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

Philipp van Kempen[1]*, Benedikt Witteler[1], Daniel Mueller-Gritschneder[2] and Ulf Schlichtmann[1]

[1]Chair of Electronic Design Automation, Technical University of Munich, Munich, Germany
[2]Institute of Computer Engineering, TU Wien, Vienna, Austria

Abstract

This paper evaluates CNN workloads on the RISC-V Vector Extension (RVV) 1.0 using GCC 14 and LLVM 19. We benchmark auto-vectorized and manually optimized TinyML kernels on a hardware platform, analyzing runtime and code size trade-offs. Results show that LLVM 19 provides a better balance between both metrics, while GCC 14 exhibits greater variability.

Introduction

With each new release of software toolchains such as GCC and LLVM, support for vector and single-instruction multiple-data (SIMD) instruction set architectures (ISAs) advances. Auto-vectorization is becoming increasingly capable of handling a broader range of workloads and patterns. At the same time, manually written vectorized code benefits from enhanced optimizations, improved heuristics, and more refined cost models. The presented results provide insights into optimizing RISC-V SW and ML deployment toolchains, as well as kernel libraries for TinyML workloads.

Background

The RISC-V Vector Extension extends the scalar RISC-V ISA by 32 new vector registers of length VLEN bits each, supporting dynamic adjustments and grouping of vector registers, while maintaining a vector-length agnostic design. Since its ratification in 2021, the first commercially available hardware to implement RVV 1.0 is the *T-Head XuanTie C908* core, integrated into the CanMV K230 board. muRISCV-NN [1] is a suite of optimized deep learning kernels for embedded systems. It addresses the need for a lightweight compute library that can utilize RVV for quantized ML workloads. While both the LLVM Framework [2] and the RISC-V GNU Toolchain offer auto-vectorization capabilities for RVV 1.0, achieving efficient vectorization remains a challenge, as highlighted in [3]. The auto-vectorization performance of LLVM 17 and GCC 14 is analyzed in [4] and [5]. This report extends their analysis by evaluating the latest compiler versions, LLVM 19 and GCC 14, and further comparing their auto-vectorization against manually vectorized kernels using muRISCV-NN on real hardware.

*Corresponding author: philipp.van-kempen@tum.de
The work is supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127.

Experiments

In contrast to the experiments performed in [1], which have been conducted on an instruction set simulator (ISS), the following analysis is based on the K230 hardware platform. The heterogeneous SoC features a C908 core with a small vector length of 128 bit and an SIMD datapath made up of two 64 bit-wide execution units. A Linux-based execution environment running on the RVV-enabled core was chosen for benchmarking, as the vendor-provided RTOS lacks compatibility with upstream versions of the RISC-V GNU toolchain.

The MLPerf Tiny benchmarking suite [6], consisting of four quantized neural networks (QNNs) optimized for microcontrollers, was employed in this study. Given the integer nature of these models , only the performance of integer vector operations is analyzed. The TensorFlow Lite for Microcontrollers (TFLM) framework [7] was used to execute the models on the embedded device. Three kernel variants were evaluated:

- **Scalar**: Standard kernel implementations compiled for `RV64GC`.
- **Vector (Auto)**: Standard kernel implementations compiled with auto-vectorization enabled (`RV64GCV`).
- **Vector (Manual)**: Manually vectorized kernel implementations (`RV64GCV`).

To ensure fair comparisons, the manually vectorized implementations were compiled with auto-vectorization explicitly disabled, preventing any unintended performance degradation due to conflicting compiler optimizations. Experiments were conducted across up to five different configurations, as summarized in Table 1. Configuration II is skipped for LLVM as it yields the same results as configuration I.

The primary performance metrics considered in this evaluation include the runtime improvement achieved through vectorization and the associated code size overhead, specifically measured using the static library `libmuriscvnn.a`. Since TinyML applications rely ex-

Table 1: *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

clusively on statically allocated memory buffers, the impact of auto-vectorization on the RAM usage is negligible and therefore excluded from the discussion.

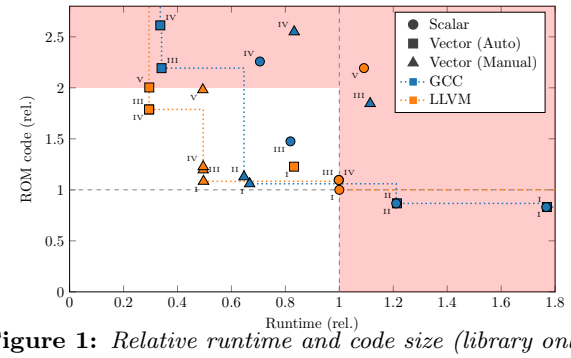

Figure 1: *Relative runtime and code size (library only). Toolchain (color) performance compared for different kernels (markers) and configurations (see Table 1).*

The results for a residual neural network (ResNet) are depicted as a scatter plot in Figure 1. Further CNN models have been evaluated and show similar results. To enable a relative comparison of the obtained performance metrics, scalar kernels compiled with LLVM without loop unrolling were selected as the baseline (dotted lines). An analysis of the scalar performance reveals that GCC generally achieves higher execution speed at the cost of increased code size in configurations III and IV. Conversely, in certain cases (I & II), GCC produces code that is $15 - 20\%$ smaller while introducing an up to 75% runtime increase compared to LLVM. To balance these trade-offs between runtime efficiency and code size, the following constraints are introduced (highlighted as red areas):

- The execution time must not exceed that of the selected baseline.
- The code size overhead must not exceed 200%.

Applying these constraints results in the elimination of certain configurations, specifically the auto-vectorized GCC implementations, as well as configurations III and IV for manually vectorized kernels compiled with GCC. Further data points can be excluded if they are dominated by others, which can be determined by constructing a Pareto front (dotted lines) within the remaining feasible region. This eliminates the configurations III-V for manually vectorized kernels compiled with LLVM. Overall, the auto-vectorization capabilities are very limited or non-existent in configurations I vs. II leading to the conclusion that optimal code size while using vector instructions can only be achieved by using manually written kernels.

Conclusion

The experimental results indicate that the impact of compiler optimization flags, loop unrolling, and vectorization on code size overhead remains consistent across the evaluated benchmarks. However, the selection of optimal compiler flags for optimal runtime performance is highly task-dependent. On the given target hardware, LLVM 19 provides the best balance between code size and performance, whereas GCC tends to favor one extreme over the other. Additionally, LLVM demonstrates greater sophistication in compiling both auto-vectorized and manually vectorized code. Considering LLVM has offered partial support for RVV auto-vectorization since version 14 in 2022, while GCC only began integrating support in 2024, GCC's performance is not significantly behind. It's also important to note that these conclusions are based on quantized CNNs and may not generalize to other workloads. As toolchain releases continue to evolve, further improvements are expected, making the comparison more competitive. The muRISCV-NN library, along with the scripts used for this analysis, are available online[1].

References

[1] Philipp van Kempen et al. "muRISCV-NN: Challenging Zve32x Autovectorization with TinyML Inference Library for RISC-V Vector Extension". In: *21st ACM International Conference on Computing Frontiers*. CF '24 Companion. Ischia, Italy: Association for Computing Machinery, 2024, pp. 75–78.

[2] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[3] Neil Adit and Adrian Sampson. "Performance Left on the Table: An Evaluation of Compiler Autovectorization for RISC-V". In: *IEEE Micro* 42.5 (2022), pp. 41–48.

[4] Eugeni Casadesus et al. "Deploying Neural Networks on RISC-V with VPU". In: *2024 39th Conference on Design of Circuits and Integrated Systems (DCIS)*. IEEE. 2024, pp. 1–6.

[5] Klara Modin. *A comparison of auto-vectorization performance between GCC and LLVM for the RISC-V vector extension*. 2024.

[6] Colby Banbury et al. "Mlperf tiny benchmark". In: *arXiv preprint arXiv:2106.07597* (2021).

[7] Robert David et al. "Tensorflow lite micro: Embedded machine learning for tinyml systems". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 800–811.

[1] GitHub Repository: `https://github.com/tum-ei-eda/muriscv-nn`