Contents

1 Abstract

2 Introduction

2.1 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.

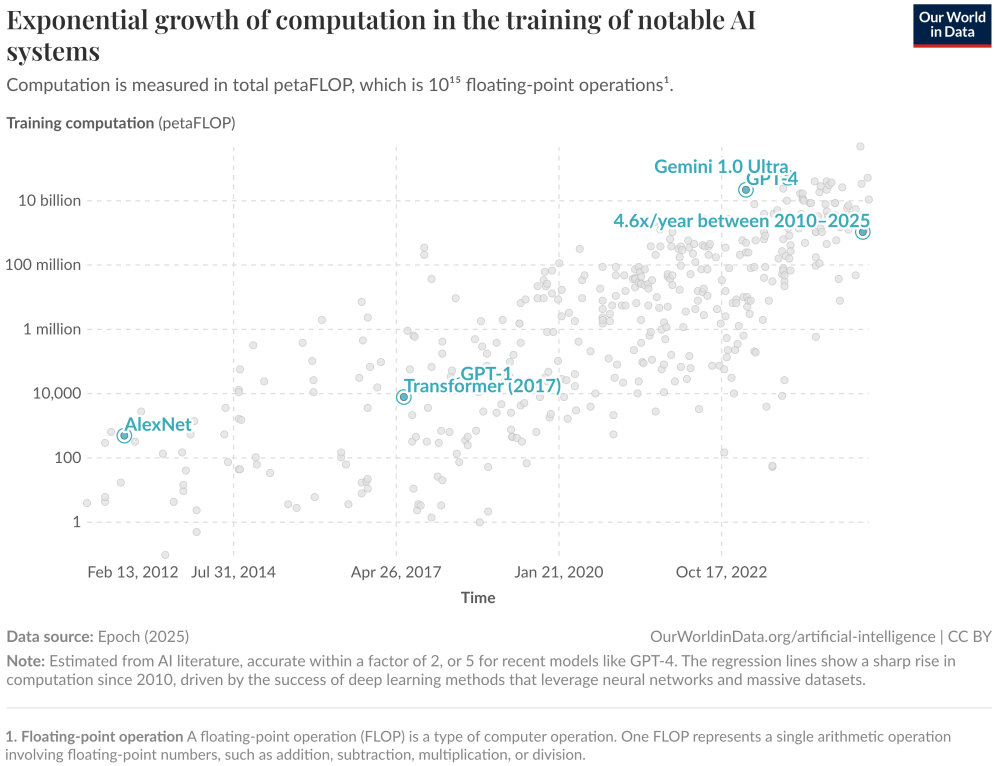

Figure 1: Exponential growth in AI model computational requirements over time, showing the dramatic increase in FLOPs required for training state-of-the-art models.

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.

2.2 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.

2.3 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.

2.4 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.

2.5 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.

2.6 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.

2.7 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.

3 Background & Related Work

3.1 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.

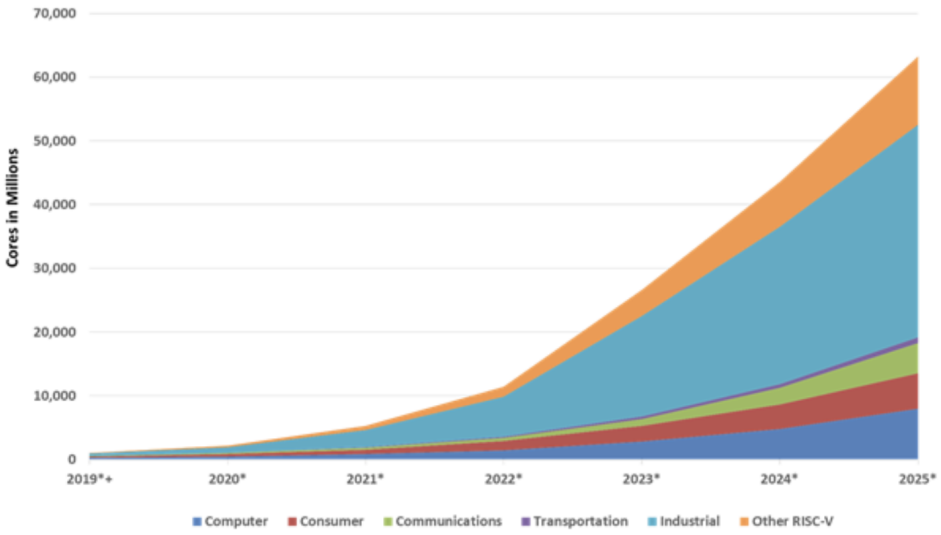

Figure 2: RISC-V ecosystem growth. Source: RISC-V International

3.2 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.

3.2.1 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.

3.3 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.

3.3.1 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)

3.3.2 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);
```

3.4 Ara RISC-V Vector Coprocessor

Ara is a scalable vector coprocessor developed at ETH Zurich that works alongside the CVA6 (formerly Ariane) scalar core to accelerate RISC-V vector operations. It supports 2–16 parallel lanes and implements the RV64GCV instruction set. The processor uses a lane-based design where each lane handles 64-bit wide operations. Vector length (VLEN) can be configured from 128 to 1024 bits depending on application needs. It achieves up to 97% FPU utilization when running a 256×256 double-precision matrix multiplication on sixteen lanes.

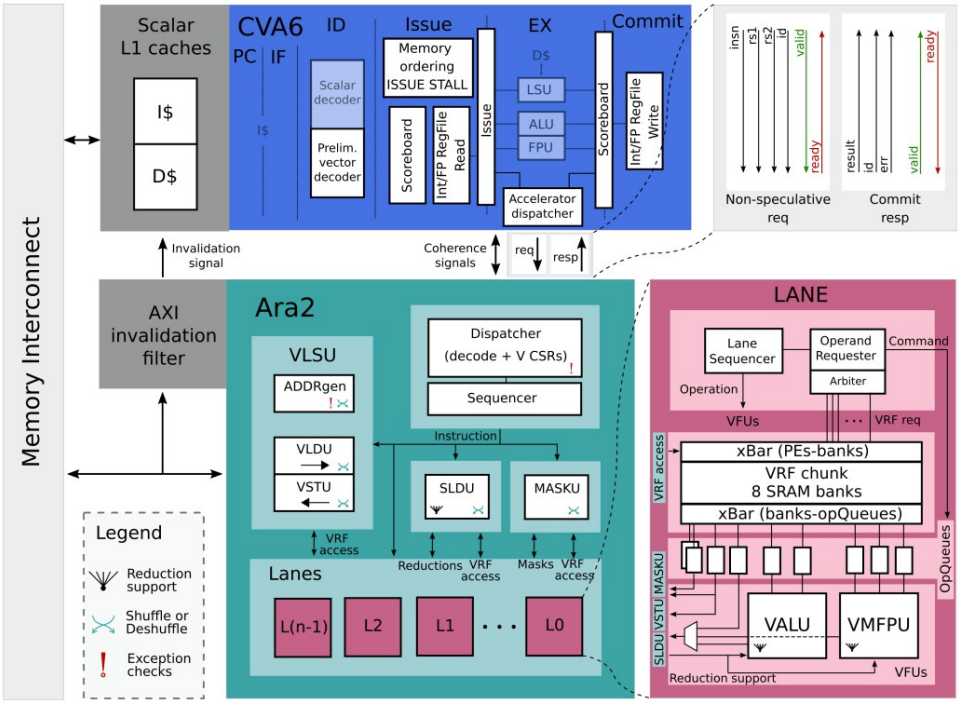

Figure 3: Top-level architecture diagram of the Ara RISC-V Vector Coprocessor

3.4.1 Architecture Components

Lane Design: Each lane operates as an independent vector processing unit with its own ALU and FPU. The lanes process different parts of vector data in parallel, with each lane handling vector elements based on its position. For example, in a 4-lane configuration, lane 0 processes elements 0, 4, 8, 12, while lane 1 handles elements 1, 5, 9, 13.

Sequencer: The sequencer acts as the control center for instruction dispatch and execution tracking, maintaining a global view of all vector instructions across lanes. It can track up to 8 parallel vector instructions and ensures correct execution order while interfacing with the CVA6 scalar core to coordinate vector and scalar instruction execution.

Vector Register File (VRF): The VRF features eight 64-bit wide banks per lane, providing 512 bits of total bandwidth per lane. Banks are accessed in parallel to supply operands to multiple units simultaneously. When multiple functional units need to access the same bank, conflicts are resolved dynamically with a weighted round-robin arbiter. The initial bank of each vector register is shifted in a "barber's pole" fashion to avoid banking conflicts when functional units fetch the first element of different vector registers.

Slide Unit (SLDU): The SLDU handles vector element rearrangement operations essential for many algorithms, performing vector slides, element insertion and extraction, and vector shuffles. It must access all VRF banks simultaneously for some operations.

Vector Load/Store Unit (VLSU): The VLSU is responsible for all vector memory operations and includes sophisticated address generation capabilities. It handles multiple outstanding memory requests and includes logic for address calculation for different stride patterns, memory request coalescing for unit-stride operations, and memory ordering constraint handling.

Queue Management: The multi-banked organization of the VRF can lead to banking conflicts when several functional units try to access operands in the same bank. Each lane has a set of operand

queues between the VRF and the functional units to absorb such banking conflicts.

3.4.2 Ara as a Performance Reference

Ara serves as the hardware reference platform for this project, providing cycle-accurate performance measurements of our vectorized kernels. By compiling and running our implementations on Ara's synthesizable RTL model, we can obtain realistic performance metrics including:

- **Cycle Counts:** Precise measurement of execution cycles for both vectorized and scalar implementations
- **Speedup Analysis:** Quantitative comparison showing the performance benefits of vectorization
- **Hardware Utilization:** Insights into how effectively our code uses the vector processing resources
- **Energy Efficiency:** Understanding of the power-performance trade-offs in our implementations

3.5 Foundational Research on RVV for Machine Learning

Several academic research projects have validated the theoretical benefits of RVV and explored microarchitectural techniques to further accelerate specific workloads, particularly Deep Neural Networks (DNNs) and machine learning algorithms.

3.5.1 SPEED: Scalable Multi-Precision DNN Processor

Wang et al. addressed the gap between general-purpose vector processors and specialized DNN inference demands in their work on SPEED. They identified that standard RVV processors struggle with modern DNNs due to limited support for low-precision data types (e.g., 4-bit), constrained computational throughput for massive MAC operations, and inefficient dataflows that underutilize hardware.

SPEED integrates a highly parameterized Systolic Array Unit (SAU) within each vector lane, acting as a dedicated matrix multiplication engine working in concert with the standard RVV ALU. The architecture is enhanced with custom instructions (VSACFG, VSALD, VSAM) to manage the SAU and a specialized dataflow strategy to maximize data reuse and computational efficiency. When synthesized for 28nm technology and compared to the Ara processor, SPEED demonstrates significant improvement in area efficiency (2.04x for 16-bit and 1.63x for 8-bit operations).

3.5.2 RVV Efficiency for ANN Algorithms

Rumyantsev et al. presented a practical and theoretical analysis of applying RVV to accelerate Approximate Nearest Neighbor (ANN) search algorithms. Their work aimed to quantify performance gains from using RVV to optimize common ANN libraries like Faiss, Annoy, and NMSLIB, where distance computation is the primary bottleneck.

They implemented and optimized key algorithms (IVFFlat, IVFPQ, HNSW) using RVV intrinsics and benchmarked their performance against scalar code on a Lichee Pi 4A board. Experimental results showed that RVV-optimized code achieved speedups of up to 2.58x over scalar versions. The paper also presents a theoretical model of a parameterized vector unit used to determine optimal hardware configuration (register length, number of functional units) for this class of algorithms, providing both real-world performance data and theoretical hardware design insights.

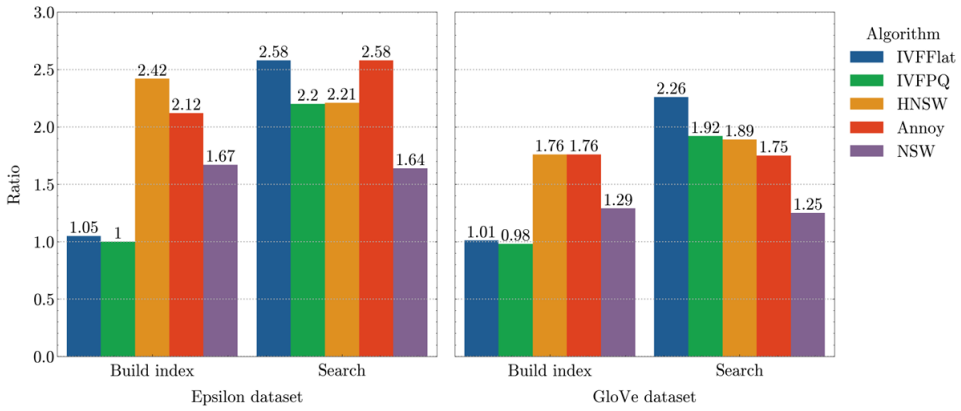

Figure 4: Performance acceleration from RVV optimization for various ANN algorithms on the Epsilon and GloVe datasets

4 Methodology: Architecture & Implementations

4.1 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.

4.1.1 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.

4.1.2 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.

4.2 Development Toolchain

4.3 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.

4.3.1 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.

4.4 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.

4.4.1 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.

4.4.2 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

4.5 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.

4.5.1 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.

4.6 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.

4.6.1 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.

4.6.2 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

4.7 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.

4.8 RISC-V Vectorization Kernels Design

[Placeholder]

4.9 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.

4.9.1 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

4.9.2 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

4.9.3 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:

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

4.9.4 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)

4.9.5 Discrete Functions Correctness Results

[Placeholder]

4.9.6 Models

[Placeholder]

5 Methodology: Performance Validation

5.1 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.

5.1.1 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.

5.1.2 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.

5.1.3 Rationale for Selecting Ara and Vicuna

To provide a comprehensive evaluation of the RISC-V vector design space, this research employs two distinct RTL cores: Ara and Vicuna. These cores were selected because they represent two divergent philosophies within the architectural spectrum: high-performance computing (HPC) and predictable real-time embedded systems.

Ara is selected as the representative for high-performance, throughput-oriented vector processing. As the first fully open-source implementation of the ratified RVV 1.0 specification, Ara targets application-class workloads. It is designed to scale to high lane counts (up to 16 lanes) and focuses on maximizing floating-point utilization for complex kernels like those found in training and heavy inference. Its inclusion enables the benchmarking of "scale-up" scenarios where raw computational power and energy efficiency are the primary metrics.

Vicuna, in contrast, is selected to represent the safety-critical and embedded domain. It implements the Zve32x subset of the vector extension (integer only) and prioritizes timing predictability over maximum average-case throughput. Vicuna's design ensures freedom from timing anomalies, making it a unique platform for studying how vectorization can be applied in real-time systems where Worst-Case Execution Time (WCET) bounds are mandatory. Its inclusion allows the research to explore the trade-offs between determinism and performance, a critical consideration for ML deployment in autonomous vehicles and industrial control systems.

By juxtaposing these two cores, the benchmarking environment covers the full breadth of the RISC-V vector ecosystem, from the cloud (Ara) to the edge (Vicuna), providing a nuanced and exhaustive analysis of hardware RTL cores for ML workloads.

5.2 Ara Vector Processor

5.2.1 Overview and Design Motivation

Ara is a 64-bit vector unit (VPU) designed to act as a high-performance coprocessor for the CVA6 (formerly Ariane) scalar core. It is engineered specifically to address the performance requirements of data-parallel workloads found in High-Performance Computing (HPC) and Artificial Intelligence (AI). The architecture is rooted in the historical lineage of vector processing—tracing its conceptual origins to the Cray-1 supercomputer—and aims to mitigate the Von Neumann Bottleneck. By leveraging the Single Instruction, Multiple Data (SIMD) execution model, Ara amortizes the high energy and latency costs of instruction fetching and decoding over large vectors of data, thereby boosting both performance and energy efficiency.

The primary target workloads for Ara are those exhibiting high degrees of data-level parallelism, such as dense linear algebra (e.g., `dgemm`, `sgemm`), convolutions for Deep Neural Networks (DNNs), and scientific computing kernels like Fast Fourier Transforms (FFT). The design motivation explicitly references the Fugaku supercomputer's A64FX processor as a contemporary proof point for the viability of vector architectures in the exascale era, positioning Ara as a RISC-V counterpart aiming for similar efficiency in the open-source hardware domain.

Ara's architectural evolution (specifically the transition from Ara to Ara2) was driven by two overriding goals: strict compliance with the ratified RISC-V Vector Extension version 1.0 (RVV 1.0) and the maximization of floating-point throughput. The transition to RVV 1.0 necessitated significant microarchitectural changes to support new behaviors for masking, element width handling, and vector configuration. The throughput scaling goal enables Ara to support a parametric number of lanes ranging from 2 to 16, allowing the processor to be instantiated in various PPA configurations while achieving high functional unit utilization (targeting $> 90\%$ on compute-bound kernels).

5.2.2 Architectural Organization

The Ara system operates as a coherent coprocessor system. The scalar host core, CVA6, is responsible for the control plane: it fetches instructions, handles interrupts, and performs scalar computations. When CVA6 encounters a vector instruction, it offloads the instruction to Ara via a dedicated accelerator interface. This interface is designed to be non-speculative, meaning instructions are dispatched only when they are committed by the scalar core, simplifying the vector unit's control logic by removing the need for complex rollback mechanisms in the event of branch mispredictions.

The system shares a unified memory hierarchy. Both CVA6 and Ara access main memory via an AXI interconnect. A critical component of this organization is the enforcement of memory consistency between the scalar and vector domains without requiring software-managed coherence (explicit fences), a significant architectural enhancement over previous iterations.

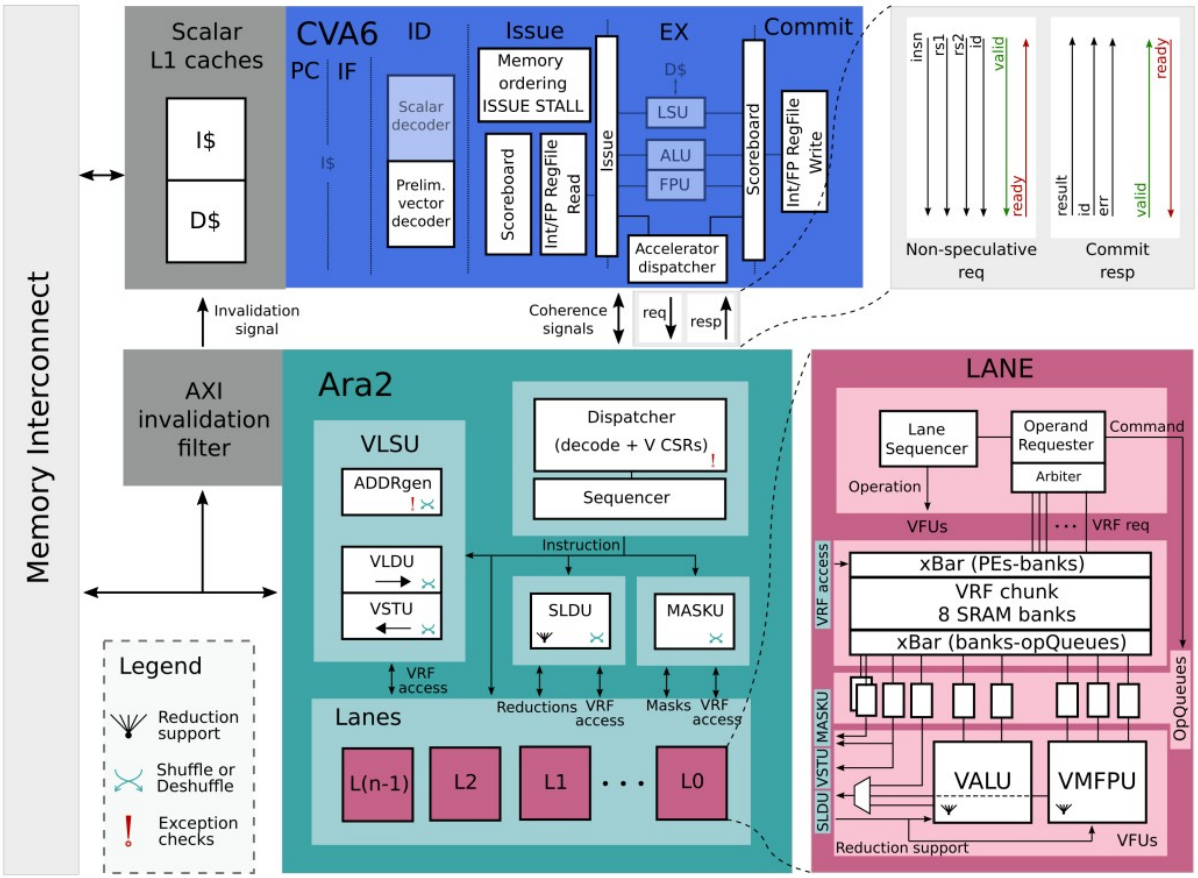

Figure 5: Top-level block diagram of the Ara2 system showing the vector coprocessor, detailed lane architecture, and host scalar core CVA6 integration.

Ara employs a scalable, lane-based architecture. The computational resources and the Vector Register File (VRF) are distributed horizontally across L lanes. Each lane acts as an independent slice of the vector machine, containing a portion of the VRF and dedicated functional units. In a 16-lane configuration, the processor can effectively compute 16 double-precision (64-bit) operations per clock cycle.

Within each lane, Ara instantiates specific execution units: a Vector ALU (VALU) for integer arithmetic and logical operations, a Vector FPU (VFPU) for floating-point arithmetic (FMA, Add, Mul) with native double-precision (FP64) support, and a Vector Multiplier (VMUL) specialized for integer multiplication. Ara utilizes a standard lane-striped layout where consecutive elements of a vector are placed in consecutive lanes (e.g., Element 0 in Lane 0, Element 1 in Lane 1).

5.2.3 RVV Implementation

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

- **Data Types:** IEEE-754 floating-point (FP16, FP32, FP64) and standard integers (INT8, INT16, INT32, INT64)
- **Reductions:** Full support for vector reduction operations, including ordered and unordered floating-point reductions requiring careful pipeline management
- **Masking:** Comprehensive support for masked execution where operations on specific elements can be suppressed based on a mask register
- **Permutations:** Instructions such as `vslideup`, `vslidedown`, and `vgather`/`vscatter` supported through on-chip interconnect

5.2.4 Vector Execution Model

Ara adheres to the Vector Length Agnostic (VLA) programming model mandated by the RISC-V specification. The hardware parameter VLEN (bits per vector register) can vary between instantiations, but the software binary remains compatible. At runtime, the `vsetvli` instruction configures the application vector length (AVL). Ara's control logic automatically stripes this AVL across the available lanes. If the requested vector length exceeds the physical capacity of the parallel lanes ($VL > Lanes$), the hardware "strip-mines" the operation in hardware, executing the vector in temporal chunks.

A defining complexity of RVV 1.0 is the support for dynamic Single Element Width (SEW) changes. Because Ara maps consecutive elements to consecutive lanes, changing the SEW changes which lane holds a specific logical element. When an instruction uses source operands with different SEWs (mixed-width operations), Ara injects reshuffle micro-operations. Before the main arithmetic operation executes, the Slide Unit (SLDU) is engaged to realign the data bytes across the lanes to match the target SEW.

The execution pipeline is decoupled. Once an instruction is dispatched from CVA6 to Ara's Dispatcher, it enters an instruction queue (Issue Queue). Instructions are issued to the functional units when operands are available and execute in a SIMD manner. Ara supports vector chaining, allowing a dependent instruction to begin execution before the predecessor has fully completed, provided the necessary elements are available. This is essential for keeping the deep pipelines of the FPU utilized during complex sequences like `vfmacc` (fused multiply-accumulate).

5.2.5 Memory Subsystem

The Vector Load/Store Unit (VLSU) is the interface between the high-bandwidth AXI memory system and the parallel vector lanes. Its primary responsibility is to fetch data from memory and align it to the correct lanes in the VRF. The VLSU is one of the most complex units in the design, with area and complexity scaling superlinearly with the number of lanes ($O(L^2)$) because it must route data from a fixed-width memory bus to any of the lanes depending on the stride and element index. It handles unit-stride loads (contiguous blocks), strided loads (regular gaps), and indexed loads (scatter/gather).

Ara2 introduces a robust hardware coherency scheme. The CVA6 data cache (L1-D) is configured in write-through mode, ensuring that any data written by the scalar core is immediately visible in main memory where Ara reads its data. When Ara performs a vector store, it sends invalidation signals to CVA6, forcing it to invalidate lines corresponding to the addresses written by the vector unit. This hardware mechanism eliminates the need for fence instructions to maintain coherence between scalar and vector memory views.

5.2.6 RTL Implementation

Ara is implemented in System Verilog and is designed to be technology-agnostic, though it is optimized for modern ASIC nodes. The reference implementation is characterized in 22nm FD-SOI technology, achieving a target frequency of 1.35 GHz for configurations up to 8 lanes. The critical path is approximately 40 FO4 (Fan-Out-of-4) inverter delays, indicating a moderately aggressive pipeline depth suitable for high-performance operation.

The design is highly parameterized with lane counts of 2, 4, 8, or 16. As lane count increases, the area of the functional units scales linearly. However, the area of the interconnects—specifically the Slide Unit (SLDU) and the Mask Unit (MASKU)—scales superlinearly. Ara2 implements a specific optimization restricting the SLDU to power-of-two strides, reducing the wiring complexity from $O(L^2)$ to $O(L \log L)$ and enabling feasibility at 16 lanes.

5.2.7 Benchmarking Suitability

Ara is exceptionally well-suited for benchmarking compute-bound ML kernels. For large matrices (e.g., 128×128), Ara achieves 97-99% FPU utilization, indicating that the microarchitecture successfully hides memory latency and control overhead. The bit-accurate nature of Ara allows researchers writing kernels using RVV intrinsics to precisely tune their code, verifying lane utilization and optimizing register allocation. Research highlights that for smaller problem sizes, multi-core configurations with smaller vector units can outperform single large units, providing critical insights for architectural trade-offs.

5.3 Vicuna RISC-V Vector Coprocessor

5.3.1 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.

5.3.2 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 and supports different internal pipeline organizations. In the **Compact** configuration, all functional units share a single pipeline. In the **Dual/Triple** configuration, units are distributed across multiple parallel pipelines (e.g., Memory Unit in Pipeline A, ALU in Pipeline B), allowing for concurrent execution of loads and arithmetic while improving efficiency without introducing unpredictable hazards.

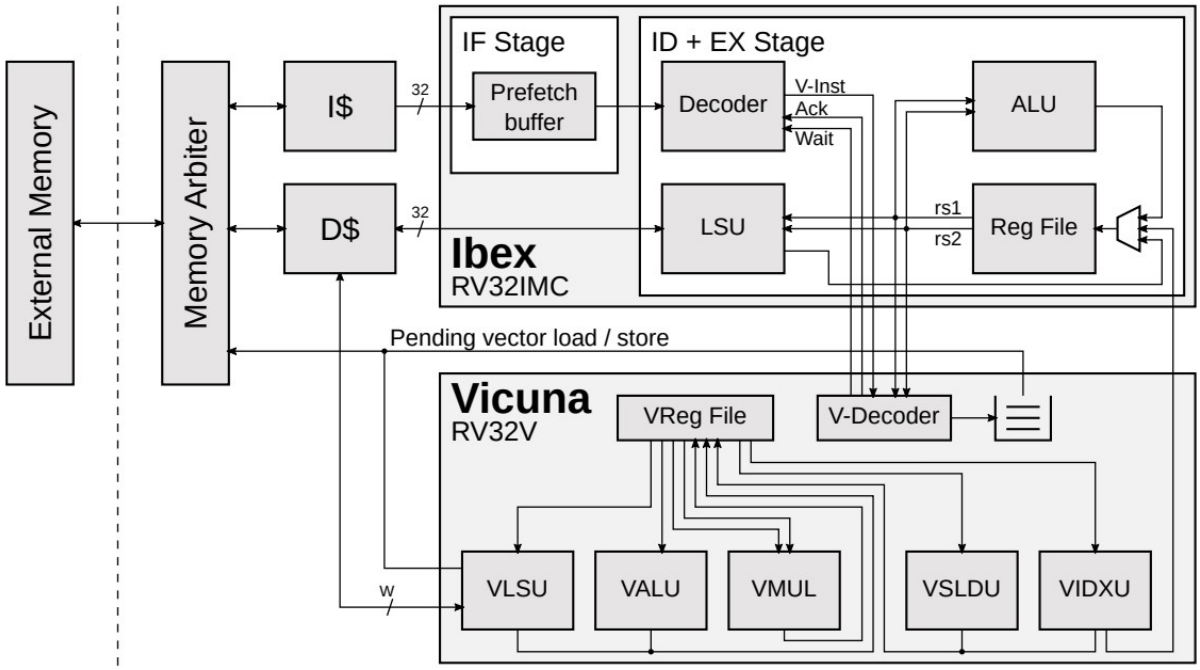

Figure 6: Overview of Vicuna's architecture and its integration with the Ibex main core. Both cores share a common data cache with predictable memory arbitration ensuring deterministic timing behavior.

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.

5.3.3 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.

5.3.4 Execution Model

Vicuna strictly adheres to in-order execution. Instructions are issued to the functional units only when all dependencies are resolved. The pipeline is designed such that a local timing variation (e.g., a stall) never induces a larger global timing variation, allowing for compositional timing analysis where the timing of the vector unit can be analyzed independently of the scalar core's complex state.

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.

5.3.5 Memory Subsystem

Vicuna supports the standard RVV memory access patterns: unit-stride, strided, and indexed (scatter/-gather). To maintain predictability, the handling of these accesses is strictly serialized or arbitrated in a fixed manner. Vicuna employs a specialized memory arbiter to ensure that vector memory accesses do not cause counter-intuitive interference with the scalar core. If a vector load is active, the scalar core might stall, but the duration of this stall is bounded and predictable, allowing system architects to calculate worst-case scenarios for critical interrupts.

5.3.6 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).

5.3.7 Benchmarking Suitability

Vicuna serves as the baseline for embedded efficiency in the thesis benchmarking suite, representing the "constrained" design point. Benchmarking on Vicuna allows for the construction of Roofline models for embedded devices, demonstrating that for compute-bound kernels like GEMM, Vicuna achieves $> 90\%$ efficiency. Since Vicuna focuses on integer arithmetic, it is ideal for evaluating 8-bit quantized neural networks, providing direct insight into how RISC-V vectors can accelerate edge AI applications without the power and area cost of floating-point hardware.

5.4 Comparative Analysis: Ara vs. Vicuna

The most fundamental difference between Ara and Vicuna lies in their architectural philosophy: Ara is an Application-Class Processor targeting maximum throughput, while Vicuna is an Embedded-Class Coprocessor prioritizing timing predictability.

Table 1: Architectural Comparison of Ara and Vicuna

Feature	**Ara (Ara2)**	**Vicuna**
ISA Compliance	RVV 1.0 (Full, incl. FP64)	Zve32x (Integer Only)
Host Core	CVA6 (Linux-capable, 6-stage)	Ibex (RTOS-capable, 2-stage)
Primary Goal	Maximize Throughput	Maximize Predictability (WCET)
Scaling Mechanism	Lane Replication (2-16)	Pipeline Parallelism
Implementation	ASIC (22nm FD-SOI)	FPGA (Xilinx 7 Series)
Data Path	64-bit (Double Precision)	32-bit (Integer)
Target Frequency	1.35 GHz	80 MHz
Peak Performance	97-99% FPU utilization	$> 90\%$ efficiency
Memory Coherency	Hardware coherent	Predictable arbitration

Execution Model: Ara's execution model allows for dynamic optimization through chaining, out-of-order writebacks (within scoreboard bounds), and complex reshuffling, maximizing utilization but making exact cycle-count prediction difficult. Vicuna's in-order, monotonic model guarantees that if a task takes N cycles once, it will always take N cycles. Ara uses a scoreboard and hazard detection logic to stall the pipeline dynamically, while Vicuna relies on a stricter structural hazard resolution strategy that prevents hazards by construction.

Memory Subsystem: Ara employs a hardware-coherent memory system where interactions between the CVA6 write-through cache and the vector unit are managed automatically, enabling seamless shared-memory programming. Vicuna uses a predictable memory arbiter with strict access ordering to ensure the vector and scalar units do not interfere in unpredictable ways, simplifying hardware at the cost of more careful software management.

Benchmarking Trade-offs: Ara excels at floating-point benchmarks (`dgemm`, `sgemm`), pushing the limits of RVV 1.0 in terms of raw FLOPs, but incurs high simulation cost and complexity due to super-linear scaling of interconnects. Vicuna serves as the definitive reference for hard real-time vectorization, proving vectors are safe for safety-critical systems, with lightweight and fast simulation, but is limited to integer workloads and cannot benchmark floating-point training kernels.

This duality ensures that the benchmarking results are robust, covering the diverse requirements of the modern computing spectrum from cloud (Ara) to edge (Vicuna) deployments.

5.5 Validation Strategy

[Placeholder]

5.6 Validation Results

[Placeholder]

6 Open Source Library Architecture

[Placeholder]

7 Conclusion & Future Work

[Placeholder]

8 Acknowledgment

[Placeholder]

References

[1] Perotti, M., Cavalcante, M., Andri, R., Cavigelli, L., & Benini, L. (2024). Ara2: Exploring single- and multi-core vector processing with an efficient RVV 1.0 compliant open-source processor. *IEEE Transactions on Computers*, 73(7), 1822–1836. https://doi.org/10.1109/TC.2024.3388896

[2] 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

9 Code Listings

[Placeholder]