Chair of Electronic Design Automation
Department of Computer Engineering
School of Computation, Information and Technology
Technical University of Munich

Benchmarking TinyML CNN Kernels on RVV 1.0 Hardware: GCC 14 vs. LLVM 19

Philipp van Kempen, Benedikt Witteler, Jefferson Parker Jones, Daniel Mueller-Gritschneder, Ulf Schlichtmann

Motivation

SIMD vs. Vector Processing

- SIMD (Single Instruction Multiple Data): Fixed length, software hard to maintain
- Vector: Length-agnostic, one software fits all

TinyML

- Run AI inference locally on low-power MCUs instead of in the cloud
- Advantages: Greater privacy, lower power and cost, reduced latency
- Disadvantages: Limited computational resources and memory

Auto-Vectorization vs. Manual Vectorization

- Capability of handling SIMD/Vector workloads automatically vs. benefits from enhanced optimizations in manually written vectorized code

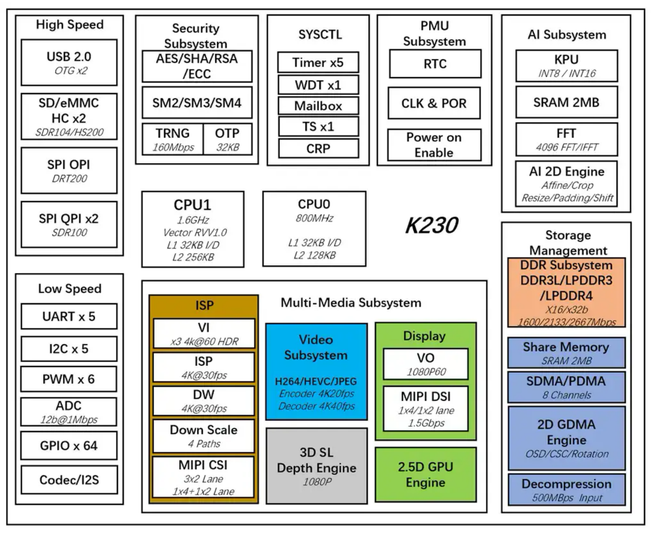
Hardware Background

RISC-V Vector Extension (RVV)

- Extends the scalar RISC-V ISA by 32 new vector registers of length `VLEN` bits each
- Supports dynamic register grouping and vector-length-agnostic operations
- Ratified in 2021 as RVV 1.0

Used Hardware [5]

- **CanMV K230 Board:** T-Head XuanTie C908 core, 512 MiB LPDDR3 RAM, microSD card
- **C908 Core:** Vector length (`VLEN`) of 128 bits and SIMD datapath composed of two 64-bit-wide execution units



Software Background

muRISCV-NN [2]

- Lightweight library of optimized ML kernels for embedded RISC-V
- Designed to leverage RVV for quantized deep learning workloads

Software Toolchains / Compilers

- **LLVM:** Partial support for RVV auto-vectorization since version 14 in 2022
- **GCC:** Initial support for RVV auto-vectorization in 2024
- **LLVM 19 and GCC 14:** This research extends prior work using LLVM 17 and GCC 14 by analyzing the most recent toolchain versions. Further, auto- and manual vectorization are compared on real hardware rather than in an instruction set simulator

Conclusion

Summary

- Compiler flags and vectorization **consistently affect code size**, while **runtime tuning remains task-specific**
- LLVM 19 shows the best trade-off between code size and performance on K230 hardware, whereas GCC tends to favor one extreme over the other
- LLVM shows more advanced handling of both auto- and manually vectorized code
- GCC's recent RVV support (since 2024) narrows the performance gap to LLVM

Challenges / Lessons Learned

- Vendor-provided RTOS requires custom GNU Toolchain ⇒ Use Linux instead

Outlook / Future Work

- Evaluate on more **powerful RVV 1.0 hardware** (e.g. $VLEN \geq 256$)
- Check further workloads/models (e.g. FP32)
- Explore compiler **heuristics/cost functions**
- Port **real-world application** (e.g. person detection) to K230 board

Experiments

Setup

- **Toolchains:** LLVM 19 vs. GCC 14
- **Models:** Quantized Convolutional Neural Networks (CNNs) from TinyMLPerf Suite [3]:
  - Image Classification (`resnet`)
  - Speech Recognition (`aww`)
  - Visual Wake Words (`vww`)
- **ML Deployment**
  - Framework: Tensorflow Lite for Microcontrollers (TFLM) [1]
  - Kernels: muRISCV-NN Scalar (Unvectorized and Auto-Vectorized) vs. muRISCV-NN Vector (Manually Vectorized)
  - Benchmarking: MLonMCU Flow [4]

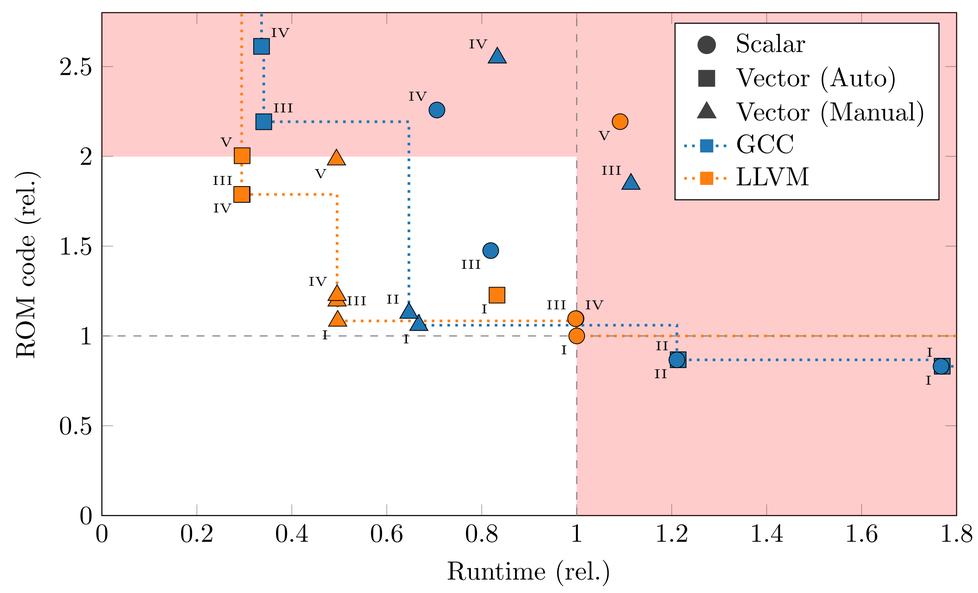
Compiler Configurations

	Optimize	Unroll	GCC	LLVM
I	Size (`-Os`)		✓	✓
II	Size (`-Os`)	✓	✓	
III	Speed (`-O3`)		✓	✓
IV	Speed (`-O3`)	✓	✓	✓
V	Speed (`-O3`)	✓[a]		✓

[a] Custom LLVM Unrolling heuristic

Results

For `resnet` model:



- ResNet and additional **CNN models** show **consistent performance** trends
- GCC achieves higher execution speeds at the cost of increased code size in scalar configurations III and IV
- In configurations I and II, GCC produces code that is 15 to 20% smaller while introducing an up to 75% runtime increase compared to LLVM
- Auto-vectorization impact negligible in configurations I vs. II ⇒ Only **manual vectorization yields optimal trade-offs** between runtime and code size

Tradeoffs Summary

Config	TC	Kernel	Vect.	Runtime	Code Size
I	GCC	Scalar/Vector	Auto	●●●●●	●●●●●
I	LLVM	Vector	Auto	●●●●	●●●
I	LLVM	Vector	Manual	●●●	●●●
III	GCC	Vector	Auto	●●	●●●●●●
III	LLVM	Vector	Auto	●	●●●●

References

[1] David, Robert, et al. "Tensorflow lite micro: Embedded machine learning for tinyml systems." Proceedings of Machine Learning and Systems 3 (2021): 800-811.
[2] van Kempen, P., Jones, J. P., Mueller-Gritschneder, D., Schlichtmann, U. (2024, May). Muriscv-nn: Challenging zve32x autovectorization with tinyml inference library for risc-v vector extension. In Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions (pp. 75-78).
[3] Banbury, C., Reddi, V. J., Torelli, P., Holleman, J., Jeffries, N., Kiraly, C., Xuesong, X. (2021). Mlperf tiny benchmark. arXiv preprint arXiv:2106.07597.
[4] van Kempen, Philipp, et al. "Mlonmcu: Tinyml benchmarking with fast retargeting." Proceedings of the 2023 Workshop on Compilers, Deployment, and Tooling for Edge AI. 2023.
[5] https://developer.canaan-creative.com/k230/dev/00_hardware/K230_datasheet.html

This work has been developed in the project Scale4Edge funded by the German Federal Ministry of Education and Research (BMBF) under contract no.16ME0127. The authors are responsible for the content of this publication.

SPONSORED BY THE Federal Ministry of Education and Research

Contact:
philipp.van-kempen@tum.de

Open Source:
https://github.com/tum-ei-eda/muriscv-nn