Contents

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 RVV64_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

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.

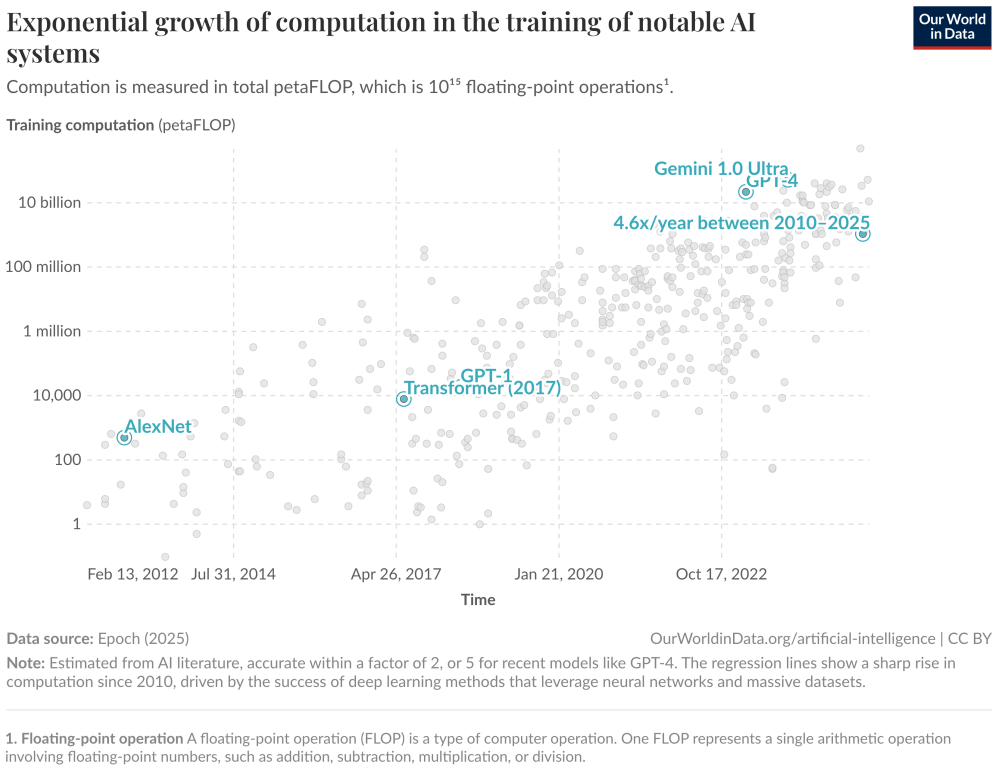


Figure 1: Exponential growth in AI model computational requirements over time, showing the dramatic increase in FLOPs required for training state-of-the-art models.

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 (CUDA, 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 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—Single Instruction, Multiple Data). 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 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. 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 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 RTL simulation environments for development and validation. This creates additional complexity in establishing reproducible benchmarking methodologies that can provide meaningful performance insights.

1.5 Research 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 optimized RVV kernels** for fundamental machine learning and DSP operations, including:
    - Matrix operations: multiplication, transposition, addition
    - Activation functions: ReLU, Softmax
    - Convolutional layers
    - Training kernels: linear regression gradient descent
3. **Establish functional verification methodologies** using ONNX (Open Neural Network Exchange) as a golden reference framework, ensuring correctness through quantitative metrics such as Signal-to-Noise Ratio (SNR) and Maximum Absolute Error.
4. **Conduct cycle-accurate performance evaluation** using RTL simulation with the Ara and Vicuna 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 Research 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** (RVV64_Library) implementing fundamental operations for neural network inference and signal processing. The library demonstrates efficient use of RVV features including strip-mining loops, vector length agnostic programming, LMUL configuration, and masked operations.
3. **Comprehensive performance characterization** of vectorized kernels on two distinct RTL platforms:
    - **Ara**: A high-performance 64-bit vector coprocessor targeting application-class workloads with configurable lane counts (2–16 lanes) and full floating-point support.
    - **Vicuna**: A timing-predictable 32-bit vector coprocessor targeting embedded real-time systems with deterministic execution guarantees.

4. **Quantitative analysis** demonstrating significant speedups achieved through vectorization, with experimental results showing up to 3.6
   times improvement for matrix multiplication and 2.6
   times for ReLU activation compared to scalar implementations.

5. **A containerized development environment** (Docker-based) ensuring reproducibility across different development systems and simplifying the onboarding process for future researchers working with RISC-V vector extensions.

1.7 Thesis Organization

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

Chapter 2: Background and Related Work establishes the necessary theoretical foundation, providing a detailed overview of the RISC-V ISA, its relevant extensions, and the architectural principles of the Vector (RVV) extension. This chapter reviews the vector-length agnostic programming model, control and status registers, and the rich instruction set that enables efficient data-parallel processing. Additionally, it examines foundational academic research on RVV for machine learning acceleration, including work on specialized processors and prior vectorization efforts.

Chapter 3: Development Tools outlines the practical tools and workflows established for implementation and validation. This includes the RISC-V GNU toolchain configuration, QEMU emulator setup for functional testing, Ara RTL simulation environment, Docker containerization for reproducible development, and ONNX framework integration for functional verification.

Chapter 4: Methodology presents the systematic three-step framework that forms the core of this research: (1) kernel design using RVV C-intrinsics with vector-length agnostic programming, (2) functional verification against ONNX golden references using quantitative metrics (SNR and MaxAbs), and (3) cycle-accurate performance evaluation using RTL simulation with Verilator. This chapter details the kernel selection criteria, vectorization design approach, verification workflow, and performance measurement techniques.

Chapter 5: Library introduces the RVV64_Library, our collection of optimized RISC-V vector kernels for machine learning and DSP applications. This chapter systematically presents each implemented kernel—including matrix operations (multiplication, transposition), activation functions (ReLU, Softmax), convolutional layers, and training kernels (linear regression)—with detailed explanations of the naive sequential approach, proposed RVV-based solution, and performance optimization strategies.

Chapter 6: Hardware (RTL Cores) details the two RTL platforms used for cycle-accurate performance evaluation: Ara and Vicuna. This chapter explains the role of RTL simulation in architectural research, describes the microarchitectural details of each coprocessor (Ara for high-performance throughput-oriented workloads and Vicuna for timing-predictable embedded systems), and justifies their selection as representative platforms spanning the full spectrum of RISC-V vector implementations.

Chapter 7: Results and Discussion presents both theoretical analysis and empirical evaluation of the implemented kernels. This chapter provides complexity analysis, theoretical speedup calculations, and experimental results from RTL simulation on both Ara and Vicuna platforms. Quantitative performance improvements are analyzed, including achieved speedups, functional unit utilization, and the impact of different LMUL configurations. The discussion interprets these results in the context of ML/DSP application deployment scenarios.

Chapter 8: Conclusion summarizes the key findings of this research, reflecting on the demonstrated effectiveness of the RISC-V Vector Extension for accelerating data-parallel workloads. This chapter synthesizes the contributions made through our systematic framework, optimized kernel library, and comprehensive performance characterization across diverse hardware platforms.

Chapter 9: Future Work outlines promising directions for extending this research, including algorithmic expansion to additional ML operators (pooling, normalization, attention mechanisms), hardware-level validation on FPGA and ASIC platforms, integration with established ML frameworks (TensorFlow Lite, ONNX Runtime), and exploration of multi-core vector configurations for improved scalability.

Supporting materials are provided in the final sections, including acknowledgments of collaborators and advisors, a comprehensive reference list, and complete source code listings for all implemented kernels in the appendix.

2 Background & Related Work

2.1 RISC-V Architecture Overview

RISC-V (pronounced "risk-five") is an open-source instruction set architecture (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, 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.

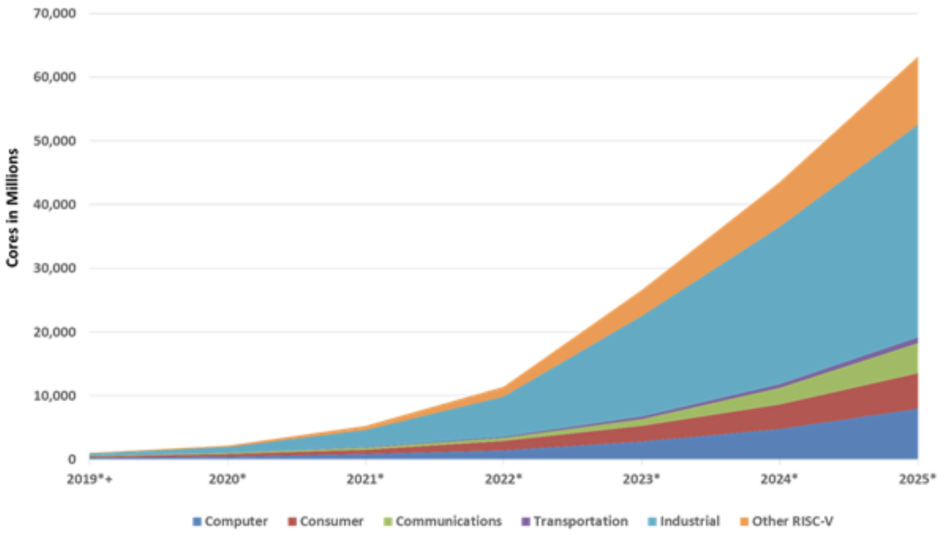


Figure 2: RISC-V ecosystem growth. Source: RISC-V International

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

M Extension (Integer Multiplication and Division): The M extension adds integer multiplication, division, and remainder operations that are fundamental for many AI and DSP algorithms. In AI applications, integer multiplication is crucial for quantized neural networks that use integer arithmetic instead of floating-point operations to reduce power consumption and increase performance.

F Extension (Single-Precision Floating-Point): The F extension provides IEEE 754 single-precision (32-bit) floating-point arithmetic, which is the most commonly used precision for AI training and many DSP applications. The extension includes fused multiply-add (FMA) instructions that are particularly important for AI and DSP workloads, as convolution operations in neural networks consist primarily of multiply-accumulate patterns that can be efficiently implemented using FMA instructions.

D Extension (Double-Precision Floating-Point): The D extension adds IEEE 754 double-precision (64-bit) floating-point arithmetic, which is important for AI training applications that require higher numerical precision and certain DSP applications that need extended dynamic range.

V Extension (Vector Operations): The V extension is the most significant addition to RISC-V for AI and DSP applications, providing comprehensive support for data-parallel vector operations. This extension represents a fundamental advancement in vector processing architecture and is the primary focus of this work.

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.

This subsection expands the descriptions for each of the selected papers to reflect their specific scientific contributions to the RISC-V and hardware–software co-design communities.

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

This paper introduces the first open-source, extensible **Virtual Prototype (VP)** based on the RISC-V architecture, implemented using standard-compliant **SystemC and 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.

The VP architecture is built around a **RISC-V RV32IM core** and a generic bus system enabling easy platform reconfiguration. System-level features include a **PLIC-based interrupt controller**, a core-local interrupt controller (CLINT), and peripherals such as a DMA controller and a terminal.

Key **SystemC performance optimizations** include:

- **Direct Memory Interface (DMI)** for bypassing bus transactions.
- **Temporal decoupling** to reduce simulation kernel synchronization overhead.

The VP achieves execution speeds of up to **20 million instructions per second**, several orders of magnitude faster than RTL simulation, enabling practical design space exploration.

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

Schlägl et al. 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.

A major contribution is an **automated instruction generation framework** that produces over **20,000 lines of C++ code**, ensuring consistency across RV32 and RV64 implementations. The resulting **RISC-V VP++** supports masking, register grouping, and widening/narrowing operations, as well as a parameterizable execution cycle model.

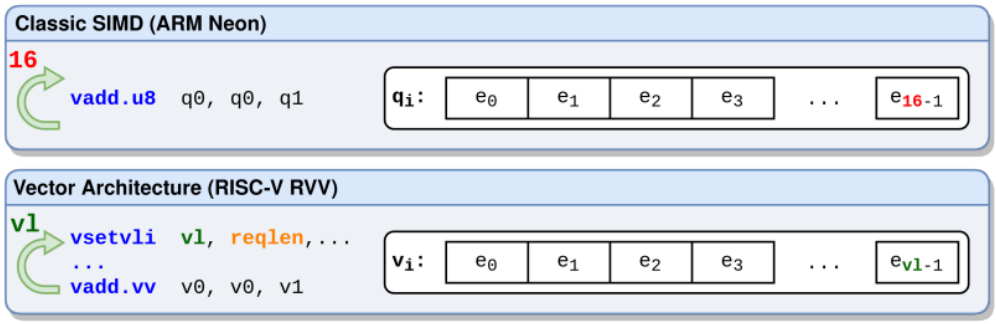


Figure 3: Classic SIMD (ARM Neon) vs. Vector Architecture (RISC-V RVV).

The authors validate correctness using a comprehensive verification chain combining FORCE-RISCV, Spike-based reference execution, and coverage analysis with riscvOVPsim, achieving 81.44% basic coverage. Design-space exploration studies reveal non-linear performance scaling due to memory interface bottlenecks.

Quiroga et al.: Reusable Verification Environment for Vector Accelerators

Quiroga et al. propose a **UVM-based, interface-agnostic verification environment** for RISC-V vector accelerators, enabling reuse across heterogeneous projects such as EPI and eProcessor.

The framework integrates constrained random generation via **RISCV-DV**, a Spike-based golden model using SystemVerilog DPI, and polymorphic wrappers to abstract interconnect protocols such as OVI and AMBA CHI. This modularity reduces simulation overhead and enables early bug discovery during RTL development.

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

Imianosky et al. analyze performance–reliability trade-offs in fault-tolerant RISC-V systems by extending the **HARV-SoC** with **Zve32x** vector multiplication support.

Fault-injection experiments simulating neutron-induced single-event upsets show that vector acceleration achieves up to **28.69**× speedup while often improving overall reliability due to reduced execution time. Reliability is shown to be highly dependent on **SEW**, with 8-bit configurations offering the most favorable trade-off.

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, memory-bound workloads, and high-performance computing systems, including machine learning workloads.

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

Wang et al. propose **SPEED**, a scalable multi-precision DNN processor designed to overcome limitations of baseline RVV implementations. SPEED integrates a highly parameterized **Systolic Array Unit (SAU)** within each vector lane and introduces custom instructions to orchestrate data movement and computation.

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.

Rumyantsev et al.: RVV Efficiency for ANN Algorithms

Rumyantsev et al. investigate the use of RVV to accelerate **Approximate Nearest Neighbor (ANN)** algorithms such as IVFFlat, IVFPQ, and HNSW. Using RVV intrinsics on a Lichee Pi 4A platform, the optimized implementations achieve speedups of up to **2.58**× over scalar baselines.

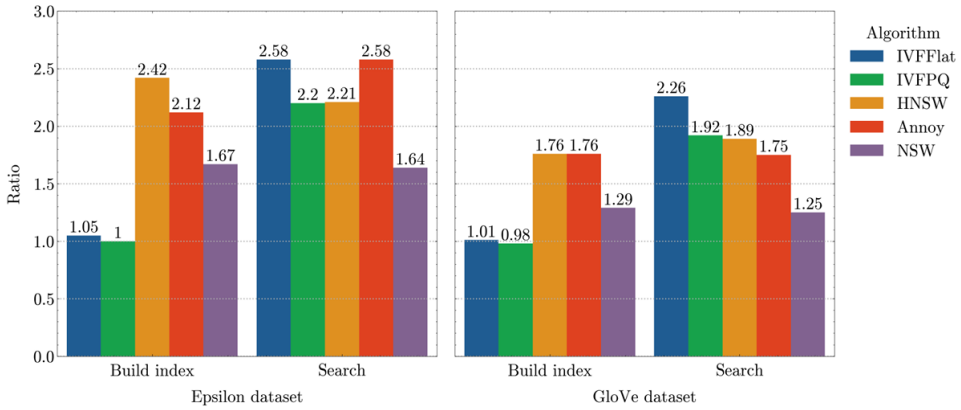


Figure 4: Performance acceleration from RVV optimization for various ANN algorithms on the Epsilon and GloVe datasets.

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

Carpentieri et al. present an empirical evaluation of **compiler-driven autovectorization** on real RISC-V hardware, comparing GCC and LLVM across TSVC benchmarks and real-world applications.

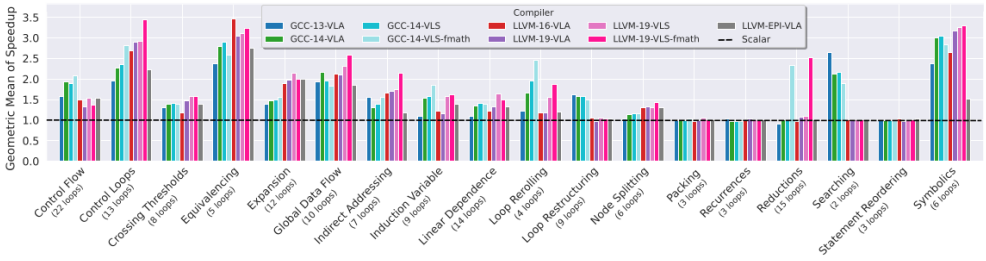


Figure 5: Geometric mean of speedup achieved through autovectorization across different loop categories and compilers using RVV 1.0.

The study shows that **Vector Length Specific (VLS)** programming often outperforms VLA due to reduced runtime overhead, and that careful tuning of **LMUL** can yield speedups of up to **3**×. LLVM 19 outperforms GCC 14 in most benchmark cases, though higher LMUL values can increase register pressure.

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

Volokitin et al. 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. Despite lower absolute bandwidth, RISC-V systems demonstrate high memory subsystem utilization efficiency.

Brown: Evaluating the Sophon SG2044 for High Performance Computing

Brown 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, positioning RISC-V as a competitive architecture for multi-core server workloads.

3 Methodology: Architecture & Implementations

3.1 Deep Learning Kernel Selection and Justification

The development of the RVV64_Library begins with careful selection of computational kernels that represent fundamental building blocks for machine learning inference and digital signal processing applications. The selection process follows a systematic prioritization strategy based on several criteria.

3.1.1 Selection Criteria

Kernels are selected based on the following criteria:

1. **Computational significance**: Operations that constitute performance bottlenecks in target applications. Profiling studies of neural network inference workloads indicate that matrix multiplication and convolution operations account for 80-95% of total execution time, making them priority targets for optimization.

2. **Data parallelism potential**: Operations exhibiting high degrees of data-level parallelism that can be efficiently mapped to RVV's vector execution model. Element-wise operations and reduction operations naturally fit this category.

3. **Memory access patterns**: Operations with predictable memory access patterns that can leverage RVV's unit-stride, strided, and indexed memory operations without excessive overhead.

4. **Composability**: Fundamental operations that can be composed to build more complex computational graphs. For example, matrix multiplication and activation functions are basic building blocks for fully-connected neural network layers.

5. **Algorithmic complexity**: A mix of simple and complex kernels to demonstrate RVV's versatility. Simple element-wise operations like ReLU provide straightforward vectorization examples, while operations like Softmax requiring normalization demonstrate handling of complex multi-phase algorithms.

3.1.2 Prioritized Kernel Categories

Based on these criteria, kernels are organized into three priority tiers tailored to AI and computer vision workloads, targeting models such as LeNet-5 and Tiny-YOLOv2.

Tier 1 (Core Linear Algebra for DNNs): Matrix multiplication, matrix addition, matrix transposition, and dot product. These operations form the backbone of convolutional and fully connected layers, dominating the computation in deep neural networks used for image classification and object detection.

Tier 2 (Activation, Normalization, and Basic CV): ReLU, Sigmoid, Tanh, and Softmax activation functions, along with simple elementwise normalization and scaling. These operations are lighter than matrix kernels but are essential for expressing nonlinearities in CNNs like LeNet-5 and for stabilizing training and inference in models such as Tiny-YOLOv2.

Tier 3 (Specialized Vision Operations): Convolution and pooling operations, as well as domain-specific kernels commonly used in computer vision pipelines (e.g., feature extraction, downsampling, and simple pre/post-processing). These operations showcase RVV's ability to handle complex multi-dimensional data access patterns typical of CNN-based AI workloads.

3.2 Development Toolchain

3.2.1 RISC-V GNU Toolchain

The RISC-V GNU Toolchain provides the foundational components necessary to build, compile, link, and debug code for the RISC-V ISA. This toolchain was selected for its active community support, regular updates, and comprehensive support for both native and emulated environments, including full support for the vector extension.

Toolchain Components **Compiler (`riscv64-unknown-linux-gnu-gcc`):** Translates C/C++ source code into RISC-V assembly and object code. The compiler includes full support for RISC-V vector intrinsics, allowing developers to write vectorized code using high-level C/C++ functions that map directly to vector instructions. Architecture-specific flags enable vector extension support, including:

- `-march=rv64gcv`: Enables the full RV64GCV ISA including vector extensions
- `-O2`, `-O3`: Optimization levels that leverage vector instructions
- `-mabi=lp64d`: Specifies the 64-bit ABI with double-precision floating-point

Assembler (`riscv64-unknown-linux-gnu-as`): Converts assembly code into object files, supporting both standard RISC-V instructions and vector extension opcodes.

Linker (`riscv64-unknown-linux-gnu-ld`): Links multiple object files and resolves external references to produce final executable binaries.

Debugger (`riscv64-unknown-linux-gnu-gdb`): Enables source-level debugging and inspection of RISC-V binaries running under emulation, with support for examining vector register contents.

Binary Utilities: Tools like `objdump`, `readelf`, and `nm` for inspecting compiled binaries, verifying instruction encoding, and analyzing symbol tables.

Runtime Libraries: Standard runtime support including newlib and glibc, providing C standard library functionality for both bare-metal and Linux environments.

3.2.2 QEMU Emulator

QEMU (Quick Emulator) serves as our primary execution environment for RISC-V code. As a full system emulator, QEMU provides comprehensive support for user-mode and full-system emulation, including the RISC-V vector extension in recent versions.

Why QEMU Over Spike While Spike is the official RISC-V ISA simulator from the RISC-V Foundation, we chose QEMU for several key advantages:

System Completeness: QEMU emulates entire Linux-based systems, not just the ISA, enabling realistic testing of our implementations including system calls, memory management, and I/O operations.

Debugging Integration: QEMU integrates seamlessly with GDB, allowing source-level debugging with breakpoints, watchpoints, and inspection of both scalar and vector registers.

Performance: QEMU uses dynamic binary translation, providing faster execution than Spike's interpretive approach, which is crucial when running complex neural network workloads.

Vector Extension Support: Modern QEMU versions include comprehensive RVV support, accurately emulating vector instructions including the latest intrinsics.

Community and Documentation: QEMU benefits from wider usage, more extensive documentation, and more active development than Spike, with a large community providing support and bug fixes.

QEMU User Mode Execution Our typical workflow uses QEMU in user-mode emulation, where RISC-V Linux binaries are executed directly on the host system:

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

This approach provides:

- Fast startup times
- Direct access to host file system
- Straightforward integration with development tools
- Configurable vector length (VLEN) for testing portability

3.2.3 Ara RTL Compilation and Simulation

Ara provides a cycle-accurate hardware model written in SystemVerilog, enabling precise performance measurement of our vectorized kernels. The Ara repository includes the complete RTL description of the vector coprocessor, testbench infrastructure, and build scripts.

Ara Simulation Environment Hardware Description: Ara's RTL is synthesizable and can be simulated using industry-standard tools like Verilator. The repository includes:

- Complete processor core with configurable lane count
- Memory system models
- Instruction trace generation
- Performance counter interfaces

Software Integration: Ara supports standard RISC-V toolchains and can execute the same binaries produced by our compilation flow. Programs are loaded into simulated memory and executed cycle-by-cycle, with detailed logging of:

- Instruction execution sequences
- Vector register states
- Memory access patterns
- Pipeline utilization
- Cycle-accurate timing

Performance Measurement: Ara's simulation environment provides precise metrics for evaluating our kernels:

- Total cycle count for kernel execution
- Vector unit utilization percentages
- Memory bandwidth consumption
- Instruction-level parallelism achieved

Compilation and Execution Flow:

1. Compile C++ code with RVV intrinsics using RISC-V GCC
2. Generate ELF binary with embedded test data
3. Load binary into Ara simulation environment
4. Execute simulation and collect performance traces
5. Extract cycle counts and utilization metrics
6. Compare vectorized vs. scalar implementations

This cycle-accurate measurement is essential for quantifying the real-world performance benefits of our vectorization efforts, as it accounts for all microarchitectural effects including banking conflicts, memory latency, and pipeline stalls that are not visible in pure ISA-level simulation.

3.2.4 Docker Development Environment

To ensure consistency across team members' development environments and simplify onboarding, we created a comprehensive Docker image containing all necessary tools: the RISC-V GNU toolchain, QEMU with vector support, and associated utilities.

Docker Advantages Reproducibility: Every team member works in an identical environment, eliminating "works on my machine" issues and ensuring consistent build outputs.

Portability: The development environment can be deployed on any system supporting Docker, regardless of host OS (Linux, macOS, Windows).

Isolation: Tool installations and configurations are isolated from the host system, preventing conflicts with other software and allowing easy rollback to known-good states.

Version Control: The Dockerfile serves as documentation of the exact tool versions and configurations used, enabling future reproducibility and facilitating environment updates.

Docker Workflow Our Docker image includes:

- RISC-V GNU Toolchain (complete build from source)
- QEMU 8.2+ with RVV support
- Development utilities (make, cmake, git)
- Text editors and debugging tools
- Pre-configured environment variables

Team members mount their local source code directory into the container, enabling:

- Code editing with familiar host-side tools
- Compilation and execution inside the container
- Direct access to build artifacts on the host
- Persistent storage of development files

3.2.5 ONNX Framework Integration

ONNX Runtime provides our functional verification framework, enabling validation of kernel correctness against reference implementations. The ONNX ecosystem includes:

Model Conversion Tools: Utilities for converting models from PyTorch, TensorFlow, and other frameworks to ONNX format.

Reference Runtime: A highly optimized C++ inference engine that serves as our ground truth for correctness validation.

Operator Testing: ONNX's operator test suite provides standardized test cases for individual operations, which we use to validate each kernel implementation.

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 fused multiply-add (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 General Matrix Multiply (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 = 0 to K-1
            sum ← sum + A[i][k] × B[k][j]
        C[i][j] ← sum
```

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 ← number of cols processed in parallel
        j ← N - remaining_columns
        acc_vector ← 0
        FOR k = 0 to K-1
            a_vector ← broadcast A[i][k]
            b_vector ← load B[k][j : j+vl]
            acc_vector ← acc_vector + (a_vector × b_vector)
        END FOR
        store acc_vector into C[i][j : j+vl]
        remaining_columns ← remaining_columns - vl
    END WHILE
END FOR
```

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 = 0 to N-1
    acc ← bias[j]
    FOR each input feature k = 0 to K-1
        acc ← acc + input[k] × weights[j][k]
    output[j] ← acc
```

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 ← number of outputs processed in parallel
    acc_vector ← load bias[j : j+vl]
    FOR each input feature k = 0 to K-1
        weight_vector ← load weights[j : j+vl][k]
                                          (strided)
        input_vector ← broadcast input[k]
        acc_vector ← acc_vector +
            (input_vector × weight_vector)
    END FOR
    store acc_vector into output[j : j+vl]
    j ← j + vl
END WHILE
```

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 ← sum + input[b][ic][ih][iw]
                                  × kernel[oc][ic][kh][kw]
        output[b][oc][oh][ow] ← sum
```

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
        FOR each output row oh, output column ow
            acc_vector ← output[b][oc:oc+vl][oh][ow]
            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]
```

```
                        weight_vector ← packed_w[ic][kh][kw][oc:oc+vl]
                        acc_vector ← acc_vector + input_val × weight_vector
        store acc_vector to output
```

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
            col[K_index][oh×OW + ow] ← input value or 0 (padding)
        END FOR
    END FOR
END FOR

// Step 2: GEMM
gemm_output ← kernel_matrix × col_matrix

// Step 3: Bias Add
FOR each output channel oc
    FOR each output position
        output[oc][pos] ← gemm_output[oc][pos] + bias[oc]
    END FOR
END FOR
```

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 3×3 Vectorization The 3×3 kernel size deserves special attention because it dominates modern CNN architectures. When we know the kernel size is fixed at 3×3, 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
    row0 ← input row oh
    row1 ← input row oh+1
    row2 ← input row oh+2
    FOR output columns ow in vector chunks
        Load vectors:
            v00, v01, v02 from row0
            v10, v11, v12 from row1
            v20, v21, v22 from row2
        acc ← v00 × k00
        acc ← acc + v01 × k01
        acc ← acc + v02 × k02
        ...
        acc ← acc + v22 × k22
        store acc to output
    END FOR
END FOR
```

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 3×3 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 ← -(inf)
                FOR h = h_start to h_end-1
                    FOR w = w_start to w_end-1
                        max_val ← max(max_val, input[b][c][h][w])
                output[b][c][oh][ow] ← max_val
```

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 < out_w
                vl ← number of columns processed in parallel
                max_vector ← -(inf)
                FOR kh = 0 to k_h-1
                    ih ← ih_start + kh
                    IF ih out of bounds: continuez
                    FOR kw = 0 to k_w-1
                        iw ← ow × stride_w - pad_w + kw
                        input_vector ← load input values
                        max_vector ← max(max_vector, input_vector)
                store max_vector to output
                ow ← ow + vl
```

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 $ow \times \text{stride}_w - \text{pad}_w + kw$. For the next output column $ow + 1$, we need input at position $(ow + 1) \times \text{stride}_w - \text{pad}_w + kw$. 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 load input values 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 "lowest-risk, highest-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 1: 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 2: 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 3: 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 4: 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 5: Scalar Bias Add pseudo-code

Scalar Tensor Add Implementation:

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

Listing 6: 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 7: 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 8: 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: 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 9: 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 10: Partially vectorized NMS pseudo-code

Vectorized IoU Computation The Intersection over Union (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 11: 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

After designing and implementing vectorized kernels, the verification phase ensures functional correctness by comparing kernel outputs against trusted reference implementations. This phase ensures that vectorized implementations produce mathematically correct results, accounting for minor numerical variations inherent in floating-point arithmetic.

3.4.1 ONNX Golden Reference Framework

The Open Neural Network Exchange (ONNX) framework serves as the golden reference standard for functional verification. ONNX defines a hardware-agnostic computational graph representation where operations are specified as named operators and data dependencies as edges between nodes.

The verification workflow follows this sequence:

1. **ONNX model creation**: A reference model is constructed in ONNX format, implementing the same computation as the RVV kernel using standard ONNX operators
2. **Test data generation**: Identical input datasets are generated for both the RVV kernel and the ONNX model
3. **Parallel execution**: The RVV kernel and ONNX model execute using the same inputs
4. **Metric computation**: Output arrays are compared using quantitative metrics to account for numerical precision differences
5. **Correctness verification**: Metrics are evaluated against predefined thresholds to confirm functional equivalence

This approach provides several advantages:

- **Hardware-agnostic validation**: ONNX references are independent of any specific CPU or accelerator
- **Industry standard**: ONNX is widely adopted in machine learning frameworks (TensorFlow, PyTorch, ONNX Runtime)
- **Reproducibility**: ONNX models can be distributed and verified independently
- **Compositional verification**: Complex kernels can be built from simpler verified kernels

3.4.2 Test Data Generation Strategy

Test datasets are carefully designed to exercise different numerical and algorithmic scenarios:

1. **Boundary value testing**: Inputs include zero, small positive/negative values, large values near representable limits, and special floating-point values (NaN, infinity) where applicable
2. **Random data generation**: Pseudo-random inputs drawn from uniform or normal distributions to test general-case behavior
3. **Structured patterns**: Regular patterns such as identity matrices, constant arrays, and linearly-increasing sequences to facilitate manual verification and debugging
4. **Real-world data samples**: Actual data from deployed models and signal processing applications to ensure practical applicability

3.4.3 Numerical Verification Metrics

The framework employs two quantitative metrics to assess functional correctness:

1. **Signal-to-Noise Ratio (SNR)**: Measures the ratio of the reference signal power to the error power:

$$\text{SNR (dB)} = 10 \cdot \log_{10} \left(\frac{\sum_i \text{ref}_i^2}{\sum_i (\text{ref}_i - \text{test}_i)^2} \right) \tag{1}$$

   SNR values greater than 100 dB indicate excellent agreement, with SNR of 40 dB or higher generally considered acceptable for signal processing applications.

2. **Maximum Absolute Error (MaxAbs)**: Captures the largest deviation between corresponding output elements:

$$\text{MaxAbs} = \max_i |\text{ref}_i - \text{test}_i| \tag{2}$$

3.4.4 Verification Threshold Definition

Acceptance thresholds for SNR and MaxAbs are defined based on the kernel type and numerical precision requirements:

- **Element-wise operations**: SNR > 100 dB, MaxAbs $< 10^{-6}$ for single-precision floating-point
- **Reduction operations**: SNR > 60 dB, MaxAbs $< 10^{-4}$ (allowing for accumulation of rounding errors)
- **Complex multi-stage operations**: SNR > 40 dB, MaxAbs $< 10^{-3}$ (accounting for multiple transformation stages)

3.4.5 Discrete Functions Correctness Results

[Placeholder: sally edit within this file]

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 and Tiny-YOLOv2. These models were selected to provide comprehensive coverage of the implemented kernel categories while representing distinct computational patterns and application domains within computer vision.

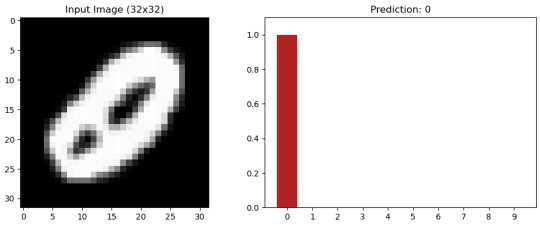
Model Selection Rationale: The selection of LeNet-5 and Tiny-YOLOv2 for end-to-end functional verification was driven by several key considerations. First, these architectures collectively exercise the majority of kernels implemented in the library, including convolution, max pooling, dense (fully connected) 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.

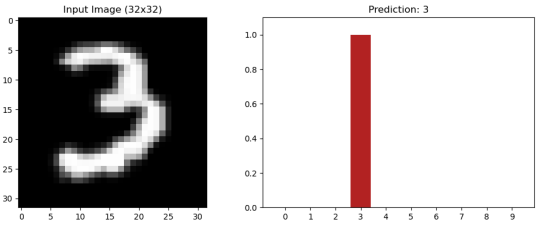
The LeNet-5 implementation directly maps to the following vectorized kernels from the library:

- `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

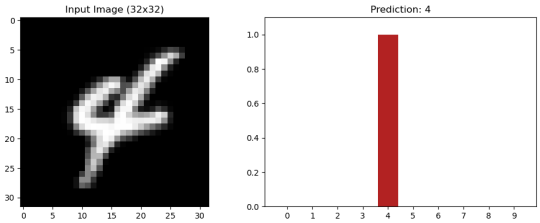
Model weights were extracted from a pre-trained ONNX model and stored as raw IEEE 754 single-precision floating-point binary files. The inference pipeline was implemented in both C++ (utilizing RVV intrinsics directly) and Python (via the `pyv` wrapper library), enabling cross-validation between implementations.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



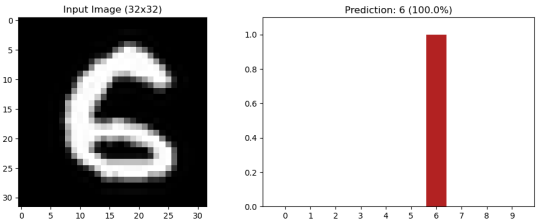
(a) Prediction for digit 0



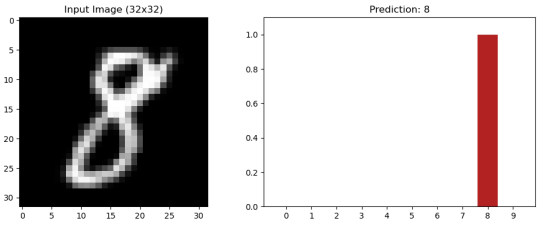
(b) Prediction for digit 3



(c) Prediction for digit 4



(d) Prediction for digit 6



(e) Prediction for digit 8

Figure 6: 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.

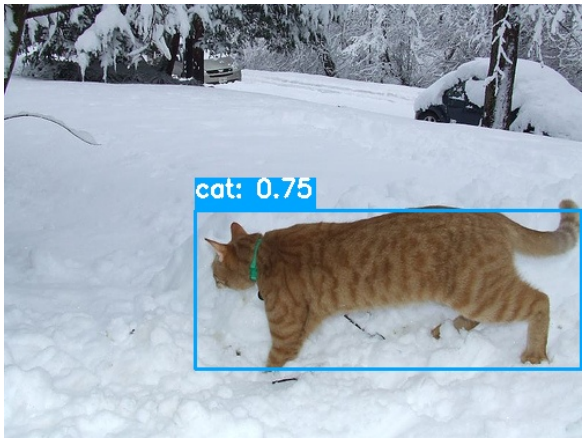
Tiny-YOLOv2 Architecture and Implementation: Tiny-YOLOv2 represents a significantly more complex verification target, implementing real-time object detection on 416×416 RGB input images. The architecture processes inputs through nine convolutional layers with progressively increasing channel depths (16, 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 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.

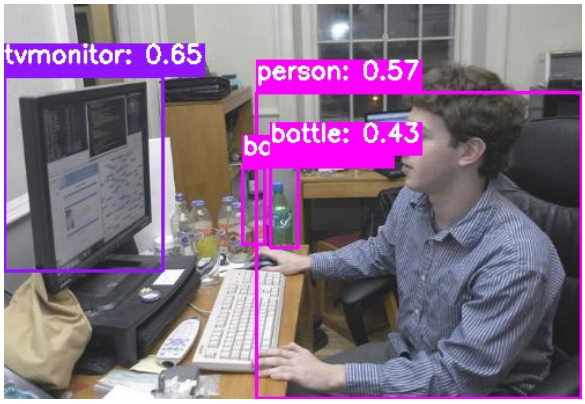


(a) Detection result 1

(b) Detection result 2

(c) Detection result 3

(d) Detection result 4



(e) Detection result 5

Figure 7: 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 6 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 7 demonstrate successful localization and classification of objects across various test images.

The successful implementation and verification of LeNet-5 and Tiny-YOLOv2 using the RVV64_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 or QEMU—provide a mechanism for validating instruction set architecture (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 LUT usage) that are grounded in physical reality rather than theoretical estimation.

For machine learning workloads, which are characteristically defined by dense linear algebra operations (GEMM), convolutions (CONV2D), and high-bandwidth memory access patterns, RTL cores reveal the true utilization of functional units (FUs). They allow for the precise measurement of "raw throughput ideality"—a metric comparing achieved performance against theoretical peaks—and facilitate the identification of non-obvious bottlenecks, such as the scalar core's instruction issue rate limiting the performance of short-vector kernels.

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.

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.

Without cycle-accurate visibility, a researcher might overestimate the performance of a matrix multiplication kernel by failing to account for the cycles lost to cache invalidations or the serialization of micro-operations within the vector unit. Moreover, ML workloads often exhibit phases of computation that are distinct: memory-bound phases (e.g., activation loading) and compute-bound phases (e.g., matrix accumulation). RTL cores allow for the construction of "Roofline" models based on empirical data, plotting arithmetic intensity against achieved floating-point operations per second.

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** 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**, 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 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. 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.

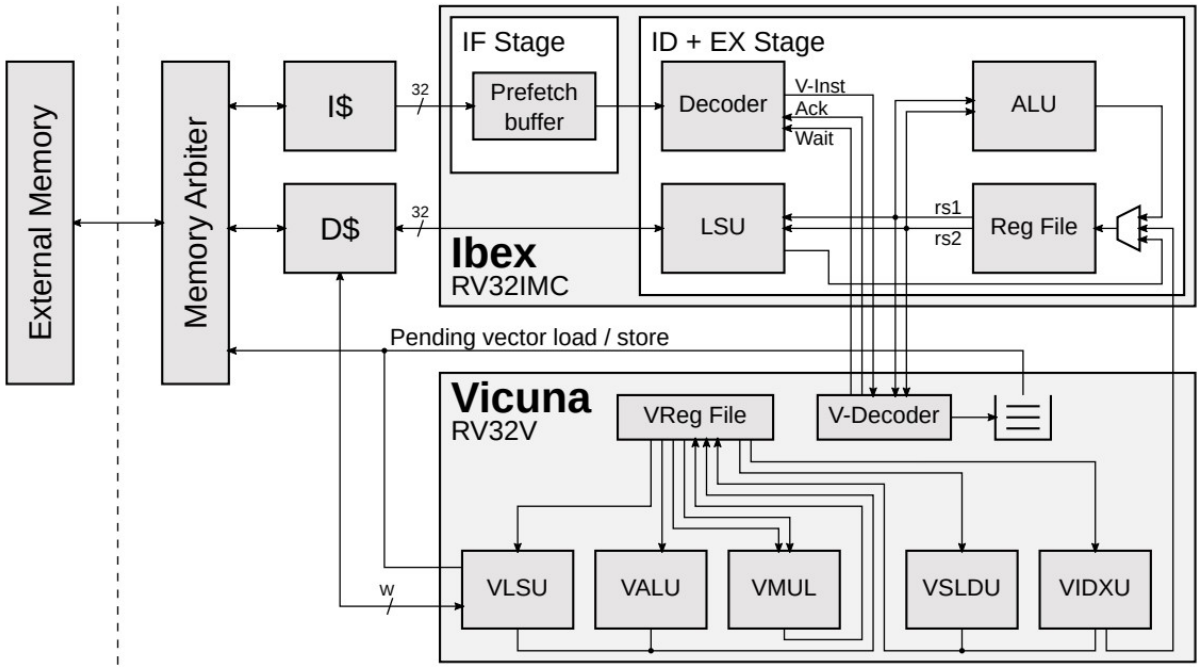


Figure 8: Overview of Vicuna's architecture and its integration with the Ibex main core. Both cores share a common data cache with predictable memory arbitration ensuring deterministic timing behavior.

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 Element (VELEM) units 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 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 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, typically CVA6 (formerly Ariane). 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, as well as contemporary systems like the Fugaku A64FX processor, positioning Ara as an open-source RISC-V vector architecture for the exascale era.

Ara's architectural evolution—most notably the transition from Ara to Ara2—was driven by two main objectives: strict compliance with the ratified RISC-V Vector Extension version 1.0 (RVV 1.0) and aggressive scaling of floating-point throughput. This evolution required substantial microarchitectural changes to support dynamic vector length configuration, masking semantics, and mixed element widths, while enabling parametric scaling across a wide range of performance and power targets.

4.3.2 Architectural Organization

The Ara system operates as a coherent vector coprocessor tightly integrated with the CVA6 scalar core. CVA6 manages the control plane, including instruction fetch, control flow, and exception handling. Vector instructions are dispatched to Ara only after they are committed in the scalar pipeline,

ensuring non-speculative execution and simplifying the vector unit's control logic by eliminating rollback requirements due to branch mispredictions.

Ara's microarchitecture is composed of N identical vector lanes, where N is typically a power of two $(2, 4, 8, 16)$. Each lane contains a proportional slice of the Vector Register File (VRF) and a set of functional units, including a Vector ALU (VALU) and a Vector Floating-Point Unit (VFPU). To provide high register bandwidth without incurring the area cost of multi-ported memories, Ara implements a *barber-pole* banking scheme. In this layout, vector elements are distributed across banks and lanes such that, for most element-wise operations, operands are locally available within each lane, minimizing cross-lane data movement.

This banking strategy allows Ara to achieve high VRF bandwidth using simple single-port SRAMs while maintaining scalability. Consecutive vector elements are striped across lanes (e.g., element 0 in lane 0, element 1 in lane 1), enabling the processor to execute up to N operations per cycle for fully vectorized workloads.

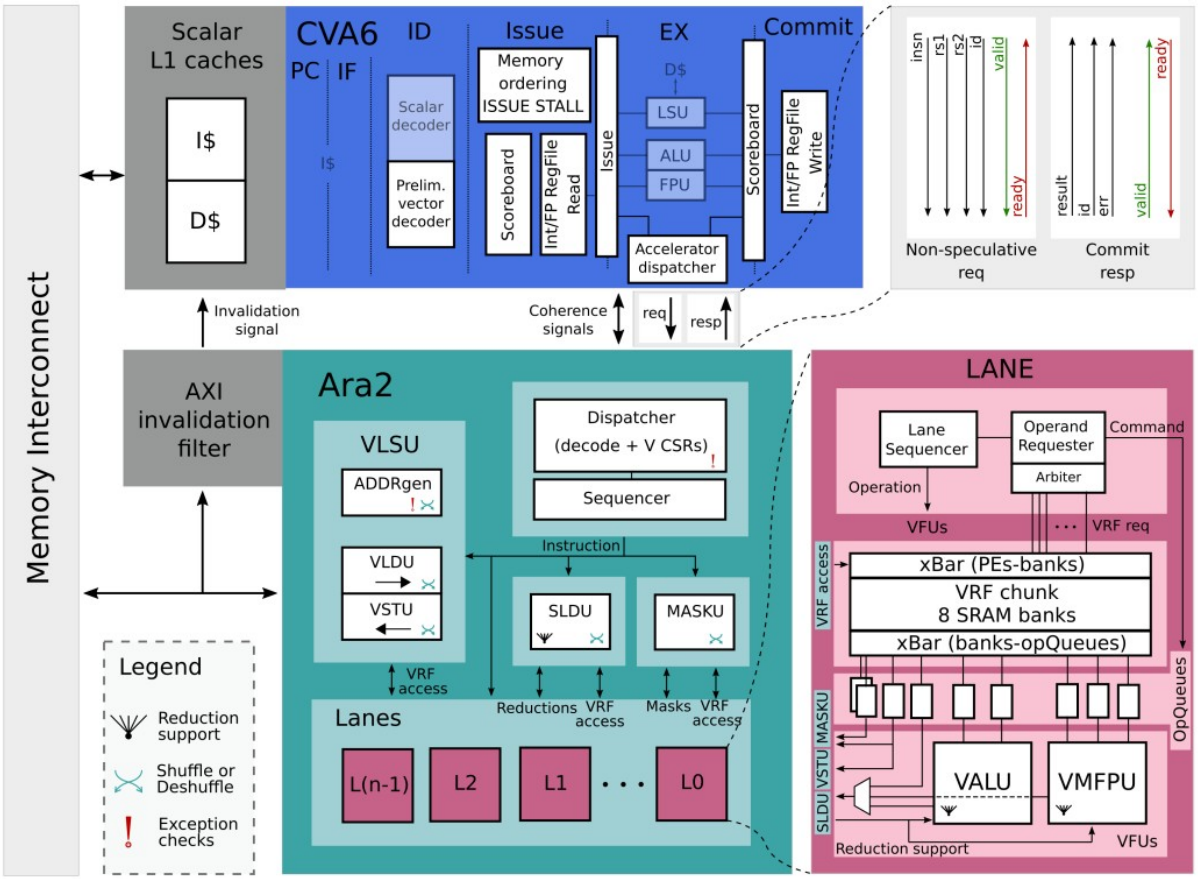


Figure 9: Top-level block diagram of the Ara2 system showing the vector coprocessor, lane-based organization, and integration with the CVA6 scalar core.

4.3.3 RVV Implementation

Ara fully implements the frozen RVV 1.0 extension with comprehensive feature support:

- **Data Types:** IEEE-754 floating-point formats (FP16, FP32, FP64) and standard integer types (INT8, INT16, INT32, INT64)
- **Reductions:** Support for ordered and unordered vector reductions, including floating-point reductions requiring precise control of execution order
- **Masking:** Full support for masked execution, enabling element-wise predication through dedicated mask registers
- **Permutations:** Support for permutation instructions such as `vslideup`, `vslidedown`, `vgather`, and `vscatter` via a dedicated on-chip interconnect

4.3.4 Vector Execution Model

Ara adheres to the Vector Length Agnostic (VLA) programming model mandated by the RISC-V specification. While the hardware vector length (VLEN) is fixed per instantiation, software binaries remain portable across implementations. At runtime, the `vsetvli` instruction configures the application vector length (AVL), which is automatically distributed across the available lanes. If the AVL exceeds the number of parallel lanes, the hardware transparently strip-mines the operation, executing the vector in temporal chunks.

While element-wise instructions execute independently within each lane, operations involving data movement—such as reductions, masking, and permutations—are handled by centralized Mask and Permutation Units. These operations require cross-lane communication and are implemented using an $O(L \log L)$ interconnect, where L is the number of lanes.

For reduction operations, Ara employs a three-step algorithm: first, partial reductions are performed locally within each lane; second, cross-lane shuffling aligns partial results; finally, the scalar result is written back to the destination register. This decoupled execution model allows arithmetic pipelines to remain highly utilized while centralized units manage irregular data access patterns.

Ara supports vector chaining, enabling dependent instructions to begin execution before their predecessors have fully completed, provided that the required operands are available. This capability is essential for sustaining high utilization of deep floating-point pipelines in sequences such as fused multiply-accumulate (`vfmacc`) operations.

4.3.5 Memory Subsystem

The Vector Load/Store Unit (VLSU) serves as the interface between the high-bandwidth AXI memory system and the vector lanes. It supports unit-stride, strided, and indexed (scatter/gather) memory accesses and is responsible for routing memory data to the appropriate lanes and VRF banks. Due to its flexible routing requirements, the complexity of the VLSU scales superlinearly with the number of lanes.

Ara2 introduces a robust hardware coherence mechanism between the scalar and vector domains. The CVA6 data cache operates in write-through mode, ensuring that scalar stores are immediately visible to Ara. Conversely, vector stores issued by Ara generate invalidation signals to the CVA6 cache, forcing eviction of affected cache lines. This hardware-based scheme eliminates the need for software-managed fence instructions while maintaining a coherent memory view across scalar and vector execution.

4.3.6 RTL Implementation

Implemented in GlobalFoundries 22FDX (22nm FD-SOI) technology, Ara (in its Ara2 iteration) achieves a clock frequency of 1.35 GHz for configurations up to 8 lanes, with a critical path of approximately 40 FO4 delays. The design is highly parameterized, supporting lane counts of 2, 4, 8, and 16.

While the area of the functional units scales linearly with the number of lanes, the interconnect structures—particularly the Slide Unit (SLDU) and Mask Unit (MASKU)—exhibit superlinear scaling. To mitigate this, Ara2 restricts slide operations to power-of-two strides, reducing interconnect complexity from $O(L^2)$ to $O(L \log L)$ and enabling feasible scaling to larger lane counts.

In a 16-lane configuration, Ara achieves over 98% FPU utilization on matrix multiplication kernels and a peak energy efficiency of 37.8 DP-GFLOPS/W. The 64-lane "AraXL" variant occupies only 3.8× the area of the 16-lane design, demonstrating near-linear area scaling and superior efficiency compared to traditional SIMD or multi-core scalar architectures.

4.3.7 Benchmarking Suitability

Ara is exceptionally well-suited for benchmarking compute-bound machine learning and HPC kernels. For large problem sizes (e.g., 128×128 matrices), Ara sustains 97–99% floating-point unit utilization, indicating that memory latency and control overhead are effectively hidden by the microarchitecture.

The bit-accurate and fully RVV-compliant implementation enables precise tuning of kernels written using RVV intrinsics, allowing researchers to analyze lane utilization, register pressure, and instruction scheduling effects. Benchmarking results further show that for smaller problem sizes, multi-core systems with smaller vector units may outperform a single large vector processor, providing valuable insights into architectural trade-offs across scalability regimes.

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 1 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 1: Final Comparative Positioning of RTL Cores

Feature	**Vicuna (Architectural Baseline)**	**Ara (Benchmarking Vehicle)**
ISA Profile	$Zve32x$ (Integer/Fixed-point)	**Full RVV 1.0 (DP 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.

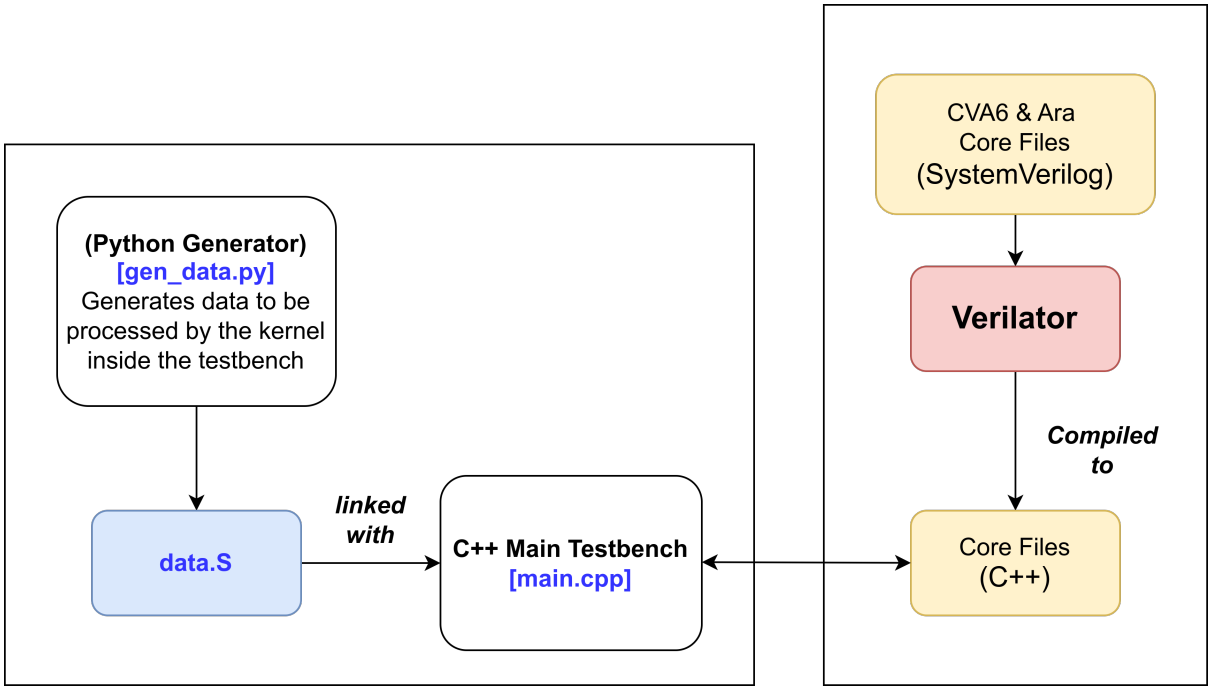
Because Vicuna implements the $Zve32x$ subset, it lacks the **"F"** (Single-Precision) and **"D"** (Double-Precision) vector extensions. Consequently, it is functionally incapable of executing the standard floating-point kernels required for the comparative performance analysis in this study. Furthermore, 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` 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 12: 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;
```

```
        float alpha = ALPHA_VAL;

        printf("\n=== LEAKY RELU [N: %d, Alpha: %.2f] ===\n", N, alpha);

        // Scalar Test
        memcpy(output_data, input_data, N * sizeof(float));
        start_timer();
        leaky_relu_scalar(output_data, N, alpha);
        stop_timer();
        printf("Scalar Cycles: %d\n", get_timer());

        // Vector Test
        memcpy(output_data, input_data, N * sizeof(float));
        start_timer();
        leaky_relu_vector(output_data, N, alpha);
        stop_timer();
        int t_vec = get_timer();
        printf("Vector Cycles: %d\n", t_vec);

        // Optional: Verify Correctness
        if(verify(output_data, golden_data, N) == 0) printf("Status: PASSED\n")
            ;
        else printf("Status: FAILED\n");

        return 0;
}
```

Listing 13: Leaky ReLU Testbench Execution Logic

4.6 Validation Results

This section presents a comprehensive performance evaluation of the RVV64_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 2 provides a summary of the peak acceleration achieved in each category.

Table 2: Peak speedup across all evaluated configurations for each RVV64_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 3, 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 3: 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 4: 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 5, 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 5: 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 6, 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 6: 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 7: 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 7. 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 8: 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 8 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 9: Additive Kernels — Best Implementation per Size

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 9, 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 3 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.

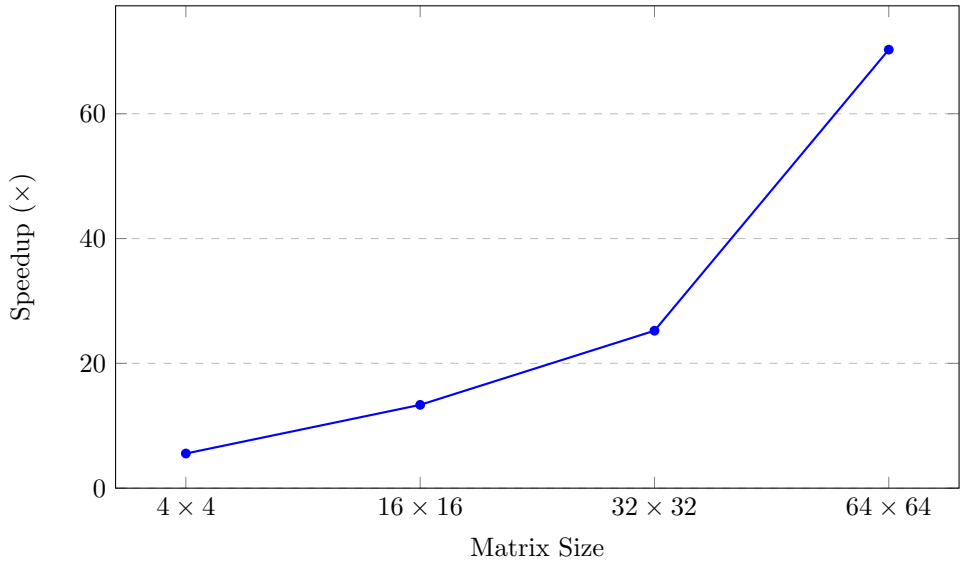


Figure 10: Speedup of MatMul across matrix sizes

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 **25.01**× 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 (To be named)

A critical aspect of this work extends beyond the hardware implementation to provide a comprehensive, user-friendly software interface for the RISC-V Vector Extension (RVV) accelerated kernels. The `(To be named)` library offers Python wrappers that bridge the gap between high-level deep learning frameworks and the low-level RVV-optimized C implementations. This approach significantly enhances usability and accessibility for researchers and developers who wish to leverage hardware acceleration without delving into assembly-level programming.

5.1 Design Philosophy and Importance

The Python wrapper library addresses a fundamental challenge in hardware-software co-design: making optimized low-level implementations accessible to a broader audience. While the underlying C/assembly kernels provide maximum performance through direct RVV intrinsics, the Python interface offers several key advantages:

- **Ease of Integration**: Researchers can integrate RVV-accelerated operations directly into Python-based machine learning pipelines using familiar NumPy array semantics.
- **Variant Selection**: Users can easily switch between scalar baseline implementations and various LMUL configurations (M1, M2, M4, M8) to benchmark and compare performance characteristics.
- **Tiled Execution Support**: Several kernels offer tiled variants that improve cache locality for large tensors, accessible through simple API parameters.
- **Rapid Prototyping**: The wrappers enable quick experimentation with different kernel variants without recompilation, facilitating performance exploration and optimization studies.

5.2 Library Structure

The `(To be named)` library follows a modular architecture organized into three layers:

1. **Backend Layer** (`backend.py`): Provides utility functions for memory management and pointer conversion between NumPy arrays and C-compatible float pointers using Python's `ctypes` module.
2. **Wrapper Layer** (`wrappers/`): Contains individual wrapper modules for each kernel type, handling the `ctypes` interface to shared libraries.
3. **Kernel API Layer** (`kernels.py`): Exposes high-level Pythonic functions with automatic shape inference, memory allocation, and variant selection.

5.3 Available Kernel Wrappers

The library provides wrappers for eleven distinct neural network operations, each with multiple implementation variants. Table 10 summarizes the available implementations.

Table 10: Python Wrapper Kernel Variants and 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	✓	–	–	–	–	–

5.3.1 ReLU Activation

The ReLU wrapper provides element-wise rectified linear unit activation supporting scalar and all LMUL variants.

Function Signature:

```
relu(x: np.ndarray, variant="M8") -> np.ndarray
```

Parameters:

- `x`: Input tensor (any shape, float32, C-contiguous)
- `variant`: One of `"scalar"`, `"M1"`, `"M2"`, `"M4"`, `"M8"`

Usage Example:

```
import numpy as np
from pyv.kernels import relu

x = np.random.randn(1, 64, 32, 32).astype(np.float32)
y = relu(x, variant="M8")  # RVV-accelerated with LMUL=8
```

5.3.2 Leaky ReLU Activation

The Leaky ReLU wrapper extends ReLU with a configurable negative slope parameter and includes tiled variants for improved cache performance on large tensors.

Function Signature:

```
leaky_relu(x: np.ndarray, alpha: float = 0.01, variant="M8") -> np.ndarray
```

Available Variants:

- Standard: `"scalar"`, `"M1"`, `"M2"`, `"M4"`, `"M8"`
- Tiled: `"tiled_scalar"`, `"tiled_M1"`, `"tiled_M2"`, `"tiled_M4"`, `"tiled_M8"`

Usage Example:

```
from pyv.kernels import leaky_relu

x = np.random.randn(1, 128, 16, 16).astype(np.float32)
y = leaky_relu(x, alpha=0.2, variant="M4")
```

5.3.3 Matrix Multiplication

The MatMul wrapper performs general matrix multiplication $C = A \times B$ with support for all LMUL configurations.

Function Signature:

```
matmul(A: np.ndarray, B: np.ndarray, variant="M8") -> np.ndarray
```

Parameters:

- `A`: Matrix of shape (M, K), float32
- `B`: Matrix of shape (K, N), float32
- `variant`: One of `"scalar"`, `"M1"`, `"M2"`, `"M4"`, `"M8"`

Usage Example:

```
from pyv.kernels import matmul

A = np.random.randn(128, 256).astype(np.float32)
B = np.random.randn(256, 64).astype(np.float32)
C = matmul(A, B, variant="M8")  # C is (128, 64)
```

5.3.4 Tensor Addition

The Tensor Add wrapper performs element-wise addition of two tensors with matching shapes.

Function Signature:

```
tensor_add(A: np.ndarray, B: np.ndarray, variant="M8") -> np.ndarray
```

Usage Example:

```
from pyv.kernels import tensor_add

A = np.random.randn(1, 64, 32, 32).astype(np.float32)
B = np.random.randn(1, 64, 32, 32).astype(np.float32)
C = tensor_add(A, B, variant="M4")
```

5.3.5 Batch Normalization

The Batch Normalization wrapper implements inference-mode batch normalization with learned scale and bias parameters. Both standard and tiled variants are available for all LMUL configurations.

Function Signature:

```
batch_norm(x: np.ndarray, scale: np.ndarray, bias: np.ndarray,
           mean: np.ndarray, variance: np.ndarray,
           epsilon: float = 1e-5, variant="M8") -> np.ndarray
```

Parameters:

- `x`: Input tensor of shape (N, C, H, W), float32
- `scale`, `bias`, `mean`, `variance`: Per-channel parameters of shape $(C,)$
- `epsilon`: Small constant for numerical stability
- `variant`: Standard variants or tiled variants (`"tiled_M1"` through `"tiled_M8"`)

Usage Example:

```
from pyv.kernels import batch_norm

x = np.random.randn(1, 64, 32, 32).astype(np.float32)
scale = np.ones(64, dtype=np.float32)
bias = np.zeros(64, dtype=np.float32)
mean = np.zeros(64, dtype=np.float32)
var = np.ones(64, dtype=np.float32)

y = batch_norm(x, scale, bias, mean, var, epsilon=1e-5, variant="tiled_M8")
```

5.3.6 Bias Addition

The Bias Add wrapper adds a per-channel bias to feature maps, commonly used after convolution operations.

Function Signature:

```
bias_add(input: np.ndarray, bias: np.ndarray, variant="M8") -> np.ndarray
```

Parameters:

- `input`: Tensor of shape (N, C, H, W), float32
- `bias`: Bias vector of shape $(C,)$

Usage Example:

```
from pyv.kernels import bias_add

x = np.random.randn(1, 128, 16, 16).astype(np.float32)
b = np.random.randn(128).astype(np.float32)
y = bias_add(x, b, variant="M8")
```

5.3.7 2D Convolution

The Conv2D wrapper provides the most comprehensive set of implementations, including standard convolutions, Im2Col+GEMM variants, and specialized 3×3 kernels optimized for common filter sizes.

Function Signature:

```
conv2d(input: np.ndarray, kernel: np.ndarray, bias: np.ndarray = None,
       stride=(1,1), pad=(0,0), variant="M8") -> np.ndarray
```

Available Variants:

- Standard: `"scalar"`, `"M1"`, `"M2"`, `"M4"`, `"M8"`
- Im2Col+GEMM: `"im2col_M8"` (optimized for larger convolutions)
- Specialized 3×3: `"3x3_M1"`, `"3x3_M2"`, `"3x3_M4"`, `"3x3_M8"`

Usage Example:

```
from pyv.kernels import conv2d

x = np.random.randn(1, 3, 224, 224).astype(np.float32)
w = np.random.randn(64, 3, 3, 3).astype(np.float32)

# Standard vectorized convolution
y1 = conv2d(x, w, stride=(1,1), pad=(1,1), variant="M8")

# Specialized 3x3 kernel
y2 = conv2d(x, w, stride=(1,1), pad=(1,1), variant="3x3_M8")

# Im2Col+GEMM approach
y3 = conv2d(x, w, stride=(1,1), pad=(1,1), variant="im2col_M8")
```

5.3.8 2D Transposed Convolution

The Conv2D Transpose wrapper implements transposed (deconvolution) operations used in decoder networks and generative models. Specialized 3×3 RVV kernels are available for common upsampling configurations.

Function Signature:

```
conv_transpose(input: np.ndarray, kernel: np.ndarray,
               stride=(1,1), pad=(0,0), variant="M8") -> np.ndarray
```

Available Variants:

- Standard: `"scalar"`, `"M1"`, `"M2"`, `"M4"`, `"M8"`
- Specialized 3×3 RVV (automatically selected for 3×3 kernels)

Usage Example:

```
from pyv.kernels import conv_transpose

x = np.random.randn(1, 256, 8, 8).astype(np.float32)
w = np.random.randn(256, 128, 3, 3).astype(np.float32)
y = conv_transpose(x, w, stride=(2,2), pad=(1,1), variant="M8")
```

5.3.9 Dense (Fully Connected) Layer

The Dense wrapper implements fully connected layer computation with bias addition.

Function Signature:

```
dense(x: np.ndarray, weights: np.ndarray, bias: np.ndarray,
      variant="M8") -> np.ndarray
```

Parameters:

- `x`: Input vector of shape $(in\_features,)$
- `weights`: Weight matrix of shape $(out\_features, in\_features)$
- `bias`: Bias vector of shape $(out\_features,)$

Usage Example:

```
from pyv.kernels import dense

x = np.random.randn(512).astype(np.float32)
W = np.random.randn(10, 512).astype(np.float32)
b = np.random.randn(10).astype(np.float32)
y = dense(x, W, b, variant="M8")  # y is (10,)
```

5.3.10 Max Pooling

The MaxPool wrapper provides max pooling with configurable kernel size, stride, and padding. Tiled variants offer improved performance for large feature maps.

Function Signature:

```
maxpool(input: np.ndarray, k_h: int, k_w: int,
        stride_h: int = 1, stride_w: int = 1,
        pad_h: int = 0, pad_w: int = 0, variant="M8",
        tile_h: int = None, tile_w: int = None) -> np.ndarray
```

Available Variants:

- Standard: `"scalar"`, `"M1"`, `"M2"`, `"M4"`, `"M8"`
- Tiled: `"tiled_M1"`, `"tiled_M2"`, `"tiled_M4"`, `"tiled_M8"`

Usage Example:

```
from pyv.kernels import maxpool

x = np.random.randn(1, 64, 112, 112).astype(np.float32)

# Standard 2x2 max pooling with stride 2
y = maxpool(x, k_h=2, k_w=2, stride_h=2, stride_w=2, variant="M8")

# Tiled variant for better cache performance
y_tiled = maxpool(x, k_h=2, k_w=2, stride_h=2, stride_w=2,
                  variant="tiled_M8", tile_h=16, tile_w=16)
```

5.3.11 Softmax

The Softmax wrapper provides numerically stable softmax computation for 1D or 2D inputs.

Function Signature:

```
softmax(x: np.ndarray) -> np.ndarray
```

Usage Example:

```
from pyv.kernels import softmax

logits = np.random.randn(10).astype(np.float32)
probs = softmax(logits)  # Sum to 1.0
```

5.4 LMUL Configuration Guidelines

The LMUL (Length MULtiplier) parameter in RISC-V Vector Extension determines how many vector registers are grouped together for operations. Selecting the appropriate LMUL involves trade-offs:

- **M1**: Uses single registers, maximum flexibility but lower throughput per instruction.
- **M2**: Groups 2 registers, doubling vector length with moderate register pressure.
- **M4**: Groups 4 registers, good balance for medium-sized workloads.
- **M8**: Groups 8 registers, maximum throughput but highest register pressure.

For most deep learning workloads with large tensors, `M8` provides optimal performance. However, kernels with complex control flow or multiple active vectors may benefit from lower LMUL values to reduce register spilling.

5.5 Backend Utilities

The backend module provides essential pointer conversion utilities:

```
from pyv.backend import ptr_f32

# Convert NumPy array to C float pointer
arr = np.zeros((100,), dtype=np.float32)
c_ptr = ptr_f32(arr)  # ctypes.POINTER(ctypes.c_float)
```

This utility ensures proper memory alignment and type checking, requiring arrays to be float32 and C-contiguous for correct operation with the underlying C kernels.

6 Conclusion and Future Work

6.1 Conclusion

This thesis presented the design, implementation, and evaluation of RVV64_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 RVV64_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 Digital Signal Processing (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 diversification would position RVV64_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 GELU activations would enable support for models such as BERT, 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.

7 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.

References

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

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

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

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

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

[6] Vicuna Project, "Vicuna Documentation," 2023. [Online]. Available: vicuna.readthedocs.io (accessed: Jul. 2025).

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

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

[9] 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`

[10] 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`

8 Code Listings

[Placeholder]