POLITECNICO
MILANO 1863

HEAP LAB

HW5: ML Compilation and Execution with IREE on RISC-V Embedded Systems - AA 2025-2026

Kristi Duro, 273213
Supervisors: Prof. Davide Zoni, Dr. Andrea Galimberti, Adriano Guarisco

1 Introduction

The deployment of Deep Learning (DL) models [5] on embedded architectures requires highly optimized compiler toolchains to bridge the gap between high-level frameworks (like PyTorch or ONNX) and bare-metal hardware. This project explores the **IREE** (Intermediate Representation Execution Environment) toolchain [1] targeting the **RISC-V** Instruction Set Architecture (ISA).

The primary objective is to evaluate the functional correctness and efficiency of the **RISC-V Vector Extension (RVV)** by comparing scalar (baseline) and vectorized implementations of various neural network architectures. We analyze execution time, memory footprint, and vector instruction utilization across a suite of models including MobileNetV2 [6], SqueezeNet [2], TinyBERT [3], and Network in Network (NiN).

2 Problem Analysis

Compilation for embedded RISC-V targets presents unique challenges compared to standard x86/ARM deployment:

1. **Graph Lowering**: Mapping complex operators (e.g., Depthwise Convolution, Self-Attention) to vector primitives requires sophisticated Intermediate Representation (MLIR) transformations.
2. **Auto-Vectorization Heuristics**: The LLVM backend must decide when to vectorize. Architectures with large, irregular kernels (like AlexNet [4]) often fail heuristic checks, falling back to scalar code.
3. **Resource Constraints**: Embedded devices have strict memory limits [8]. Enabling vectorization often increases binary size and runtime memory overhead, creating a trade-off between performance and footprint.

3 Method

3.1 Experimental Environment

The experiments were conducted on a host machine featuring an **Apple M3 Pro (ARM64)** processor running macOS. To ensure reproducibility and resolve cross-platform compatibility issues, the entire compilation and execution pipeline was encapsulated within a **Docker** container running **Ubuntu 22.04 LTS**.

Rationale for Docker Usage:

- **Toolchain Compatibility:** The IREE compiler and LLVM RISC-V toolchain are primarily distributed as x86_64 Linux binaries. Running these natively on macOS (ARM64) is non-trivial. Docker provides a standardized Linux environment.
- **Architecture Translation:** Since the host is ARM64 and the toolchain is x86_64, Docker Desktop on macOS utilizes **Rosetta 2** to

translate the x86 container instructions to the host ARM silicon.

Emulation Stack: The final execution involved a multi-layered translation stack, which significantly impacts performance profiling:

1. **Target:** RISC-V Vector code (rv64gcv).
2. **Emulator:** QEMU User Mode (`qemu-riscv64`).
3. **Container:** Ubuntu 22.04 (x86_64) running via Rosetta 2 translation.
4. **Host:** Apple M3 Pro (ARM64).

3.2 Model Pipeline

Models were sourced from the ONNX Model Zoo or exported from PyTorch. A custom preprocessing pipeline was established:

- **Opset Standardization**: Models were upgraded to ONNX Opset 17 using `upgrade_model.py`, with specific exceptions (e.g., MobileNetV2) where upgrading altered the graph structure and broke vectorization.
- **Manual Definition (NiN)**: Due to the obsolescence of official repositories for Network in Network, the canonical ImageNet architecture was manually redefined in PyTorch and exported to ONNX to ensure compatibility.
- **Compilation**: Each model was compiled twice using `iree-compile` to isolate the performance impact of vectorization.

3.3 Compilation Strategy

We employed a two-pass compilation strategy to strictly isolate the impact of the vector backend.

1. Scalar Baseline (SISD): The first pass utilized standard RISC-V extensions (`+m,+a,+f,+d`) to generate a generic 64-bit baseline.

- `+m, +a`: Enable hardware integer multiplication and atomic memory operations.
- `+f, +d`: Enable single and double precision floating-point units.

2. Vector Optimized (SIMD): The second pass enabled the RISC-V Vector Extension (RVV) with the flags `+zvl512b,+v`.

- `+v`: Enables the vector instruction set (e.g., `vle32.v`, `vfmacc`), allowing the CPU to use the 32 vector registers.
- `+zvl512b`: Explicitly informs the compiler that the target hardware (QEMU) has a fixed vector length of **512 bits**. This allows the LLVM backend to perform optimal loop unrolling and register packing.

4 Results

The benchmark results are summarized in Table 1. A clear dichotomy is observed: modern architectures successfully vectorized, while legacy models reverted to scalar execution.

Table 1: Scalar vs. Vector Benchmarks.

Model	**Type**	**Time** (ms)	**Mem** (KB)	**Instr** (Vec)
MobileNet	Scalar	10,516	208,248	0
	Vector	104,727	459,160	**307k**
SqueezeNet	Scalar	8,313	212,596	0
	Vector	100,169	464,632	**286k**
TinyBERT	Scalar	400	296,984	0
	Vector	64,720	393,332	**18k**
AlexNet	Scalar	120	22,612	0
	Vector	90	22,476	0
NiN	Scalar	370	64,928	0
	Vector	320	64,960	0

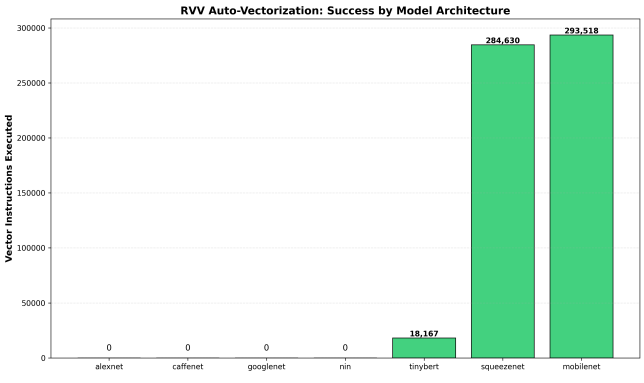

Figure 1: Vector Instruction Count. MobileNet and SqueezeNet show massive auto-vectorization success, while legacy models (AlexNet, NiN) fail completely.

4.1 Vector Utilization Analysis

Success Cases
MobileNetV2 and SqueezeNet achieved high vector counts. This is attributed to their heavy use of 1×1 pointwise convolutions, which IREE lowers into regular, dense loops that map efficiently to vector instructions. TinyBERT also vectorized successfully due to its reliance on matrix multiplication primitives.

Zero-Vectorization Cases
AlexNet, CaffeNet, GoogleNet [7], and **Network in Network (NiN)** showed **0 vector instructions**. The failure of NiN is particularly notable. Despite introducing 1×1 "Mlpconv" layers, the architecture's large 11×11 input kernel with stride 4 produces strided memory access patterns that inhibit LLVM vectorization.

5 Discussion

5.1 Emulation Overhead & Docker Analysis

A prominent anomaly is the significant slowdown of vector code compared to scalar code, as shown in Figure 2. This is an artifact of the complex emulation stack described in the Method section. QEMU must translate every complex 512-bit RISC-V vector instruction into hundreds of x86 instructions, which Rosetta 2 then translates to ARM64 micro-ops.

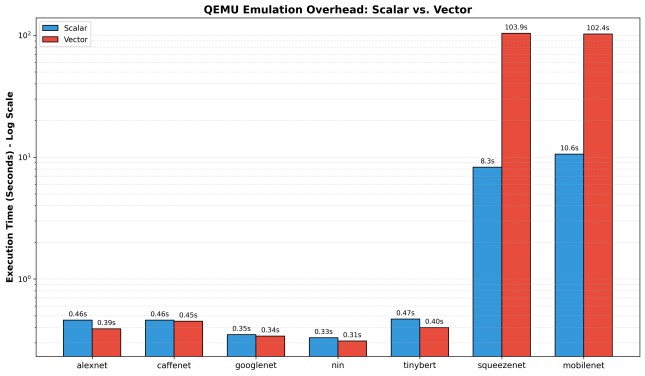

Figure 2: Execution Time (Log Scale). The 10x slowdown in vector mode confirms heavy emulation overhead.

The Docker resource monitor (Figure 3) confirms that the workload is compute-bound. The distinct blue CPU spikes (reaching 400-1000% usage) correspond to the multi-threaded translation engine struggling with vector workloads. Notably, the memory usage shows distinct "plateaus" where vector runs consume significantly more RAM than scalar runs.

Figure 3: Host Resource Usage. Blue spikes indicate high CPU load during QEMU vector emulation.

5.2 Edge Suitability & The Memory Wall

Based on the memory profiling results (Figure 4), we can categorize the suitability of these models for different classes of edge hardware.

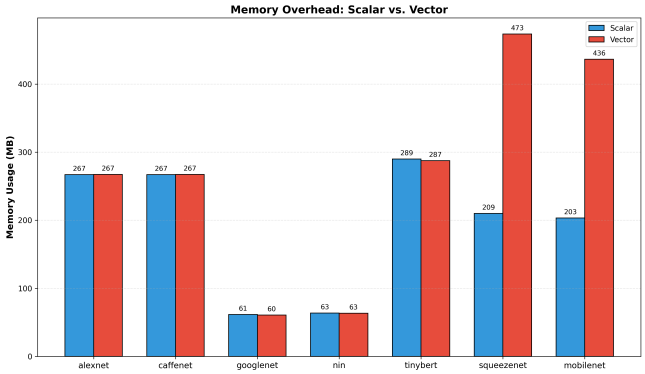

Figure 4: Memory Footprint. Vectorization introduces significant runtime memory overhead (Red bars).

1. High-End Edge (e.g., Jetson Orin, RPi 5): Models like MobileNetV2 (requiring 460 MB in Vector mode) are fully suitable. These devices typically feature 2GB+ of RAM, comfortably accommodating the runtime overhead.

2. Mid-Range Edge (e.g., RPi Zero 2 W): With typically 512MB of RAM, these devices are at the limit. Running MobileNetV2 (459 MB) leaves

almost no room for the operating system. To deploy here, we would need to revert to the Scalar implementation (208 MB).

3. Low-Power/Microcontrollers (e.g., STM32): None of the tested vector configurations are suitable. The 400MB requirement far exceeds the SRAM limits (¡1MB) of microcontrollers.

6 Conclusions

This study demonstrated that while modern architectures like MobileNet and Transformers map efficiently to the RISC-V Vector extension, they introduce a significant **Time-Space Tradeoff**. The current IREE implementation doubles the memory footprint to achieve vectorization.

To contextualize these results, Table 2 highlights the fundamental differences between our experimental simulation and theoretical silicon performance. Future work should involve deployment on physical hardware (e.g., SiFive X280) to validate the expected 4x-8x performance gains masked by simulation overhead.

Table 2: Simulation vs. Silicon Performance

Metric	QEMU (Current)	SiFive X280
Execution	Serial Emulation	Parallel Hardware
Throughput	10x Slower	4x-8x Faster
CPU Load	100% (Host)	Offloaded (NPU)

References

[1] T. I. Authors. Iree: Intermediate representation execution environment. `https://github.com/iree-org/iree`, 2024. Accessed: 2026-01-18.

[2] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[3] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[5] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[6] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[8] D. Zoni. *Hardware Projects: Embedded Systems A.Y. 2025/2026*. Politecnico di Milano, 2025. [HW5] ML compilation and execution with IREE on RISC-V, Page 21.