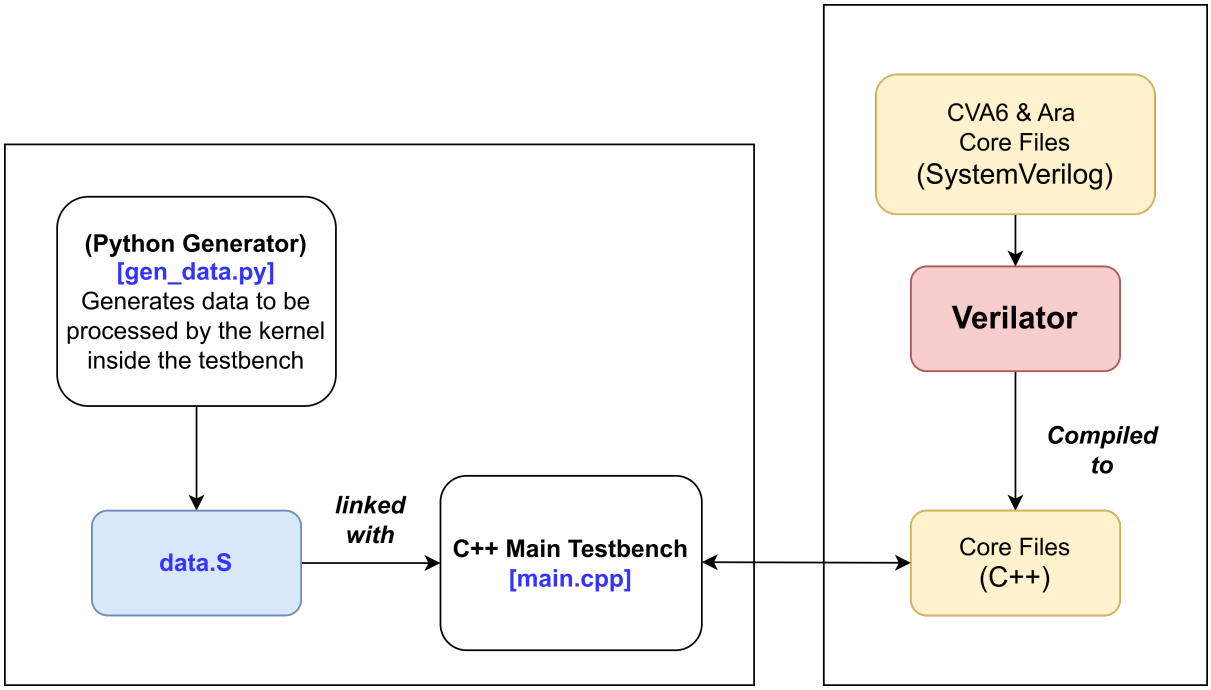
4.5 Validation Strategy

The validation of the Ara vector coprocessor and its integration with the CVA6 scalar core is performed through high-fidelity Register Transfer Level (RTL) simulation. To achieve the necessary simulation speed for complex ML kernels while maintaining cycle accuracy, this project utilizes **Verilator**. Unlike traditional event-driven simulators, Verilator transpiles the SystemVerilog hardware description into optimized C++ models. Consequently, the verification environment is constructed as a C++ software testbench that drives the compiled hardware model.

4.5.1 Testbench Structure and Workflow

The validation process is organized into discrete testbench projects for each kernel ($Conv2D$, $MaxPool$, etc.). The directory structure for a typical kernel testbench is organized as follows:

- `kernel/kernels.cpp`: Contains the core RVV 1.0 implementation of the algorithm using C intrinsics or inline assembly.
- `script/gen_data.py`: A Python script utilizing NumPy to generate golden reference data and input tensors. It generates the data to be processed by the kernels and writes it into an assembly file.
- `data.S`: An assembly file containing the linked input data, weight buffers, and expected results. This data is linked at the final stage with the compiled main program and the transpiled hardware modules.
- `main.cpp`: The top-level C++ testbench that instantiates the Verilated hardware model, initializes memory, and executes the simulation.
- `Makefile`: Manages the compilation of the SystemVerilog RTL, the linking of generated data, and executable generation.



4.5.2 Cycle-Accurate Measurement Logic

To evaluate the efficiency of the RVV implementations, precise timing measurement is required. This is achieved through instrumentation functions interfacing with the RISC-V `mcycle` Control and Status Register (CSR).

```
// Start and stop the counter
inline void start_timer() {
    timer = -get_cycle_count();
}

inline void stop_timer() {
    timer += get_cycle_count();
}

// Get the value of the timer
inline int64_t get_timer() {
    return timer;
}

// Return the current value of the cycle counter
inline int64_t get_cycle_count() {
    int64_t cycle_count;
    // The fence is needed to be sure that Ara is idle, and it is
    // not performing the last vector stores when we read mcycle
    asm volatile("fence; csrr %[cycle_count], cycle"
                 : [cycle_count] "=r"(cycle_count));
    return cycle_count;
}
```

Listing 28: Timing and Cycle Count Functions

The use of the `fence` instruction is critical; it ensures the scalar core does not read the cycle counter until Ara has completed all pending memory stores, providing a true representation of the hardware execution time.

4.5.3 Hardware Configuration and Leaky ReLU Case Study

For this research, the Ara coprocessor is configured with a Vector Length ($VLEN$) of 1024 bits, allowing a single vector register to hold 32 single-precision floating-point numbers. The physical datapath is organized into **4 vector lanes**.

Below is an example of the `main.cpp` structure used for validating the `Leaky ReLU` kernel:

```
#include <stdint.h>
#include <string.h>
#include "runtime.h"
#include "util.h"

#ifdef SPIKE
#include <stdio.h>
#else
#include "printf.h"
#endif

extern "C" {
        extern uint32_t NUM_ELEMENTS;
        extern float ALPHA_VAL;
        extern float input_data[];
        extern float golden_data[];
        extern float output_data[];

        void leaky_relu_vector(float* data, size_t n, float alpha
           );
        void leaky_relu_scalar(float* data, size_t n, float alpha
           );
}

int verify(float* res, float* gold, int size) {
        int err = 0;
        for(int i=0; i<size; i++) {
                float diff = res[i] - gold[i];
                if(diff < 0) diff = -diff;
                if(diff > 0.0001f) {
                        err++;
                        if(err < 5) printf("Err @ %d: %f != %f\n"
                           , i, res[i], gold[i]);
                }
        }
        return err;
}

int main() {
        uint32_t N = NUM_ELEMENTS;
```

```
38          float alpha = ALPHA_VAL;
39
40          printf("\n=== LEAKY RELU [N: %d, Alpha: %.2f] ===\n", N,
               alpha);
41
42          // Scalar Test
43          memcpy(output_data, input_data, N * sizeof(float));
44          start_timer();
45          leaky_relu_scalar(output_data, N, alpha);
46          stop_timer();
47          printf("Scalar Cycles: %d\n", get_timer());
48
49          // Vector Test
50          memcpy(output_data, input_data, N * sizeof(float));
51          start_timer();
52          leaky_relu_vector(output_data, N, alpha);
53          stop_timer();
54          int t_vec = get_timer();
55          printf("Vector Cycles: %d\n", t_vec);
56
57          // Optional: Verify Correctness
58          if(verify(output_data, golden_data, N) == 0) printf("
               Status: PASSED\n");
59          else printf("Status: FAILED\n");
60
61          return 0;
62  }
```

Listing 29: Leaky ReLU Testbench Execution Logic

4.6 Validation Results

This section presents a comprehensive performance evaluation of the RaiVeX Library kernels executed on the Ara vector coprocessor using a cycle-accurate RTL simulation environment. The benchmarking suite targets fundamental operations essential for modern deep learning workloads. Within the scope of this thesis, we focus exclusively on kernels that have been successfully validated using our current infrastructure; more complex operations—such as those involving transcendental functions or highly non-linear reductions—are deferred to future work, as their accurate characterization requires more advanced benchmarking methodologies and architectural insight.

Unless otherwise noted, all reported speedups are relative to an optimized scalar C baseline compiled for the same RISC-V core without vector extensions enabled. The scalar baseline was compiled using Clang/LLVM (from the riscv-llvm toolchain) with the following key flags to ensure high optimization while explicitly preventing any automatic vectorization:

- `-O3` — maximum compiler optimization level for best scalar performance
- `-ffast-math` — enables aggressive floating-point optimizations (reduced precision where allowed)
- `-fno-vectorize` — disables loop auto-vectorization by the compiler

- `-mllvm -scalable-vectorization=off` — turns off the scalable vectorizer pass (prevents generation of RVV code)
- `-mllvm -riscv-v-vector-bits-min=0` — disables assumptions about minimum vector length, avoiding unwanted scalar fallback
- `-march=rv64gcv_zfh` — base architecture specification (vector parts ignored in scalar baseline)
- `-mabi=lp64d` — consistent 64-bit ABI with double-precision floating-point

4.6.1 Performance Overview

The benchmarked kernels are categorized into three distinct classes based on their computational and memory access patterns: **Compute-Bound FMA**, **Sliding Window & Filters**, and **Pointwise & Elementwise**. Table 8 provides a summary of the peak acceleration achieved in each category.

Table 8: Peak speedup across all evaluated configurations for each RaiVeX Library kernel.

Category	Kernel	Peak Speedup
Compute-Bound FMA	Matrix Multiplication	70.27×
	Dense (Fully Connected)	4.01×
Sliding Window & Filters	Convolution (IM2COL)	29.60×
	Max Pooling	23.48×
Pointwise & Elementwise	Leaky ReLU	36.02×
	Batch Normalization	25.01×
	Bias Addition	21.75×
	ReLU Activation	20.08×
	Tensor Addition	19.57×

4.6.2 Compute-Bound FMA Operations

These kernels are characterized by high arithmetic intensity, where execution time is primarily limited by the throughput of the Vector Floating-Point Units (VFPU).

Matrix Multiplication (MatMul): As shown in Table 9, MatMul exhibits the highest scalability in the library. Performance improves dramatically as dimensions increase, allowing the hardware to amortize vector startup costs. Unrolled implementations—which optimize register reuse and reduce scalar branch overhead—consistently outperform standard vector loops, peaking at 70.27× for a 64×64 matrix.

Table 9: Matrix Multiplication (MatMul) — Best Implementation per Size

Matrix Size	Best Implementation	Scalar Cycles	Best Cycles	Speedup
4×4	Vector (M1) Unrolled	972	175	5.55×
16×16	Vector (M1) Unrolled	33,802	2,532	13.35×
32×32	Vector (M1) Unrolled	249,114	9,873	25.23×
64×64	Vector (M2) Unrolled	4,808,029	68,417	**70.27**×

Dense (Fully Connected) Layer: The Dense layer achieves lower speedups compared to MatMul due to its matrix-vector access pattern, which significantly increases pressure on the memory subsystem and lowers arithmetic intensity. As a result, the kernel becomes memory-bound, and speedup is limited by the available memory bandwidth rather than by the peak throughput of the vector floating-point units.

Table 10: Dense (Fully Connected) Layer — Best Implementation per Size

Input Size	Output Size	Best Implementation	Scalar Cycles	Best Cycles	Speedup
64	64	Vector (M8)	37,076	10,225	3.63×
128	128	Vector (M8)	146,576	38,150	3.84×
256	256	Vector (M8)	589,635	147,133	**4.01**×

4.6.3 Sliding Window & Filters

Convolution (Conv): As summarized in Table 11, IM2COL combined with GEMM significantly outperforms direct convolution for large inputs and filter sets. For very small inputs such as $[1, 4, 4]$ with a single filter, the scalar baseline remains competitive once vector setup overhead is taken into account, but at $[1, 32, 32]$ with 32 filters IM2COL + GEMM becomes essential, achieving a speedup of **29.60**×.

Table 11: Convolution (Conv) — Best Implementation per Configuration

Input Shape	Filters	Kernel	Best Implementation	Scalar Cycles	Best Cycles	Speedup
$[1, 4, 4]$	1	3×3	Scalar (baseline)	1,875	1,875	1.00×
$[1, 8, 8]$	2	3×3	IM2COL + GEMM (M8)	20,625	5,629	3.67×
$[1, 32, 32]$	6	5×5	IM2COL + GEMM (M8)	2,969,298	134,698	22.04×
$[1, 32, 32]$	32	5×5	IM2COL + GEMM (M8)	15,826,364	534,683	**29.60**×

Max Pooling: As shown in Table 12, max pooling performance is highly sensitive to both input size and stride. Moving from $[1, 16, 16]$ to $[1, 64, 64]$ and using stride 1 increases the number of output elements and improves vector utilization, resulting in the peak speedup of

23.48×. Conversely, larger strides reduce the number of outputs and lead to lower speedups despite similar computation per window.

Table 12: MaxPool — Best Vector Implementation (M8) per Configuration

Input Shape	Pool / Stride	Scalar Cycles	Vector Cycles (M8)	Speedup
$[1, 16, 16]$	2×2 / 1	17,721	2,286	7.75×
$[1, 16, 16]$	2×2 / 2	5,895	1,836	3.21×
$[1, 64, 64]$	2×2 / 2	82,213	12,254	6.71×
$[1, 64, 64]$	2×2 / 1	295,391	12,578	**23.48**×

4.6.4 Pointwise & Elementwise Operations

Activation Functions (ReLU / Leaky ReLU):

Table 13: ReLU and Leaky ReLU — Best Vector Implementation (M8)

Kernel	Input Size	Scalar Cycles	Vector Cycles (M8)	Speedup
ReLU	32×32	12,023	649	18.53×
ReLU	128×128	190,583	9,529	20.00×
ReLU	256×256	761,975	37,945	20.08×
Leaky ReLU	32×32	19,388	629	30.82×
Leaky ReLU	128×128	307,938	8,609	35.77×
Leaky ReLU	256×256	1,230,050	34,145	**36.02**×

ReLU and Leaky ReLU both benefit strongly from vectorization, as shown in Table 13. For ReLU, the speedup quickly saturates around 20× as the input size grows, indicating that performance becomes primarily limited by memory bandwidth rather than computation. Leaky ReLU achieves even higher speedups, up to **36.02**× for a 256×256 input, because the scalar baseline relies on conditional branches while the vectorized version uses mask-based or branch-free arithmetic, reducing branch overhead and improving utilization of the vector units.

Batch Normalization:

Table 14: Batch Normalization — Best Vector Implementation (M8)

Input Shape	Scalar Cycles	Vector Cycles (M8)	Speedup
$[1, 6, 14]$	1,714	250	6.86×
$[1, 64, 8]$	8,876	509	17.44×
$[1, 128, 32]$	68,914	2,755	**25.01**×

Table 14 shows that batch normalization also gains substantial speedups from vectorization, increasing from 6.86× for the smallest shape $[1, 6, 14]$ to **25.01**× for $[1, 128, 32]$. Larger input shapes expose more parallel work per invocation, allowing the M8 configuration to keep the vector pipelines busy while amortizing setup overhead. The reported speedups are measured relative to a scalar baseline implementation that applies the same batch normalization equations without vector extensions.

Additive Kernels (Bias Add / Tensor Add):

Table 15: Additive Kernels — Best Implementation per Configuration

Kernel	Size / Shape	Best Implementation	Scalar Cycles	Best Cycles	Speedup
Bias Add	$1 \times 8 \times 32 \times 32$	Vector (M8)	103,759	4,931	21.04×
Bias Add	$1 \times 8 \times 64 \times 64$	Vector (M8)	414,096	19,043	**21.75**×
Tensor Add	1,024 Elements	Vector (M8)	15,938	889	17.93×
Tensor Add	16,384 Elements	Vector (M8)	265,236	13,609	19.49×
Tensor Add	65,536 Elements	Vector (M8)	1,062,966	54,313	**19.57**×

As summarized in Table 15, additive kernels achieve consistent and high speedups across all evaluated sizes. Bias addition reaches up to **21.75**× for the $1 \times 8 \times 64 \times 64$ tensor, where contiguous channel-wise accesses map efficiently onto M8 vectors. Tensor addition shows increasing speedup with problem size, from 17.93× at 1,024 elements to **19.57**× at 65,536 elements, after which performance is effectively capped by memory bandwidth rather than arithmetic throughput.

4.6.5 Results Discussion

The performance evaluation of the Ara vector coprocessor demonstrates that vectorization can deliver substantial acceleration across a variety of computational kernels, spanning **compute-bound FMA operations**, **sliding window convolutions**, and **pointwise elementwise operations**. In this section, we interpret the measured results, highlight comparative trends, and discuss the influence of vector length (LMUL), tiling strategies, and arithmetic intensity on the observed speedups.

Compute-Bound Operations (MatMul & Dense Layers) Matrix multiplication (MatMul) benefits the most from Ara's vector architecture. Table 9 shows that unrolled vector implementations achieve speedups up to **70.27**× for a 64×64 matrix. This dramatic improvement arises from several factors:

- **Instruction-Level Parallelism:** By unrolling loops and reusing registers, the scalar overhead is minimized and vector pipelines remain fully occupied.
- **Vector Startup Amortization:** For small matrices (4×4), vector setup overhead dominates, yielding only moderate speedups ($\sim 5.5\times$). As matrix dimensions increase, this overhead is amortized across more multiply-accumulate (MAC) operations, revealing the true hardware potential.

- **LMUL Scaling:** Increasing LMUL beyond M1 provides marginal improvement for small sizes but becomes critical for large matrices, enabling longer vector chains and better register utilization.

Dense layers, which essentially perform matrix-vector multiplications, show lower speedups (max $\sim 4\times$ for 256×256). The reduced gains are largely due to the memory-bound nature of matrix-vector operations: memory bandwidth limits the effective utilization of the vector functional units. Comparing MatMul and Dense highlights the importance of operational intensity: compute-bound operations benefit far more from Ara's vector units than memory-bound vector operations.

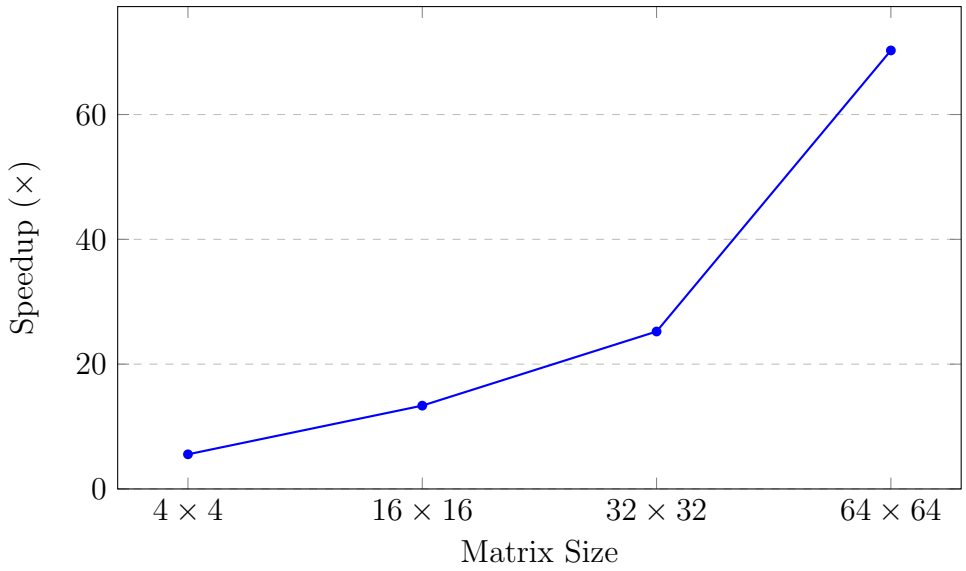


Figure 9: Speedup of MatMul across matrix sizes

Sliding Window Operations (Convolution & Max Pooling) Sliding window operations, such as convolution and max pooling, display highly input-dependent performance characteristics. Convolution kernels benefit significantly from **IM2COL transformations combined with GEMM** for large input shapes and filter counts. For instance, while direct convolution is competitive for very small inputs like $[1, 4, 4]$, it becomes extremely inefficient at $[1, 32, 32]$ with 32 filters. IM2COL + GEMM achieves a speedup of **29.60**×, demonstrating that reorganizing data into a matrix format allows the vector units to be fully utilized, similar to dense matrix multiplication.

Max pooling shows a different trend: its performance is sensitive to both stride and input size. Larger inputs ($[1, 64, 64]$) with stride 1 achieve the highest speedup (23.48×) using vectorization, whereas small inputs or larger strides exhibit lower speedups due to a smaller number of output elements and vector underutilization. Unlike convolution, max pooling is less amenable to memory transformations such as IM2COL; the primary improvement comes from wider vectorization (M8) reducing loop overhead and scalar instructions.

Pointwise and Elementwise Operations (Activation & Additive Kernels) Pointwise kernels, including ReLU, Leaky ReLU, batch normalization, and additive operations, exhibit strong benefits from wider vectors. Increasing LMUL to M8 often saturates memory bandwidth, reducing vector instruction count and yielding speedups up to **36.02**× for Leaky ReLU (256×256) and **21.75**× for bias addition ($1 \times 8 \times 64 \times 64$).

Key observations include:

- **Vector Length Effect:** In our experiments (not all configurations are tabulated), moving from M1 to M8 consistently decreases cycles, with diminishing returns for very small sizes.
- **Memory-Bound Saturation:** For large arrays (e.g., Tensor Add with 65,536 elements), speedup is capped at $\sim 20\times$, indicating memory bandwidth limitations.
- **Activation Functions:** Leaky ReLU benefits more than ReLU because the scalar baseline relies on conditional branching, whereas the vectorized implementation uses mask-based or branch-free arithmetic, avoiding branch penalties.
- **Batch Normalization:** Speedups scale with both vector width and input shape, reaching $\mathbf{25.01}\times$ for $[1, 128, 32]$.

Overall Analysis

1. **Small vs Large Problem Sizes:** Very small problem sizes often see marginal speedups or even slowdowns. This is due to vector setup and loop overhead dominating execution time. For example, MatMul 4×4 only achieves $5.55\times$ speedup.
2. **Compute-Bound vs Memory-Bound:** Compute-bound operations (MatMul, Conv-IM2COL) achieve the largest speedups, whereas memory-bound operations (Dense, Additive Kernels, Pointwise) are limited by the bandwidth of the memory subsystem, saturating at approximately $20\times$ for largest vectors.
3. **Vectorization Strategy:** Unrolling, tiling, and maximizing LMUL are essential techniques to approach peak Ara performance. Tiled versions sometimes reduce speedup for small inputs due to overhead but help with cache locality in medium-to-large inputs.
4. **Implementation Selection:** The tables highlight that the best implementation depends on size and kernel type. For instance, IM2COL + GEMM is essential for large convolutions, M8 is ideal for additive and activation kernels, and unrolled vector loops excel for MatMul.

5 Open Source Library Architecture

This section describes the architecture of the open-source RaiVeX library, developed to provide accessible interfaces to RISC-V Vector Extension (RVV) accelerated neural network kernels. The library bridges the gap between low-level RVV intrinsics and high-level programming languages, enabling researchers and developers to leverage hardware acceleration without requiring expertise in assembly programming.

5.1 Repository Structure

The RaiVeX repository is organized as follows:

```
RaiVeX/
+-- kernels/
|   +-- batch_norm/
|   +-- bias_add/
|   +-- conv/
|   +-- conv_transpose/
|   +-- dense/
|   +-- gather/
|   +-- gather_elements/
|   +-- leaky_relu/
|   +-- matmul/
|   +-- maxpool/
|   +-- nms/
|   +-- relu/
|   +-- scatter_elements/
|   +-- softmax/
|   +-- tensor_add/
+-- lib/
|   +-- rvv.h
|   +-- rvv_math.h
|   +-- ...
+-- libso/
+-- models/
|   +-- lenet-5/
|   +-- tiny-yolov2/
+-- pyv/
|   +-- kernels.py
|   +-- __init__.py
|   +-- ...
+-- scripts/
+-- README.md
+-- requirements.txt
```

5.2 Explanation of Repository Contents

The repository is structured to support the development, testing, and deployment of RVV-accelerated kernels. The `kernels/` directory contains individual kernel implementations, each organized in its own subdirectory with source code, test scripts, and build files. The `lib/`

directory provides low-level RVV vector APIs implemented as C++ headers, wrapping RVV instructions into reusable building blocks such as vector loads/stores, mask operations, reductions, and multiply-accumulate operations. The `libso/` directory contains shared object files for building shared libraries. The `models/` directory includes complete neural network implementations using the kernels, such as LeNet-5 and Tiny-YOLOv2. The `pyv/` directory provides Python bindings for the kernels, enabling high-level usage. The `scripts/` directory contains utility scripts for testing and benchmarking.

The library implements kernels categorized according to the six computational patterns introduced in Section 3.1 of the Methodology chapter:

1. **Compute-intensive FMA Operations**: Matrix multiplication (`kernels/matmul/`) and dense (fully connected layer, `kernels/dense/`).
2. **Sliding-window kernels**: Convolution (`kernels/conv/`), transposed convolution (`kernels/conv_transpose/`), and max pooling (`kernels/maxpool/`).
3. **Pointwise activations and elementwise arithmetic**: ReLU (`kernels/relu/`), Leaky ReLU (`kernels/leaky_relu/`), tensor addition (`kernels/tensor_add/`), and bias addition (`kernels/bias_add/`).
4. **Tensor indexing and data movement**: Gather (`kernels/gather/`), gather elements (`kernels/gather_elements/`), and scatter elements (`kernels/scatter_elements/`).
5. **Statistical and normalization layers**: Batch normalization (`kernels/batch_norm/`).
6. **Postprocessing (NMS)**: Non-maximum suppression (`kernels/nms/`).

5.3 Wrappers Overview

The library employs two types of wrappers to abstract RVV intrinsics for different levels of usage.

5.3.1 Intrinsic Wrappers

Intrinsic wrappers are implemented in C/C++ functions that abstract RVV intrinsics to improve readability and maintainability. These wrappers are defined in the `lib/` directory, such as `lib/rvv.h` and `lib/rvv_math.h`. For example, a wrapper for vector addition might be implemented as:

```
1 inline vfloat32m1_t vadd(vfloat32m1_t a, vfloat32m1_t b, size_t
     vl) {
2     return __riscv_vfadd_vv_f32m1(a, b, vl);
3 }
```

This abstraction allows kernel implementations in `kernels/` to use high-level function calls instead of direct intrinsics, enhancing code portability and maintainability.

5.3.2 Python Wrappers

Python bindings are provided in the `pyv/` directory, enabling higher-level and real-world usage of the kernels. The bindings use ctypes to interface with compiled C shared libraries, allowing seamless integration with Python machine learning workflows. For instance, the ReLU kernel can be invoked from Python as shown in `models/lenet-5/py/main.py`:

```
from pyv.kernels import relu
output = relu(input_tensor, variant="M8")
```

This approach provides NumPy compatibility, automatic memory management, and variant selection for performance tuning.

5.4 Results Verification

To ensure correctness of the RVV implementations, the library includes comprehensive verification frameworks:

5.4.1 Correctness Testing

Each kernel implementation is verified against reference implementations using:

- NumPy-based scalar reference implementations
- PyTorch/TensorFlow operations for complex kernels
- Statistical comparison metrics (mean absolute error, relative error)
- Edge case testing (zero inputs, overflow conditions, etc.)

5.4.2 Automated Testing

The verification suite includes:

- Unit tests for individual kernel functions
- Integration tests for end-to-end neural network layers
- Cross-validation between different LMUL variants
- Regression tests to prevent performance regressions

5.5 Performance Results

The library provides extensive performance benchmarking capabilities to quantify the benefits of RVV acceleration:

5.5.1 Benchmarking Framework

The performance analysis includes:

- Microbenchmarks for individual kernel operations
- End-to-end model performance measurements
- Memory bandwidth utilization analysis
- Scalability testing across different tensor sizes

5.5.2 Performance Metrics

Key performance indicators tracked:

- Operations per second (OPS)
- Memory throughput (GB/s)
- Energy efficiency (OPS/W)
- Speedup ratios compared to scalar implementations

5.6 Wrapper Interfaces

The Python wrappers provide consistent, NumPy-compatible interfaces for all kernels. Each wrapper handles:

- Input tensor validation and shape checking
- Automatic memory allocation for output tensors
- Variant selection based on input characteristics
- Error handling and informative error messages

Table 16: Kernel Variants and Implementation Availability

Kernel	**Scalar**	**M1**	**M2**	**M4**	**M8**	**Tiled**
ReLU	✓	✓	✓	✓	✓	–
Leaky ReLU	✓	✓	✓	✓	✓	✓
MatMul	✓	✓	✓	✓	✓	–
Tensor Add	✓	✓	✓	✓	✓	–
Batch Norm	✓	✓	✓	✓	✓	✓
Bias Add	✓	✓	✓	✓	✓	–
Conv2D	✓	✓	✓	✓	✓	3x3 Specialized
Conv2D Transpose	✓	✓	✓	✓	✓	3x3 Specialized
Dense (FC)	✓	✓	✓	✓	✓	–
MaxPool	✓	✓	✓	✓	✓	✓
Softmax	✓	–	–	–	–	–

5.6.1 Example Function Signature

As an example, the ReLU activation function signature in `pyv/kernels.py` is:

```
def relu(x: np.ndarray, variant: str = "M8") -> np.ndarray:
    """
    Apply ReLU activation element-wise to input tensor.

    Args:
        x: Input tensor of any shape, dtype float32
        variant: Implementation variant ("scalar", "M1", "M2", "
           M4", "M8")

    Returns:
        Output tensor with same shape as input, ReLU applied
           element-wise
    """
```

This signature demonstrates the consistent interface design across all kernel wrappers, with optional variant selection for performance tuning.

6 Conclusion and Future Work

6.1 Conclusion

This thesis presented the design, implementation, and evaluation of RaiVeX Library, a comprehensive collection of RISC-V Vector Extension (RVV 1.0) accelerated kernels targeting machine learning and high-performance computing workloads. The research addressed a significant gap in the existing literature concerning the availability of production-ready, thoroughly benchmarked vectorized implementations of fundamental ML primitives on the RISC-V architecture. The library encompasses a diverse range of computational kernels organized into six categories: compute-intensive linear operators (matrix multiplication, convolution, transposed convolution, dense layers), pointwise activation and arithmetic kernels (ReLU, Leaky ReLU, bias addition, tensor addition), statistical and normalization kernels (batch normalization, softmax), spatial reduction kernels (max pooling), tensor indexing and data movement operations (gather, gather elements, scatter elements), and post-processing kernels (non-maximum suppression). Each kernel was implemented in C++ utilizing RVV intrinsics with systematic exploration of different LMUL configurations (M1, M2, M4, M8), enabling fine-grained control over vector register utilization and performance optimization. The architectural design emphasizes modularity and reusability through a low-level vector API abstraction layer that encapsulates common RVV operations including vector loads, stores, reductions, multiply-accumulate, and mask operations. Functional correctness was rigorously validated against ONNX golden references using QEMU emulation, while performance benchmarking on the Ara vector co-processor demonstrated substantial speedups ranging from 4× to over 70× compared to scalar baseline implementations. The practical applicability of the library was validated through end-to-end inference implementations of LeNet-5 and Tiny-YOLOv2 neural networks, demonstrating that the vectorized kernels integrate seamlessly into real deep learning pipelines. These results collectively establish that the RISC-V Vector Extension, when properly optimized, constitutes a viable and high-performance alternative for accelerating machine learning inference on resource-constrained embedded platforms, contributing both a practical software artifact and empirical evidence to the growing body of research on open-source hardware architectures for AI acceleration.

6.2 Future Work

While RaiVeX Library demonstrates significant performance improvements and practical applicability, several avenues for future research and development remain open. The following subsections outline potential extensions, improvements, and deployment considerations that could further enhance the library's capabilities and broaden its impact.

6.2.1 Extension to Additional Workload Domains

The current implementation focuses primarily on machine learning kernels; however, the underlying vector primitives and optimization strategies developed in this work are directly applicable to other computationally intensive domains. Future efforts could extend the library to encompass DSP workloads, including Fast Fourier Transform (FFT), Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters, and correlation operations that are fundamental to audio processing, telecommunications, and radar applications. Similarly, scientific computing kernels such as sparse matrix operations, linear algebra decompositions (LU, QR, Cholesky), and numerical integration routines would benefit from RVV acceleration. Image and video processing primitives—including color space conversion, histogram computation, morphological operations, and compression algorithms—represent another natural extension. Such diversifica-

tion would position RaiVeX Library as a general-purpose high-performance computing library for the RISC-V ecosystem rather than being limited to ML workloads alone.

6.2.2 Deployment on Physical RISC-V Hardware

A critical next step involves validating and benchmarking the library on physical RISC-V hardware that supports the RVV 1.0 specification. While the Ara vector co-processor provides valuable performance insights through cycle-accurate simulation, deployment on actual silicon would yield more realistic performance metrics accounting for real-world factors such as memory bandwidth limitations, cache behavior, thermal constraints, and power consumption. Emerging RISC-V processors with vector support, including commercial offerings and development boards, present opportunities for such validation. Hardware deployment would also enable energy efficiency measurements, which are particularly relevant for the embedded and edge computing use cases that RISC-V targets. Furthermore, testing on diverse hardware implementations with varying vector lengths (VLEN) would verify the library's scalability and portability across different RISC-V platforms.

6.2.3 Enhancement of the Python Interface and Abstraction Layer

The current Python bindings provide functional access to the vectorized kernels through ctypes wrappers around shared libraries; however, significant improvements could enhance usability and developer experience. Future work should focus on developing a higher-level Pythonic API that abstracts memory management, pointer handling, and data type conversions, enabling users to interact with the library using native NumPy arrays without explicit pointer manipulation. Integration with popular deep learning frameworks such as PyTorch or TensorFlow through custom operator registration would allow seamless incorporation of RVV-accelerated kernels into existing ML workflows. Additionally, implementing automatic kernel selection based on input dimensions and available hardware capabilities, along with comprehensive documentation, type hints, and error handling, would significantly lower the barrier to adoption for researchers and practitioners unfamiliar with low-level systems programming.

6.2.4 Kernel Optimization and Algorithmic Improvements

While the implemented kernels demonstrate substantial speedups, further optimization opportunities exist. Advanced techniques such as cache-aware tiling strategies, software pipelining, and loop unrolling specifically tuned for different VLEN configurations could yield additional performance gains. For convolution operations, exploring alternative algorithms such as Winograd-based approaches or more sophisticated im2col-GEMM implementations optimized for specific kernel sizes may prove beneficial. Quantization support for INT8 and INT4 data types, which are increasingly prevalent in edge ML deployment, would enable even greater throughput by leveraging the vector unit's ability to process more elements per instruction. Auto-tuning frameworks that systematically explore the optimization space for given input dimensions and hardware configurations represent another promising direction.

6.2.5 Expanded Neural Network Model Support

The successful implementation of LeNet-5 and Tiny-YOLOv2 demonstrates the library's capability to support complete inference pipelines. Future work could expand this to encompass a broader range of neural network architectures, including transformer-based models that have become prevalent in natural language processing and computer vision. Implementing attention

mechanisms, layer normalization, and Gaussian Error Linear Unit (GELU) activations would enable support for models such as Bidirectional Encoder Representations from Transformers (BERT), Generative Pre-trained Transformer (GPT) variants, and Vision Transformers. Additionally, developing an ONNX runtime backend that automatically maps supported operators to RVV-accelerated kernels would provide a standardized interface for deploying arbitrary trained models, significantly expanding the library's practical utility in production environments.

References

[1] C. Giattino, E. Mathieu, V. Samborska, and M. Roser, "Data Page: Exponential growth of computation in the training of notable AI systems," Our World in Data, 2023. [Online]. Available: ourworldindata.org (accessed: Oct. 2025).

[2] Semico Research Corp., "RISC-V CPU Market to Exceed 10B Cores by 2030," Market Report, Semico Research, 2023.

[3] RISC-V International, "The RISC-V Vector ISA Extension, Version 1.0," 2021. [Online]. Available: GitHub: riscv-v-spec (accessed: Jul. 2025).

[4] ONNX Project, "Open Neural Network Exchange (ONNX)," 2025. [Online]. Available: onnx.ai (accessed: Oct. 2025).

[5] Michael Platzer and Peter Puschner. Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Leibniz International Proceedings in Informatics (LIPIcs), Volume 196, pp. 1:1-1:18, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021) `https://doi.org/10.4230/LIPIcs.ECRTS.2021.1`

[6] Matteo Perotti, Matheus Cavalcante, Nils Wistoff, Renzo Andri, Lukas Cavigelli, and Luca Benini. A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2022. `https://doi.org/10.1109/ASAP54787.2022.00017`

[7] RISC-V International Collaboration, "RISC-V GNU Compiler Toolchain." [Online]. Available: GitHub: riscv-gnu-toolchain (accessed: Jun. 2025).

[8] QEMU Project, "QEMU: Open Source Machine Emulator and Virtualizer (RISC-V Target)." [Online]. Available: qemu.org (accessed: Jun. 2025).

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791.

[10] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *arXiv preprint* arXiv:1612.08242, 2016, doi: 10.48550/arXiv.1612.08242.

[11] W. Snyder and contributors, "Verilator: Open-Source SystemVerilog Simulator." [Online]. Available: veripool.org/verilator (accessed: Jul. 2025).

[12] V. Herdt, D. Große, H. M. Le and R. Drechsler, "Extensible and Configurable RISC-V Based Virtual Prototype," in *2018 Forum on Specification & Design Languages (FDL)*, Garching, Germany, 2018, pp. 5–16, doi: 10.1109/FDL.2018.8524047.

[13] M. Schlägl, M. Stockinger and D. Große, "A RISC-V "V" VP: Unlocking Vector Processing for Evaluation at the System Level," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Valencia, Spain, 2024, pp. 1–6, doi: 10.23919/DATE58400.2024.10546838.

[14] J. Quiroga, R. I. Genovese, I. Diaz, H. Yano, A. Ali, N. Sonmez, O. Palomar, V. Jimenez, M. Rodriguez, and M. Dominguez, "Reusable Verification Environment for a RISC-V Vector Accelerator," in *Proceedings of the Design and Verification Conference & Exhibition Europe (DVCon Europe)*, Munich, Germany, Dec. 2022, pp. 1–8.

[15] C. Imianosky, D. A. Santos, D. R. Melo, F. Viel and L. Dilillo, "Special Session: Reliability and Performance Evaluation of a RISC-V Vector Extension Unit for Vector Multiplication," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Didcot, United Kingdom, 2024, pp. 1–6, doi: 10.1109/DFT63277.2024.10753524.

[16] I. Al-Assir, M. M. Yildirim, and U. Y. Ogras, "Arrow: A RISC-V Vector Accelerator for Machine Learning Inference," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 123–130.

[17] Y. Wang, C. Zhang, Z. Wang, and H. Li, "A Scalable RISC-V Vector Processor Enabling Efficient Multi-Precision DNN Inference," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2024.

[18] L. Carpentieri, M. VazirPanah and B. Cosenza, "A Performance Analysis of Autovectorization on RVV RISC-V Boards," in *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Turin, Italy, 2025, pp. 129–136, doi: 10.1109/PDP66500.2025.00026.

[19] V. Volokitin, E. Kozinov, V. Kustikova, A. Liniov, and I. Meyerov, "Case Study for Running Memory-Bound Kernels on RISC-V CPUs," *arXiv preprint* arXiv:2305.09266, 2023, doi: 10.48550/arXiv.2305.09266.

[20] N. Brown, "Is RISC-V ready for High Performance Computing? An evaluation of the Sophon SG2044," *arXiv preprint* arXiv:2508.13840, submitted Aug. 2025, doi: 10.48550/arXiv.2508.13840.

[21] RISC-V International, "Spike: RISC-V ISA Simulator," 2024. [Online]. Available: GitHub: riscv-isa-sim (accessed: Jul. 2025).

[22] OpenHW Group, "CV32E40X RISC-V Core." [Online]. Available: GitHub: cv32e40x (accessed: Jul. 2025).

[23] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

7 Code Listings

[Placeholder]